PROGRAM-LEVEL SUPPORT FOR PROTECTING AN APPLICATION FROM UNTRUSTWORTHY COMPONENTS

Nuwan Abhayawardena Goonasekera

B.Sc. (Hons) Information Systems

Submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Information Security Institute

Faculty of Science & Engineering

Queensland University of Technology

November 2012

Keywords

Component Isolation, System Call Interpositioning, Hardware Virtualization, Application Isolation

Abstract

Many software applications extend their functionality by dynamically loading executable components into their allocated address space. Such components, exemplified by browser plugins and other software add-ons, not only enable reusability, but also promote programming simplicity, as they reside in the same address space as their host application, supporting easy sharing of complex data structures and pointers.

However, such components are also often of unknown provenance and quality and may be riddled with accidental bugs or, in some cases, deliberately malicious code. Statistics show that such component failures account for a high percentage of software crashes and vulnerabilities. Enabling isolation of such fine-grained components is therefore necessary to increase the stability, security and resilience of computer programs.

This thesis addresses this issue by showing how host applications can create isolation domains for individual components, while preserving the benefits of a single address space, via a new architecture for software isolation called LibVM. Towards this end, we define a specification which outlines the functional requirements for LibVM, identify the conditions under which these functional requirements can be met, define an abstract Application Programming Interface (API) that encompasses the general problem of isolating shared libraries, thus separating policy from mechanism, and prove its practicality with two concrete implementations based on hardware virtualization and system call interpositioning, respectively. The results demonstrate that hardware isolation minimises the difficulties encountered with software based approaches, while also reducing the size of the trusted computing base, thus increasing confidence in the solution's correctness.

This thesis concludes that, not only is it feasible to create such isolation domains for individual components, but that it should also be a fundamental operating system supported abstraction, which would lead to more stable and secure applications.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I am very grateful to my doctoral supervisors, Professor Bill Caelli and Professor Colin Fidge, who have supported and guided me throughout the course of this research project. This has included, amongst many things, long and frequent meetings to discuss and hone the direction of this project, patiently reading through many documents, helping me to overcome difficult patches in the project, and cheering and guiding me throughout the whole process. Above all, they have been mentors, both intellectually and personally. Even when my personal circumstances required that I shift to Melbourne in the last few months of my candidature, they provided me with all the support that I needed. I cannot thank them enough for their efforts.

I would also like to thank my wife Santushi, for bearing with the unusual hours I had to keep, for patiently staying up all night proof-reading various documents and for allowing me to act precious under the perennial excuse of "I'm working on my thesis".

My sincere thanks also go to Dr. Tony Sahama, who has helped and guided me in numerous ways at many points in my candidature, including its inception.

Finally, I would like to thank the rest of my family, who have seen or heard very little of me during the last stretch of this research project. I dedicate this thesis to them.

Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signature: 

Date: 6/11/2012

List of Abbreviations

API	Application Programming Interface
ASLR	Address Space Layout Randomization
AUXV	Auxiliary Vector
CC	Common Criteria for Information Security Evaluation
CERT	Computer Emergency Response Team
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-the-shelf
COW	Copy On Write
CS	Code Segment
DLL	Dynamic Link Library
ELF	Executable and Linkable Format
EPT	Extended Page Table
GB	Giga Byte
GCC	Gnu C Compiler
GDT	Global Descriptor Table
GE-645	General Electric 645
I/O	Input Output
IPC	Inter-process Communication
JIT	Just-in-time
JNI	Java Native Interface
JVM	Java Virtual Machine
KVM	Kernel Virtual Machine
Multics	Multiplexed Information and Computing Service
NaCl	Google Native Client
NPT	Nested Page Table
ORB	Object Request Broker
OS	Operating System
PCC	Proof Carrying Code
PDF	Portable Document Format
PIE	Position Independent Code
POSIX	Portable Operating System Interface
PPL	Page Privilege Level
PPL	Protection Profile
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RPC	Remote Procedure Call
SFI	Software Fault Isolation
SIP	Software Isolated Process
SMM	System Management Mode

SO	Shared Object
SPL	Segment Privilege Level
SSE	Streaming SIMD Extensions
SYSCALL	System Call
TOCTOU	Time Of Check To Time Of Use
TOE	Target Of Evaluation
VDSO	Virtual Dynamic Shared Object
VM	Virtual Machine
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Monitor
VMX	Virtual Machine Extensions
VMXOFF	Virtual Machine Extensions Off
VMXON	Virtual Machine Extensions On
VSYNC	Vertical Synchronization
XPCOM	Cross Platform Component Object Model

Publications and Conference Presentations

Publications

The following peer-reviewed papers have been published (or are in submission) as a result of this programme of research.

