Protecting Software Execution by

DYNAMIC ENVIRONMENT HARDENING

Per Mellstrand

Department of Interaction and System Design
Blekinge Institute of Technology
Protecting Software Execution by DYNAMIC ENVIRONMENT HARDENING

Per Mellstrand
Protecting Software Execution by

**DYNAMIC ENVIRONMENT HARDENING**

Per Mellstrand
Contact Information
Per Mellstrand
Department of Interaction and System Design
School of Engineering
Blekinge Institute of Technology
Box 520
372 25 Ronneby
Sweden
Email: per.mellstrand@bth.se, per@mellstrand.com
Abstract
Software has an important role in many systems, in particular in critical systems where the correct function of the larger system is of utmost importance. Software malfunction, or software exploits, that allow other system components or an adversary to control the execution of software, can cause significant damage to the system, and hence to what is controlled by the system.

Due to the complexity in, and construction of, modern software systems, it is not sufficient to only analyze software source, such as source code, trying to find vulnerabilities that might enable exploits when executing. Instead, software protection must be extended with methods that consider the actual software execution, typically by executing in parallel with the software, and that protect the software as it executes. By using such methods, we can detect unwanted execution and protect against many exploits where the underlying vulnerability is difficult or impossible to find using only pre-execution, or static, methods.

In this thesis, we present a methodology, and model for protecting software execution. We argue for the merits of dynamic methods, i.e. methods that consider the actual software execution, and we present a number of principles for how dynamic methods can be constructed. We also present three experiment systems that we have constructed, and show how the dynamic methods represented through these systems can be used to protect software execution and hence protect a system against exploits.
Acknowledgements

This work was partly funded by EC grant ENK8-CT-2002-00673, Distributed Intelligence in Critical Infrastructures for Sustainable Power, “CRISP”

I would like to thank all those who have helped me form my interest in computer science, security issues, and science in general. Especially, I would like to thank my supervisor Rune for many interesting, and sometimes upright thought provoking, discussions that not only have helped me in my studies, but also broadened my horizon and made me see many things in a different light. Also, I would like to thank my secondary supervisor Martin for much valuable input, reflections, and inspiring discussions about laborative research.

Finally, I would like to thank my parents and good friends for their support, and for being a part of an environment where we discuss things that matters, and never discuss sports.
# Table of Contents

Chapter 1 Introduction 1  
Introduction ......................................................... 1  
Protecting Software Execution ................................. 3  

Chapter 2 Methodology 5  
Introduction ......................................................... 5  
Building Executable Binaries from Source Code .............. 7  
Traditional Building and Execution of Software ............... 8  
Building Modern Software ........................................... 9  
Executing Modern Software ........................................ 10  
Software Vulnerabilities and Exploits ......................... 11  
Preventing Software Exploits ...................................... 13  
Chapter Summary .................................................... 15  

Chapter 3 Model 17  
Introduction ......................................................... 17  
A Model For Execution Environment Behavior ................. 17  
A Model for Execution Environment Hardening ................ 22  
Validating The Hypotheses .......................................... 26  
Chapter Summary .................................................... 26  

Chapter 4 Principles of Protection 27  
Introduction ......................................................... 27  
Construction of Dynamic Environment Hardening Tools ...... 27  
Principal Function of Dynamic Methods ....................... 30  
Levels of Dynamic Run-Time Protection ....................... 35  
Chapter Summary .................................................... 37
CHAPTER 1

INTRODUCTION

1.1 Introduction

Software is used to control many different types of systems. Some of these systems are what we consider traditional software systems, while other are embedded systems where the software play an essential but indirect role. Software has an important role for many such systems; both for systems where the software is explicitly exposed, as well as for systems where the software is deeply embedded. Should a software component fail, or, even worse, become exploited meaning that other system components or an adversary can control its execution, this will have a large impact on such a system as a whole. Due to inherent properties in modern software, and because of the way we build software systems, it is not always possible to determine in advance the complete chain of consequences for a system, should a software component become exploited. The lack of protection in software systems is a highly relevant problem for many critical systems, and software failures, or exploits, cause serious problems for systems whose function are critical for the society.

Traditionally, critical systems, i.e. systems whose correct function is critical in one or more aspects, have been isolated stand-alone systems. Such systems have had none or very restricted means of communicating with its environment. Recent years have seen a trend of increasing openness in such systems, where critical systems no longer are isolated islands, but communicate using open standards and protocols. For traditional software systems, we have seen a strong trend to connect systems to the Internet or private Internet-like intranets, where important, and possibly business-critical communication, is exchanged between system components over open networks. This trend is not unique to traditional software systems, but is also highly relevant for deeply embedded systems, where we see the vision to build highly communicative systems to enable production of complex services at low prices.
No matter if we look at traditional software or at communicating embedded systems, we see that many complex systems become increasingly interconnected. This is also true for what we refer to as critical systems, where a system malfunction would likely have a large impact on the society. We have already seen some of the effects caused by problems in the electrical power grid, in the form of blackouts, and the difficulty for the society to handle this type of events. For example, the 2003 blackout in the United States and Canada affected 50 million persons, was estimated to cost the society (US $) 50 billion, and was largely caused by software malfunction [22]. The cost for the Swedish and Danish blackout 2004 was estimated to (SEK) 2 billion [21].

Also, traditional software systems have had massive problems related to malfunctioning software. In an era where business critical information is being used on a computer that is directly connected to the Internet, the network that connects the computer to competitors as well as teenage hackers, and where the software used on the computer is inherently insecure and never was designed for this type of usage, we should not be too surprised that certain problems arise. The challenge is to increase the dependability in such systems, as the systems already exist and there is little chance that these will be completely redesigned.

1.1.1 Background

Traditionally, many critical systems were protected by means of isolation - a system that does not communicate with other systems, or that communicates with only a very restricted set of systems, is to some degree protected from problems originating from the outside world. Should some form of simple communication be needed, this could be accomplished using trivial signalling protocols which resulted in comparably strong borders between different systems, and made it possible to limit and predict damage caused by the failure of a single component. Systems that used difficult and obfuscated protocols, where significant work was required just to find the standard specifying the protocol, was sometimes believed to further reduce the risk of someone or something causing problems to the system - if for no other reason, for the sheer difficulty in communicating with the system.

Systems that do not communicate are in a sense quite simple. They are simple in regard to the number of possible states that the executing software must handle, and they are simple to protect as there is no channel for unwanted, or hostile, data to reach the system. In some cases, simple is good; if a problem can be solved using a simple system, there is no need to solve it using a complex one. That being said, there are often other requirements on a system that makes the design open, and the openness has many advantages.

Simple systems can, to some extent, be proven or tested for correctness. If the number of states a software program must handle is very small, and the software can be developed using formal methods, there is a chance that the correctness can be formally proven or at least validated. This approach is used for some high-profile systems, like controlling aircraft operation, and for certain academic toy programs. However, the vast majority of software is not developed using this type
of formal methods, and the correctness of such software cannot formally be proven or even validated to be correct.

Other approaches include testing software for correctness, or to apply some development method in an attempt of creating software that is free from defects. This might possibly work for extremely small and simple programs where the entire state space can be handled. For open and interconnected systems, however, such methods are difficult, in reality meaning impossible, to apply, as the number of states and possible points of interaction become very large. While these methods still are relevant, we believe that it is not realistic to require software in complex systems to be free from all possible defects no matter context. Instead, we must find other ways to protect software execution so that a single vulnerability in a software component cannot damage an entire system.

Open and interconnected systems change the scene of how we can protect execution; systems communicate, and this is a feature, not a problem. Communication is the key property that makes open systems so attractive, but it also makes the software part of such systems more complex and opens up for a new class of problems. In many open systems, there are more problems than just those that indirectly originate from the openness; one such example is that software is used in environments for which it was not originally designed - sometimes in a more hostile environment. Protecting complex, and interconnected software systems is clearly not an easy task, but there are methods and tools that greatly can advance the current protection mechanisms - something that we explore in this thesis.

1.2 Protecting Software Execution

Many protection mechanisms focus on software source, or sometimes at an even higher level, i.e. on how such source is created. In this thesis we will refer to such methods as “static methods”. These methods are static, because they fail to reckon one of the most important aspects of software - the actual execution. We will argue that it is in executing software that problems can arise, and, as we will discuss, in many cases it is not feasible or even possible to predict the actual flow of execution that will take place in a complex software system.

There are other types of protection mechanisms for software, and our work presented in this thesis is focused on what we refer to as “dynamic methods”. A dynamic method is a method that takes the actual execution into account, typically by operating inside an executing program, and protects the program based on information it extracts from the actual execution. Using such information, dynamic methods can take the actual execution into account, which eliminates the need to predict or guess this important feature of software. Also, many dynamic methods have other advantages, such as the ability to operate on software without requiring access to source code, or requiring recompilation of programs.

We will discuss this aspect of protection both from a theoretical view, and using three practical experiment systems that we have designed and used to prevent exploits on live software systems.
1.2.1 Research Issues

The research issue that we address in this thesis is;

- the development of a comprehensive methodology enabling secure execution environments

which we intend to validate by means of practical experimentation. Our approach is to address the research issue in the context of a single executing node, with an emphasis on UNIX / POSIX-based environments, however the methodology should not be restricted to only such environments. For an example of how this methodology can be applied to distributed systems using our principal approach, please see [1].

1.2.2 Outline of Thesis

In the next chapter, 2. Methodology, we describe background information, including the function of the tool chain used to transform source code into executable binaries, and how this can be extended to support dynamic protection tools. In this chapter we also describe how modern software execute, which is in a rather complex environment and an environment that is important for how dynamic security methods operate. We also define terms that we will use throughout the thesis, including “vulnerability”, “exploit”, “static methods”, “dynamic methods” and “environment hardening”. In the chapter following that, 3. Model, we describe our model for software execution, as well as observations made on this execution. In this chapter we also present our three hypotheses on dynamic software protection methods, and our model for hardening an execution environment. In chapter 4. Principles of Protection, we extend the discussion on models, present eight principles for protecting software execution, and describe the principle function of our, and many other, dynamic tools. In the chapter 5. Experiments, we present three experiment systems that each illustrate a different layer of dynamic software protection, and we also describe practical experimentation we have performed using these systems. Finally, in chapter 6. Conclusions, we present conclusions from this work, and in chapter 7. Discussion we discuss further possibilities using methods and techniques described in this thesis, as well as possible future work.

1.2.3 Relevance and Intended Audience

In this thesis we present both practical experimentation on increasing the dependability in software systems, in several cases without modifying the binary executable file, as well as theoretical reasoning about such protection and underlying mechanisms. It is our intention that this should be relevant both for the academic security community, as well as for developers/engineers working with more practical software security issues.
2.1 Introduction

To get an understanding of how dynamic methods can improve system reliability, the first step is to actually understand what the term dynamic methods mean. The short explanation is that dynamic methods are focused on actual program execution rather than on program source, such as source code. This is, however, not a particularly useful explanation, because we must specify what we mean by program execution, and how this can be used to improve system reliability. To give a better explanation to what dynamic methods really are, we should first look at the process of how modern software is created and how such software executes. The execution is highly dependent on how the program is created, so we start with an introduction to how programs are created, and then look at how they execute.

Modern programs are created using complex tools, stored in complex files, and also execute in complex environments, so we really start with the traditional way of creating and executing programs, and then look at the more complex modern ways. From a high level view, the process of creating and executing software can be divided into three simple steps as illustrated in Figure 2-1. In this figure, each step is represented by an arrow;

1. The source code is written
2. The source code is transformed into an (executable) binary
3. The binary is executed
Readers with a background in software engineering might argue that in a typical software development process there are actually quite a few steps involved before the source code is written. While this might be true, under some circumstances, it is of absolutely no interest for the discussion on dynamic software protection, and hence we will leave it to the software engineers. Rather than focusing on the process that leads to source code being developed, and how this process could possibly affect the source code, the focus for this discussion is on the steps used to transform the source code into an executing binary, and the intermediate results, shown as boxes in Figure 2-1.

With just the simple illustration in Figure 2-1 we can extend the view of dynamic methods, and another type of methods, static methods, and where these fit in the simple three-step figure.

The static approach is best described as a forward-chaining method. Using static methods the focus is on different sources for a software program, and how we can find weaknesses, or vulnerabilities, in these sources. The main idea is that if a vulnerability can be found, and removed, from the source it will not be present in the executing program either, and hence cannot be exploited. These methods are of the forward-chaining type because the focus is not on the actual problem, exploits, but on vulnerabilities that under some circumstances and in conjunction with a run-time condition might cause an exploit.

Dynamic methods, are focused on the execution of software, and on how unwanted execution, exploits, can be prevented and effectively dealt with. This is a backward-chaining approach as the problem is known, but not all sources to the problem. In Figure 2-2 the high level view of creation and execution of software is extended with a high-level view of static and dynamic methods.

![Figure 2-2: High Level View of Static and Dynamic Methods](image)

Most static methods focus on high-level source, such as the source code for a program, but some methods use information from a lower level, such as the machine code for a program. Dynamic methods use a backward-chaining approach, so the focus is primarily on the executing binary, but dynamic methods may use information from higher levels, such as the machine code for a program, or even the source code.

For the discussion on how dynamic methods can improve system reliability, we must return to the prerequisites for the original problem; creation and execution of
traditional and modern software programs, and then return to how this can be used to protect execution. We provide this background in the next few sections.

2.2 Building Executable Binaries from Source Code

Executable programs are built using a chain of several tools where the output from one tool is used as input for the next. The compiler is arguably the most well-known tool, perhaps because programmers tend to see the compiler as the most central part of the tool chain. The compiler typically also sets the requirements on other tools used in the chain.

In a traditional UNIX environment these “other tools” typically mean the pre-processor and the linker. For example, the following steps are carried out when creating an executable binary from a single file of C source code:

1. The file is processed by the pre-processor, creating a pre-processed source code file.
2. The output from the pre-processor is compiled, creating an object file.
3. The object file is given to the linker which modifies the file and combines it with other files necessary to create an executable file.

These steps form a tool chain as each tool performs a transformation of its input data to its output. Each tool can be modified, independently of other tools in the chain, and doing so will likely affect the program that is being transformed into an executable file. For example, installing a new version of the compiler and re-building a program will most likely yield a different executable file than what the previous tool chain produced.

Some building systems use utility tools as well, such as the ubiquitous make program. In this context, however, make is not a part of the actual tool chain as it merely executes the tools in the chain, and does not directly modify the transformations used to create the executable program. Installing a new version of make and rebuilding a system will typically not yield a different executable binary file than the system using the previous version of make did.

The behavior of most tools in a typical building tool chain can be modified by changing parameters. In the same way as switching compilers likely will modify the executable file produced by the tool chain, so will changing relevant compiler parameters. Hence, for each tool in the chain there are at least three variables that affect the output from the tool;

1. The Tool Itself
2. Input Data
3. Configuration

Because of the way a tool chain is constructed with several independent tools, where the output from one tool is the input to the next, the output of the final tool is dependent on all other tools in the chain, and the respective configuration for each such tool. Modifying a tool early in the chain, for example to transform a program to be resistant against a certain type of attack, modifies the output from that tool which, hopefully, finally will result in a different executable binary.
2.3 Traditional Building and Execution of Software

The final output produced by the building tool chain is some sort of executable file. Traditionally, an executing binary had only primitive ways to communicate with its executing environment, normally by using system calls to the operating system kernel, either directly or through small stub functions [9]. This communication is trivial in the sense that it is uni-directional, i.e. the program calls the kernel, requesting a particular function to be carried out, and that only basic functions with extremely well-defined interfaces can be called. Functionality exposed in this fashion include file handling, process creation, and network communication [12]. This type of executables are still occasionally used and are often called “static” executables.

Because executing programs had only trivial ways of communicating with their environment, programs were stored in simple file formats, where each program contained everything it needed during execution. In this sense, the executable file was static - once created by the linker it was not modified, and the only way to alter the behavior of a program was to either change its source code, a tool or parameter in the building tool chain, and rebuild the program. Since most programs were written in a high-level language the supporting libraries were also linked into the binary file and never modified.

With this type of simple programs, should a vulnerability be found, the vulnerability had to be fixed in the source code and the program had to be rebuilt. If there was a vulnerability in a library used by several programs, the library first had to be rebuilt and then every program that used the library also had to be rebuilt.

Most programs today are built using more modern techniques that allow a larger degree of modification to be made to the executable binary, even after the program is built, and without access to the source code for the program. This is an important property of modern systems that many dynamic security methods rely upon. While this type of late modification enables security-enhancing techniques, it is also important to point out that this environment makes it particularly difficult to predict exactly how a program will execute. A program that uses standardized functions from the C library would behave very differently from one execution to another, should the library be modified. This is important for understanding the function of dynamic executable programs, as even though the program has not been modified between executions, the behavior of the program has. This is something that makes static methods, i.e. methods that do not concern the actual execution, difficult to apply, but, as we will argue in this thesis, is a strength for dynamic security methods.

1. There is support for a simple mechanism often called signals that enable the kernel to signal certain conditions to the executing program, but this facility is also comparably simple.
2. The use of the term “static” in this sense has no direct connection to the term used to describe static security methods.
2.4 Building Modern Software

The traditional way of building programs into static executables have successively been replaced by other methods resulting in less static, but more complex executable programs. Performance and maintenance requirements have been important factors for moving from static binaries, however for the discussion on dynamic protection we are not primarily interested in neither of these properties, but in the ability to analyze and modify binary programs.

One of the first steps in moving from static executable programs was support for loadable libraries. When using loadable libraries, code that is used by several programs is placed in a separate file, the loadable library, and then loaded into an executing program when needed. This gives two advantages over static binaries;

1. Commonly used code could be shared between different programs which was particularly important on computers with small amounts of RAM memory.
2. Modifications to the library applied directly to programs that used the library and did not require a separate rebuild.

Early support for loadable libraries depended on special code, stub code, being placed in programs that used such libraries. When the program attempted to call a function in the loadable library, it actually called the stub code which, if needed, loaded the library and transferred control to the actual function inside the library. This required no special support from the operating system but still enabled sharing of binary code between programs.

The concept of loadable libraries was later extended to what is often called dynamic libraries (and dynamic executables). Dynamic libraries offer more functions than loadable libraries and has fewer obscure side effects, but require more complex binary files and special support from the operating system to execute. In section 2.5 Executing Modern Software we briefly describe how these programs execute.

A number of modifications were done to the building tool chain to support dynamic executables and libraries. The process of building a dynamic executable (Step 2 in Figure 2-1) is extended as shown in Figure 2-3.

![Figure 2-3: Building Dynamic Executables](image)

1. Source code is written.
2. The source code is pre-processed and compiled.
   This step actually consists of both the pre-processing and compilation step, but in modern compilers these steps are practically combined into
Executing Modern Software

one single step. The compiler also adds low-level support code to the object files, such as support for runtime-information.

3. The output from the compiler is linked to a dynamic executable file. The linker combines one or more object files with static libraries and links this file to a dynamic executable.

The third step is the most interesting in this context; the linker combines object files and static libraries, and transforms these into a format the can be executed. Whenever the linker finds a reference to a function that is defined in an external, dynamic, library it creates a special note in the output file so that the runtime environment can replace this with a reference to an actual function prior to executing the program. The final result, the dynamic executable, is not really an executable file, but a file that given the correct runtime environment and external dynamic libraries can be loaded in such a way that it can be executed.

2.4.1 Typical Features of Dynamic Executables
Dynamic executables offer a large number of advantages over static binaries, far to many to exhaustively describe here. For the discussion on dynamic protection, however, we are primary interested in two such features; the ability to change which function is mapped to a particular name, and the ability to load arbitrary libraries into an executable.

2.5 Executing Modern Software
The process of preparing an executable file to execute is often called loading. Historically, this was as simple as copying the file from hard disk to RAM memory and start the execution at the first byte [10]. In a modern system using dynamic executables this is quite a complex process, often so complex that it is moved from the operating system kernel to a special program, called the run-time linker.

The run-time linker determines which dynamic libraries that are required to run the program, and if any library require further libraries to be loaded. Then, for the program and each required library, it maps different parts of the respective file to an appropriate memory address. In some cases, it is necessary to relocate a part of the program or library to an address for which it was not built, which require the run-time linker to dynamically rewrite parts of the program. The process of loading and executing an executable file is described in Figure 2-4.

![Figure 2-4: Loading and Executing a Binary](image-url)
During the process of loading an executable file, the run-time linker asserts that all functions that can be called by the program or a library are defined. Typically, the exact address is not determined when the program initially is loaded, but the first time a call is made to a particular function. This lazy loading is useful because most programs call only a small minority of all functions defined, and because a program can manually load additional dynamic libraries that may alter the mapping between function names and addresses.

When a modern program executes it does so in quite a complex environment. There is no way of guaranteeing the exact memory address of any part of the program or any libraries it may use. Actually, in most cases we can ignore the fact that some functions are defined in libraries and others in the program itself. In some environments, such as most modern Unix systems, the same function can be defined in several libraries and given a priority. When the program calls such a function the one with the highest priority will be invoked automatically. In this thesis we will refer to this environment as the programs “execution environment”. In the next chapter, 3. Model, we discuss this concept in more detail.

Because of features in this complex environment and the large possibility to modify the behavior of a program by affecting its execution environment, we consider tools that modify this environment as a part of the software building tool chain. The result of such tools will typically not be saved in the executable file but applied later by the dynamic linker, or as we will describe in this thesis, even later in the process. For this reason, it is essential to understand that all steps shown in Figure 2-1, (Writing source code, compiling and executing) are important for the behavior of the executing software program.

2.6 Software Vulnerabilities and Exploits

When discussing software security, vulnerabilities and exploits are amongst the most commonly used terms. Unfortunately, these terms are often used interchangeably, but for the discussion on dynamic software protection it is important that we keep a traditional view on the meaning of these terms, and that we under no circumstances use them interchangeably.

2.6.1 Vulnerabilities

A vulnerability is a kind of weakness somewhere in a system. In most cases, this means a weakness in the source code for a program how a particular situation, that may occur in run-time, is handled. When this source code is compiled to an executable file, the vulnerability would typically, although not necessarily, be transferred through the tool chain to the executable as well.

Vulnerabilities are not restricted to source code weaknesses, though. For example, there could be a vulnerability in how a compiler optimizes code so that a program is compiled into a form that does not match its source code. This is typically a problem when using compilers that aggressively optimize their output to compile programs that handle sensitive data, such as passwords, and where the program needs to overwrite memory once used [38]. One could argue, that the actual vulnerability is in the source code to the compiler, which might be true.
Software Vulnerabilities and Exploits

However, given this view, compiling a program using such a compiler transfers the vulnerability to the object code of the program being compiled, and then through the linker to the executable file. No matter what one considers to be the "root" vulnerability, a weakness, or vulnerability, is created when compiling the program.

Another type of vulnerability might be in the processor that actually executes the binary code for a program. When the program executes on that particular CPU, the CPU is a part of the system and hence are vulnerabilities in the CPU also vulnerabilities in the system.

One could expand this discussion even further, claiming that problems in understanding the requirements for a system and having developers write source code that does not match the actual system requirements causes a vulnerability; however, because of the focus in this thesis, starting with source code and ending with execution, we will not consider this case.

2.6.2 Exploits
An exploit is a form of unwanted execution that is made possible because of run-time conditions and one or more vulnerabilities. The typical exploit from a software security perspective, is caused by the unsafe use of functions, or failure to check bounds of a memory buffer combined with hostile input resulting in an adversary executing arbitrary machine code on a server.

There are other types of exploits though; for example a particular vulnerability in handling simultaneous execution combined with a specific runtime situation could result in a race condition, causing unwanted execution. Given the definition of exploits, we also consider so called denial-of-service attacks as exploits, as they consist of a vulnerability in the system combined with a run-time situation and results in unwanted execution, although in this case unwanted execution typically means no execution (of relevant, or intended, code).

2.6.3 Vulnerabilities, Exploits and Execution Environments
For the discussion on dynamic protection, the concept of exploits is by far more important than that of vulnerabilities. A vulnerability can be hidden anywhere in a system; in the source code of a program, in a library it uses, in the compiler, in the processor or in some other part. Given the right circumstances, it might never cause problems, or even execute.

An exploit, on the other hand, can occur if, and only if, there is one or more vulnerabilities in the system and given a particular run-time situation. If one of these requirements for an exploit are removed, the exploit will not occur.

The run-time situation that can cause an exploit is dependent on several factors, where aspects such as hostile input frequently is discussed. However, the run-time situation for a program is also highly dependent on the environment in which it executes. By modifying the execution environment, it is possible to change how a system will behave when executed, and hence, in some cases, create an envi-
ronment that protects an executing program from run-time situations where the program otherwise would be exploited.

2.7 Preventing Software Exploits
There are a large number of methods that make software more secure. Given this, there are different ways to classify, or categorize, these methods. The terms we use in this thesis are “static methods” or “static analysis”, “dynamic methods” or “dynamic analysis” and “environment hardening” which we consider a special form of dynamic analysis.

2.7.1 Static Analysis
The term static analysis, or static methods, are used to describe methods that;

1. Operate on some kind of source to a program.
2. Does not take actual program execution into consideration.

The vast majority of static analysis is in the form of tools that automatically analyze the source code for a program looking for vulnerabilities. Some tools analyze other source data to a program, such as the machine code. We consider the machine code to be a form of intermediate source for the program, as the program has not yet been executed. (See 2.5 Executing Modern Software) Less often manual analysis, or inspection, is used to find vulnerabilities.

Automated forms of static analysis are typically used to find vulnerabilities that require little context to be identified, such as buffer overflow vulnerabilities. Manual inspection is typically used on smaller pieces of source to find vulnerabilities that are difficult to find using automated tools. However, even for vulnerabilities that require a comparably small context to find, such as buffer overflows, there are issues with using automated tools.

“Not all vulnerabilities are found by automatic or manual inspection of source code. Because of how modern programming languages are constructed, overflow vulnerabilities may be hidden almost anywhere in a program, which makes manual audit time consuming and difficult. Vulnerabilities can also be dependent on runtime conditions, something that make automated static analysis difficult, if not impossible.

Very few programs can easily be inspected. Many legacy systems are still used for which the source code is not even easily available. Also, there might be business or political reasons that makes a company reluctant to providing customers will all source code required for an inspection.” [4]

Manual inspection also has some important limitations, perhaps most importantly in that the quality of the inspection is highly dependent on the knowledge, or competence, of the inspector, and that some types of vulnerabilities are difficult to find manually. However, because a successful static analysis can reveal several vulnerabilities and because there are a number of automated tools that perform
Preventing Software Exploits

static analysis is this a good starting point to find the most trivial vulnerabilities in a program.

2.7.2 Dynamic Analysis
Dynamic analysis, or dynamic methods, are methods that operate on executing software. This can be done in the form of low-level hardware support, such as processors that limit which memory regions that may be executed, but it can also be in the form of more high-level methods such as those described in the experiment chapter of this thesis.

From a security perspective, dynamic and static methods have fundamentally different properties because the underlying difference in operating on executing software or software source.

"Inspecting programs during execution, dynamic inspection, has the advantage over static inspection methods that the actual state of the program is always known. When a function call is made from the program it is possible to [analyze] the actual parameters sent from the program to the function, as well as the value of other variables used in the program. A dynamic inspector can also use other process information, such as the process' privileges and file system root as basis for taking different actions.

Dynamic inspection is done at the actual time of execution, and inside the executing program, so it is important that the inspector does not impose vast overhead on the program, and does not introduce new vulnerabilities during inspection. A vulnerability in the inspection may result in as severe problems as if the vulnerability were in the program itself." [2]

2.7.3 Environment Hardening
Environment Hardening, in general, refers to a method or technique that makes an environment more resilient to some kind of exploit. Because the only domain we discuss in this thesis is software execution, environment hardening is equivalent to (software) execution environment hardening.

A static method could constitute hardening of an execution environment, for example by statically searching for buffer overflow vulnerabilities in a system library. However, because the way static methods operate on software source rather than execution, the result of static analysis typically are vulnerabilities rather than exploits, and for this reason most static environment hardening constitutes an indirect form of hardening.

Dynamic methods, however, operate directly on executing software and for this reason can harden the execution environment directly (i.e. by preventing exploits rather than vulnerabilities.) In this work the focus is on dynamic environment hardening, meaning environment hardening by using dynamic, as opposed to static, methods. The purpose of the hardening in this context is to lower the risk of
a vulnerability in a program being exploitable, and should an exploit be possible reduce the consequences for the system.

Using dynamic methods to prevent a program from being exploitable often also reveals the underlying exploit, which can be logged for later analysis. In chapter 5. Experiments we describe one such system, SLIBC, which in addition to immunizing a system from certain kinds of buffer overflow exploits also provides information that can be used to find the actual root vulnerability.

2.8 Chapter Summary

Executable programs are created by applying a chain of tools operating on the source code where each tool performs a specific transformation, and the output of one tool is the input to the next. Modern programs are stored in complex formats that enable modifications to the program’s execution behavior without the need to re-build the program from source code. Because of the complex post-build transformations, tools that perform such transformations should be considered a part of the tool chain used to build the software. The complex environment in which modern software execute is called the software’s execution environment.

A software vulnerability is a weakness somewhere in the system. An exploit is unwanted execution that takes place because of one or more vulnerabilities and a run-time condition. Some security methods aim at creating more dependable software by analyzing the software source, normally the high-level source code, looking for vulnerabilities. These methods are called static methods. Other methods operate on executing software and primarily prevent exploits rather than vulnerabilities. These methods are called dynamic methods.

Environment hardening in the context of software execution is a process or method that makes the execution environment more resilient to exploits. This can be done by static methods, but is more often done by dynamic methods. The methods and tools described further on in this thesis are dynamic methods that harden a program’s execution environment to make it harder to exploit vulnerabilities in the program, and should they still be exploitable, reduce the security consequences for the system.
CHAPTER 3

MODEL

3.1 Introduction
As discussed in the previous chapter, modern software programs execute in a complex environment, which we call the program’s execution environment. Execution environments have a number of properties that can be analyzed and modified, and by using these properties in the right way, we can create more dependable software execution; the topic of the next few chapters. However, before discussing which properties in the environment that are usable for increasing the dependability and how this can be done, we must first establish a model of typical software execution behavior. Once this is established, the model must be extended to cover hardening of execution environments as well, the two topics of this chapter.

3.2 A Model For Execution Environment Behavior
From an external view, most execution environments can be seen as the inside of an operating system process. The operating system process encapsulates the executing program and all its resources, and hence encapsulates the typical execution environment. This is, however, not a complete view, and also not very practical for modelling dynamic environment and execution behavior. This is true for two reasons;

1. An execution environment can, under some circumstances, span several operating system processes.
2. When using dynamic methods in general, and dynamic environment hardening in particular, we are interested in the internal structures in an executing process. While this state is encapsulated in an operating system process, the actual contents is not considered, but just the fact that the program stored data in a particular region of memory.
A Model For Execution Environment Behavior

Rather than modelling an execution environment as being the inside of an operating system process, we consider the relevant parts of the executing process to be a property of the execution environment. In this case the relevant parts are those that affect the execution, and not operating system internal data, such as scheduling and accounting information.

![Figure 3-1: An Execution Environment](image)

Figure 3-1 illustrates the contents of an execution environment, which includes the executing binary, libraries, and process data. By "process data" we mean inherited, relevant, properties from the operating system process, such as:

- Memory layout and the actual memory contents
- Operating system handles such as file descriptors
- CPU registers, including the program counter
- Process privileges

The process data is dependent on the particular operating system, and might change when a program is executed on a different operating system. For this reason, we cannot give a exhaustive list of the process data contents. This is, however, not necessary for our model of execution environment behavior.

**Definition (I).** An execution environment is the complete context in which a program executes on a particular operating system. It includes properties inherited from the process, such as memory contents and layout, kernel objects such as file handles, and a set of privileges which the process executes under. It also includes other hardware-specific resources such as registers.

The execution environment differs from the process concept in the sense that it also includes the layout of binary formats internal to the process, functions, call mappings, and local data that the operating system disregards. Given the definition of an execution environment, we will return to discussions on practical program behavior to reflect on a number of observations on how execution environments behave.

### 3.2.1 Communication

The vast majority of programs are written in high level programming languages, most often the language C [9]. Because of the common heritage between the C programming language and the Unix operating system, all library functions use the function calling convention from the C programming language. This way of calling functions is also the de-facto standard when designing interfaces between programs written in different programming languages and is commonly called the
C calling convention. To call the operating system kernel a special type of interface is used, as this type of communication require a controlled form of privilege escalation that is specific for each CPU and typically cannot be accessed directly from high-level languages. In BSD-derived systems this interface is similar to the C calling convention while on other systems, such as Linux, this interface bears little resemblance to a normal function call.

The POSIX standard defines a number of interface functions between a program and its environment. The standard does not specify which functionality should be implemented as system calls, and which should be implemented in library functions, but but leaves this to the implementation [14, 15]. For this reason, system calls are exposed through helper functions (stub functions) in the C library, where the stub function invokes the actual system call. A program call functions in the C library, both actual functions and stub functions that invoke the kernel, using the C calling convention. A program may communicate directly with the kernel, but that interface is not standardized, and can typically not be called from high-level languages, but only from functions implemented directly in assembly.

Observation (I). Essentially all program communication is done by means of function calls, and when a program calls a function in a system library, it uses the C calling convention. The vast majority of programs are written in the C programming language and these use the C calling convention for internal calls, as well.

An executing program can communicate directly only with functions in its own address space. To communicate with other functions, the program must call the operating system kernel which sets up a communication channel. Two execution environments can hence communicate with each other only indirectly through the operating system kernel. To handle this type of communication, the kernel will create a context which will be made visible to an execution environment through the means of an operating system handle, most often a file descriptor.

Some types of inter-process (i.e. inter execution environment) communication is almost transparent to the execution environments that use it, for example shared memory, but this communication channel is still created through the kernel, and the kernel creates a context in the respective execution environment to support this form of communication.

Observation (II). An executing program can only communicate directly with functions in its own execution environment, and with the operating system kernel. All other forms of communication, however transparent they may seem, are actually mediated by the kernel which creates state in the execution environment to support this communication.
A Model For Execution Environment Behavior

Figure 3-2 illustrates two executing programs with their respective execution environments. While it may seem as the two processes can communicate directly through means of pipes, sockets or shared memory, this is actually a part of the illusion created by the kernel and propagated to the executing programs through the concept of operating system processes; all such communication is actually mediated by the operating system kernel.

3.2.2 Internal Execution Environment State

An executing program can modify its execution environment in several ways. Because memory layout and contents is a part of the execution environment, the environment is modified as soon as the program modifies the value of a single variable located in that memory. Actually, because hardware specific resources such as registers, including the program counter, are included in the execution environment, the environment is modified for essentially every machine code instruction the program executes, even if that instruction does not modify the contents of any variable. This type of modifications to the execution environment typically have a very small, if any, side effect on the programs execution.

Observation (III). An executing program constantly changes its execution environment. The small modifications that are done to the environment by normal execution typically do not cause unwanted side effects on the execution.

Execution environments that use a dynamic executable format have the ability to load arbitrary libraries into the execution environment and then call functions in such libraries (See 2.4.1 Typical Features of Dynamic Executables) This causes a drastic modification to the execution environment for two reasons;

1. A new library is loaded into the current execution environment.
2. The new library may be called directly from the program or other libraries in the execution environment, and the new library may directly call the program or other libraries in the execution environment.

Loading arbitrary dynamic libraries, i.e. libraries that was not originally passed to the linker, is a commonly used technique in modern programs. This is, for example, the fundamental approach used to load plug-in functions in many programs,
and the basis to lightweight object sharing such as Microsoft COM and clones, such as XPCOM [13].

When loading a library dynamically into an execution environment, the library is actually loaded into several parts of the environment as the library consists of different type of data (machine code, constant data, static variables, etc.). Since libraries can be loaded in different order when a program is executed a second time, and because a library could have changed since the last time it was used, the exact placement in memory for such libraries are not only possible, but also highly likely, to change between executions.

**Observation (IV).** Programs that dynamically load libraries are likely to have large differences in their execution environments between different executions. This is due to inherent properties in dynamic executable formats.

### 3.2.3 External Environment State

The operating system kernel can, in some cases, create state in an executing process’ execution environment. For example, when a program successfully opens a file, the kernel creates a handle which it returns to the calling program in the form of a positive integer. This type of state normally has very few side effects on the program execution, as long as a small number of assumptions of pre-opened files hold true.

In some cases, however, the kernel can cause large modifications to the address space in an execution environment. This is the case when using shared memory between processes or mapping files or devices into the address space. In some cases, the main program perform these calls to the kernel and hence expect these changes to its execution environment, but in other cases these calls are made from dynamic libraries and the modification done to the execution environment may not be expected by other parts of the program sharing the same environment.

**Observation (V).** Some calls to the kernel create state in the execution environment. Often this is in the form of handles, or file descriptors, that only marginally modify the environment, but in other cases, specifically when using shared memory or mapping files or devices into the address space, the kernel can cause significant modifications to the execution environment.

Another type of external cause that affects the execution environment lies in differences between different operating systems. For example, a program written using only standardized Posix function calls may behave very differently on different operating systems even though both operating systems meet the Posix requirements. This also holds true for operating systems that implement the Win32 interface in standard compliant, but different, ways [24].

As an example, different operating systems handle file system objects, such as path names, in different ways. This have caused exploits in some programs when ported to a new operating system that used a different way of handling file names.
A Model for Execution Environment Hardening

Observation (VI). Differences in operating systems can affect the execution environment of a program, even if the program uses only standardized function calls, and both operating systems implement all such functions in a way that is compliant with the standard.

3.2.4 Modelling Execution Environment Behavior
The complex behavior of modern software makes it difficult to create a model for deciding correct and incorrect forms of execution. It would be useful to inspect the execution environment of a program and determine if the execution was correct or incorrect given a set of rules. While this might be possible given a trivial program or trivial rules, we believe that the dynamic behavior of executing software makes this approach very difficult to generalize.

Rather than modelling correct or incorrect behavior from a complete execution environment perspective we must use other, simpler, approaches. One such approach is to look only at a very small piece of state that we believe is relevant for a particular action taken by the executing program, and depending on this state take what we believe is an appropriate action. Similarly, we could modify a small sub-set of the state that we believe affect a possible exploit, without considering the entire state of the execution environment.

By using this model we also make some claims on the normal execution environment;

Hypothesis (I). The execution environment can be divided into small subsets, which we can identify, and by analyzing only one or a few subsets we can make a good decision on whether a specific action is likely to be harmful to the program.

The execution environment is constantly modified, and in that sense never static, so the hypothesis also implies that, given a certain action, it is possible to analyze the execution environment at the right time when the relevant state is present and in an analyzable form.

3.3 A Model for Execution Environment Hardening
As discussed in the previous chapter, Methodology, the term environment hardening refers to any method or technique that makes an environment more resilient to some kind of exploit. To reduce the wide range of meanings this definition can have, we only consider the case of software execution, which makes environment hardening equivalent to software execution environment hardening. (See 2.7.3 Environment Hardening) This is still a far to wide definition, requiring further classification based on how the actual methods, or techniques, operate. We will use the terms “Static Hardening”, “Dynamic Hardening”, “Generic Dynamic Hardening” and “Domain-Specific Dynamic Environment Hardening” with relations as described in Figure 3-3 to describe the different types of environment hardening.
Figurer 3-3: Classification of Environment Hardening Methods

**Static Hardening.** We use the term “Static Hardening” to describe methods or techniques that use static methods for hardening the execution environment. This include replacing library functions with more tolerant implementations, removing vulnerable interfaces the environment, or statically removing unwanted functionality in the environment altogether.

A practical example of static hardening is the modifications to the malloc C library function found in the OpenBSD operating system. In situations where there are several candidates for which memory block to use, the function chooses one at random, rather than the first available [29]. By randomizing blocks, it is difficult for an adversary to predict the address of a particular memory buffer which makes a vulnerable program harder to exploit.

**Dynamic Hardening.** This refers to methods or techniques that make the execution environment more resilient to exploits by actively taking program execution (i.e. state) into consideration. The dynamic hardening methods we consider analyze the actual execution and do not speculate in possible execution by simulations.

**Generic Dynamic Hardening.** We use the term Generic Dynamic Hardening to describe dynamic hardening methods, that while taking program execution into consideration, does not take the execution of a specific program or domain into consideration. This include CPU features such as no-exec protection, and the SLIBC system described in the Experiments section of this thesis.

**Domain-Specific Dynamic Hardening.** This term refers to dynamic environment hardening methods or techniques that not only take program execution into consideration, but also have specific knowledge about a particular program, environment or both and make decisions using both the execution state and this domain-specific knowledge. Knowledge of “environment” in this context is broader than we otherwise have used the term environment, and include such cases as when a program is used for a particular task, typically embedded, where external requirements are different than the typical requirements. An example of such a system, PLIBC, is given in the Experiments section. PLIBC is used to apply an execution policy to a non-cooperative program, which enables standard programs to be used in environments with higher dependability requirements, such as in controlling critical infrastructure.
3.3.1 Observations on Environment Hardening

Environment hardening, by definition, has a focus on preventing exploits rather than addressing vulnerabilities. When using static environment hardening methods, the border is typically more diffuse than when using dynamic methods. For example, randomizing return addresses does not address a particular vulnerability but makes it harder in general to exploit a vulnerable program. One could argue, though, that the deterministic behavior of computer programs is the actual vulnerability, and that randomization attacks this particular vulnerability at its root. Also, consider the case of removing functionality in system libraries as a method of environment hardening. For example, does removing a function that is standardized but known to be inherently unsafe, such as the gets function, remove a vulnerability or harden the environment from a potential exploit?

This becomes a question of definition, and we consider both removal of gets and randomization of malloc buffers to constitute hardening of the environment, because the exploitability is more concrete than the vulnerability. In the case of randomizing a program, we do not know the actual part of a vulnerable program that will be exploitable, but the method is based on the assumption that should a vulnerability exist, it becomes harder for an adversary to exploit if he has less information about the execution environment. In the case of removing functions that cannot be used in a safe way, the real question is what we consider to be the vulnerability. If we consider the sole existence of unsafe functions to be a vulnerability removing such functions is a form of static security method. On the other hand, if we consider using unsafe functions a vulnerability, and thus opening for a potential exploit, removing such functions from the environment is a form of hardening, although a static form of hardening.

Observation (VII). The border between environment hardening and traditional static analysis is sometimes diffuse. We consider methods or techniques that address a concrete exploit but only an indirect, or unknown, vulnerability to be environment hardening.

In many cases, removing functionality using static environment hardening causes problems for correct systems. While some functions, such as the gets C library function, have no known safe uses, many other functions can be used in both safe and unsafe ways, and removing all functions that are potentially unsafe is not realistic. Also, many functions that are designed to be especially secure can be used in insecure ways given the right, or wrong - depending on your view, parameters or runtime state. In these cases, dynamic environment hardening is an effective method to move the decision from a static one to a dynamic. Dynamic methods can analyze the execution environment and make informed decisions about a specific call that could not be done without runtime information.

Observation (VIII). A large class of functions can be used in safe and unsafe ways, and removing such functions is typically not an option. Also, even functions that are designed to be very safe can often be used in unsafe ways.

1. Correct with regard to implementing a particular system in accordance with some specification or standard.
Dynamic hardening methods are especially well-suited in these cases, since such methods can determine if a particular action is safe or unsafe based on specific runtime conditions.

Some types of Dynamic Hardening is implemented in hardware, typically no-exec protection \[18, 19\], but most types are implemented in software only. Dynamic methods always operate in real-time as they analyze the execution environment of a given program to make decisions on which operations that are safe. Typically, although not necessarily, dynamic methods also operate inside the execution environment of the program they protect, which enables fast access to the execution environment. This result in tools that have access to all available state, and that can operate with a very low overhead, but also that vulnerabilities in the tool can propagate directly to the program being protected.

“Dynamic inspection is done at the actual time execution, and inside the executing program, so [it is] important that the inspector does not impose overhead on the program, and does not introduce new vulnerabilities during inspection. A vulnerability [in] the inspection may result in as severe problems as [if] the vulnerability [was] in the program itself.” \[2\]

Observation (IX). Dynamic hardening tools can protect a program from several types of exploits that are difficult to protect from using other methods, but because of the way in which these tools typically are implemented, a vulnerability in the tool can propagate to the protected program.

3.3.2 Modelling Environment Hardening

Execution environments can be hardened using a variety of methods, where each method attack a class of abstract environment or program vulnerabilities. We divide environment hardening first into “static” and “dynamic” hardening, and then further divide the latter type into “generic” and “domain-specific” hardening. This classification can be thought of as a scale ranging from the pure static protection, to the generic dynamic, and further to domain-specific dynamic, where more and more dynamic, or run-time, information is taken into consideration.

As we observed, using runtime information places high requirements on the tool both in terms of overhead and safety, but having access to more information at the time of dynamic analysis gives the possibility to make better, in the sense of more informed, decisions.

Hypothesis (II). The complex and changing execution environment makes a pure static, as in completely pre-execution, methods inherently inferior in protecting system execution compared to dynamic methods, that take the actual execution environment state into consideration when making decisions on which operations that are safe.

However, to build working dynamic methods, we also must combine the hypothesis above with the hypothesis of execution environment behavior;

Hypothesis (III). It is possible to build dynamic environment hardening methods that while using only one or a few sub-sets of execution environment, can pro-
Validating The Hypotheses

tect programs from exploits where the underlaying vulnerabilities are very difficult to find using static methods.

3.4 Validating The Hypotheses
We have performed a number of practical experiments to validate the correctness of the hypotheses presented in this chapter. In the next chapter, 4. Principles of Protection, we describe the method used when developing the experiments which are described in chapter 5. Experiments. In the experiment chapter, we describe three systems, “LPS”, “SLIBC” and “PLIBC” where each respective system is an example of static environment hardening, generic dynamic environment hardening and domain-specific dynamic environment hardening. We provide a discussion on our interpretation of the results from this practical experimentation in chapter 6. Conclusions.

3.5 Chapter Summary
The concept of execution environments are fundamental for environment hardening, especially when using dynamic methods. Slightly simplified, an execution environment is the complete state of an executing program, which includes state from an operating system process. Different actions taken either by a program directly, or a library loaded by the program, causes modifications to the execution environment, although some modifications are small and can be disregarded. Because of the complex execution environment in a modern program it is difficult to separate correct execution from incorrect by looking at the entire executing environment. Instead, we can use simpler methods that consider only a small part of the execution environment. It is our hypothesis that this is possible.

Execution environments can be hardened using a variety of methods. We classify such methods according to which underlying technology that is used, into “static” and “dynamic” environment hardening, and then further classify “dynamic” methods into “generic” and “domain-specific”. Because of how modern programs are constructed, and because of the complexity of the execution environment, we have two hypotheses regarding environment hardening. Firstly that static methods are inferior to dynamic in regard to protecting system execution, because dynamic methods have more information available and can hence make more informed decisions. Secondly that it is possible to construct dynamic environment hardening tools that by using small pieces of execution environment can protect a program from exploits where the underlaying vulnerability is difficult to find.
4.1 Introduction

In order to build tools that dynamically protect program execution, we must first establish the principles that describe the fundaments for how such tools operate. In this chapter we present these basic principles. The main objective for these principles as well as the fundamental function of the tools are on protecting system execution - this is done either by immunizing a system, detecting unwanted execution or by reducing consequences for a system should an exploit occur.

The principles described in this chapter are based on the model from the previous chapter, especially the observations on how modern programs behave and which execution environment subsets that can reliably be analyzed.

4.2 Construction of Dynamic Environment Hardening Tools

Dynamic hardening can be done at different layers in a system, and the choice of layer affects the type of protection a particular method can provide. The first such layer that is relevant for dynamic protection, and that we have performed most experimentation on, is the communication between different parts within the same execution environment. All parts of an execution environment share the same memory space and operating system kernel objects, such as open files, which makes several types of communication possible. However, as we noted in Observation (I) on page 19, essentially all program communication is done by the means of function calls, and that this is also the standardized way of communicating with system libraries.¹

1. Some variables, such as errno are also exposed, although in many cases, these are actually defined as macros that expand to function calls. This is, however, of no particular relevance for this work, and we will hence not further elaborate on the construction or behavior of such variables or macros.
Functions in system libraries have no special privileges compared to functions in other libraries, or functions in the main program. If needed, any function found in a system library could be placed in another library. Assuming that the function still use the same interface and provide the same functionality, a program would normally not notice the difference even when calling the function. This is true even if the function use another, perhaps more dependable, implementation of the same interface as the original function in the system library. This is a fundamental property of how functions behave in a modern execution environment, and the way function calls are executed is of interest to us for two reasons;

1. We can create more dependable implementations of a certain interface, that normally behave identical to the standard implementation, but may differ if it detects a run-time situation that it considers harmful for the program.
2. A program can implement a function locally and call that function instead of calling a corresponding function in a system library. If we have a more dependable version of such a function in the system library, this function will not be called.

As a direct, and important, consequence of (2) it follows that;

3. There is no way of enforcing a program to call a certain library function to carry out a particular action. The program can call some other function, or implement the functionality in some other way. There is no way in run-time where we reliably know or control this.

In contrast to this type of library based communication, an execution environment also communicates with the operating system kernel through another type of interface. This type of communication forms the second layer that is relevant when designing dynamic protection methods. To call a service in the kernel, the program normally calls a function in a system library that contains the operating system and hardware specific machine code that invokes the kernel. A program can, however, invoke the kernel directly. Calling either local functions in an execution environment or calling functions in the operating system kernel are the only two ways an executing program can communicate with its environment (See Observation (II) on page 19).

Observation (X). The operating system kernel executes in a special, privileged, mode and can allow or deny operations requested by a program in an enforceable way. Functions exposed by the kernel cannot be accessed in any other way, which forces a program to call a particular function in the kernel for a certain action to be taken. Because the kernel executes in privileged mode and a program does not, there is no way to implement the same functionality in the program itself. This means that the interface to the kernel can enforce properties or policies on an executing program, and that there is no way for a program to circumvent such policies.

Most operations, i.e. all non-privileged operations, that a program perform do not require calling the operating system kernel. Calling the kernel takes long time compared to a normal function call, which, among other reasons, makes it advantageous to place code that do not require special privileges outside the kernel. The relative overhead imposed by calling the kernel compared to calling a normal
function call depends on several factors, such as the actual operating system and architecture. For an example of the overhead imposed by normal function calls, calls to the operating system kernel, and fast calls to the operating system kernel in the Solaris operating system, please see [11].

The difference between the enforceable interface to the kernel and the non-enforceable interface between functions in an execution environment are fundamental when designing dynamic protection methods and tools.

4.2.1 Dynamic Methods and Layers

When creating a dynamic protection tool, the first question to ask is whether the tool will be handling hostile, or just non-cooperative, code. In some, rare, cases we design tools that handle cooperative code, but we will not specially consider this case. Instead, we merely consider this as a sub-class of non-cooperative code. Hostile code include programs that we do not trust, typically because the program has been exploited. If this is the case, the protection must be placed on the interface to the kernel, since this is the only interface that can enforce a policy.

Principle (I). All dynamic methods that handle potentially hostile code must be placed on the interface to the kernel, or use the existing protection mechanisms in the kernel, as this is the only interface that can enforce a policy on a program. In many cases, already existing mechanisms in the kernel are sufficient, as all communication outside an execution environment must be done through the kernel (See Observation (II) on page 19).

Placing all dynamic methods on the kernel interface is not very useful, as the kernel is invoked only for comparably few operations. Many exploits are caused by programs using library functions in a non-safe way. These library functions do not call the kernel so placing the protection at the kernel layer would not be useful. In these cases, the dynamic protection must be placed in the execution environment, as this is the only interface where the exploit effectively is detectable.

Principle (II). Dynamic methods placed in the execution environment can detect run-time situations that are difficult to detect by the operating system kernel. While a hostile program can choose simply not to call such functions, there is no reason to design a program to circumvent a system designed to protect the programs’ execution. Designing a dynamic method in this fashion enables
Principal Function of Dynamic Methods

protection of non-cooperative programs, but not of hostile programs, including any exploited program, that actively avoids the protection.

As a direct consequence of Principle (II), any dynamic method operating inside the execution environment must detect a potential exploit before the exploit is executed (i.e. while the program is not hostile). This detection can, however, be just before the program normally would be exploited.

In some cases, we combine Principle (I) with Principle (II) to create a hybrid protection where a cooperative or non-cooperative system needs to execute code that we believe has a high risk of being exploited, given some run-time conditions.