Goonasekera N, Caelli W.J, Sahama T., "50 Years of Isolation," in *Proceedings of the 2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, Brisbane, Australia, 2009, pp. 54-60.

Goonasekera N, Caelli W.J, Fidge C, "A Hardware Virtualization Based Component Sandboxing Architecture," *Journal of Software,* vol. 7, pp. 2107-2118, 2012.

Goonasekera N, Caelli W.J, Fidge C, "LibVM: An Architecture for Shared Library Sandboxing," *In Submission*.

Conference Presentations

This research has been presented at the following conferences.

Goonasekera N, Caelli W.J, Sahama T., "50 Years of Isolation," Presented at the Symposia and Workshops on Ubiquitous Autonomic and Trusted Computing in Conjunction with the UIC09 and ATC09 Conferences (2009), University of Queensland, St. Lucia, Brisbane, Australia, 7-9th July, 2009.

Chapter 1: Introduction

1.1 MOTIVATION

A "process" is the key software abstraction supported by modern operating systems for protecting and managing separate applications. However, with the rapid spread of component based software, a contemporary application typically extends its functionality by loading components dynamically into its process address space. For example, operating system kernels load device drivers, web browsers load browser plug-ins, and many applications support some form of extension components to provide or augment their basic functionality. Although operating system processes have well-defined isolation boundaries and inter-process communications mechanisms [130], current operating systems provide insufficient mechanisms for isolating or protecting components of a particular application from each other [93]. This is clearly demonstrated by the fact that component based software extensions often *decrease* the reliability of the hosting application; a badly-written or misbehaving component can damage the containing host, and other components, either accidentally or deliberately.

The statistics are revealing: over 85% of Windows XP crashes are due to faulty device drivers [125], and Linux drivers have 3 to 7 times the bug count of the kernel itself [20]. Such failures are not limited to kernel drivers; software applications also suffer similar problems. Zeigler [145] indicates that over 70% of crashes in the popular browser *Internet Explorer* are due to 3rd party add-ons. In addition, over 50% of CERT-reported security threats are due to buffer overflow vulnerabilities [77]. The Java Virtual Machine (JVM) has similar vulnerabilities, because any misbehaving Java Native Interface (JNI) component has the potential to overwrite critical memory regions of the JVM, bringing down the entire virtual machine [16]. Clearly, as many researchers have emphasised, a critical need is better component isolation so that hosts are isolated from any extension components they incorporate [57, 59, 82, 126].

Modern trends in browser architectures also emphasise the gravity of this issue; both Microsoft's *Internet Explorer* [145] and Google's *Chrome* [8] browsers have

changed to multi-process architectures in which program components are isolated into several, disparate operating system processes. The Mozilla *Firefox* web browser too, has introduced limited support for isolating plugins into disparate processes, in order to reduce the possibility of vulnerabilities in the plugins leading to the whole system being compromised [98].

This thesis addresses this issue, and revisits the component isolation problem, with a view to preserving the key benefits of component based programming while preventing errant or misbehaving components from compromising their host.

1.1.1 Definition of a Component

Before discussing component isolation further, the term 'component' must be defined in a suitable way, as a general consensus on what constitutes a software component remains somewhat vague. We provide an extensive discussion of these definitions in Chapter 2, but for the purposes of this thesis, we limit a *component* to be any executable binary unit which is loaded by an application into its own private address space, with communication taking place between the component and its host application via a well-defined interface. Primarily, this will be in the form of dynamic link libraries (DLL)/shared object (SO) libraries, which form the primary means of composability in modern operating systems and applications. Therefore, this thesis focuses on component isolation at the DLL/SO level, with the expectation that isolation at this lower level will enable higher level constructs to be build upon it. Future references to the term "component" therefore, will specifically refer to this narrower definition.

1.1.2 Benefits of Components in Shared Address Spaces

While the benefits of components as units of composition are well understood [93], a key benefit in DLL/SO components is that they share the same address space as their host, which is a critical function for enabling easy data exchange between a component and its host container, or between several different components. These advantages are best contrasted by comparing the benefits of private address spaces against shared address spaces.

Chase et al. [15] identify three main advantages to private address spaces.

1. They increase the amount of address space available to all programs.
2. They provide hard protection boundaries.
3. They permit easy cleanup when a program exits.

However, private address spaces come with the following disadvantages.

1. It becomes increasingly difficult to share data between address spaces.
2. There are significant performance costs in crossing address space boundaries.
3. Synchronization between processes is usually required when address spaces are switched.