We have not performed any experiments with pure operating system kernel-only approaches, but the methods we use are either pure execution environment methods, or hybrid methods.

4.3 Principal Function of Dynamic Methods

We have divided the functionality of a dynamic hardening tool into three distinct steps where each step is dependent on the result of the previous steps. These steps and their relations are shown in Figure 4-2, and this also represents the fundamental, high-level, design of our experiment systems.

Figure 4-2: Principal Dynamic Environment Hardening

For a dynamic method to operate correctly, it is not sufficient just to implement the right interface or function, the method must also be invoked in such a way that it can analyze relevant execution environment state (i.e. at the right time). An interception step must hence be designed to take over program execution at a time when the execution environment contains relevant and analyzable data (See Observation (III) on page 20), and perform this interception without modifying the execution environment in such a way that the executing program is affected.

Tools built using certain methods of dynamic environment hardening require the program (actually the execution environment) to invoke the tool directly to perform a specific function. In this case the interception step is trivial (NULL interception), as the program explicitly call the tool, and we know that the program expect the tool to carry out its function. In most cases, however, it is necessary to forcefully take over the execution from a program not originally designed to call the tool. This is the typical interception step, and this is somewhat more problematic to perform with reasonable performance and without damaging the executing program.
4.3.1 Interception Primitives

On most operating systems that use dynamic executables, there is a simple and effective way to intercept calls to functions in a system library, by using features in the dynamic linker. The dynamic linker typically supports injection of additional libraries into a program when the program is loaded, and a function in such a library normally has precedence over a function with the same name in the system library [17]. This technique is often called "overloading of a function". The main disadvantage of this method is that all functions that should be modified must be present in the library when the program is loaded, and cannot be modified later. Hence, we must \textit{statically} (i.e. pre-execution) know which functions to modify.

We can overcome this limitation by overloading all functions that might become interesting during execution, and then in each function determine which action to take. This method is comparably slow, but as we show in experiments with the PLIBC system, the overhead can be greatly reduced by using low-level techniques in the interception code, rather than using high-level functions. It would also be possible to construct special-purpose dynamic linkers that have a more configurable behavior than the default linker. Since the dynamic linker is a normal, although not dynamic, program, this can be done without special support from the operating system kernel. This overloading approach is effective only on dynamic executables, as other types of programs are loaded and executed by the kernel directly.

It is also possible to intercept functions in some non-dynamic (static) programs as well, but this require patching the machine code for the binaries prior to executing them, or to execute the binaries in a virtual machine. We have not performed any experimentation on this type of binaries, and will not further elaborate on methods to intercept function calls in such environments. However, in principle we believe that given the appropriate interception primitive, such environments could be hardened using the same principles that we use on dynamic executables.

4.3.2 Preserving Execution Environment During Interception

When intercepting a function with another function that uses the same interface, the calling convention and prologue code generated by the compiler recreates parameters and variables to the function, and we do not need to know how these are sent from the calling function. In this case, the interception step is done simply by using an interception primitive (for the actual interception) and the compiler (in the prologue code generated automatically, as the interfaces are identical).

When intercepting large amounts of functions with generic interceptors that don’t have prologue code generated, and where the prologue code would more than double the overhead, the interceptors must be designed in a special way so they don’t damage important state in the execution environment. The exact way of doing this is dependent on the hardware which the system executes on and the
Principal Function of Dynamic Methods

calling convention used. We give an example of how this can be implemented on IA32 computers in the section on the PLIBC system.

**Principle (III).** When intercepting a single function with a one-to-one map to another function having the same interface, and the replacement function is available when the program is loaded, it is sufficient to use just interception primitives for the interception. For all other cases, we must use more generic interception techniques that cannot rely on compiler-generated prologue code or other high-level artifacts.

### 4.3.3 When to Intercept a Program

For the interception to be useful, it must be done when relevant run-time state is present in the execution environment. For some functions, the only run-time state required to separate safe from unsafe execution are the parameters sent to the function. In this case, the interception can always be done just before the normal function would execute. This is true because the calling function has stored all parameters to the function according to the calling convention, and these parameters can easily be inspected. For more complex cases, where parameters alone are not a sufficient subset of state to separate safe from unsafe execution, more complex interception is required, which typically means use of several interceptions where each saves state which is later used by a final full dynamic analysis step. Because of the constantly changing execution environment, it is often difficult to extract state that we don’t know is stable (See Observation (III) on page 20), which makes intercepting function calls and parameters to functions a more reliable source of information.

In some cases, it would be useful to intercept the program after the call has executed, for example to determine if important memory data has been overwritten, but we have not performed any experimentation with this type of interception.

**Principle (IV).** When intercepting function calls, it is effective to intercept the execution just before a function is called, as this guarantees that all run-time information about parameters to the function is available. For complex analysis it might be necessary to save state from several interceptions for later use by the dynamic analysis.

### 4.3.4 Unsuccessful Interception

If the dynamic tool fails to intercept the program, the default action will be taken (See Figure 4-2 on page 30). If the interception is done when the execution environment does not contain the relevant state, the error is most likely propagated to the next step of the dynamic tool. Intercepting the program when the state cannot be reliably analyzed is clearly a problem in the interception, but the consequences will appear in the dynamic analysis or in the action step.

### 4.3.5 Dynamic Analysis

In the dynamic analysis step, the tool must find the relevant subsets of state, and analyze these to determine which action should be taken. Typically, there is no way for the dynamic analysis logic to determine if the program was intercepted at
the right time, so when designing this step we must assume that the interception was successful but also be careful not to cause damage to the execution environment should this not be the case.

Which sub-sets of the execution environment that are relevant for a particular dynamic analysis are highly dependent on the type of exploit that should be prevented. Some relevant and commonly used subsets are:

- Which function performed the call that was intercepted
- Which function was intercepted
- Which parameters were sent to the function
- Process Data, typically permissions

As we will show in the next chapter, 5. Experiments, it is possible to detect a large class of exploits by analyzing only these and a few other subsets of the execution environment. Tools that use dynamic analysis techniques analyze the actual state, so there is no need to check more than the actual data - more specifically; there is no need to assume program flow, predict state or guess parameters - something that would require significant time and resources, and still not necessarily result in a correct prediction.

However, when intercepting a large amount of function calls the interception and dynamic analysis still can take a significant amount of time compared to the time the actual function require to execute. This is the case in the PLIBC experiment, where we designed a tool (PLIBC) which enables a user to write execution policies that, depending on these and a few other execution environment sub-sets, can take certain actions. In this case, there were a large number of interceptions done and the performance of the analysis (and interception) was crucial. We have identified two ways to optimize the analysis step:

1. The tool should perform dynamic analysis only on the state required by the particular function being analysed. This optimization is useful on all functions that don’t require every analysis step to determine safe from unsafe execution.

2. If the successful analysis of a specific function shows that a particular decision can be generalized to a class of function calls, this information can be propagated to the interception step, such that future function calls falling into this newly created domain of generalizability can be handled without further invocations of the dynamic analysis step.

In the next chapter, Experiments, we will give a practical example of how these optimizations can be used in a tool, and provide performance measurements on how this optimization affects a typical executing program.

**Principle (V).** We can prevent a large class of exploits by using comparably simple forms of dynamic analysis on a large amount of functions. To get acceptable run-time performance for this type of dynamic tools, we must use highly optimized interception and analysis code.

In some cases, the dynamic analysis is far more complex and cannot be based on matching patterns in relevant execution environment state, but require specialized analysis code for each function. The performance of such tools is also impor-
Principal Function of Dynamic Methods

tant, but in our experience this type of specialized analysis tools execute less often than the more generic type, so the performance is not as critical.

4.3.6 Dynamic Analysis Result
After a successful interception, the relevant subset of the execution environment has been analyzed and the tool has made a decision on what action should be performed. If the analysis was unsuccessful, it may result in an incorrect decision or in the analysis damaging the execution environment.

4.3.7 Action
The final step taken by a dynamic method is to effect the action, decided by the dynamic analysis. Most dynamic tools execute in environments where the executing program is not designed to operate together with a dynamic tool, and for this reason there is no way for the tool to propagate information in a structured way to other parts of the execution environment.

There are two principal approaches on how to effect the action in a non-cooperative environment;

1. Take a simple action that can be effected by the operating system kernel
2. Forcefully patch the execution environment without informing the program or any other part of the execution environment about the modification

A dynamic tool operates inside the execution environment, so any communication with the operating system kernel can be made directly, as the kernel ignores which part of an execution environment that requests a particular action to be taken. However, since the function that was intercepted cannot be informed of the action taken, only a small set of simple actions make sense to take. These include;

- Termination of the process(es)
- Closing and/or reopening operating system handles
- Changing process permissions or file system root
- Logging messages to the system log (informing an operator about some condition)

In many cases, the first action is taken, assuming that a terminated program is better than an exploited one. However, randomly killing suspect programs is not always a good approach; denial of service is a form of unwanted execution, and we consider this as a form of exploit, so if possible we want to take a less destructive approach (See 2.6.2 Exploits).

Principle (VI). Simple actions can be taken by the dynamic tool calling the operating system kernel requesting operations to be carried out on the process(es) that are connected to the execution environment. When detecting run-time situations that are very likely to cause exploits, a safe way is to request the kernel to terminate the process, but this can cause a form of denial-of-service attack, so we prefer other methods, if available.
The second main approach, to forcefully patch the execution environment, can be used to change the execution flow of a program, or to prevent the program from performing a potentially dangerous operation. Forcefully patching an execution environment can, however, cause a number of unexpected, and unwanted, side effects. Randomly patching the (unknown) executing main program is typically not a good approach since there is little, if any, information on how the program will react to the modified environment. However, when intercepting a call to a well-known function, such as functions in system libraries, the designer of the dynamic tool has a good knowledge of how the function will respond when changing some state, such as parameters. This form of patching a program has fewer potential problems, but this method always has a certain level of risk and should be used with great care. We have performed some experimentation on modifying parameters, that we present in the next chapter.

**Principle (VII).** A dynamic tool can effect complex actions by modifying the execution environment directly. This type of actions can cause unwanted execution if used incorrectly, but by using information about common interfaces it is possible to modify state in a comparably safe way. Patching the execution environment can, if used correctly, remove the run-time condition that otherwise would have caused an exploit, and still let the program continue execution.

4.4 Levels of Dynamic Run-Time Protection

A number of external factors determine to which extent a system can be protected using dynamic environment hardening methods. Also, methods that are based on fundamentally different technologies can in some cases be used to protect against the same type of exploits. When comparing different methods it is beneficial to use a classification that is based on how the method modifies system behavior, rather than on the technology that the method uses to protect the system. For reasons of simplicity and convenience, we use such a classification system that contains three classes, “Immunization”, “Detection”, and “Reducing System Consequences”;

4.4.1 Immunization

Immunization represents the highest level of protection, and the best possible result we can get from a dynamic environment hardening, if we disregard possible side effects of logging.

**Definition (II).** A method is said to be immunization of a pair \langle program, exploit \rangle if, and only if, the method modifies an execution environment in such a way that, for a particular program and exploit: The executing program that given a runtime condition, such that the program would be exploited when executed without the method, given the same run-time condition and when using the method is not exploited, and not terminated, but continues execution in a way that is not significantly less dependable than it was before the run-time condition arose.
Levels of Dynamic Run-Time Protection

A slightly less formal, and hopefully more readable, version of the definition above is that a method is of the type immunization if it affects an execution environment so that a program that would normally be exploited, is not exploited but can continue normal operations. An immunizing tool may perform other operations than just to immunize the execution environment, for example write a diagnostic message to a log file.

4.4.2 Detection
Detection represents the second best protection type we can get from a dynamic environment hardening method, and hence from a tool implementing the method.

Definition (III). A method is said to be of the type detection for a pair <program, exploit> if, and only if, the method modifies an execution environment in such a way that, for a particular program and exploit: The method modifies an execution environment in such a way that, for a program executing in the execution environment given a run-time condition such that the execution environment executing without the method would have been exploited, is modified in such a way that the execution environment when executing with the method detects that the run-time condition, possibly combined with relevant subsets of the execution environment would cause or would likely cause an exploit.

The slightly less formal way of expressing this is that a method is of the type detection if it detects an exploit before the system has a chance to become exploited.

Typically, but not necessarily, detection methods take some action to prevent the system from being exploited, most often by killing the process(es) associated with the execution environment.

4.4.3 Reducing System Consequences
Reducing the consequences for a system is the third class of dynamic environment hardening methods, that represent the lowest level of protection.

Definition (IV). A method is said to reduce system consequences of a system for a given pair <program, exploit> if, and only if, the method modifies an execution environment in such a way that, for a particular program and exploit: The method modifies the execution environment in such a way that the program given a run-time condition that would have caused an exploit resulting in a number of unwanted events, given the method, causes fewer or less unwanted events.

The less formal definition is that a method reduces system consequences if it modifies an execution environment so that if exploited, the consequences for the system are not as bad as would have been the case without using the method.

4.4.4 Levels of Protection and Dynamic Methods Layers
One of the external factors that affect the degree of protection that a given dynamic method can offer is the layer (or interface) where the method is active.
Methods that reduce system consequences when an exploit occur must rely on some type of enforcement mechanism (See Principle (I) on page 29) as the actual exploit still occurs with the method active.

There are a variety of methods that reduce system consequences, where changing the file system root (chroot, jail) of a process (and hence an execution environment) is perhaps the oldest and most well-known. However, as described in Principle (II) on page 29, it is difficult to detect some types of exploits from the operating system, and for this and other reasons, such as performance, some dynamic methods operate inside the execution environment only. These methods can detect the exploits that would be difficult for kernel-based methods to detect, but since these methods operate inside the execution environment it is not possible to enforce a security policy (See Principle (I) on page 29). Combining these two methods with the classification of dynamic methods we get;

**Principle (VIII).** Dynamic methods that immunize a system from an exploit, or that detects some type of exploits typically rely on sub-sets of execution environment state that can only easily be obtained from inside an execution environment, and hence these methods must operate on this layer. Operating on the execution environment layer means that it is not possible to enforce a security policy, so while such methods can operate on non-cooperative programs, they cannot operate on hostile (exploited) programs. For this reason, this type of methods must detect a potential exploit before the program is exploited, or the protection can easily be circumvented.

### 4.5 Chapter Summary

Dynamic environment hardening can operate at different layers, and we consider two different types of methods based on which layer they operate at; those operating entirely inside an execution environment and those that in some way use the operating system kernel services to protect the environment. Methods that use the kernel can enforce a security policy which is required when handling hostile code, but methods inside the execution environment can detect some events that are difficult to detect from the kernel.

The principal function of a dynamic environment hardening tool is divided into three distinct steps; intercept, dynamic analysis, and effecting (executing) an action. There are a number of interception primitives that we use, but for all but the simplest forms of interception these must be combined with other techniques. We deal further with such techniques in chapter 5. Experiments.

The dynamic analysis must identify which subsets of state that are relevant and analyze these to make a correct decision on which action that should be taken. This is possible only if the interception is done when the state is present in the execution environment, and if the dynamic analysis logic can analyze this state correctly. We described our method of how this can be done when analyzing function parameters.

Some type of exploits can be prevented by using simple, but many, interception points, while other types are better attacked with few points of interception but
Chapter Summary

complex analysis. When using many interception points, there are a number of optimizations that can be done to greatly improve performance. As is the case with interception techniques, we deal further with these techniques in chapter 5.

Experiments.

Because most programs are designed without mechanisms for communicating with dynamic tools, such communication must be done without relying on the possibility to inform the program about what is happening. There are simple forms of actions that the kernel can carry out, typically to kill relevant processes, and more complex forms that involve the tool patching the execution environment. Applying the latter type of action can lead to serious problems, but can be made safer by using existing knowledge about how functions in system libraries operate.

We use a system to classify the type of protection a dynamic environment hardening method achieves. In this system there are three classes: immunization, detection, and reducing system consequences. Due to the properties of kernel-based dynamic methods and methods that operate entirely inside an execution environment most methods that achieve immunization or detection must operate inside the execution environment. For this reason, they cannot enforce security policies and must hence detect possible exploits before the exploit is executed.
CHAPTER 5

EXPERIMENTS

5.1 Introduction

In this chapter, we apply three different methods based on previously described principles of protection, to dynamically protect software execution. Each method operates at a different level in the execution model; immunization, detection and reduction of system consequences (See 4.4 Levels of Dynamic Run-Time Protection), resulting in three quite different protection methods. For each method we have also constructed a prototype implementation, or experiment system, which we have used to perform practical experiments on software programs.

This chapter is organized as follows; for each method, we present a background to the relevant technology, and a description of the method as well as the experiment system. After that, we describe the technical challenges we faced when designing and implementing the system and how these challenges were handled. Then, we provide details on practical experimentation including applicability for the method and results from performance benchmarks. Finally, we discuss the method and the protection offered in terms of protection principles and earlier observations on software security.

On the next page we start with the LPS system, which is designed to reduce system consequences should a system component be compromised. This is done by means of transparently separating privileges in a program, while not requiring the program to be extensively modified for this separation.

On page 51, we describe the PLIBC system which is used to apply a domain-specific execution policy to detect unwanted execution. This is done entirely within an execution environment, and can also be used to modify an execution environment should that be necessary.

Finally on page 67, we describe the SLIBC system which protects against a class of buffer overflows. This type of protection can be used to immunize vulnerable programs so they cannot be exploited but resume execution.
5.2 Experiment 1: Reducing System Consequences

The first experiment system separates a program into several parts, where distinct parts execute with different sets of permission, i.e. experiments in transparent separation of privilege. By dividing a program into several parts, the consequences of a potential exploit in all but the most privileged parts of the program can be restricted, so that the exploit cannot cause damage to the system as a whole, hence reducing consequences of the exploit for the larger system.

This type of protection is suitable for situations where it is likely that a certain part of a program has a high risk of being exploitable, but there is little or no information on what type of exploit that could occur. Without any detailed information on which type of exploit to prevent, privilege separation methods do not attempt to prevent the actual exploit from occurring, but restricts which operations a possibly exploited program is authorized to perform.

Since privilege separation methods handle hostile (potentially exploited) programs, it is necessary to use kernel-assisted methods to enforce a security policy on the program (See Principle (I) on page 29).

5.2.1 Principal Privilege Separation

Many privilege separation methods use operating system processes to take advantage of existing protection mechanisms in the kernel. To achieve this, the execution environment of the protected program can be divided into one (or more) privileged and one (or more) non-privileged processes. The privilege separation system must then establish a communication channel so the different processes can exchange state components in a controlled way. The high-level principal function of a privilege separated system is quite simple and illustrated in Figure 5-1.

![Figure 5-1: Principal Privilege Separation](source_image)

The privilege separated program is compiled and executed normally, but during execution the program divides itself into several parts and establishes a communication channel. Should an exploit occur, this limits the damages the exploit can cause if the following conditions are met:

1. The exploit occur in a non-privileged part.
2. The communication channel between the privileged and non-privileged part cannot be used to propagate the exploit.
Background on Lightweight Privilege Separation

3. The operating system kernel can enforce restrictions on the non-privileged part.
4. The system as a whole can recover from a single component failure, i.e. the system can handle the situation where the privilege separation indicates that an error occurred.

If an exploit is detected in a system where these conditions are met, it is possible to determine the maximum extent to which the system has been damaged, as it is known under which permissions the exploit executed. In a typical privilege separation system, the non-privileged parts execute with the lowest enforceable permissions, resulting in that an exploit typically cannot modify any local files which makes recovery of such a system significantly easier compared to other exploited systems.

5.2.2 Background on Lightweight Privilege Separation

To separate privileges in a program, the program must be modified to some extent. A common and straightforward way to separate privileges, is to divide a program into two processes, i.e. a single non-privileged and a single privileged process, where the vast majority of functionality executes in the non-privileged process. The privileged process is typically only used to create operating system handles (by opening files and sockets) when requested by the non-privileged part. These objects are then transferred to the non-privileged part, using some kernel-assisted method.

Using this type of privilege separation has two main advantages:

1. A program execute with the vast majority of its functionally in the non-privileged process.
2. The communication channel typically uses optimized communication primitives which results in a low performance overhead.

The main disadvantages of this approach is that it requires extensive modification to existing programs, and that transferring operating system handles between processes is dependent on non-standardized kernel support. Using this type of privilege separation is a good choice when designing new systems, or when it is possible to modify existing systems to such an extent required, and when these systems should execute on a single or a few operating systems only.

There is a large class of programs that could take advantage of privilege separation, but where it is not possible to perform this extensive modification, or where portability requirements make this type of separation impractical. We designed the Lightweight Privilege Separation system, LPS, to experiment with separating privileges in such systems, and to validate certain of our hypotheses on dynamic protection. Because of the requirement that LPS should not require extensive modification to existing systems, such as re-writing extensive amounts of code, it uses a fundamentally different approach with a single privileged and several non-privileged processes.
The LPS Experiment System

The fundamental design principle of the LPS system is described in the following quote from one of our technical reports [3];

“Typically, there is a small set of functions that contain most of the vulnerabilities in a given program. For example, functions that parse data from a human readable format into a machine readable format (parsers) have been known to contain many vulnerabilities. This fact has been known for quite some time and still there are new vulnerabilities found in parsers that have already been reviewed several times. Many network servers need to parse a user name and a password before they can switch to a lower privilege and hence, some parsing of potential hostile input is done under super user privileges. By placing code that we believe is more likely to contain vulnerabilities in a low-privilege access environment, we can reduce the consequence if that code in fact should contain a vulnerability and be exploited. By allowing the majority of a program to execute under high privileges, and place restrictions only on carefully selected parts, we can impose a privilege separation on the most risky parts with minimal, if any, impact on other parts of the program.” [3]

5.3 The LPS Experiment System

The LPS system separates privileges in a program using a single privileged and multiple non-privileged processes. The main functionality of the program execute with high privileges, and at certain points the LPS system create the non-privileged processes and execute selected parts of the program in such a process. To provide this protection, the LPS system has two main features;

1. Dynamically creating new processes where high-risk parts of the program can execute in an environment that is very close to the original execution environment.
2. Transferring selected subsets of the execution environment from a non-privileged process back to the privileged main process in a dependable way.

By using this approach, the privilege separation can be made transparent to the vast majority of the execution environment, as the separation is done only on a small part of the functionality each time and because LPS transfers relevant state back to the privileged process which then resumes normal execution.

Following the principal function of our model for dynamic methods, we divide the function of LPS into distinct three steps - Interception, Dynamic Analysis and Action; (See 4.3 Principal Function of Dynamic Methods)

5.3.1 Interception

The first step, interception, is a NULL-interception where the program explicitly calls the LPS system, requesting a privilege separated call to be made. Explicitly calling the interception is the only modification typically done when adapting a program to use the LPS system, and this is typically a small modification done at a
single place in the source code only. Consider the example in Figure 5-2 that illustrates how a simple function call is modified to use the LPS system.

When invoked, the LPS system creates a clone of the current process using the `fork` system call, and applies restrictions to this process as requested, or, as in the example above, using default values. We assume the reader to be familiar with the semantics of the `fork` call, and how processes are created in a Unix-like environment, and will not further elaborate on how we obtain a clone of the existing process.

Once the new process is created and restrictions are applied, LPS transfers control to the code that should execute with low privileges. When this code has finished its execution, the LPS system transfers selected subsets of the execution environment back to the privileged process. The privileged process waits for this data, and once received, begins the dynamic analysis phase. This mimics the behavior of a normal function call, and is, in most cases, transparent to the privileged process.

5.3.2 Dynamic Analysis

In the dynamic analysis step, the data received from the non-privileged process is analyzed and verified against the specification in the `LPS_END_SEPARATION` macro. The data is serialized in a simple format by the LPS system in the non-privileged process, and transmitted through a pipe to the privileged process. If the data received meet a number of requirements such as size and type, it is applied to the execution environment in the privileged process.

When integrating the LPS system into an environment, it is important to identify which state that should be transferred back to the privileged process. As observed in the technical report [3], there are three different type of state that should be considered:

1. State modified by the non-privileged code that is required by code in the privileged part to continue execution.
2. State modified by the non-privileged code that is temporary and not required by other parts of the program (scrap buffers).
The LPS Experiment System

The first and second type of state are handled respectively by transferring, and not transferring the state back to the program. The third type is the most important, and includes state such as a variable that determines if a user has been authenticated successfully. This type of state must not be transmitted back to the privileged process, but rather the primitives for such state (such as a user name and a password) so that the privileged process can make the decision whether the user is authenticated. If this type of state is transmitted to the privileged process there is a risk that this can be used by hostile code executing as a result of an exploit.

Once the state has been verified and applied to the execution environment of the privileged process, LPS returns control to the program which can resume execution. To the program this will look like only a normal function call had been made.

5.3.3 Action
Under normal circumstances, the LPS system perform the action of applying relevant state, which is performed directly in conjunction to the dynamic analysis. If an error occur LPS will, however, perform some actions to assert that the error does not propagate into the protected part. We now describe some of these situations and how the LPS system handle these.

5.3.4 Challenges
We encountered a number of challenges when implementing the lightweight privilege separation system. The different challenges and our way of handling them are described below;

**System Errors.** A normal function call cannot fail. A function might return an error value, but this is essentially a return value like any other. When changing a normal call to a privilege separated one, a number of errors can occur, and these must be handled in such a way that the system does not introduce new exploits in the program. The following situations where errors can occur were identified;

1. Before the new process is created, typically as a result from errors in the *fork* or *pipe* system calls.
2. Before executing code in the non-privileged process, typically because the kernel refused to lower privileges for the to-be non-privileged process.
3. If the non-privileged process does not terminate correctly, typically because of an exploit.
4. If the data received for dynamic analysis is invalid.

The first and second situations are handled using a special error-return variable (See Figure 5-2), so if a system call fails prior to creating the non-privileged process, an error is returned. If the non-privileged process is unable to reduce its privileges it uses the pipe to the privileged process to indicate this situation, causing the LPS system in the privileged part to return this as an error.

There is no way for the privileged process to determine if an error has occurred in the non-privileged process. Instead, there is a maximum time specified that the
privileged process will wait for data. After this time, it kills the non-privileged process and return an error through the error-return variable.

In the dynamic analysis step, a number of tests are applied to the data received in order to determine whether the data should be applied. If an error is detected, the LPS system uses the error-return variable to indicate this situation, and does not apply more state to the process.

When integrating the LPS system into a program, it is important to test the error-return variable after privilege separation to determine if there was an error. In many cases, the program should be terminated if an error is detected, but this is not done automatically by the system as doing so could introduce a denial-of-service situation if not handled correctly (See Principle (VI).) Privilege separation requires some modification to the source code, and for this reason we also require the developer to handle potential error situations in a way suitable for the specific program being separated.

Cloning a Process and Cloning the Execution Environment. By cloning the privileged process and then applying restrictions to the new process, an execution environment is created that closely reassembles the original environment. However, some type of state that cannot reliably be cloned in this way, despite the fact that the process was cloned, and other state that might affect execution in the non-privileged process. Such state is a part of the process data (See 3.2 A Model For Execution Environment Behavior), most often file descriptors and System V semaphores.

To create a secure privilege separation and still create an execution environment similar to the original, we must consider each file descriptor and in which way it is used. The default behavior in LPS is to close standard input, standard output and standard error descriptors, and then open /dev/null on the corresponding descriptor number. This ensures that the non-privileged part of a program does not communicate directly with the user, something that is particularly important for network servers where these descriptors typically are mapped to a network socket. In the default configuration, LPS does not modify any other descriptors, but leaves the decision on how such descriptors should be handled to the developer that integrates the privilege separation.

It is important that the developer that performs this integration understands how the kernel enforces security policies when a file is opened but not when it is used. Because of this semantics, a file descriptor can be left open in the non-privileged process if it is suitable that the non-privileged part should be able to use the object represented by the descriptor, and suitable to close the descriptor if this is not the case. Once such descriptor is closed it cannot be re-opened, because the process is then executing with lower permissions that do not permit this action.

Complex State. When integrating the LPS system, it is not sufficient to consider only the state that should be transferred to the non-privileged part. We must also consider how relevant state components should be transferred back to the privileged part once the non-privileged code has finished executing.
The LPS Experiment System

In most cases, it is possible to place the separation at such a place in the program that it is easy to determine which state components to transfer back. Typically, the separation should be placed around a single function that has a well-defined interface in regard to what type of data it returns. For such cases, we only need to consider this data and any global data that the function modifies.

Not all type of state can be transferred from the non-privileged process back to the privileged. For example, transferring shared memory regions, System V semaphores, or open file handles is not supported. It is up to the developer integrating the LPS system to determine where in the program the separation of privileges should be done so that there is no need to transfer this type of state.

Waiting for, and Transferring State. The framework in LPS permits transferring all primitive data types found in the C language, as well as arbitrary memory buffers of a fixed size, and C-style NULL-terminated strings of a pre-determined maximum length. It also offers a call-back mechanism to transfer complex data such as structures that are decomposed into primitive types by the call-back and then recomposed into complex data structures by corresponding code in the privileged process. By only transferring primitive types, it is possible to copy data binary between different processes without considering how the data is represented, and also apply simple but effective validation on the data.

The LPS system transfers state components in the following way to protect the privileged process;

1. LPS waits a pre-defined maximum time for a data head.
2. The data head contains two data items, size of the data body and an error code. If the error code is non-zero, which indicate problems in the non-privileged part, an error is generated. Otherwise, the size is verified against pre-configured minimum and maximum values.
3. LPS waits a pre-defined maximum time for the data body.
4. The entire data body is read and verified to be of the exact size specified in the head.
5. Each datum transferred is decoded according to type and size, which is also verified against the separation macro.

By using a simple format, we avoid placing a complex parser in the privileged part, but perform only simple validations of binary data. By applying time-out values and restrictions on data size, it is possible to handle possible denial-of-service situations.

5.3.5 Performance

LPS uses a model with a single privileged and multiple non-privileged processes and also serializes state over a pipe when transmitted back to the privileged process. Using this model, each privilege separation requires a number of system calls to be made; a new process must be created (fork), a pipe opened between the processes (pipe) and privileges reduced in the new process. (setuid, seteuid, chroot, jail etc.) In some cases, open file descriptors must also be closed (close). All these operations are system calls, and there is no other way to call these than to invoke the kernel which has a large performance overhead.
compared to normal function calls (See 4.2 Construction of Dynamic Environment Hardening Tools) Because of the slow system call interface, and the large number of system calls that must be made to separate just a single call, one might expect quite high overhead in the LPS system.

For the original technical report describing LPS, we performed extensive performance measurements, and the results from these measurements were presented in the report [3]. We provide a summary of these results below;

For the benchmark two test cases were used; test case one being a simple function calculating the value \( f(x) = x + 1 \) using int type as parameter and return value, hence using 4 bytes of state to transfer back to the privileged process. The second test case consisted of a function that filled a 1024-byte buffer with a byte pattern. This buffer then had to be transmitted back to the privileged process, resulting in a pay-load of 1024 bytes (1 Kb) of state. The test cases were selected based on the following three conditions;

1. We wanted to determine the effect the amount of state had on the overhead.
2. Each function executes very fast compared to the overhead in just a single system call. For this reason, we can disregard the time taken by actual execution of the function.
3. Each function is defined in such a way that it cannot be removed by optimization code in the compiler; the actual code has to execute to generate the respective results.

Each test case was executed on two different computers under three different operating systems. The first computer (A) was a 400Mhz Pentium II Compaq with 160Mb RAM, and the second (B) a 2.7Ghz Pentium 4 Dell with 512Mb RAM. These were the slowest and fastest computers respectively that were easily available to us at the time. The benchmark was performed on FreeBSD (4.8), OpenBSD (3.2) and a RedHat Linux 8 (Using a 2.4.18-14 kernel) operating systems and the benchmark program was compiled using the system default compiler [30, 31]. See Table 5-1 for the results from the benchmark.

Table 5-1: Performance Benchmark Results in Milliseconds

<table>
<thead>
<tr>
<th>Test Case</th>
<th>System A (400Mhz Pentium II)</th>
<th>System B (2.7Ghz Pentium 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FreeBSD</td>
<td>OpenBSD</td>
</tr>
<tr>
<td>One (4 byte)</td>
<td>5.179</td>
<td>1.881</td>
</tr>
<tr>
<td>Two (1 Kb)</td>
<td>5.198</td>
<td>2.127</td>
</tr>
</tbody>
</table>
The LPS Experiment System

From these results we make two observations;

1. LPS executes significantly slower on FreeBSD than on OpenBSD and Linux, and that using a faster computer only marginally improves the performance.
2. On Linux and OpenBSD the execution time on a modern computer is well below one millisecond, and that the amount of state transmitted does not vastly affect the execution time.

We did not perform any experimentation on the newer FreeBSD 5 series at the time, but from what we understand this version has different performance characteristics than the 4 series, so it is not necessarily so that our results can be generalized to the newer version.

5.3.6 Discussion

Separation of privileges is a type of dynamic method that is comparably easy to use, and is also quite a common way to make a program or system more resilient. By separating privileges, we don’t actually protect the program or system from an exploit, but control the consequences of what harm an exploit could do. Because the protection operates at this level (See 4.4 Levels of Dynamic Run-Time Protection) the system to be protected must be built in such a way that it can recover from a situation where the privilege separation exists with an error (See 5.2.1 Principal Privilege Separation) If this is not the case, but the program crashes, the only part of the system that is protected is the surrounding operating system and other programs that otherwise could have been affected by hostile code executed as a part of a privileged exploit. We provide a more thorough discussion on this topic in the final chapter; see 7.1 Further Observations on Dynamic and Static Methods.

Some common privilege separation tools require quite extensive modifications of the source of the program that should be protected, while the system we have experimented with, and described above, is designed to be almost transparent to the program. Independently of approach, however, the privilege separation is a dynamic method, as it targets the execution of the program and the consequences an exploit could have for the (presumably larger) system, and does not perform off-line validation of a program source. See also Observation (VII) on page 24 for a discussion on concrete exploits and abstract vulnerabilities that closely relate to this topic, although in this case the consequence is concrete, the exploit abstract, and the vulnerability even more so. Given this view, privilege separation is a dynamic method even for tools that require large modifications of a program source to practically integrate the protection into a program environment.

While privilege separation is a dynamic method, it is a form of static hardening of the execution environment (See 3.3 A Model for Execution Environment Hardening) This is because by separating privileges, we apply a set of static restrictions to the execution of a certain part of the program. While some tools, such as LPS, allow the privileges to be specified dynamically before separation, and these
restrictions can be different each time, the restrictions do not change depending on the actual execution in the non-privileged part, and is in this sense static.

From a more theoretical point of view, we observe that not all forms of privilege separation necessarily are a static form of environment hardening. For example, consider a privilege separation tool that analyses the actual execution (of the non-privileged part) and applies different security policies depending on the live results from such analysis. Such a tool would provide a dynamic form of environment hardening. Practically constructing such a tool would likely require a virtual machine or some form of enforceable single stepping support in the CPU, since the method would still handle possibly exploited code, that is likely to perform hostile operations and actively avoid cooperation with an analysis tool. We are currently not aware of any tools that operate in this fashion.

Another important aspect of a privilege separation tool is the actual dependability of the tool. As noted in Observation (IX) on page 25, a vulnerability in a dynamic inspection tool can propagate to the program being protected. As privilege separation typically is applied to parts of a program that we believe to be particularly risky, a vulnerability in the separation tool could open for exceptionally problematic exploits in the protected program. However, it is also important that a separation tool does not impose a vast overhead on the execution, as this might cause unexpected behavior in a program where the developer assumed high speed internal function calls.

Our LPS system is designed for simplicity for several reasons; using a simple model with a single non-privileged process per separated call, and by using a simple way to transmit data back to the privileged process is it less likely to be vulnerabilities in the LPS enhanced system. In LPS, data is transmitted in binary form over a pipe to the privileged process, with pre-established bounds and minimal binary meta-data. This means that the size of each component in the data stream is known as well as the total size, and that there is no need for a complex parser to interpret the data. Parsers are traditionally problematic from a security perspective, and are often a source of vulnerabilities. The low speed in LPS is a direct consequence of the high number of system calls (calls to the kernel) required to separate privileges, and we don’t believe there is much room for optimization inside the tool. For this reason, LPS and tools that are based on similar techniques are not suitable to integrate in real-time software that modifies its behavior based on wall-clock measurements.

5.3.7 Protection Profile

The LPS system, like any other privilege separation system, should be used when there is little or no hard information about what exploit or vulnerability to protect a system against. The level of protection is in the form of reducing system consequences, which is the weakest form of protection we consider in this thesis. The performance overhead of using LPS is about 1 ms on OpenBSD and Linux systems, and higher on FreeBSD. This is far more than the other methods we discuss. In our opinion, the low level of protection offered by the method, and the comparably high overhead is the price for the extreme flexibility in what kind of
The LPS Experiment System

exploits the method can protect against and the relative ease with which the method can be integrated into existing systems.

We summarize the principal protection offered by privilege separation in general, and by the LPS system in particular, by returning to the principles of protection and the relevance for each principle on the LPS experiment as described in Table 5-2.

Table 5-2: Protection Principles for LPS

<table>
<thead>
<tr>
<th>#</th>
<th>Summary of Principle</th>
<th>Relevance for Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Dynamic Methods that handle hostile code must rely on the operating system kernel for enforceability (See page 29.)</td>
<td>High</td>
</tr>
<tr>
<td>II</td>
<td>Some types of exploits can only easily be detected from inside the execution environment (See page 29.)</td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>For simple interception, interception primitives alone are sufficient (See page 32.)</td>
<td>Medium</td>
</tr>
<tr>
<td>IV</td>
<td>A function can be intercepted just prior to the normal invocation to obtain the parameters (See page 32.)</td>
<td>Medium</td>
</tr>
<tr>
<td>V</td>
<td>Using only simple analysis on a large amount of functions is useful to prevent some types of exploits (See page 33.)</td>
<td></td>
</tr>
<tr>
<td>VI</td>
<td>Simple actions can be taken by invoking the operating system kernel on the process(es) (See page 34.)</td>
<td>High</td>
</tr>
<tr>
<td>VII</td>
<td>Complex actions can be taken by the tool modifying the execution environment (See page 35.)</td>
<td></td>
</tr>
<tr>
<td>VIII</td>
<td>Dynamic methods that operate inside the execution environment must operate before a program is exploited (See page 37.)</td>
<td></td>
</tr>
</tbody>
</table>

The first principle is highly relevant for LPS and any other privilege separation; the sole purpose of separating privileges is to reduce consequences a hostile program can perform, so enforceability is an absolute requirement. This could be solved in other ways by using the operating system kernel, but the concept of processes is convenient as the kernel already handles permissions on a per-process basis. LPS uses a very simple interception method where a program actually must be modified to explicitly call the system (NULL-interception.) It might be possible to enhance the interception mechanism to support privilege separation without access to the source code, but this require a way to propagate errors back to the program, and a reliable way to transfer state back to the privileged part. This would also require a more complex form of interception. Thanks to the simple interception, there are no issues in obtaining required state components such as variables. Principle VI is also relevant, as the security properties of privilege separation is dependent on the operating system kernel performing actions on the processes in which the program execute.
5.4 Experiment 2: Detecting Potential Exploits

For the second experiment system, the focus is placed on protecting program execution by detecting, and responding accordingly to, exploits just before the exploit normally would execute. This type of protection thus require analysis of run-time state just before a vulnerable function is executed, and a way to specify which run-time conditions should result in a certain action being taken.

For these experiments, we have designed an experiment system, PLIBC, that allows an administrator to apply a policy for the execution of a program. In this policy a number of conditions can be specified, and which actions the system should take if these conditions are met. We describe this system in the next section, 5.5 The plibc Experiment System, where we also present results from performance benchmarks and practical experimentation results from using the system. In the next section we also provide a background to the technology as well as a discussion on where and when this type of policy-based execution is suitable.

5.4.1 Principal Policy-Based Execution

Policy-based execution is a wide term and can technically refer to any type of technology that allows parameterized control over some form of execution. A common form of policy-based execution is software at one layer of a system that enforces a policy on a program executing at a higher level. For example, this is the case in some virtual machines, such as Java JVM, and when using an operating system kernel to enforce a policy on a possibly hostile program. For the experiment and due to the particular context in this thesis, we will not consider kernel or virtual machine assisted policy-based execution, but focus entirely on policies within an execution environment.

Using a policy at the execution environment layer implies that there is no way to enforce the policy on a hostile program (See Principle (I) on page 29). As the policy cannot be enforced, execution environment based methods must detect an exploit before the exploit execute. Using kernel-assisted policy methods are advantageous, because of the enforceability properties, but using execution environment execution policies might seem a bit more obscure. However, using policies at the execution environment interface means that it is possible to detect situations that could not easily be detected using kernel-based methods (See Principle (II) on page 29), and as we have observed in the PLIBC experiment, policies at this level impose a comparably low overhead on the performance. Also, using execution environment based policies it is possible to apply more complex forms of decisions than what could easily be done from the kernel alone, something that we also experimented with practically in PLIBC.

This type of policy-based execution have quite different properties than methods based on enforceable security, and from a high-level view, we can see the addition of an execution policy as yet another component in the execution environment that does not originate from the source code for the main executing program. See Figure 5-3 on page 52 for an overview of how execution environ-
Detecting Potential Exploits

ment policies fit in the construction of modern software, and compare with Figure 2-1 on page 5 which describe the steps involved in building such software.

There are several ways in which an execution policy can be transformed ("compiled") so it can be used in an effective way in the executing binary. A policy-compiler could transform rules to either a dynamic library that can be loaded directly by the dynamic linker or transform the rules to another format. Technically, the compiled policy could be identical to the policy written by the administrator (NULL-compilation), however in the experiments with PLIBC we observed that the performance of the system could be greatly improved by using a highly optimized format for the policy.

Uses of Policy-Based Execution. There are a number of situations where execution environment based execution policies are suitable. For example, when using general-purpose software in environments for which the software was not originally developed, there might be other requirements on the software execution. This is, for example, the case when using software to control high-level aspects of critical infrastructures;

"Much general-purpose software is used today in business sensitive or critical environments, for which the software was not originally designed. In these environments, there are typically different requirements on the software execution properties than what might normally be the case. For example, software used on a closed intranet to monitor device output might have high requirements on reliability and logging of erroneous states, but lesser requirements on performance and fully supporting standards that are only partially used." [2]

Typical environments where there is an interest for using general-purpose software for tasks more critical than the software originally was designed for include to perform non-realtime control of the electrical power grid, and some military applications [20].

There are more uses of execution policies than just controlling the electrical power grid or for military purposes, and we have no intention of listing all uses exhaustively, but conclude that in situations where it is possible to trade some run-time performance for dynamic control over program execution, especially with respect to applying domain-specific rules for the execution, using execution policies is generally a technique worth considering.
5.5 The PLIBC Experiment System

The PLIBC system is an execution environment based system for applying non-enforceable execution policies to non-cooperative programs. Policies are written in a special-purpose high-level scripting language, compiled to a binary format and then used by the dynamic inspector. Using PLIBC, an administrator can apply a simple type of policies that allow or deny a particular function call based on runtime conditions as well as policies that use more complex actions, such as runtime patching of the program.

From a technical point of view, the principle function of PLIBC is to intercept a vast amount of function calls and based on the execution policy perform comparably simple dynamic analysis. Since the tool intercepts a large set of functions, high performance of the interception part is crucial for the usability of the tool (See Principle (V) on page 33), but with simpler dynamic analysis we can restrict the sub-sets of the execution environment that can be analyzed. The focus on PLIBC is on unsafe uses of library functions (hence the acronym PLIBC meaning protecting library calls), which results in the dynamic analysis handling parameters to the function (See Principle (IV) on page 32) and only a few other subsets of execution environment state.

5.5.1 Execution Policies In PLIBC

From the operators perspective, the first step when using PLIBC is to write an execution policy for a particular program, and to compile this policy into a binary format. An execution policy is written in a special script language, which is designed to allow a high level optimization when compiled. A policy consists of several rules that are evaluated in order, and where each rule has the format;

```
action function-name [condition] [predicate]
```

For example, consider the following rules that use the simple “allow” and “deny” actions;

```
allow gets if not euid 0
deny gets
deny printf if param-match 1 "%n"
```

Example 5-1: Simple PLIBC Execution Policy Rules

When applied to a program, the first and second rule will allow only programs that do not execute with super-user permissions to use the `gets` function. The third rule will deny a program to call the `printf` function if the first parameter to `printf` points to a null-terminated string that contain the characters “%n”. It is also possible to use more complex actions than “allow” and “deny”, for example;

```
map gets alt_gets if buffer-origin 1 stack
replace-param printf 1 ptr “repl” if param-match 1 “%n”
log syslog “is” if param-match 2 “%s”
```

Example 5-2: Complex Execution Policy Rules
The plibc Experiment System

In this example, the first rule modifies calls to the function “gets” if the first parameter points to a memory address on the stack. In the second rule the first parameter to printf is changed to “repl” if the original first parameter pointed to a string containing the sub-string “%n”. The third rule causes plibc to log the first two parameters sent to syslog if the pointer in the second parameter referred to a string that contains the characters “%s”. This rule might seem a bit strange at first, but it is useful to log when a program fails to specify the length of a string sent as a parameter to syslog.

When compiled, the rules are combined by the function referred (function-name) into an entity called rule-set, but retain their internal order for the particular function. This enables efficient run-time look up for all rules that apply to a certain function, but also enables “first match” rule semantics that is used in other rule languages and easy to understand. The compiler canonizes all data to primitive types and constants in order to reduce the parsing necessary at run-time. The compiled execution policy is stored in a binary format, but the principal contents and organization of the file is described in Figure 5-4.

![Figure 5-4: Contents of a Compiled Execution Policy](image)

5.5.2 Loading Rules in Run-Time

PLIBC can be loaded with a program in several ways, although the most common way is as a dynamic library. From a conceptual view, we consider the compiled execution policy to be included as any other object into the loaded binary which then become a part of the execution environment. This would also be the case, in reality, if the PLIBC rule compiler built a dynamic shared object from the textual rule file. For several reasons, the rules are not compiled into a dynamic shared object, but to a much simpler format, that cannot be loaded directly by the run-time linker. Instead, the run-time linker loads the PLIBC dynamic shared object together with other objects and execute the program in this environment. The PLIBC library is constructed in such a way that it is invoked before the dynamic linker calls the main function in the program. As soon as PLIBC is invoked, it loads the file with compiled rules and applies the execution policy.
5.5.3 Interception

It is possible for an administrator to define an execution policy that involves any function from a protected library, meaning that PLIBC must have a way to intercept arbitrary function calls in the protected library. To do this, PLIBC actually intercepts all function calls in the library and then by dynamic analysis determine which action should be taken. This requires quite a complex interception mechanism compared to that of our previously described experiment system - LPS - as well as SLIBC which is the experiment system described in the next section.

As illustrated in Figure 5-4, a rule-set contains all rules for a particular function, identified by the name of the function. Identifying functions by name rather than by address is convenient and resistant to changes in the library layout, but require mapping between names and addresses in run-time.

Information about function names are available in all libraries, but is stored in a way that makes it time-consuming to obtain.

Also, because all library calls are intercepted, PLIBC must handle each call with a very low overhead or the system will not be usable. This makes reading function names from the library prohibitively expensive, and also requires a mechanism to avoid unnecessary invocations of the dynamic analysis step.

To meet performance requirements, and still provide function names in run-time, each function in a protected library is replaced with a unique function in PLIBC. These replacement functions are called “hook functions” as they are the first hook into the dynamic analysis in PLIBC. Hook functions are automatically created when PLIBC is configured to protect a particular library. In the typical case, protecting the C library, several hundred hook functions are created. Each function is created from a template, but contain two unique state elements that are used in run-time;

- Name of the function intercepted
- Cache
The plibc Experiment System

The name of all functions are easily available when PLIBC is built (before the program is executed), and hence extracted and stored in each hook function. Each hook function also contain a piece of cache that is used to optimize empty or simple rules.

When a function call is intercepted by PLIBC, a number of steps are taken, as illustrated in Figure 5-6:

**Cache Look-up.** Firstly, the hook function checks the state of its cache. If this particular function has already been called once, and the decision from the dynamic analysis is generalizable to all calls to this function, then the cache contains a pointer to the function that should be invoked. If this is the case, the entire interception, analysis and action steps are finished, and control is transferred to that function.

Since hook functions are not implemented in a high-level language, but directly in assembly, the overhead for looking in the cache is only four machine code instructions, and can safely be done without modifying the environment. This is one of the factors that make the overhead extremely low (See 5.5.9 Performance)

**Obtain Run-time Information.** The second step is taken only if there was no cached result available. This means that the dynamic analysis step should be invoked to determine which action to take. The dynamic analysis is written in a high-level language (C) and the hook function must create a context so it can call high-level code without destroying state for the intercepted function. Additionally, the hook function obtain run-time information that is used by the analysis. Dynamic information obtained by the hook function include:

- The name of the intercepted function (compiled into the hook function)
- Parameters sent to the intercepted function
- Pointer to the cache

The dynamic analysis step also obtain other state information if required by the policy, but these elements are always obtained by the hook function as they cannot easily be obtained from the high-level code. Once there is a context for the call and the run-time information is available, the hook function invokes the...
dynamic inspection part, described in the next section 5.5.4 plibc Dynamic Analysis.

**Restore Environment.** When finished, the dynamic analysis returns to the hook function, which restores the execution environment to be equivalent to that before the analysis. The hook function then transfers control to the appropriate function as determined by the dynamic analysis, which will effect the action.

Hook functions are generated from a template, are small (12 machine code instructions long) and efficient. The hook function never modifies its cache (this is done by the dynamic analysis) but merely prepares state for the dynamic analysis or effects generalizable actions. Using this approach gives a significantly lower overhead than to search a library for function names, or even to look up a name in a hash table. Just invoking a high-level function would typically require four or five machine code instructions overhead, which is a considerable overhead in this context.

### 5.5.4 PLIBC Dynamic Analysis

The dynamic analysis evaluates the execution policy to determine which action should be taken given the current execution environment. The dynamic analysis also determines if the action is generalizable and, if so, updates the cache for the hook function. The following steps are taken by the dynamic analysis;

The first step is to find the rule-set for the current function. The interception step has already determined the name of the function, and the compiled policy contain an optimized representation of the rule-sets available. If no rule-set is found, a single rule, unconditionally allowing the call, is assumed. The dynamic analysis handles functions by name (rather than by address) only. For a discussion of the implications of this, please see 5.5.10 Discussion

Secondly, each rule in the rule-set is evaluated in the order declared in the execution policy file. If the condition and predicate match the current run-time state, the action part of the rule is effected. Some actions are final and will halt further rule evaluation, and these are returned to the hook function. Other rules only modify some aspect of the execution environment and can be combined with more rules. The following actions can be taken from a rule;

<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow</td>
<td>Allow the call. This is the default action, and the way a program would behave without PLIBC</td>
<td>Final</td>
</tr>
<tr>
<td>deny</td>
<td>Terminate the program.</td>
<td>Final</td>
</tr>
<tr>
<td>map</td>
<td>Modify the call so control is transferred to another function than the program intended to call.</td>
<td>Final</td>
</tr>
</tbody>
</table>

Table 5-3: PLIBC Rule Actions
The plibc Experiment System

### Table 5-3: PLIBC Rule Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Explanation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>Log information about the call, including parameters sent to the function.</td>
<td>Modifies Execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Environment</td>
</tr>
<tr>
<td>replace-param</td>
<td>Change the value of a parameter to the function.</td>
<td>Modifies Execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Environment</td>
</tr>
<tr>
<td>nop</td>
<td>Do nothing.</td>
<td>Modifies Execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Environment</td>
</tr>
</tbody>
</table>

The nop instruction does nothing and is a special type of instruction used to prevent optimization, normally not used in live systems. Also, the deny action is generalizable to the optimizer, but as the program immediately will be terminated whenever a deny-rule is executed the practical use of the generalizeability is small.

When a final action has been set, the last step for the dynamic analysis is to determine if the rules can be generalized to the next time this particular function is executed. PLIBC uses a conservative estimation, and considers an action to be generalizable if, and only if:

- the final action is set by the first rule in the rule-set, and
- the rule had no condition set.

This means that empty rule-sets and unconditional allow-rules always are generalized, and that map actions that statically map one function to another also are generalized. This method does, however, not optimize for conditions that are guaranteed to be the same throughout the execution of the program, although such optimization would be possible. However, no such conditions currently exist in PLIBC.

#### 5.5.5 Effect Action

The last step is to effect the action decided by the dynamic analysis. Possible actions are:

- Terminate the program (deny)
- Transfer control to some function (allow and map)

These actions are effected by PLIBC without invoking the C library, as there are situations, such as unwanted recursion, where the C library cannot reliably be called. Also the control transfer mechanism is handled directly by the hook function to reduce overhead.

#### 5.5.6 Challenges

We encountered a number of challenges in the work with designing and implementing the PLIBC system. These were handled as described below:

**Performance.** When performing massive interception, performance becomes an important issue. Most of the challenges we faced were in some way related to
performance and how a certain problem could be handled without imposing a vast overhead on run-time performance.

**Obtaining Run-time Information.** The first approach to obtaining run-time information about a function (primarily the function name), was to build some kind of data base mapping function addresses, which can easily be obtained, to function names. This approach would require searching the data base each time a call was intercepted, which would have taken a significant amount of time. The overhead of searching the data base would have been significant even when using a highly optimized algorithm. This is true because the overhead in just calling an empty function is slightly higher as the current approach, so even if the algorithm could find the function name using zero machine code instructions, it would impose a higher overhead. This is the background to the “hook functions” that recreate this information in a more effective way.

**Obtaining Additional Run-time Information.** When logging a function call, information about the type of parameters sent to the function are crucial. Because of the way functions are called using the C calling convention there is no such information available in run-time, but this information must be known by the calling function and the function being called. When intercepting such calls there is no way to determine the actual type of parameters, but this information must be obtained else were.

To make matters worse, parameter information is not available in a compiled library, but only in header and source code files. Analyzing all header files for a library would be a complex and time-consuming task. Also, many from a security perspective interesting functions use variable arguments, meaning that the type of arguments are not even specified in the header or source code file, but determined in run-time based on the value of other parameters. Assuming the wrong type for a parameter in run-time could cause unexpected execution, most likely causing the program to crash.

To handle this in an effective way PLIBC does not attempt to guess types, analyze all functions, or trying to determine the type a particular parameter has, but requires additional information to be specified in the execution policy file. When using rules that require information about parameter types, this information is specified in the policy, and assumed to be correct. Most rules in PLIBC does not require information about parameters to operate, and for the few that does, this information must be obtained manually by the administrator.

**Recursive Calls and Multi-threading.** Some functions in PLIBC use functionality from the C library, which is most often the library being protected. To avoid the situation of uncontrolled recursion where an intercepted function makes PLIBC call the C library, which is intercepted by PLIBC (... etc.), PLIBC temporarily disables the interception mechanism when active. Consequently, calls made from PLIBC to a protected library are never verified against the execution policy and thus avoid the possible situation of recursion, but, more importantly, also implies that PLIBC assumes a single threaded operation. If a call made from one thread is being verified by PLIBC, calls made from all other threads pass through without verification, as long as PLIBC is active in the first thread. This problem can easily be avoided by
using thread-private data to store the temporary-disable lock, but this requires knowledge about the type of threading the program uses. The current version of PLIBC does not handle any type of multi-threaded programs, but given knowledge of the threading model a particular program uses, it would be trivial to add support for multi-threaded programs as well.

5.5.7 Experimentation

To experiment with PLIBC in a real-world setting, we applied an execution policy to ISC DHCPd version 3.0 [32]. This version is known to contain a “format string” type of vulnerability [35], that we will perform the practical experimentation on. The protection policy for DHCPd, fits two scenarios;

1. A report about a newly found vulnerability has reached relevant security mailing lists. There is not yet much information about the exploit available, especially no information on which operating systems the vulnerability is exploitable or which proof-of-concept exploits that circulate. We also assume that the program in question (dhcpd) is deployed on several servers, and that it is compiled using a customized build process. Hence, an administrator cannot instantly switch to a new version. The administrator can, however, write an execution policy that terminates the program if the vulnerable function is called as described in the vulnerability report.

2. A new service (dhcpd) is added to a number of servers in an organization. There are no public reports about vulnerabilities available, but because of the dhcp protocol the server must execute with super-user privileges. Also, these servers will be used in critical environments, so it is important that the administrator can control which actions are taken if functions that are known to be problematic are called with parameters that are particularly risky.

Both scenarios make it suitable to write an execution policy for dhcpd, although the actual policies will likely differ. In the first case we have some knowledge about which function is vulnerable and by reading the vulnerability report or looking at a proof-of-concept exploit the vulnerable run-time parameters can easily be identified. In the second case we might consider an execution policy that prevents some dangerous situations from executing, but there is little or no information about a particular exploit we are attacking.
5.5.8 Test Cases

From these scenarios, and with this background, we use the following three test cases;

**Baseline.** The system is executed without any PLIBC protection at all.

**Denying Rule-set.** In this case we use a rule-set that denies seldomly used and potentially dangerous uses of ANSI C string formatting. The following policy rules are applied;

allow vsnprintf if not euid 0
 deny vsnprintf if param-match 3 "%n"
 deny vsnprintf if param-match 3 "%hn"

**Example 5-3: Rule-set to Deny vsnprintf Calls Based on Format String**

The first rule allow use of vsnprintf regardless of parameters if the program does not execute with super-user privileges. The second and third rule apply only when the program executes with super user privileges and denies use of the vsnprintf function if "%n" or "%hn" is found in the third parameter. This can be seen as a generic rule-set although the rules typically would match many more functions than just vsnprintf. For readers unfamiliar with "%n"-style formatting it is worth mentioning that this is used to write the number of bytes stored so far in the string to a pointer provided by the user. Essentially all format-string exploits use this facility to modify program execution, but the directive is seldom used otherwise. It is, however, a part of the ANSI C standard, and removing it completely can cause problems for programs which use the functionality in a legit way.

**Patching Rule-set.** Given some knowledge about DHCPd, the actual vulnerability and knowledge of how this can be exploited, we also use a specialized rule-set that logs execution that could be caused by an exploit, and then patches the program in run-time to make it immune to the exploit. The following rules are used;

log vsnprintf "pisp" if param-match 3 "%n"
log vsnprintf "pisp" if param-match 3 "%hn"
replace-param vsnprintf 3 ptr "call protected by plibc" ¬ if param-match 3 "%n"
replace-param vsnprintf 3 ptr "call protected by plibc" ¬ if param-match 3 "%hn"

**Example 5-4: Rule-set to Patch Program Execution**

**Experiment Execution and Results.** DHCPd was executed normally for each test case, and the publicly available proof-of-concept exploit was executed. We did not modify neither DHCPd nor the exploit.

Executing the test for the baseline case resulted, as expected, in injection of hostile code that could execute with super-user privileges. However, because the exploit was not modified for our environment, DHCPd crashed rather than execute the actual exploit code.
The plibc Experiment System

When using the denying rule set, DHCPd was terminated by PLIBC, and the program state was saved in a file (core dump). This was also an expected result as the “%n” and “%hn” directives are used in essentially all format string vulnerabilities, and these were analyzed for all vulnerable functions. DHCPd was terminated just before executing the vsnprintf function and no hostile code was ever executed. An investigation with a debugger of the saved core file reveals the chain of calls that lead to the point where the program would normally have been compromised. This information can often be used to trace the root problem in the vulnerable program. For a longer discussion on tracing the vulnerability from the core file, please see our original technical report [2].

Using the patching rule set, DHCPd continues to execute normally when attacked, but some messages sent to the system log are changed to "call protected by plibc". The reason for this is that the attack against DHCPd targets only logging functions and when PLIBC modifies the parameters sent to vsnprintf the function is no longer exploitable. Modifying parameters in this fashion is not always a good decision from a security perspective, for a discussion on this please see 7.1 Further Observations on Dynamic and Static Methods

5.5.9 Performance
To measure the performance of PLIBC, we performed three benchmarks, which was required because the extensive caching has an important role for the performance. Thus, we must determine;

- Time to execute a complete intercept - analysis - action rule
- Time to execute a cached rule

These values are obtained from results of a synthetic test where different rule-sets caused PLIBC to intercept different functions. The third measurement required to get a good view of real-world performance is;

- The number of calls that typically are made from a program to the C library.

This value is highly dependent on the program analyzed, however as an example, we benchmarked DHCPd to determine the number of interceptable calls that were made to service a single request from a client. Given this figure it is possible to determine the total overhead imposed by using the PLIBC system.

To perform the benchmark, a special policy that logged all calls to the C library was used. This policy was applied to DHCPd which ran on a dedicated network where 1000 requests were sent. The network was silent with exception of the DHCP traffic. The total number of calls recorded was 163950, including a small number of calls made when the DHCP daemon started up and shut down. The exact number of calls made for each request is not interesting in this case, so we did not separate calls made during start up and shut down from those made to serve requests, but conclude that on average 163950 / 1000 \( \approx \) 164 calls were made to handle a request.

**Synthetic Benchmark.** To determine the execution overhead imposed by using PLIBC, we designed a small program that repeatedly called four functions in the C library. Each function was called ten million times and execution time for the pro-
gram was recorded on two different computers running with and without PLIBC. The results are presented in Table 5-4.

**Table 5-4: Execution Time in Seconds Without PLIBC, and Using Empty Rules**

<table>
<thead>
<tr>
<th></th>
<th>Without PLIBC</th>
<th>With PLIBC</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium II, 400Mhz</td>
<td>49.04</td>
<td>50.59</td>
<td>1.55 (3.16%)</td>
</tr>
<tr>
<td>Pentium 4, 2.8Ghz</td>
<td>11.21</td>
<td>11.39</td>
<td>0.18 (1.61%)</td>
</tr>
</tbody>
</table>

In a sense, this is the worst-case scenario for PLIBC, as the program has essentially no logic but just calls functions in the C library. On the other hand, as no rules were loaded, the rule “allow” is used for all functions, resulting in PLIBC caching the calls after the first invocation. In regard to the caching, this represents the best possible case.

To determine the impact of the cache, and the time required to perform a full dynamic analysis, the same program was executed using four different execution policies. The first policy used a “nop” rule for one of the four functions, causing ten million of the forty million calls to be intercepted. The second rule used “nop” for two functions, the third for three, and so forth - resulting in 10, 20, 30 and 40 million complete interceptions and invocations of the dynamic analysis respectively. The execution time for this program using these respective policies are shown in Table 5-5.

**Table 5-5: Time in Seconds for Complete Interception and Dynamic Analysis**

<table>
<thead>
<tr>
<th></th>
<th>10 Million</th>
<th>20 Million</th>
<th>30 Million</th>
<th>40 Million</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium II, 400Mhz</td>
<td>123.06</td>
<td>202.01</td>
<td>277.68</td>
<td>372.59</td>
</tr>
<tr>
<td>Pentium 4, 2.8Ghz</td>
<td>27.03</td>
<td>46.95</td>
<td>67.15</td>
<td>89.71</td>
</tr>
</tbody>
</table>

**Analysis.** These results indicate that the overhead imposed by using PLIBC is linear to the number of non-cacheable function calls. Using the data from Table 5-4 and Table 5-5, we can also determine the average overhead imposed per call. This is difficult to determine on a per call basis as the values are so small the resolution of the real-time clock becomes an issue.

**Table 5-6: Average Overhead In Seconds Per Call**

<table>
<thead>
<tr>
<th></th>
<th>Cache Hit</th>
<th>Full Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium II, 400Mhz</td>
<td>$3.88 \times 10^{-8}$</td>
<td>$8.09 \times 10^{-6}$</td>
</tr>
<tr>
<td>Pentium 4, 2.8Ghz</td>
<td>$4.50 \times 10^{-9}$</td>
<td>$1.96 \times 10^{-6}$</td>
</tr>
</tbody>
</table>
From these results we make a number of conclusions. Firstly, that the average overhead for a full dynamic analysis is about 2 microseconds on a Pentium 4 and about 8 microseconds on a Pentium II. We also conclude that the overhead for cacheable calls are extremely small, and will most likely not be noticeable in a typical program.

For our final conclusion on the performance of PLIBC, we combine the results from the DHCPd experiment with those from the performance benchmark. On a modern computer, the Pentium 4, the overhead of analyzing every single function in the C library impose an overhead of 0.32 seconds for handling 1000 DHCP requests. Typically, only a very small minority of the functions have non-optimizable rule-sets, so the overhead should be even smaller. For many systems, this is quite a small overhead, making execution policies a good alternative for the protection of these systems.

5.5.10 Discussion

Applying an execution policy to a program is a domain-specific form of dynamic environment hardening (See 3.3 A Model for Execution Environment Hardening) This is true because the execution policy is dependent on the domain, and the sole purpose of using an execution policy is to apply the domain-specific protection. An execution policy can either be tailor-made to prevent a specific exploit from occurring in a program or, more commonly, to specify generic rules on how the system should behave when potentially dangerous operations are executed. Using a policy to prevent a specific exploit is typically used before a permanent fix for an exploit is available for the vulnerable program.

Using a more generic execution policy is applicable when standard programs are used in critical environments, or in environments that the program was not originally designed for. In this case, an administrator can define a set of rules based on the current environment where, for example, it might be better to terminate the program than to allow it to use the system in certain ways.

There are several forms of policy-based restrictions on program execution, which operate on different type of software. Most such systems operate from a lower level and enforce the policy on the program; this is for example the case for virtual machine or kernel-based methods. The PLIBC system is a pure execution environment based tool giving it one major disadvantage - it cannot enforce a policy on a hostile program (See Principle (I) on page 29). However, operating from the execution environment means that some forms of unwanted execution can be detected that would be difficult to detect from the kernel (See Principle (II) on page 29) and that the system operates with a comparably low overhead. In practice, this means that PLIBC can detect possible exploits such as format string attacks, unsafe uses of library functions, and parameters that are out of range. This makes the function of PLIBC more like that of some debugging tools (purify, bounds checker) than a typical execution policy tool.

Another related approach is to statically remove all unwanted functionality in a library, rather than using an execution policy. This is useful for functions that are inherently unsafe and never should be used (gets), but hard to generalize. Some
correct programs depend on using functions that other programs use in unsafe ways, and removing or modifying such functions might lead to unexpected behavior. An execution policy provide a good compromise when using a program in an environment it was not designed for, as the administrator can log programs actions, and develop a policy that fits that particular program in the specific environment where it executes.

In some cases, a more robust implementation of an interface can be used, and if that implementation does not impose too much overhead it can be used statically in a library. If the implementation has too much overhead or other consequences it can be combined with an execution policy that redirects call based on certain conditions.

For the particular domain of protecting against format string vulnerabilities, there have been some progress on using the C pre-processor to detect the number of parameters sent to a function, and match these in run-time with the format string [7]. This type of protection requires recompilation of all code that should be protected (since it uses the pre-processor) and, while it is believed that these modifications cohere to the ANSI C standard [16], some type of programs; typically being subjected to portability and compatibility issues make assumptions that no longer hold true given these modifications, and such programs will cause problems.

5.5.11 Protection Profile

Using the PLIBC system it is possible to perform scriptable dynamic analysis inside the execution environment of a non-cooperative program. Placing the protection inside the execution environment result in a comparably low overhead and a good source of information, but lack of enforceability. This makes execution environment policies suitable when adapting programs for use in new environments where it is possible to trade run-time performance for domain-specific control. As previously discussed, the function of PLIBC is similar to that of some debugging tools, but the system is tuned to have a lower overhead as it is used during normal execution and not during the debugging phase. The effectivity of PLIBC relies heavily on the quality of the policy and to which extent an interface can be monitored in run-time. For a longer discussion on the degree to which an interface can be monitored in runtime and our ideas of how this can be increased, please see 7.2 Hybrid Methods.

We summarize the principal protection offered by execution policies through the PLIBC system, by returning to the principles of protection and the relevance for each principle on the PLIBC experiment as described in Table 5-7.
The plibc Experiment System

<table>
<thead>
<tr>
<th>#</th>
<th>Summary of Principle</th>
<th>Relevance for Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Dynamic Methods that handle hostile code must rely on the operating system kernel for enforceability (See page 29.)</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>Some types of exploits can only easily be detected from inside the execution environment (See page 29.)</td>
<td>High</td>
</tr>
<tr>
<td>III</td>
<td>For simple interception, interception primitives alone are sufficient (See page 32.)</td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>A function can be intercepted just prior to the normal invocation to obtain the parameters (See page 32.)</td>
<td>High</td>
</tr>
<tr>
<td>V</td>
<td>Using only simple analysis on a large amount of functions is useful to prevent some types of exploits (See page 33.)</td>
<td>High</td>
</tr>
<tr>
<td>VI</td>
<td>Simple actions can be taken by invoking the operating system kernel on the process(es) (See page 34.)</td>
<td></td>
</tr>
<tr>
<td>VII</td>
<td>Complex actions can be taken by the tool modifying the execution environment (See page 35.)</td>
<td>High</td>
</tr>
<tr>
<td>VIII</td>
<td>Dynamic methods that operate inside the execution environment must operate before a program is exploited (See page 37.)</td>
<td>High</td>
</tr>
</tbody>
</table>

The possibility to observe potential exploits inside an execution environment is a key-mechanism for PLIBC, making Principle (II) highly relevant. It is by performing simple analysis on a vast amount of function calls, and using information from the execution environment that the system detect exploits - which is the core of principle (IV) and (V). The PLIBC tool exposes quite complex actions to an administrator, such as modifying parameters to a function, which is relevant for both protection Principle (VII) but also for Observation (IX) on page 25. Should there be a vulnerability in the PLIBC system or in the execution policy this can clearly harm the execution of a program.
5.6 Experiment 3: Immunizing Systems

The third, and final, experiment which the principles in this thesis are based on, is on hardening a system, e.g. to immunize an executing program from a class of buffer overflow exploits. We use the term immunization as described in Definition (II) on page 35, although in a somewhat speculative approach. In short we can prevent a class of exploits, and in some cases, this protection is sufficient to allow the program to resume execution.

Buffer overflow exploits are well-known, well-understood and still a very common type of exploit. While the cause of the underlying vulnerability can be discussed, buffer overflows are exploitable because of the lack of enforceable run-time validation of buffer sizes. This is the reason why buffer overflow exploits are common in programs written in the C or C++ programming language both which lack run-time validation of buffer sizes, and uncommon in programs written in other, typed, programming languages.

Many different approaches are used to make software more resilient to buffer overflow vulnerabilities; static approaches are primarily used to find underlying vulnerabilities in source code, and various dynamic approaches are used to detect and prevent the actual overflowing of buffers. The approach we are taking with the third experiment, SLIBC, is to immunize an executing program by preventing a class of buffer overflow exploits without depending on static analysis or other pre-generated information about the executing program.

As previously mentioned, buffer overflow exploits are only common in programs written in languages that don’t enforce buffer boundaries in run-time. While this include other languages than C and C++, we are only considering programs written in either of these languages. Programs written in C and C++ not only lack run-time enforcing of buffer boundaries, but also use little other run-time information. To immunize such programs against buffer overflow exploits, without requiring extensive analysis of program source, we make a few assumptions regarding enforcement of buffer bounds in construction of programs, as we will further describe. Also the SLIBC approach only protects against a class of buffer overflow exploits, that are caused indirectly by incorrect usage of library functions, as we describe in the next section.

5.6.1 Direct and Indirect Overflow Vulnerabilities

There are several ways to classify buffer overflow vulnerabilities and exploits. One classification that is important for our approach is that of direct and indirect overflows.

A direct overflow is, unsurprisingly, caused directly by a program, typically as the result of a logic error. For example consider the code fragment in Example 5-5, which contains a direct buffer overflow vulnerability;

```c
for(i=0; i <= (sizeof(buf)/sizeof(buf[0])); ++i)
    buf[i] = other[i];
```

Example 5-5: A Direct Buffer Overflow
Immunizing Systems

Executing this code will cause a direct overflow of the buf buffer because one element too much will be copied. This is so because the sizeof operator will return the size in bytes for the buffer buf, which divided by the size of the first element gives the number of elements in the buffer. The first element in a buffer is at position zero (buf[0]) and not one, so the last valid index is the number of elements of the buffer minus one. Using the less-than-or-equal operator, as above, causes one additional element to be stored in the buffer buf independently of the type and number of elements buf is declared to hold.

The actual number of bytes that will overflow the buffer is dependent on the type of buf. For example, if buf is declared to hold objects of the type char, one additional byte will be stored, and if buf is declared to hold int objects typically four bytes extra (sizeof int) will be stored.

This is a direct error, or vulnerability, in the program. The vulnerable code will be transformed into machine code instructions by the building tool chain and then executed on some hardware. The error is propagated from the source code through all steps in the tool chain and will finally be present in the machine code (See 2.1 Introduction) On most processors, the code fragment above will not call any helper functions, but the logic is transformed directly into machine code instructions that are placed inside the function where they are declared.

In contrast to the direct overflow, consider the code fragment in Example 5-6 that contains an indirect buffer overflow vulnerability;

```
char buf[10];
sprintf( buf, "%.10s", srcstr );
```

Example 5-6: An Indirect Buffer Overflow

When executing this code fragment, sprintf will format a string, as specified by the second argument, and store the result in the buffer pointed to by the first argument. The somewhat obscure notation of "%.10s" means that sprintf should consider the third argument (srcstr) as a pointer to a NULL-terminated string and copy no more than 10 characters from this string to the destination. Using this way to limit the number of bytes copied is common in older source code developed for UNIX-like systems. Executing this code will overflow the buffer buf unless the length of the string in srcstr is nine characters or less. If the string is longer than nine characters, sprintf will overflow the buffer buf by exactly one byte. This is true because ten characters will be copied into buf, as specified in the second parameter, and then a terminating NULL-character will be appended.

This is an indirect overflow, because the actual overflowing of the buffer is not done directly by the program, but indirectly by a library function. It is important to point out that the library function (sprintf) is not incorrectly implemented, but

---

1. This is typically done to make code portable to older UNIX(-like) systems that lack the snprintf function.
follows the standard, and instead it is the incorrect use of `sprintf` that causes the buffer to be overflowed.

Buffer overflows caused by incorrect usage of libraries are quite common, and typically cause large problems to executing software. When illustrating this type of vulnerability, the most common example is the `gets` function, for which there is no safe use. Library-based, or indirect, overflows are however caused by many other functions, although often as a result of incorrect usage of the respective function. See also Observation (VIII) on page 24 for a discussion on safe and unsafe uses of functions. Since many functions can be used in both safe and unsafe ways, it is not realistic to remove all these functions, but the protection must be based on the context of such functions.

5.6.2 Buffer Origins

Modern programs execute in complex execution environments that consist of many different parts, and where it is possible to map memory in many different ways. This makes it possible for a program to use very complex memory models, something that might be relevant for a data base engine or other programs that rely on special memory arrangements. However, that being said, most programs use the facilities provided by the programming language and run-time environment. For the discussion on overflow protection, we are only considering programs that are written in the C and C++ programming language and that use the standard memory facilities provided by the programming language and run-time environment.

The following three different pools of memory, or origins for buffers, are used by programs written in the C programming language;

1. Stack based - This includes local variables and parameters to a function.
2. Heap based - Memory allocated by `malloc` or some function that indirectly use `malloc`, such as the C++ `new` and `new[]` operators.

There have been several exploits against vulnerabilities in handling stack-based memory, in our opinion most likely because many architectures store the return address on the stack as well, making it comparably easy to modify the execution flow by overwriting data on the stack. However, depending on run-time conditions, memory from the heap and global buffers can also be used to change the execution flow of a program. We give one example of such a case in our experimentation with the TinyProxy program [34].

1. The keyword `static` is used for several different types of declarations in the C programming language; neither has any direct relation to static analysis.
5.7 The SLIBC Experiment System

The SLIBC experiment system protects an executing program from indirect buffer overflows originating from incorrect use of functions in the C library, and other libraries. The principal function of SLIBC can be divided into two main features;

1. Approximating the bounds of stack- and heap- based memory buffers.
2. Enforcing the maximum buffer size restrictions of stack- and heap- based buffers at run-time.

This functionality is provided by SLIBC by means of intercepting mainly low-level memory copying operations and approximate the size of buffers prior to copying the memory. Because of how memory copying is handled inside the C library, this means that essentially all functions in the C library are protected, and since many other libraries rely on these primitives for copying memory, functions in these libraries are protected as well. The validation is done prior to copying any data, which means that violations are detected prior to memory being overwritten. Because of this, the root cause of the violation, the vulnerability, can be traced as we describe in section 5.7.6 Experimentation.

5.7.1 Buffer Size Approximation

To prevent memory violations in run-time, SLIBC determines the size of an allocated buffer to make sure a program does not write more bytes to a buffer than the buffer safely can hold. It is not possible to determine the exact size of an arbitrary buffer at run-time, because there is simply not enough information available. Instead, SLIBC approximates the maximum size of a buffer based on information in the execution environment and compiler artifacts present in the program. Consequently, by means of buffer size approximation and run-time information hereof, it is possible to effectively deal with overwriting of crucial data, e.g. return addresses as well as to prevent data written in one dynamically allocated buffer to overflow into another adjacent buffer.

Different memory regions, such as the stack and the heap, have different layouts and by means of determining the origin of a pointer, SLIBC can use artifacts specific to the particular region to determine the maximum size for a buffer. The first step when analyzing a pointer to a buffer is to determine from which region the buffer originates. The following four pointer types are applicable for such a procedure;

1. Stack-based - This class matches pointers that refer to buffers allocated on the stack, which typically includes variables local to the currently executing function, or local to a function higher up in the call chain.
2. Heap-based - Pointers to a buffer allocated directly or indirectly through the malloc function. Indirect uses of malloc typically involves the new or new[] operator in C++, or objects allocated automatically by a library for a program.
3. NULL Pointer - This class matches pointers referring the address zero.
4. Unknown - Pointers to buffers that do not match any of the previously mentioned classes. This class of pointers typically includes global or static variables, function pointers and pointers to buffers inside mmap-ed objects.
NULL pointers are classified as valid and of the size zero. The address zero is by definition not valid, so any attempt to dereference such a pointer will cause a program to crash. Still it is legal to copy zero bytes to this address. While the need for our null pointer might seem a bit awkward, it effectively deals with some unusual, but correct, program behavior like `memcpy(0, 0, 0)`.

Finally, the pointer type of unknown buffers are assumed to be valid and unbounded, as there is no information about these available. This is how all pointers are handled without SLIBC. Stack and heap based pointers, which make up the vast majority of analyzed pointers, are further investigated as described below to approximate their respective maximum size.

### 5.7.2 Dynamic Analysis for Stack-Based Buffers

The layout of the stack is dependent in particular on the computer architecture (typically the CPU) and the compiler. For programs written in the C programming language and compiled for the IA32 architecture (pc) return addresses and saved frame pointers are stored on the stack. The memory layout of the stack is important for many exploits that overwrite the return address, but also makes it possible to expand the frame pointers to determine which part of the stack is used by which function. This is the same fundamental approach as used by the well-known libsafe library and similar tools [6]. For example, many debuggers use this information to extract information from the core dump of a crashed program.

![Figure 5-7: Stored Frame Pointers as a Linked List of Memory Regions](image)

By expanding the list of frame pointers, it is possible to determine the total size of local variables used by a function, but not the size of each individual buffer. Also, it is not possible to determine the size for variables in some specially optimized functions. Such limitations apply to all tools that rely on analysis of frame pointer chains. In this context it is however possible to determine the maximum size so that the return address is not overwritten. Also, because it is known that the buffer originated from the stack a more conservative decision can be taken when an attempted overwrite is detected.

### 5.7.3 Dynamic Analysis of Heap-Based Buffers

For dynamic memory, the `malloc` function divides allocation of objects into two main categories; small and large objects. An object is large if it occupies more than half a memory page, meaning more than 2Kb on IA32. Large objects are allocated from a heap of raw memory pages, resulting in the size of each object effectively being rounded up to 4 Kb. The size of small objects are rounded up to the closest power of two that is larger than or equal to 16 and smaller than or equal to 2048, and allocated from a special sub-heap that only hold objects of
The slibc Experiment System

this particular size. Each sub-heap allocates 4Kb pages from the heap of large objects when full.

Because each sub-heap hold objects whose size is rounded up to a power of two, and the page size is a power of two, each page will contain a number of objects that is also a power of two. For example, each page in the sub-heap for 16 byte objects contains \(2^{12}/2^4 = 2^8 = 256\) objects. Placing objects in this way has several advantages; each object is aligned in memory which enables fast access, and it is easy to use a bitmap to represent which slots on the page that are used and which are free. This bitmap is kept separate from the data in the FreeBSD malloc. The disadvantage of using this layout from a SLIBC perspective is that there are no unused bytes on a page, so unlike stack-based buffers there is no single byte for SLIBC to consider invalid and to trigger on. For a detailed discussion on heap functions in FreeBSD please see [27].

The SLIBC tool uses meta-data already present in the C library to determine the size of a buffer, and used by the normal heap functions, rather than relying on special memory objects that should never be referred. The following procedure is applied to determine the size of a heap-based pointer;

1. Determine the base of, and offset into, the allocated object that the pointer refers to.
2. Locate which sub-heap the buffer was allocated from.\(^1\)
3. Verify that the object is marked as being "in use".\(^2\)
4. The offset into the buffer is subtracted from the buffers size and this value is considered the safe buffer size.

This procedure is similar to the normal C library implementation of free where step 3 is very similar to the built-in support against "double free" attacks\(^3\). However, because SLIBC is invoked for all memory copying operations incorrect use of already-freed pointers are detected at the first attempt to use the buffer. We consider this a positive side-effect of using SLIBC.

The separation of buffer and offset in step 1 and 4 differ significantly from the normal implementation of free. The reason for this difference is because it is valid to perform memory operations on pointers inside a memory buffer, but it is not valid to call free with such a value. Consider the program in Example 5-7;

\[\frac{212}{28} = 256\]

---

1. Knowing which sub-heap gives the size for small objects. malloc saves special information about allocated pages for large objects which SLIBC uses in the same way.
2. Objects that are not in use have not been allocated (bogus pointers that just happen to point inside the heap region) or, more likely, have been free()-d. The default action for SLIBC is to terminate the program if it finds such pointers.
3. When an allocated buffer is released by calling free more than once.
Enforcing Buffer Bounds

Example 5-7: Heap Safe Sizes and sprintf

The first memcpy has the base of ptr and the offset of zero, meaning that it refers to the beginning of a buffer. The second memcpy has the same base, but an offset of four. In this case it is essential to first separate the offset from the base to determine which buffer the pointer refers to, and when the size of this buffer has been determined (16) subtract the offset to get the remaining number of valid bytes (12).

Figure 5-8: Using Heap Information to Determine the Safe Size of a Buffer

We are aware of two shortcomings with this method. The first is the granularity of the protection; when a program dynamically allocates a structure only the heap block is protected, not the members of the structure. This is problematic to solve because there is no way for a program to communicate which type of data it allocates memory for, and hence no way for malloc to record this information or for SLIBC to use it. For a discussion on this, please see 7.2 Hybrid Methods.

The second shortcoming concerns cases when malloc is called twice with the same size and two adjacent memory areas are returned. In this case it is valid to treat these two blocks of memory as a single large block and perform operations on it as such. When using SLIBC this will not be permitted. As there is no way to guarantee that malloc will behave in this way, and considering that most malloc implementations mix data with meta data which completely eliminates the possibility of this case, this behavior is highly unlikely in a program. We have not found a single program that behaves in this way. We are currently not aware of a good way to remove this limitation.

5.7.4 Enforcing Buffer Bounds

For SLIBC to enforce functions in the C library to respect the size of a buffer, it relies on intercepting relevant function calls at run-time and compare parameters sent to these functions to the approximated size of a buffer. The normal C library in FreeBSD provides about 1050 functions, although not all handle memory buffers. Re-implementing all functions that in some way handle memory buffers

```c
int main( int argc, char **argv ) {
    char *ptr = (char*)malloc(16);
    memcpy( ptr, "hello", 5 ); /* Safe size is 16 */
    memcpy( ptr+4, "world", 5 ); /* Safe size is 16-4 */
    return 0;
}
```
The slibc Experiment System

would be ineffective and problematic to maintain. Protecting just the most common functions would be easy to maintain, but this would leave many functions completely without protection, which is not a good decision from a security perspective.

Rather than re-implementing all such functions, our tool SLIBC relies on how memory copying and zeroing is handled inside the C library. There are three functions in the C library for copying memory (memcpy, memmove and bcopy) which essentially provide the same functionality. Similarly, there are two functions for writing a byte-pattern to a memory buffer (bzero and memset) which, for the most common byte pattern, zero, provide the same functionality. These functions are exposed by the C library so that programs can use them, but, more importantly, they are also used internally for essentially all memory copying and zeroing operations. The SLIBC tool enforces buffer size restrictions by providing an alternative implementation of these and a small number of other key-functions, such as gets, inside the C library.

Many buffer overflows are caused by incorrect use of functions other than simple memory copying and zeroing. Functions that are often incorrectly used are those for string formatting (printf family) or error logging functions (syslog). The SLIBC tool protects against incorrect use of these and other library functions as the vast majority of functions in the C library use one of the simple copying functions to copy actual bytes to the user-provided buffer, which is the cause of the overflow. Consider the program shown in Figure 5-9;

```c
int main( int argc, char **argv ) {
    char ptr[SIZE];
    sprintf( ptr, "hello, %s\n", "world" );
    return 0;
}

memcpy(ptr+0, "hello, ", 7 );
memcpy(ptr+7, "world", 5 );
memcpy(ptr+12, "\n\0", 2 );
```

Figure 5-9: Expanded sprintf and memory operations

The call to sprintf will expand to three calls to memcpy, the first copying "hello, ", the second "world" and the third the new-line and string terminating NULL character. Using SLIBC all three memcpy calls will verify the size of their respective output buffer before copying any bytes.

5.7.5 Challenges

There were a number of challenges we faced when designing the SLIBC experiment system. Below we describe, what we believe, the most important such challenges and how they were handled;

**Primitive Memory Functions and Generalizability.** The construction of SLIBC relies on the fundamental assumption that memory is copied and zeroed through a number of functions, that we have referred to as low-level memory functions.
Some of these functions are defined in the ANSI C90 [16] and newer standards (\texttt{memcpy}, \texttt{memmove}, ...), while others have a 4.2BSD heritage (\texttt{bcopy}, \texttt{bzero}). While the behavior of the ANSI-functions are not identical to the BSD counterpart in regard to some base cases, they are implemented by the same function in the FreeBSD C library\(^1\). For this reason, the implementation in the existing C library can only make assumptions that hold for all standardized interfaces that map to a particular function. This is, for example, the reason that \texttt{memcpy} handles overlapped memory in the same way as \texttt{memmove}, where the standard only requires the latter function to correctly handle this case, and allows the first yield an undefined result. This is relevant for SLIBC, because in reality there exists only one function to copy memory, and one function to write a byte-pattern to a buffer. The different standardized functions are implemented as variants of these two basic functions.

The C Library internally relies heavily on the memory copying and writing functions. For example the \texttt{printf} family of functions first format a string in a \texttt{STDIO} buffer and then, for the functions where this is relevant, copies this string to the user-provided buffer using the memory copying function. Essentially all other functions in the C library that copies memory or writes particular byte patterns to a memory buffer rely on these primitive functions.

The enforcing mechanism in SLIBC relies on the assumption that these low-level functions are used for copying memory and writing byte patterns to memory buffers. Using SLIBC these functions first validate each memory access, and then perform the requested operations. A few functions in the C library use other functions for these tasks, typically because requirements to copy byte-by-byte (\texttt{gets}) or because of some special optimizations in copying C style \texttt{NULL}-terminated strings. For this reason, SLIBC implements a small number of other functions as well.

The FreeBSD C library provides a good ground for experiments like our SLIBC system. The source code is clean, easy to understand and, which is important for our experiment, uses primitive functions heavily. The FreeBSD C library already provides many features that we believe are good from a security perspective, such as the forgiving implementation of \texttt{memcpy} previously mentioned. The SLIBC system would probably not be as effective if implemented over some other library that was not organized in this way.

There is a challenge when designing dynamic methods, because, in order for the method to function effectively, we must make some assumptions on the execution environment. In this case, the assumptions are on the construction of the C library, which makes it possible to build the system with comparably low requirements on the program being protected. It does, however, also mean that a system like SLIBC cannot be moved to a different environment without analyzing that environment and adjusting the method to function in the new environment.

\(^1\) Technically, this is done through the C pre-processor which duplicates the logic inside each respective function, but adjusts for differences in calling interfaces. Most likely this is done to avoid the overhead of an additional function call. This has no practical relevance for the SLIBC approach.
Optimization of Primitive Memory Functions. The different standardized functions that copy memory and write a byte pattern to buffers are essential for the correct operation of the SLIBC library. These functions are also candidates for automatic compiler inlining, meaning that the compiler, under some circumstances, will remove calls to the respective function and replace that call with machine code that performs the requested operation. Such code will, under some circumstances, execute faster than code that calls the actual function, because the process of calling a function takes considerable time in comparison to writing a few bytes in a memory buffer.

Typically, memory copying that does not handle overlapping buffers, and that is used with a constant size argument is a candidate for this type of optimization, but other optimizations are also possible.

The function of the SLIBC system is dependent on modifying functions that copy memory, and for this reason is such optimization problematic. Calls that are replaced (or inlined) by the compiler will not call any memory copying function, and hence will not be protected by the library. Fortunately, most calls are not optimized and there are two ways to handle optimized calls:

1. Re-compile the program with relevant compiler options
2. Statically analyze the executable binary for inlined functions, and modify these

The first approach is by far the simplest, as many compilers can be instructed not to perform this particular optimization. Requiring a re-compilation of all components in a complex system is, however, sub-optimal from our perspective. For some components, a complete recompilation might be possible, which will result in better protection for these components. If the components are re-compiled, additional compiler-based protection schemes, such as canary values [8], can be included as well.

The second approach would require analysis of the executable binary and a mechanism to identify the machine code that has replaced the function call. Because of how compilers work, the function call will typically be replaced with what is called a compiler idiom, that can easily be identified using traditional decompiler techniques [23]. Once the idiom is identified, it can be replaced with a mechanism to call the corresponding low-level function or otherwise invoke relevant dynamic protection mechanisms.

Optimization of Function Prologues. The second type of compiler optimization that is problematic from a SLIBC perspective is that of optimizing function prologues so that no frame pointer is used.

As described in section 5.7.2 Dynamic Analysis for Stack-Based Buffers, SLIBC rely on the structure of the frame pointers to determine which region of stack memory that belongs to which function, and from this determine zones that are illegal to write into. When compiling a program without automatic frame pointers, this type of analysis is not possible, and functions compiled using this option will hence not be protected.
Challenges

This type of optimization is normally not used, since in addition to making SLIBC type of analysis impossible, it also makes debugging of programs more difficult. The main reason for this is that if the program crashes, a debugger cannot determine which function the program crashed in. This type of optimization is, however, sometimes used to deliberately make it harder to analyze the execution of a program.

Performance. The design of SLIBC, where each primitive memory copying operation is validated, often results in several validations for each function call. For example, consider the call to sprintf, as illustrated in Figure 5-9 on page 74; this single call is expanded to three calls to memcpy, each being validated by SLIBC. Operating on the lower-level memory copying functions is fundamental for the enforceability of SLIBC, but one might expect significant performance penalties as a direct result of this design.

Performance is always an issue when designing dynamic protection, and the SLIBC system was designed with this in mind. We have executed a performance benchmark on the SLIBC system, which we present in the next section 5.7.6 Experimentation. Before we present these results, however, we want to point out that other factors than just the number of interceptions done affect the overall performance. Some SLIBC design decisions were made specifically to lower the total overhead imposed by using the SLIBC system.

For analysis of stack-based buffers, SLIBC expands the frame pointer chain to determine safe and unsafe regions of memory. It is possible to perform this expansion by using extensions to the C language found in the GNU C compiler, which some tools rely on. This extension, however, requires a constant parameter determining the depth of the stack, making high-level language constructions such as a loop impossible. Instead, a repeated conditional must be used causing such code to be large and slow. Also, the extension is somewhat unreliable, as using a too high value for the stack depth typically causes the generated code to crash the entire program.

Rather than relying on this extension, SLIBC obtains this data without assistance from the compiler, which requires an assembly language implementation specific for each platform that is supported (currently only one; IA32). This implementation does, however, not need to comply with the restrictions above, but can be highly optimized. While manually optimizing machine code is not always an effective way to spend developer time, this particular function is involved for every validation of stack-based buffers, so in our opinion performance is critical. The current frame pointer expansion function in SLIBC uses less than 20 machine code instructions, and could probably be even further optimized, if needed.

The SLIBC system uses only existing meta-data that describe the layout of the heap. For this reason, the implementation of the malloc and free functions are identical to their respective original implementation, and using SLIBC will not introduce an overhead in either of these functions. Also, there is no overhead imposed by, for example, misalignment problems regarding heap layout, as it is identical when executing with or without SLIBC. A typical reason for ineffective use of the heap and misalignment is the use of “magic cookies” (canaries) or debug data on
The slibc Experiment System

the heap to detect overruns, which we further discuss in the section 5.7.8 Comparisons.

Validating access to memory from the heap require more analysis than for stack-based buffers. This is true because the structure of the heap meta-data is more complex than just expanding a linked list\(^1\). The data is, however, organized quite effectively, but we don’t believe there is as much room for optimization as for stack-based buffers.

5.7.6 Experimentation

We designed an experiment to investigate the behavior of slibc on real-world programs. Firstly, we wanted to determine the effectiveness of the protection without any program-specific tuning, and secondly we wanted to determine the quality of service offered by using such a protection mechanism. In short we wanted to investigate the usefulness of slibc by examining a trace produced by the tool just before the supervised program normally would be exploited.

We used the publicly available programs “TinyProxy” (version 1.3.3) and the “htpasswd” utility program included in the popular Apache web server (version 1.3.31) as test cases \([34, 33]\). These test cases were selected because they have quite typical vulnerabilities and because this is the type of vulnerability slibc should be effective against. Tinyproxy is known to contain a heap-smash vulnerability because of incorrect use of sprintf when reporting errors back to the client \([36]\). This can in some cases result in code injection in the program. The htpasswd utility contains a quite typical stack smash vulnerability as it can be tricked into using strcpy on a user-provided input string, without checking the length of the string \([37]\). This can also be exploited to inject and execute arbitrary code in the program.

We executed both test cases on a clean FreeBSD system, first as a base-line test without using slibc, and then using slibc protection. We used identical binaries for each test case, and no modifications to the test system were done, except the addition of slibc. Each test case was given special, hostile, input that is publicly available, in an attempt to exploit certain vulnerabilities. We did not specifically adapt the input for the execution environment, which means that a vulnerable program was expected to crash rather than execute the input code.

In the baseline test, without slibc, both programs crashed when given hostile input. Using slibc neither program crashed, but both produced a diagnostic message.

**Analysis.** In the baseline test case both programs crashed, which we expected, as we had not modified the return-address from the publicly available exploit strings. A closer examination of the exploits and core dump files from the test cases reveals that both test cases were indeed exploited by the buffer overflow.

Using the slibc test case, both programs resumed execution given the hostile input string. As both programs contain quite typical heap and stack smashes, we

1. Which, in fact, is the structure of the frame pointer chain (See Figure 5-7 on page 71)
were not particularly surprised to find that the environment made the execution environment immune. To further investigate the run-time situation that caused the baseline test to crash, we investigated the output from SLIBC to trace the root cause of the vulnerability. The following output was produced when executing htpasswd:

```
strcpy: size of dest < size of src
   ptr: '0xbfbfe8ec'
   dest size: '280'
   src len: '288'
return_address[0]: '0x2808221b'
return_address[1]: '0x8049637'
return_address[2]: '0x8048bd5'
return_address[3]: '0x4'
```

Figure 5-10: SLIBC Trace Message

From this output we can make a number of observations on the vulnerability that made the exploit possible.

1. The problem was reported from the function `strcpy` meaning that the program attempted to copy a NULL-terminated string.
2. The pointer with such a high address (`0xbfbfe8ec`) refers to a stack-based buffer. Resolving the call stack in a debugger reveals that `0x2808221b` is in `strcpy` and that `0x8049637` is in `main`.
3. Finally we observe that the pointer referred to a variable 280 bytes from the beginning of an activation frame.

Given this information we know that somewhere in `main` there is a direct call to `strcpy`. It must be a direct call because there are no activation frames between `main` and `strcpy`. We also know that the destination buffer is about 280 bytes from the beginning of an activation frame. We don’t know the exact address because the pointer could be the result of pointer arithmetic, and we don’t know from which activation frame the memory is allocated, i.e in which function the variable is declared. As the call is made from `main` it is however most likely that the buffer is also allocated from `main`.

Analyzing the source code to htpasswd we see that there are four places where calls to `strcpy` can be done. It is possible to determine which call is vulnerable by disassembling the function `main` and see which of the calls that return at address `0x2808221b`. However, in this case that is not necessary; each call to `strcpy` from `main` has a unique destination buffer associated with it, so by looking at the variables in `main` and determining which one is placed 280 bytes from the beginning, we know which destination buffer was attacked, and hence can identify which call is vulnerable. It turns out to be the third variable in `main`, `char user[MAX_STRING_LEN]` that is located on this address, and hence we can deduce that the vulnerable call is `strcpy(user, argv[i+1])` on line 420 in `htpasswd.c`. 
The slibc Experiment System

5.7.7 Performance

We used a micro benchmark to determine the average overhead imposed per function call when using the SLIBC system. The benchmark repeatedly called the `sprintf` function to build a string in a heap-based memory buffer. The principal code for the benchmark is shown in Example 5-8.

```c
int n = 10000000;
char *buf = (char*)malloc( 16 );
while( n-- ) sprintf( buf, "hello, %s", "world" );
```

Example 5-8: Micro-benchmark for SLIBC Performance Test

Each call to `sprintf` results in three calls to `memcpy` being made from the C library\(^1\), but the overhead was measured per call to `sprintf`, not `memcpy`. We executed the test on three different computers; a Pentium II, a Pentium III and a Pentium 4. On the Pentium 4 we also rebuilt the entire operating system environment including all libraries with a compiler option that disables any automatic inlining to determine the overhead when even the simplest memory operations were validated by SLIBC. For each computer we measured the execution time with and without SLIBC, and the results are shown in the table below.

**Table 5-8: Execution Time and Overhead**

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Baseline</th>
<th>SLIBC</th>
<th>Overhead per Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium II (266Mhz)</td>
<td>325.8613s</td>
<td>478.1960s</td>
<td>1.52(\mu s)</td>
</tr>
<tr>
<td>Pentium III (1Ghz)</td>
<td>73.9152s</td>
<td>112.6937s</td>
<td>0.39(\mu s)</td>
</tr>
<tr>
<td>Pentium 4 (2.8Ghz)</td>
<td>39.6451s</td>
<td>49.2588s</td>
<td>0.10(\mu s)</td>
</tr>
<tr>
<td>Pentium 4 without builtins (2.8Ghz)</td>
<td>39.0743s</td>
<td>48.1188s</td>
<td>0.09(\mu s)</td>
</tr>
</tbody>
</table>

From this simple benchmark we find that the overhead per call ranges from about 0.09\(\mu s\) to 1.52\(\mu s\) depending on computer configuration. The relative overhead per call is lower on faster computers which we interpret as the SLIBC intervention is a less CPU-intensive operation than the actual copying of bytes. Because of the low overhead per call the accuracy of the real-time clock is not very good, which was the reason why we measured a loop of ten million iterations. It is possible that the highly speculative Pentium 4 CPU was able to optimize execution because of the high number of iterations. We were surprised to see that the Pentium 4 executed both the baseline and SLIBC test case faster without inlining optimizations, and we don’t have an explanation for why this is the case.

\(^1\) See Figure 5-9 on page 74 for an example of how `sprintf` uses `memcpy`
Because of the low overhead per call we did not perform any performance macro benchmark, as it is likely that the results would be disturbed by noise.

5.7.8 Comparisons

Before the general discussion on SLIBC and other dynamic methods that make exploiting buffer overflows hard, we provide a comparison between these different methods and the respective relation to SLIBC. In this comparison we mainly consider different implementations, and not the underlying methodology of each respective tool.

Libsafe and other Stack-walkers. SLIBC uses frame pointers in the same principal way as libsafe and debuggers to determine which part of the stack that belongs to which function [6]. From a technical perspective, the implementation in SLIBC is somewhat different than that in many other tools, because of the high performance requirements, as discussed in section “Performance” on page 77.

The SLIBC tool validates low-level memory copying operations, and some especially problematic functions in the C library. Because of this, many functions local to a program will be validated as they internally use these low-level copying operations. Also, because of features in the FreeBSD C library, complex functions in the C library, such as printf always call low-level memory copying functions, which means that SLIBC can protect programs from indirect overflows caused by these functions. This is an important feature as incorrect usage of string formatting functions is a common cause of exploits.

Beside protecting stack-based buffers, SLIBC protect heap-based buffers and other types of run-time instrumentation of program execution as well. This is done by using existing meta-data from the heap functions, and because there is more run-time data available, this protection is more fine-grained than the protection of stack-based buffers. This protection is not traditionally offered in stack-walkers.

Canary Values. One approach to detect buffer overflows is to place a cookie value, or canary value, at the end of the buffer and at the right time assert that this value has not been modified [8, 26]. Some tools use this approach to detect stack-based overflows at run-time when a function return, to assert that the return address and frame pointer has not been overwritten. This approach is also commonly used by debug tools to help programmers detect incorrect memory use when debugging their programs.

There are some important differences between SLIBC and the canary value approach. Most canary value approaches require the program to be re-compiled to take advantage of the protection. This is particularly true for methods that protect the stack. The SLIBC tool operates on software without requiring the software to be recompiled. Canary values can also be used to protect the heap, and this can be done without recompiling protected programs. In this case, however, it is not obvious where to place the canary value validation. Also, using canary values on the heap makes it more difficult to use optimized and simple allocation algorithms, like the one used in the FreeBSD malloc.
When detecting a destroyed canary value, there is no way to determine the exact point in the system where the canary was destroyed, or which buffer that was used. SLIBC validates memory copying operations, prior to copying the memory. This means that the exact point in the program that performs the operation is always known, and that the original buffer often can be determined. This information is logged by SLIBC and useful in tracing the original vulnerability. This is especially useful when tracing hard-to-find vulnerabilities that only occur under unusual run-time conditions.

Stack-based canary values can be used to detect off-by-one type of logic errors in a program. This type of vulnerability cannot be identified by SLIBC. Also, the SLIBC approach imposes an overhead on all memory copying operations, which is not the case when using canary values. However, using canary values on the heap, require extensive modification to the heap structure and that either current algorithms are modified or that the memory is used less effectively.

Canary value approaches cannot protect against some types of indirect overflows where one buffer is overflowed into an adjacent pointer that is used for a second overflow, for example as described in [25]. The SLIBC approach can protect against such exploits under some circumstances; when each low-level memory operation is verified. This type of vulnerability is however not yet as common as traditional buffer overflows that both canary values and SLIBC provide a good protection against.

5.7.9 Discussion

Buffer overflow vulnerabilities is a serious problem, if and only if, these vulnerabilities can be exploited. This is the fundamental approach behind all dynamic methods that deal with buffer overflows. This is a slight simplification, since no tool can entirely protect a program from all types of overflow exploits, so finding and removing the underlying vulnerability also is valuable from a security perspective.

Some tools protect a program from overflow exploits by using information about some particular underlying structure, such as the layout of the stack. In the approach we have described for this experiment, we take this one step further. Using information about the execution environment, both in terms of stack and heap layout as well as using information about the internal structure of the FreeBSD C library, it is possible to extend the dynamic protection. Fundamentally, we rely on the FreeBSD C library to use its own functions for memory copying internally, which is the primary interception step for SLIBC. This is also the essence of Observation (I) on page 19, as the enforceability of the method relies in this design.

Also, Observation (VIII) on page 24 about safe and unsafe uses of functions is relevant when discussing dynamic protection against buffer overflows, and especially the SLIBC approach. This is so because while some methods, for example canary values, protect against buffer overflows by detecting the overflow after it has occurred, our approach protects the function just before the memory is overwritten. Hence an indirect buffer overflow is essentially an unsafe use of a memory copying, or zeroing, function.
Attacking the problem at this level means that some types of overflows, which cannot be detected using canary values, such as double overflows, are detected. It also means that the overflow is detected prior to any memory being overwritten. This is very different from a canary value approach, or any other approach that detects the exploit after the memory has been overwritten, but before the exploit has executed. This is so because in the SLIBC case the execution environment is not damaged when the overflow is detected. We have already discussed the advantage of this in regard to tracing the vulnerability, but there is one additional important advantage; speculative immunization of a system.

**Speculative Immunization Against Buffer Overflows.** When detecting an overflow, it is possible to determine the origin of the buffer. This gives the granularity of the size approximation. For example, in the current SLIBC implementation the granularity is finer for heap-based than for stack-based memory buffers. In some cases it would be possible to get a better granularity of stack-based buffers, by using a static pre-analysis phase. However, just knowing the granularity is sufficient for the speculative approach.

In this situation, when the granularity of the protection is known, and all run-time information is available, a dynamic tool can take one of several actions. First, it can choose whether to write logging information or not, but, more importantly, the final action on terminating the program or allowing the program to resume execution can be made based on this information and some local policy.

The simplest action is that of terminating the program. This is also the only action that reliably can be taken if the overflow is detected after the memory has been overwritten, as is the case for canary approaches. Since this is not the case with SLIBC, but the execution environment is assumed to be intact, the tool can attempt to speculatively immunize the program.

Speculative immunization in this case would mean to modify how many bytes is copied into the destination buffer (for primitive memory operations), the destination string for direct string manipulation buffers, et cetera. After this is done, the dynamic tool would allow the program to resume execution. This is a speculative approach, because it protects programs that overflow buffers for which we have perfect size approximation, but causes for the dynamic tool, unpredictable modification to the execution for other programs. We will return to the concept of speculative immunization as a general discussion for dynamic protection tools in 7.1 Further Observations on Dynamic and Static Methods.

**5.7.10 Protection Profile**

The method we have presented, demonstrated through the SLIBC tool, provides protection for a class of exploits caused by buffer overflows that originate from incorrect usage of certain functions, a class we refer to as indirect buffer overflows. This protection can be applied to a program without recompiling or modifying the program, but the efficiency of the method is dependent on the construction of the C library used by the program.
The slibc Experiment System

We summarize the principal protection offered by our method and the SLIBC system, by returning to the principles of protection and the relevance for each principle on the SLIBC experiment as described in Table 5-9.

**Table 5-9: Protection Principles for SLIBC**

<table>
<thead>
<tr>
<th>#</th>
<th>Summary of Principle</th>
<th>Relevance for Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Dynamic Methods that handle hostile code must rely on the operating system kernel for enforceability (See page 29.)</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>Some types of exploits can only easily be detected from inside the execution environment (See page 29.)</td>
<td>High</td>
</tr>
<tr>
<td>III</td>
<td>For simple interception, interception primitives alone are sufficient (See page 32.)</td>
<td>High</td>
</tr>
<tr>
<td>IV</td>
<td>A function can be intercepted just prior to the normal invocation to obtain the parameters (See page 32.)</td>
<td>High</td>
</tr>
<tr>
<td>V</td>
<td>Using only simple analysis on a large amount of functions is useful to prevent some types of exploits (See page 33.)</td>
<td></td>
</tr>
<tr>
<td>VI</td>
<td>Simple actions can be taken by invoking the operating system kernel on the process(es) (See page 34.)</td>
<td></td>
</tr>
<tr>
<td>VII</td>
<td>Complex actions can be taken by the tool modifying the execution environment (See page 35.)</td>
<td>High</td>
</tr>
<tr>
<td>VIII</td>
<td>Dynamic methods that operate inside the execution environment must operate before a program is exploited (See page 37.)</td>
<td>High</td>
</tr>
</tbody>
</table>

The method we have described is highly dependent on analyzing sub-sets of the execution environment and interception that is only possible from inside the execution environment. It would not be possible to address this class of exploits from the kernel. We consider the interception in SLIBC to be fairly simple because only a few functions are actually intercepted, and these are replaced with one-to-one matching functions inside SLIBC. The fact that these functions are called from inside the C library and that this is the key mechanism for protection, does not change our classification.
5.8 Chapter Summary

We have described three different dynamic methods that all improve program execution security, but where each method operates at a different level of the execution environment. For each method we have built an experiment system and performed practical experimentation.

Firstly we described the LPS system which is designed to reduce security consequences for a system, should a component be compromised. This is done by separating privileges in a program not originally designed for this, and transparently executing unreliable components with lesser privilege. When such a component has finished executing the results are transferred back to the privileged part in a controlled fashion. The second experiment system, PLIBC, is used to detect unwanted execution inside an execution environment. This is done by means of applying a non kernel-enforced execution policy on the program which in addition to detecting unwanted execution also can be used to dynamically modify an execution environment should unwanted execution be detected. SLIBC, the third experiment system, prevents a class of library-based, or indirect, buffer overflow exploits. This is done by analyzing memory copy operations and validating each operation against existing meta-data to separate safe from unsafe use.

All three experiment systems are designed to operate on existing programs requiring only minimal or no modification. The LPS system is integrated by recompiling a program, but requires only minimal modifications to the existing source code. The SLIBC and PLIBC systems operate on binary programs without requiring any modification, or recompilation, although the level of protection varies depending on some compilation options.

In the practical experimentation we saw that the overhead imposed by using dynamic protection on a modern CPU ranges from about 5 ms for privilege separation to about $0.1\mu$s for buffer overflow protection. In the next chapter we provide a discussion on these results and the relation to protection principles and earlier observations.
Chapter Summary
6.1 Classification Strategies

Early in this thesis, we provided a number of classifications of protection models and methods which we used to categorize the function and level of protection offered.

**Static and Dynamic Methods.** The first classification, which is fundamental for our work, was first introduced in chapter 2. Methodology. Here we defined the concepts “static” and “dynamic” methods where the latter refers to a method that focus on software execution, and the former to a method that is used when analyzing a source (typically source code) for a particular program (see 2.1 Introduction.)

**Execution Environment and Model.** Following that, in chapter 3. Model we defined the concept of an execution environment (See Definition (I) on page 18.), described our model for execution environment behavior, and provided a classification of environment hardening methods. Using this classification, we separate between;

- Static Environment Hardening (that do not consider actual execution), and
- Dynamic Environment Hardening (that do consider actual execution).

We then further divided the class of dynamic environment hardening methods into (See 3.3 A Model for Execution Environment Hardening);

- Generic (methods that do not have specific knowledge about the program being protected), and
- Domain-specific (that do have such specific knowledge about the program being protected).

It is important to stress that all environment hardening methods, including those we classify as static, by definition constitute dynamic methods. This is true because environment hardening methods always target the execution of software
as opposed to program source. Slightly simplified, the classification of environment hardening methods into “static”, “generic” and “domain-specific” describe the degree to which the hardening method uses dynamic techniques.

**Level of Protection.** The third classification introduced in this thesis, concerns the level of protection a particular environment hardening method provides. Using this classification, we group methods and tools into three classes (see 4.4 *Levels of Dynamic Run-Time Protection*);

1. methods that immunize a program,
2. methods that that detect unwanted behavior, and
3. methods that reduce consequences for the larger system.

In this context, a tool is a set of methods tailored to a specific purpose. The classification dividing into protection levels is orthogonal to that of method type, since this criteria is based on the *effect* of the method, where the type is based on the *technical function or fundament* for the respective method.

### 6.1.1 Classifying our Experiment Systems

The experiment systems described in chapter 5, *Experiments* were categorized in terms of method type and protection level. We summarize this classification in Table 6-1 and discuss the rationale behind the classification below;

<table>
<thead>
<tr>
<th>Experiment System</th>
<th>Type of Method</th>
<th>Protection Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPS</td>
<td>Static Hardening</td>
<td>Reducing System Consequences</td>
</tr>
<tr>
<td>PLIBC</td>
<td>Domain-Specific Dynamic Hardening</td>
<td>Detection</td>
</tr>
<tr>
<td>SLIBC</td>
<td>Generic Dynamic Hardening</td>
<td>(Speculative) Immunization</td>
</tr>
</tbody>
</table>

**LPS.** The LPS system uses operating system kernel assisted methods to reduce privileges for a part of an execution environment. The sole purpose of this protection scheme is to reduce the consequences for the larger system, should an exploit occur in a component. This allows building more dependable systems from unreliable components. The level of protection offered by the method is hence quite clear - the method reduces system consequences of exploits.

Using LPS as a stand-alone method provides a static form of environment hardening. This is true because the method as such does not take the *actual execution* into consideration for parameterizing its operation. This does, however, not imply that there are *only* static uses for the method. The method is designed to be transparent to the system in which it is integrated, and because of this it would be trivial to integrate the method for use in a dynamic manner. For example, consider a program that dynamically load modules in the form of libraries (i.e. plug-ins in this context) and execute functions from these libraries. If a particular library did
or did not match a specific condition the program would split into a non-privileged part and execute the function in this context.

**PLIBC.** We classify the PLIBC system as being a domain-specific dynamic method. This is because not only does the method take the **actual execution** into consideration, but also validates the execution environment against an administrator-provided **local execution policy** that can be adopted for each system as necessary. The system hence considers the actual execution for the particular system installation.

The PLIBC system primarily detects unwanted execution; i.e. the system detects unwanted execution and then perform some comparably simple action to prevent the particular execution. In some cases, however, this action can immunize the system as well. For an example of such a rule-set, please See “Patching Rule-set” on page 61. We consider this the less typical use of the system, primarily because of the comparably simple rule language, which is the rationale behind the classification.

**SLIBC.** For the final experiment system, SLIBC, we consider the system to provide a form of immunization, albeit in a speculative fashion, and the method to be a generic form of dynamic environment hardening.

The SLIBC system immunizes the execution environment in the sense that a class of buffer overflow vulnerabilities are detected in addition to the execution environment being modified in such a manner that the program can resume execution. Programs written in the C programming language lack much run-time information which often makes it difficult to determine in which cases speculative immunization actually can be used. While this is highly relevant from a practical standpoint, as we discussed as a challenge for SLIBC, it is less important given a theoretic view.

The SLIBC system is a generic dynamic method as it does not use any specific information about the execution environment it protects. It does, however, rely on specific knowledge of other components, most notably the C library. If we were to include the C library into what we consider to be the system, the method would arguably be of the domain-specific dynamic hardening type.

### 6.1.2 Usefulness of the Classification

We use the classifications for reasons of **convenience**; there might be other principles for classifying environment hardening methods. This classification system, however, provides a simple and straightforward way of separating firstly the **function** of the method depending on the amount of dynamic information it uses, and secondly the **level of protection** the execution environment enjoys when using the specific method.

As we discussed for each respective experiment system, a static classification of a tool is not sufficient. This means that when we claim LPS is a static form of environment hardening, we are actually referring to the **typical use** of the method. As we observed, there might be dynamic uses of the method as well. This does not reduce the quality of the classification - but illustrates the dynamic properties of
this type of tools. While we could argue that using LPS in a dynamic fashion actually is combining several tools where the first tool is static and the second is dynamic, this adds little to the purpose of the classification.

This holds true for PLIBC, SLIBC and other hardening tools as well. We can use these tools in several ways, but a tool as such provide a certain level of flexibility because of its technical construction. This means that the tool belongs to a particular class or type but given a complex external system, the tool can be used in other ways as well. This is true not only for using static environment hardening tools in a dynamic fashion, but holds true the opposite direction as well. For example, consider using the PLIBC system with the same set of static rules for all programs. This would be equivalent to recompiling the programs with that particular restriction those rules imposed, and provide a static form of environment hardening, rather than the domain-specific form of dynamic hardening the tool can offer.

6.1.3 Principles of Protection

Based on the observations discussed in chapter 3. Model, we provided eight principles for building tools that hardens the execution environment. We discussed these principles in conjunction with the experiment systems, focusing on the relevance of each principle for the particular experiment system as well as in more general terms for similar tools. In Table 6-2, we provide a summary of our qualitative classification of these principles in regard to the three different experiment systems. Following that, we discuss the different principles again in regard to all three experiment systems, and our conclusions.

<table>
<thead>
<tr>
<th>#</th>
<th>Summary of Principle</th>
<th>Relevance for LPS</th>
<th>Relevance for PLIBC</th>
<th>Relevance for SLIBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Dynamic Methods that handle hostile code must rely on the operating system kernel for enforceability (See page 29.)</td>
<td>High</td>
<td></td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>Some types of exploits can only easily be detected from inside the execution environment (See page 29.)</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>III</td>
<td>For simple interception, interception primitives alone are sufficient (See page 32.)</td>
<td>Medium</td>
<td></td>
<td>High</td>
</tr>
<tr>
<td>IV</td>
<td>A function can be intercepted just prior to the normal invocation to obtain the parameters (See page 32.)</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>V</td>
<td>Using only simple analysis on a large amount of functions is useful to prevent some types of exploits (See page 33.)</td>
<td>High</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6-2: Protection Principles and Experiment Systems

<table>
<thead>
<tr>
<th>#</th>
<th>Summary of Principle</th>
<th>Relevance for LPS</th>
<th>Relevance for PLIBC</th>
<th>Relevance for SLIBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI</td>
<td>Simple actions can be taken by invoking the operating system kernel on the process(es) (See page 34.)</td>
<td>High</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VII</td>
<td>Complex actions can be taken by the tool modifying the execution environment (See page 35.)</td>
<td></td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>VIII</td>
<td>Dynamic methods that operate inside the execution environment must operate before a program is exploited (See page 37.)</td>
<td>High</td>
<td></td>
<td>High</td>
</tr>
</tbody>
</table>

Principle (I) and (II). We consider the first principle to have high relevance for the LPS system and of no particular relevance for the two other experiment systems. This might be a bit surprising, as this principle could be considered fundamental for all environment hardening tools and while this might be true, we should consider Principle (I) together with Principle (II) when classifying a dynamic tool.

Principle (I) from page 29. "All dynamic methods that handle potentially hostile code must be placed on the interface to the kernel, or use the existing protection mechanisms in the kernel, as this is the only interface that can enforce a (security) policy on a program. In many cases, already existing mechanisms in the kernel are sufficient, as all communication outside an execution environment must be done through the kernel (See Observation (II) on page 19)."

Principle (II) from page 29. "Dynamic methods placed in the execution environment can detect run-time situations that are difficult to detect by the operating system kernel. While a hostile program can choose simply not to call such functions, there is no reason to design a program to circumvent a system designed to protect the programs’ execution. Designing a dynamic method in this fashion enables protection of non-cooperative programs, but not of hostile programs, including any exploited program, that actively avoids the protection."

When we consider these principles together we observe an almost exclusive-or relation between the enforceability offered by kernel-assisted methods on the one hand and the possibility to detect certain situations offered by the execution environment on the other hand. Hence, a tool that operates from the kernel can enforce a decision on an executing program, but not detect situations when the kernel is not invoked to perform. Similarly, a tool that operates from inside the execution environment can detect certain situations but not enforce a decision on the executing program.

For some type of execution behavior this creates a situation were we must decide whether to enforce, meaning the kernel, or to detect, meaning the execution environment, when designing an environment hardening tool. These two approaches give completely different properties for the tool, and hence we select the principle with the highest relevance for the actual objective - and disregard the other.
Principle (III) and (IV). Principle (III) and (IV) concern interception, the first step of a environment hardening tool. The first, Principle (III) describes how we can perform simple interception;

Principle (III) from page 32. "When intercepting a single function with a one-to-one map to another function having the same interface, and the replacement function is available when the program is loaded, it is sufficient to use just interception primitives for the interception. For all other cases, we must use more generic interception techniques that cannot rely on compiler-generated prologue code or other high-level artifacts."

The SLIBC system uses quite a simple interception, in fact it can be loaded into a system using only the existing ELF dynamic linker [17]. Each intercepted function is replaced with a new function that uses the same interface, creating a simple one-to-one mapping. This type of interception is what we describe in Principle (III), which hence is highly relevant for this particular tool.

We also consider this principle to be relevant for the LPS experiment system. This is because while the LPS system requires manual integration in the source code, the manually prepared NULL interception is typically done as a one-to-one mapping at a simple interface. While it is possible to separate privileges of a program at any point at execution, it is not very practical, but interception would typically be done at a well-defined interface, effectively creating a simple one-to-one mapped interception.

An important observation of Principle (III) is that the principle as such actually depends on Principle (IV). This is true because in Principle (III) we make one fundamental assumption, namely that a function with an arbitrary interface can be replaced with an arbitrary function that uses the same interface. This is essentially an invariant of Principle (IV);

Principle (IV) from page 32. "When intercepting function calls, it is effective to intercept the execution just before a function is called, as this guarantees that all run-time information about parameters to the function is available. For complex analysis it might be necessary to save state from several interceptions for later use by the dynamic analysis."

This principle is relevant for all experiment systems we designed. The SLIBC system, which uses a comparably simple interception, is dependent on compiler-generated prologue code to reconstruct parameters, see Principle (III), but is also dependent directly on Principle (IV). This is because it intercepts low-level function calls and re-constructs higher-level information from parameters sent to these functions.

The PLIBC system is highly dependent on Principle (IV) as well, because its main function is to intercept massive amounts of function calls and perform comparably simple dynamic analysis of these (Principle (V)). Principle (IV) is also relevant for the LPS system because of how it is typically integrated into a program.

Principle (VI) and (VII). Principle (VI) and Principle (VII) have a direct connection to the exclusive-or relation between tools designed for kernel-based and execution-based operation;

Principle (VI) from page 34. "Simple actions can be taken by the dynamic tool calling the operating system kernel requesting operations to be carried out on the process(es) that are connected to the execution environment. When detecting run-time situations that are very likely to
Usefulness of the Protection Principles

cause exploits, a safe way is to request the kernel to terminate the process, but this can cause a form of denial-of-service attack, so we prefer other methods, if available."

**Principle (VII)** from page 35. "A dynamic tool can effect complex actions by modifying the execution environment directly. This type of actions can cause unwanted execution if used incorrectly, but by using information about common interfaces it is possible to modify state in a comparably safe way. Patching the execution environment can, if used correctly, remove the run-time condition that otherwise would have caused an exploit, and still let the program continue execution."

A tool can effect simple decisions by requesting the kernel to perform an action on the process(es) in which the execution environment live. For example, the tool could request the process(es) to be terminated should an overwritten canary value be detected. A tool could also patch the execution environment or perform some other operation on the execution environment to effect a more complex action. The former of these principles is essential for the LPS system since it operates using kernel-assisted security mechanism, while the latter is important for the SLIBC and PLIBC system.

**Principle (VIII).** The last principle also regards the separation of kernel and execution environment based methods. This is relevant for all methods that detect or immunize an execution environment;

**Principle (VIII)** from page 37. "Dynamic methods that immunize a system from an exploit, or that detects some type of exploits typically rely on sub-sets of execution environment state that can only easily be obtained from inside an execution environment, and hence these methods must operate on this layer. Operating on the execution environment layer means that it is not possible to enforce a security policy, so while such methods can operate on non-cooperative programs, they cannot operate on hostile (exploited) programs. For this reason, this type of methods must detect a potential exploit before the program is exploited, or the protection can easily be circumvented."

6.1.4 Usefulness of the Protection Principles

The principles of protection form the link between our methodology, and observations on one side, and the practical experimentation at the other - hence the link between theory and practical experimentation. This is a bidirectional relation as the observations on software execution form the foundation for the protection principles, but results and experiences gained from the practical experimentation have also affected the principles as such. This feedback loop from effect to design is an important mechanism for improvement of the software design and maintenance process.

Like our classification strategy, we use the protection principles primarily for reasons of convenience. We believe some principles to be highly generalizable, such as those that concern our observations on kernel- and execution environment based methods, while others primarily are technical patterns usable when designing and implementing dynamic tools. Our primary use of the protection principles is to describe and categorize the foundations for, technical construction of, and expected protection by means of different dynamic methods.

From the perspective of using protection principles as patterns for constructing environment hardening tools, it is important to observe that these principles also relate to our model of how dynamic methods function. While this is a fairly simple
and straightforward model, dividing the function of a tool into three distinct steps ("interception", "dynamic analysis" and "action"; see 4.3 Principal Function of Dynamic Methods), we have assumed this principal function of dynamic tools when discussing the principles of protection.

In this context, we can divide the different principles into four categories based on which step the principle concern (interception, dynamic analysis or action) or if the principle concerns foundations for execution environment based hardening:

1. Principles that regard how programs are intercepted with focus on methods that operate inside the execution environment (III, IV)
2. Principle that regard dynamic analysis with focus on methods that operate inside the execution environment (V)
3. Principles that regard effecting an action (VI, VII)
4. Principles that regard execution environment and kernel based methods, and the fundamental properties and differences between these approaches (I, II, VI, VII)
6.2 Protecting Software Execution by Dynamic Environment Hardening

The execution of modern software is a complex process involving a large state space, memory, a vast amount of program instructions, i.e. machine code, and complex interactions with other software. Additionally, many programs execute with no run-time validation, with unnecessarily high privileges and on systems that for one reason or another are not up to date. In this environment, we find that programs are exploited and that this situation causes unwanted system behavior. In this thesis, we have outlined a set of models and principles to harness this environment.

In the second chapter of this thesis, we described our Methodology in which we provided an introduction to how software programs are created from high-level source, and how these programs are transformed, "compiled", from textual source code into binary programs that later can be executed. We provided a background on older, simple, binary formats, the transition to modern more complex formats, and how these complex binary programs execute. We described the concept of exploits and vulnerabilities, and how exploits occur.

In the second chapter we also described the rationale behind static and dynamic methods where static methods attempt to find software vulnerabilities prior to executing the program and dynamic methods operate on the actual executing program, typically to detect or prevent exploits from occurring. Finally, we introduced the concept of environment hardening.

Following 2. Methodology, we described our observations on software execution in chapter 3. Model which resulted in eight principles of protection and a model of the principal function for a dynamic tool in general, and environment hardening in particular. We presented these principles in chapter 4. Principles of Protection. Following that, we described three experiment systems in the chapter 5. Experiments. The relation between observations, principles of protection, our model for dynamic methods and experiments is illustrated in Figure 6-1.

![Figure 6-1: Outline of this thesis](image-url)
6.2.1 Execution State and Paths

To summarize our findings in a formal framework we introduce some basic notations and relations. We model computations in any execution environment as state transition systems. A computation is a sequence of states generated by applying operators on a state to generate the next state. In short:

1. A state is a triple \( s = \{r, t, h\} \) with components from registers, stacks and heaps. The state space is \( S = R \times T \times H \).
2. The transitions are enabled by \( o_i \) where the operators \( o_i \) are selected from the set \( P \times LIB \times OS \), i.e. from program calls, calls to libraries and calls to the operating system kernel.
3. A computation is the sequence \( C = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots \rightarrow s_n \). We can \textit{decouple} the state sequence and the operator sequence and have a state sequence or operator sequence partial view on computations.

A computation can be modelled as a \textit{path} in the state space (See Figure 6-2). At any state \( s_i \) along that path an allowable operator \( o_i \) is applied to generate the next state \( s_{i+1} \). At that point of choice we typically would like to choose an operator that does not lead the rest of the execution path into an unwanted state. That is, models and methods to restrict the choice of operators at critical states to avoid run time failures. This thesis addresses that difficult task as a matter of applying models and methods described in previous chapters.

![Figure 6-2: Execution Path and State](image)

The general setting is as follows:

- Firstly, since we are making the operator choice decision at run time, we are under performance restrictions. The deliberation time should not be harmful to the expected time critical restrictions.
- Secondly, the decisions should be as well-informed as possible.

Those restrictions implies that we cannot neither take the whole state nor the whole operator set into consideration during our deliberation at points of selections. The critical issues are then reduced to proper choices of sub states and to provide means for adequate predictions regarding operator effects. Therefore, we have chosen to examine restrictions imposed on states in the R, S, H components (depending on our models and methods) at selected points of choice.
Furthermore, we have chosen to model the predicted effects of operators as interface streams between selected components of state, i.e. registers, stacks and heaps. The rationale for this choice being that functionality of operators typically are defined by their interfaces $o_{i, \text{inter}}$.

A bundle $b_{\text{crit}} = \sum o_{i, \text{inter}}$ of critical operator interfaces then allows us to predict and inspect the effects of applying the corresponding operators at a given selected point of choice. The sequences of critical bundles form a critical subsequence of the operator sequence of the operators.

Given the nature of our task, we cannot formally prove or even validate our methods, due to the inherent fact that we only can have partial information of the executions at any state. However, we claim that our explorative approach already have given substantial results, and also allows us to enhance our models and methods in a comprehensive and principled way towards hardening execution environments.

Unwanted Execution Paths. In a given actors’ perspective, some particular paths in a state space typically constitute an unwanted from of execution. For example, an exploit is typically an unwanted form of execution from a system owners’ perspective, but the exploit as such is actually a particular branch of an execution path which takes place because the state space at a specific time affected the sequence of operations which make up the executing program.

A dynamic method operates inside an execution environment and hence, theoretically, has access to the entire state space. Since the method operates during the program execution it can also, by means of modifying the state space, change the sequence of operations which the program execution consists of. Operating during the execution of a program also means that there are restrictions on the execution time for the method as such, i.e. the overhead imposed by using the method. Because of the size of the entire state space $S$ it is difficult to consider the entire space when making a prediction $p$ on whether a particular operation $o_i$ performed by the program is likely to cause an unwanted branch in the execution path. Instead, when making the prediction $p$, a dynamic method must identify a subset of state $s_n$, e.g. such that $s_{n,p}$ can be identified within the required time frame and that $s_{n,p}$ contains sufficient elements of the state space necessary to make an efficient prediction.

In some circumstances, a dynamic method can identify a particular subset of execution state $s_{n,p}$ necessary to make the prediction $p$ in the time frame allowed, by analyzing in addition to $s$, also the current program operation $o_i$. If this operation uses a standardized interface $I$, the dynamic tool can use information about the particular interface in conjunction with the actual execution state $s_n$ to create $s_{n,p}$ relevant for the particular $p$. In this case, the method uses not only information derived from $s_n \xrightarrow{o_i} s_{n+1}$ and $o_i$ but also static information about $I$, the interface stream.
6.2.2 Extracting Information from an Execution Environment

Returning to chapter 3. *Model* where we presented not only our observations on software execution, but also three hypotheses on dynamic methods and environment hardening.

Our first hypothesis concerns how to extract information from an execution environment. Given the fact that many programs use a very large state space, memory, and that every bit of memory theoretically can affect the execution, a dynamic method must within reasonable time identify a small sub-set of state which it analyzes to make a prediction of the state of the entire program (or execution environment). We formulated the hypothesis as;

Hypothesis (I) from page 22. "The execution environment can be divided into small subsets, which we can identify, and by analyzing only one or a few subsets we can make a good decision on whether a specific action is likely to be harmful to the program."

To prove the correctness of this hypothesis we must find a way to quantify what "a good decision" is and then show at least one tool that is able to provide such a "good decision" using only a subset of execution state. Because of the nature of the problem domain, and the non-formal definition, we will not take a formal approach to either task. Given the less formal view, we must also consider how to identify the sub-sets of state that the tool is dependent on.

Sub-sets of Execution Environment. There are very few formal rules for how an executing program behave; as long as the program executes only valid machine code instructions and does not request any service from the operating system it can execute as freely as can be expressed in machine code. Most programs, however, are written in high-level languages and use services from the operating system kernel as well as from other libraries in the system.

A program that does not communicate with its environment cannot be exploited. From a more formal view, this is true because all external input to the program originate from the operating system kernel, and a program that is not exposed to any external input cannot be said to be exploited. From a more practical view, we can disregard this type of toy programs.

For the vast majority of programs, those written in a high-level language, the program communicates using functions, or a derivate such as class methods, and typically depend extensively on helper libraries to communicate with its environment. On a UNIX system, essentially all programs use the C standard library (libc) in some way. This can be used as an interface stream (see 6.2.1 Execution State and Paths) since a dynamic tool can use static information about the interface (i.e. function prototype) to determine the relevant subset of state required to predict the new execution state (i.e. dynamic analysis to determine if the action is likely to harm the execution). In more practical terms, this means that even though it is hard to determine which memory buffer a particular program uses to store a specific datum, as soon as the program calls some function in a standardized library or in the kernel, it is possible to reliably analyze the datum given context information from the interface.
This simple form of identifying sub-sets of execution environment is what we describe from a practical view in Principle (IV), and is the fundamental principle behind the PLIBC experiment system. In PLIBC, the parameters to a large number of functions in the C library are analyzed and validated against a domain-specific policy to separate wanted from unwanted execution.

There are other examples of how to obtain a sub-set of execution environment that is sufficient for analyzing an unknown program in terms of secure execution. For example, in the SLIBC experiment system, we analyze the heap structure to predict if a particular memory copying operation is safe. This decision is made using a very small sub-set of the execution environment.

**Making a good decision.** The question of a “good decision” should, in this context, be reduced to that of making an informed decision given a reasonable time frame (see 6.2.1 Execution State and Paths). If we disregard any highly pointless tool that deliberately does not use information it has access to when making a decision, the quality of the decision is directly dependent on the accuracy of the information available. Hence, if the sub-set of execution environment we identify is “good”, it is possible for a tool to make an informed, meaning good, decision.

Because of the dynamic properties in the methods we discuss, we cannot prove that a tool is effective against all exploits belonging to any but the most trivial particular class, but by analyzing the technical function of a tool we can determine that it is likely that the tool is effective on an entire class of exploits. The quality of the analysis in the experiment systems we have presented are dependent on the domain in which we imagine the respective tool to operate. If we use SLIBC and/or PLIBC on a normal internet server, the tool will provide a significant addition to the normal (non-existent) run-time protection, and we could classify the dynamic protection of many, but not necessarily all, exploits as “good” in comparison to no protection at all.

Hence, we argue that we in this thesis have shown two example systems, namely PLIBC and SLIBC, that for some domains provide a good dynamic analysis of an execution environment sub-set, which enables the corresponding tool to make an informed, meaning good, decision.

### 6.2.3 Constructing Dynamic Tools

The third hypothesis we presented also concerns execution environments and building environment hardening tools. For this reason, we discuss this hypothesis at this stage as it closely relates to our first hypothesis. In short, we claim that it is in fact possible to build environment hardening tools that use execution environment information to protect software execution. We formulated the hypothesis as;

**Hypothesis (III)** from page 25. "It is possible to build dynamic environment hardening methods that while using only one or a few sub-sets of execution environment, can protect programs from exploits where the underlaying vulnerabilities are very difficult to find using static methods."

For Hypothesis (I) we argued that it is possible to extract relevant state information using only a small sub-set of the complete execution environment, and the usefulness of such information. Hence, we will not further argue for the possibility..."
Protecting Software Execution by Dynamic Environment Hardening

to extract relevant information, but only for the possibility to create methods and tools that when using such information can protect a program from certain exploits that are difficult to prevent using static tools.

**Protecting a Program from an Exploit.** We must consider how it is possible to protect a program or system from an exploit. Given our definition of protection levels (see 4.4 Levels of Dynamic Run-Time Protection) we consider a program to be protected from a particular exploit if we can immunize the program, detect the exploit or reduce the system consequences.

**Difficult Vulnerabilities.** In general, vulnerabilities that occur as the result of a complex execution flow, especially with a vast amount of state (variables) that cannot be determined pre-execution are difficult to detect using static methods, since the best such a tool can perform is to either guess the execution or, theoretically, consider all possible execution flows.

From a formal view, we may consider a program which has an execution flow that is very complex and constructed in such a way that a large part of state is required to determine its execution. From a more practical view, this can be a small program written in the C programming language using quite typical pointer aliasing, function pointer calling and a simple function whose result cannot be determined, for example because it depends on network input. Given some runtime condition the program dynamically builds a format string which it uses in conjunction with stdio routines [16] to format an error message, which it then prints.

This program will be very difficult to statically analyze as the execution flow is essentially arbitrarily complex. The more resources the static tool has access to, the more non-predictable iterations does the program perform.

Returning to a more practical view, constructions in the C programming language, such as lack of type enforcing and restriction-less pointer usage make the number of states in a program grow fast. As far as we know, there is no static tool that attempts to handle all possible execution flows for the type of programs we consider, but instead static tools guess the execution flow and use only a small window of execution context to guess if a particular construction is unsafe. For this class of tools, the required execution window must simply be one larger than the tool can handle.

**Using Dynamic Tools.** A dynamic tool executes in run-time and hence considers the actual execution of a program. For this reason, a complex execution flow is not an issue. The dynamic tool must, however, have means to extract relevant sub-sets of the execution environment to determine if a particular action is likely to be harmful for the program. We argued that it is possible to obtain such information in Hypothesis (I) and for this discussion assume this to be the case.

The first, and simplest, case for our argument is for tools that reduce consequences for the larger system by separating privileges, such as our LPS system. This is a simple case because the tool uses kernel-assisted methods of enforcing restrictions on the executing entity (process) in such a way that it is not possible for an exploit to cause unrestricted damage (consequence) to the system. This is
enforced on the basis of system calls and is hence completely independent of the possibly arbitrary complex execution state that lead to the invocation of the particular system call. Hence, dynamic methods, such as the LPS system can protect from exploits where the underlying vulnerabilities are very difficult to detect using static methods.

**Extending the Protection.** The focus of our work in this thesis is not only on kernel-assisted dynamic methods, but also on pure execution environment-based methods and tools. For this reason, we extend the discussion from a more practical viewpoint on building dynamic methods that protect a program from exploits where the underlying vulnerabilities are very difficult to find using static methods.

Both the PLIBC and SLIBC tools can be used to target exploits where finding the underlying vulnerability using static methods would require a comparably large context. The SLIBC tool detect memory violation, which is difficult to find statically because the unrestricted use of pointers, including pointer arithmetics, and lack of type and size enforcement. By using the heap layout it is possible to find many situations where the program normally would have been exploited, and where the underlying vulnerability might be dependent on a vast amount of execution state.

The PLIBC tool is very flexible and can be used to find several different types of vulnerabilities. In the experiments chapter we gave an example where we detected unwanted usage of string formatting directives. This type of exploits must be divided into two sub-types when we consider static and dynamic approaches:

1. Programs that use static (constant) formatting strings. These require a comparably small context to identify, and will likely be found by a static tool.
2. Programs that dynamically construct formatting strings which then are used to format the final buffer. This class requires a larger context and are more difficult to detect using static methods.

Since PLIBC is a dynamic tool, it does not matter how the formatting string is constructed, and, as we illustrated earlier in the thesis, this type of unwanted use of formatting strings can easily be detected. In the particular case of DHCPd, the latter type of formatting strings are used where a special routine firstly applies a program-specific format transformation before calling functions in the C library. Analyzing this statically would require an extensive context and would hence be very difficult.

From a more formal view, both these examples are difficult to argue for, but on the other hand not necessary for our hypothesis. From a more practical view, we see that all three experiment systems we have designed are useful for finding exploits where the underlying vulnerability require extensive context to analyze statically. This makes it very likely that these vulnerabilities are difficult to find using static tools.
6.2.4 The Applicability of Static and Dynamic Methods

Our second hypothesis concerns the applicability of purely static methods versus methods that to some extent use dynamic information. We formulated the hypothesis as;

Hypothesis (II) from page 25. "The complex and changing execution environment makes a pure static, as in completely pre-execution, methods inherently inferior in protecting system execution compared to dynamic methods, that take the actual execution environment state into consideration when making decisions on which operations that are safe."

At this stage it is important to point out that our hypothesis only states that methods which are "pure static, as in completely pre-execution" are inherently inferior to methods that take the actual execution into consideration, in regard to protecting system execution. This does not imply that information extracted by a pure static method by necessity is less usable or relevant than information extracted by other methods. It does, however, state that methods that use only static means to protect execution are inferior in regard to protecting system execution in comparison with methods that do not have this restriction.

Actual Execution. The key property for a dynamic method is that an implementation of the method has access to the execution environment of the program it protects, and hence considers the actual execution. For this reason, there is no need to predict the execution of a program.

In this thesis we have given several examples from our experimentation, which shows that a dynamic method can extract relevant information from the execution environment (Hypothesis (I)), and make a good decision based on such information (Hypothesis (III)). We have also shown in our examples that this decision can be effected. Hence, we know that it is not impossible to use dynamic information to protect execution. It is important to stress that we do not claim our experiment systems to extract the optimal sub-set of information from the execution environment, nor can we prove that any dynamic method can do so. For our experiment systems, the situation is actually the opposite - there are some information that the tools do not extract and that would be useful in order to make a good (as in informed) decision.

Informed Decisions. Hence, so far we have established that dynamic methods have access to at least one important aspect of a program that a pure static method does not - the actual execution, in the form of the execution environment. We have also established that, under some circumstances, it is possible to extract relevant sub-sets of execution environment (Hypothesis (I)), and results of practical experimentation) that, under some circumstances, can be used to find exploits where the underlying vulnerability is difficult to find using static methods (Hypothesis (III)).

Returning to our hypothesis; a dynamic method can use information about the actual execution, which we have shown can be useful for protecting program execution. There is no requirement that a dynamic method cannot also use pre-generated static data. For example, a dynamic tool might load debug data for a program, or some other data that has been pre-generated for it. Hence, a dynamic method might have access to the same information a static method
The Applicability of Static and Dynamic Methods

does, but also has access to the actual execution, which we have shown can be useful to find exploits where the underlying vulnerability would be difficult to find using static methods.

Since a dynamic method has access to more information about a particular executing program, and this additional information, under some circumstances, is useful for making a decision on whether a particular action is likely to be harmful to the program, a method that considers actual execution, a dynamic method, can make a more informed decision than a pure static method could, which makes pure static methods inferior to dynamic methods.

Practical Reflections. From a more practical view, we note that while a dynamic tool indeed may perform a complete static analysis, this is, in reality, very uncommon. Instead, static and dynamic methods are used together to protect a program, but typically no information from the static tool is shared with the dynamic. This results in the static tool failing to find vulnerabilities because it does not have access to the execution environment, and the dynamic tool failing to find exploits because it does not have sufficient static information.

While our definition allows for a dynamic tool to re-create static information, such tools are perhaps better referred to as hybrid tools. We will further discuss hybrid models, methods and tools and our views on using such technologies for future work in the next chapter 7. Discussion.

6.3 Final Conclusions
In section 1.2.1 Research Issues we presented the research issue for this work, namely the development of a comprehensive methodology enabling secure execution environments.

We have described our theoretical model of, and practical work with, protecting software execution using methods that consider execution and not only program source, dynamic methods. We discussed how software programs traditionally were constructed and how modern programs are constructed and execute. We described the fundamentals that make dynamic methods in general, and environment hardening in particular, work.

In addition to our model we presented eight principles of protection which include observations we made from a theoretic view as well as more practical observations on how software function, and how it can be protected.

Using our model and protection principles, we described three experiment systems that we have constructed. We described these systems both in terms of technical construction, but also in terms of the protection offered and how the system can be used.

We argued for three hypotheses that regard environment hardening dynamic methods - the first about extracting relevant information, the second about the inherent strengths in methods that consider the actual execution compared to methods that do not, and the third about using extracted information to protect a program from exploits.
Final Conclusions

If there is one final conclusion we have drawn from this work, it is that dynamic methods constitute a real and important complement to methods that do not consider execution, and that the actual program execution is a rich source of information. Using this information and dynamic methods, we can create secure execution environments that allow us to execute software in a more dependable way - the research issue for this work.
7.1 Further Observations on Dynamic and Static Methods

An important observation that we made on the function of static and dynamic methods is that of which assumptions that in fact are made when using a particular method. For example, when using a static method, we make some assumptions on the actual execution, or execution flow, for a particular instance of the executing program. From a high level viewpoint, we can imagine this as if the method is predicting the actual execution based on high-level artifacts it identifies in the programming language.

Some parts of an execution flow are comparably stable, such as the order and value of certain function parameters (For an example of this, please see 5.5 The plibc Experiment System), and it is hence fairly safe to make assumptions on the actual flow of execution that will result from these. For this discussion, we will call assumptions that are based on this type of stable parts of a program conservative assumptions. An assumption is conservative if it is based on only stable parts of a program or class of programs, since that means that the assumption will hold true for all or essentially all executions.

Some execution flows are very difficult to predict using static methods, and methods that still make assumptions on such flows are hence less conservative. For static methods, it is typically difficult to handle situations where the possible execution flow become complex and several state components in a program mutually depend on each other. Because of the complex flow, it is difficult to determine how the program actually will execute, and hence difficult to separate safe from unsafe execution. A method that makes assumptions on this type of execution does not make a conservative assumption, but rather speculates on possible outcomes of an execution flow. We will hence call methods that make assumptions that are not at all conservative for speculative methods. This does not imply that
the method simply guesses the outcome from a particular piece of information, but that the assumption does not hold true for all, or essentially all, instances of execution so there must be a way to handle the situation if the assumption is wrong.

It is important to point out that even though dynamic methods have access to the execution flow of a program, such methods still need to make certain complementary assumptions. Some of these assumptions are very conservative while others are of a more speculative kind. However, with dynamic methods, we make assumptions both on the method as such as well as when using the method;

**Assumptions In Dynamic Methods.** Many dynamic methods make assumptions on the layout of data in the execution environment. As we pointed out in our conclusions, this is done either because there is not enough information available (the data has been removed by the building tool chain; see 2.2 Building Executable Binaries from Source Code) or because of restrictions on execution time. Much like static tools, some assumptions are quite conservative and hold true for the vast majority of execution, while other assumptions are less conservative, or to some extent speculative.

Dynamic methods that protect more formally defined interfaces, such as calls to the operating system kernel, can use relevant interface streams (see 6.2.1 Execution State and Paths) to perform dynamic analysis making only comparably conservative assumptions on the executing program. This type of conservative dynamic methods share certain properties with the corresponding conservative static methods. Both type of methods rely only on assumptions that hold true for essentially all executions, greatly reducing the risk for false positives. On the other hand, using only this type of interface streams that results in assumptions that hold true for all, or almost all programs, increases the risk for false negatives (i.e. the tool not detecting certain unwanted execution).

When designing a method, we must balance the risks of false positives (a reported problem that is in fact not a problem) with that of false negatives (failure to report an actual problem). When designing secure systems, it is often preferable to use a method that emits false positives, as these can manually be inspected, rather than false negatives, but this situation is somewhat more complicated with dynamic methods.

**Assumptions When Using Dynamic Methods.** When using a dynamic method on a particular software system, we make some assumptions on the software system as such. For example, when integrating privilege separation into a system, we assume there is a way to propagate an error such that the larger system can recover from a failure in the privilege separated part. Hence, for most technical systems, we assume there is a way to effect an action such that the system in a larger context can resume execution.

Independently of the actions a particular method supports, we make an assumption when integrating a tool representing the method into a system. For simple actions (see 4.3.7 Action), such as to terminate the process in which an error was detected, we make the assumption that the, possibly larger, system can handle some processes being killed by the security frame-work. The assumption is
Consequences of Speculative Software Protection

recursive in regard to this property; if the larger system cannot handle a single component failure but this causes the entire system to fail, the larger system of which the current system is only a component must handle this situation.

When using more complex actions, it is not easy to predict which consequences a particular action will have for the larger system. This can be either as a consequence of assumptions in the method or tool itself or in the use of the specific tool. For example, the PLIBC experiment system which we described earlier in this thesis, can be used to very freely modify an execution environment when a certain condition is met. It is hence difficult to, in general terms, predict which exact consequences incorrect usage of the tool would have for the executing program.

7.1.1 Consequences of Speculative Software Protection

Every assumption made by a protection method that to some extent is speculative result in the protection as such being speculative to a certain degree. This does, however, not mean that we can generalize without restriction from speculative protection offered by one method to that offered by another. This is particularly true when comparing static methods with dynamic; for example, consider a typical static analysis tool that identifies source code-level vulnerabilities. Such a tool will typically be used by a developer/auditor that manually reviews the output from the tool and for each vulnerability reported makes a decision, firstly if the vulnerability is a false positive, and, if not, how the vulnerability should be removed. While static tools by definition do not require a human operator to make the final decision, this is usually the case.

This is in stark contrast to how many dynamic tools operate. When such a tool effects an action (the final step of a dynamic tool; see 4.3 Principal Function of Dynamic Methods) it typically does so in real-time and without informing an operator about its decision. The tool might write logging information about effected decisions but such information is typically only reviewed by a human auditor at a later time. For this reason, we expect false positives to be more problematic for dynamic than for static methods. Also, because many programs are designed in such a way that there is no easy and structured way to inform the program about a failure, a dynamic method must under some circumstances use a less fine-grained action than desirable.

Because of the difference in consequence when a static method makes a speculative decision and when a dynamic method effects an action based on a speculative analysis it is important to find ways that;

1. make the analysis as such less speculative without having vastly negative effects on system performance
2. enable more fine-grained actions to be effected, even on non-cooperative programs

In this thesis, we have described techniques that use function prototypes, pre-computed static information (in the form om PLIBC rules) and information about typical memory usage (in SLIBC - for handling adjacent heap buffers) as basis for dynamic analysis. We have also used some actions that effect the execution environment in a more fine-grained fashion than merely to terminate relevant pro-
Hybrid Methods

cesses. Using these techniques, we have been able to create tools that protect an executing program from some kind of exploits where the underlying vulnerabilities would be difficult, or impossible, to find using static methods.

There are likely other data that a pure dynamic, or almost pure dynamic, method could use to further advance the protection of software execution, however we believe it would be beneficial to combine the inherent strengths of dynamic analysis with other methods and tools to be able make more conservative decisions in run-time, and apply these decisions to an executing program in a less destructive fashion. Such a tool could use either pre-computed data generated by means of static analysis or leverage results from other tools and hence create a hybrid form of protection, where the real-time sensitive dynamic part would effect decisions based on a more informed dynamic analysis; a form of hybrid dynamic protection.

7.2 Hybrid Methods

We use the term hybrid method to describe a method that uses more than one principal way to obtain data and/or to effect an action. In this context, “principal way” has a rather broad meaning, so we consider a method that uses both static and dynamic analysis as a form of hybrid method, but we do not consider a method that uses several dynamic analysis steps to be a hybrid method. Given this definition, a hybrid method does not need to use any dynamic techniques at all, although in this discussion we only consider methods that combine dynamic analysis with one or more additional, and principally different, source of information.

7.2.1 Sources of Information

There are several different sources of information that we believe could be relevant when building next generation hybrid protection tools. We have considered the following three principal techniques and the possibilities a hybrid with each respective type could give;

**Static Methods.** Perhaps the most apparent combination for a hybrid protection is the combination of traditional static and dynamic methods. A dynamic tool could, for example, take advantage of high-level static data such as fine-grained information about complex data structures used within the program. Using a static tool to extract information from this high level would save time in the dynamic step, as well as enable extraction of data that the building tool chain removes when transforming the source code into an executable program. We have seen some work in this direction, where custom static pre-analyzers are being used; for example in [5].

A dynamic tool could also take advantage of other static data extracted from a lower-level than the source code. Typical such data would include meta-data about functions (boundaries, addresses and calling graphs) as well as which idioms where used by the compiler when the machine code was optimized. This type of data hence lies on the border between normal static data about the program and information generated by a compiler, which brings us to the next interesting source of information.
Effecting an Action

Building Tool Chain-assisted Methods. When building a program different parts in the tool chain (see 2.2 Building Executable Binaries from Source Code) can add transparent or pseudo-transparent security mechanisms that will be present in the final executable file. This type of methods can be used to detect certain kinds of buffer overruns and other similar situations. The compiler and linker process additional information that would be useful for a dynamic protection. Such information is by its nature static, making a tool chain that extracted such information a special class of static tool as it would operate at a lower level than most other tools. We would be particularly interested in using data regarding;

- Calling conventions, such as the number and type of parameters
- Layout of the compiler run-time library allocated function activation frames
- Hints about functions that show an atypical execution behavior (i.e. `alloca` and some compiler extensions like label pointers)
- Language-level constraints that could be propagated to run-time (i.e. pointers that are not permitted to overlap)

Much of this data is already used in the building process, and other data should be trivial to extract when building the executable binary. However, extracting this information from a pure dynamic method is difficult, which is the reason we believe a hybrid method could likely perform better than both a static and a dynamic tool together.

Hardware and Virtual Hardware Enforced Restrictions. It would be beneficial for a dynamic tool to operate on a hardware architecture that had stringent requirements on separation between code and data, and an instruction set that effectively could be parsed in run-time. There has been some recent improvements on the separation of code and data, using a simple but efficient protection mechanism [18, 19] as well as software that uses existing mechanisms in new and creative ways such as OpenBSDs [31] W^X. However, it is still difficult to parse machine code instructions in run-time within reasonable performance bounds. The situation is somewhat better on certain RISC-based computers, but the instruction set on ia32 is far too complex to analyze within reasonable time and without static assistance.

To further increase the amount of usable information provided by the executing machinery, it might be possible to use virtual machines or other virtualization techniques when executing certain type of programs. Currently, many such approaches impose a vast performance overhead, and are hence only to be used under special circumstances.

7.2.2 Effecting an Action

Another aspect of hybrid tools is the possibility to effect more fine-grained decisions (actions) on an execution environment. This is not an easy task, since the current "kill rather than tell"-approach used by many tools is guaranteed not to let an exploit execute. On the other hand, this is a very coarse level of protection and we believe it would be beneficial to have a more structured way of informing a program or system about the situation at hand.
Final Thoughts

In this thesis we have presented some experiments where we effected more complex actions than just to kill (terminate) the relevant processes, but using such techniques without having proper background knowledge of the system and possibly also the exploit could, in some situations, cause unwanted execution. This would be the case if a tool, such as PLIBC, was used to patch the execution environment in an inappropriate way. Rather than using the administrators knowledge of the system construction, as to some extent is the case when using PLIBC, it would be beneficial to have a more structured way to propagate security-related errors to the host program.

Run-Time Security Exceptions. Exceptions of different kinds are widely used both in certain programming languages as well as on a lower hardware level to propagate information about erroneous, or exceptional, state to a program. If there was a way for a dynamic, or hybrid, method to use such a facility to communicate with the executing program, it would be possible to give the program a chance to recover from the situation before reverting to less cooperative ways of handling unwanted execution.

Commercial implementations have shown that the normal exception framework is not always suitable for handling security related messages [28]. Hence, security exceptions would likely need to be handled separately from other exceptions as it is difficult to know which state in the programs execution environment that still is in a usable/reliable form.

7.3 Final Thoughts

Software execution is a complex process, and we must use all methods available to increase the dependability of this execution. In this thesis we have argued for the merits of dynamic protection, i.e. methods that protect execution rather than find vulnerabilities. We believe dynamic methods provide an important complement to other protection mechanisms, and that dynamic protection methods should be considered when designing dependable systems.

Dynamic methods does not mark the end of the road in regard to what we can achieve using methods that analyze execution. We can combine the strengths from dynamic methods with strengths of other methods to create new ways of protecting software execution. There is most likely no single silver bullet that can prevent all existing and new forms of unwanted software execution. There will most likely be new forms of exploits and other problems in software. However, we have a growing class of methods, models and tools that allow us to create better tailor-made protection, and hence at the end of the day make better software. The future is bright.
REFERENCES

Journals

Papers


Books

Standards
15 IEEE P1003.1, “Portable Operating System Interface for Computer Environments (POSIX)”, Institute of Electrical and Electronic Engineers (IEEE).
19 AMD, AMD64 Architecture Programmer’s Manual (See Volume 2: System Programming: Section 5.6.1).

Miscellaneous
20 European Commission EUSUSTDEV Project No. NNE5-2001-00906, Distributed Intelligence in Critical Infrastructures for Sustainable Power, “CRISP”.


Resources
Some references in this section are links to Internet web pages or similar online resources. All such references were valid July 2005.

Anonymous (“Bulba” and “Kil3r”), “Bypassing StackGuard and StackShield”, in Phrack Magazine issue 56, [http://www.phrack.org].

Anonymous (“Vendicator”), StackShield (Description of the StackShield Extension to the GNU Compiler Collection for ia32 targets), [http://www.angelfire.com/sk/stackshield/].

Kamp, P-H., “Malloc(3) Revisited”, (A technical description of the malloc implementation used in FreeBSD, OpenBSD and other systems) [http://phk.freebsd.org/pubs].


OpenBSD implementation of ‘malloc_guard’ to randomize returned pointers; described in malloc.c in OpenBSD Distribution (See also [27]).

The FreeBSD Project, [http://www.freebsd.org].

The OpenBSD Project, [http://www.openbsd.org].


Apache HTTP daemon open source project, [http://httpd.apache.org].

Tinyproxy open source http proxy server; [http://tinyproxy.sourceforge.net/].

ISC DHCPd Remote Format String Vulnerability; BugTraq Id 4701 [http://www.securityfocus.com].

TinyProxy heap overflow vulnerability; BugTraq Id 2217, CVE-2001-0129 [http://www.securityfocus.com].