With regard to the first disadvantage, the main issue is that pointers have no meaning beyond the address space in which they were originally created, requiring that they be specially manipulated or readjusted before use. Consider a simple data structure with a pointer to another data structure, such as a linked list. Any attempt to pass this data structure unmodified between private address space boundaries would render the data structure invalid as the memory address to which the structure is copied cannot be guaranteed to be the same. All pointers within the structure would have to be readjusted so that they remain valid in their new location. Therefore, some additional technique is needed to share data structures between address spaces, such as marshalling or pointer "swizzling" [140], adding overhead and complexity to the process and making data sharing between address spaces awkward.

Secondly, address space switching has traditionally come at significant cost. While address space switching with schemes like hardware supported memory segmentation do not have such costs, they have fallen out of favour due to the increase in complexity mentioned above. The currently available schemes, such as operating system supported context switches, incur significant overhead, and synchronization is required when crossing address spaces [19].

Therefore, the benefits of address space sharing have given rise to research into single address space operating systems, particularly in 64 bit environments, where address space is abundant [15, 53, 75].

Many of the approaches to component isolation do not address the problem of shared address spaces satisfactorily, falling back to traditional RPC mechanisms and shared memory pages in order to pass data between a component and its host [8, 32, 144, 145]. This, of course, incurs a performance and complexity penalty in terms of parameter marshalling across process boundaries, or reduced functionality (the inability to use pointers) when using shared pages.

Thus, the very benefits of fine-grained components are lost in the process, as simple and efficient sharing of data structures is hampered. Therefore, this issue too was a key motivating factor in this thesis.

1.2 RESEARCH PROBLEM

1.2.1 Problem Statement

Current component based software may decrease the reliability of the hosting application, as a badly written or misbehaving component can damage the containing host and other components. Part of the problem is that the Operating System (OS) provides insufficient mechanisms for isolating program components from each other [93], a different situation from OS processes, which have well-defined isolation boundaries and inter-process communications mechanisms between them [130]. This research fills this gap as follows.

1. By creating protection domains for individual components, so that the interaction between components can be better controlled.
2. By preventing an errant or untrusted component from corrupting critical memory regions or having adverse effects on the hosting process.
3. By preserving the benefits of shared address spaces for components.

These mechanisms together serve to reduce the chances that a failure of a single component translates into an application-wide failure, increasing the reliability of the containing application. It should be noted that this research develops mechanisms for isolation, and not a policy. Additionally, it does not guard against vulnerabilities in the components themselves, but provides a mechanism by which the effects of such vulnerabilities can be confined to the protection domain/sandbox within which the component executes.

1.2.2 Research Questions

The research problem outlined above leads to several research questions.

1. Can protection domains for individual components be created with acceptable overheads?
2. What functionality should an isolation container provide?
3. What conditions must be satisfied in order to have confidence that an implementation of an isolation container provides the desired functionality?
4. Can a generic API be designed for this purpose, in such a way that it separates isolation policy from isolation mechanism?
5. Can the benefits of shared address spaces be preserved during its implementation?

1.2.3 Research Aims and Contributions

The main aim of the research described in this thesis is to isolate untrusted application components from their host, so as to increase the reliability and security of the host application, while preserving the benefits of shared address spaces. Untrusted components include components which:

a. are not intentionally malicious, but may adversely impact the host due to a failure, such as a mistaken buffer overrun which overwrites and corrupts the host application's state;
b. contain exploitable vulnerabilities, such as buffer overflow vulnerabilities [23], which can be exploited by a malicious attacker to compromise the host application; or
c. potentially malicious components dynamically loaded from unknown sources, which may present a security risk to the host.

This thesis analyses and describes solutions to the issue of dealing with such untrusted components, by providing mechanisms to create isolation domains and to constrain the execution of components executing within them, thus protecting the host from such components.

Towards this end, the thesis analyses the causes and symptoms of failure in a component, as shielding against component failure necessitates determining the causes of component failure. This task however, is made complicated due to the fact that a host and its components may interact in very complex ways.

The main mechanism for a host to invoke functionality in a component is via an agreed upon interface which is in essence, a local procedure call. Such a procedure call could fail due to any number of reasons, from a bug in the component itself, to a rarely triggered edge condition or a transient reason such as a lack of host system resources. The thesis therefore makes a distinction between the classes of failure that an isolation framework can realistically deal with and classes of failure that cannot be handled. For example, a logical bug in the component cannot be protected against, but an illegal attempt by the component to access memory outside of its allotted region could be intercepted and appropriate measures taken to abort the offending component, thereby preventing the failure of the host application.

The thesis also determines the protection mechanisms which are applicable and practical for protecting components. Current research has identified a variety of methods which can be used to isolate program components from each other, such as hardware enforced protection mechanisms and software-based protection schemes. Suitable mechanisms are identified for implementing protection, while guarding against the causes of component failure which have been identified. The available mechanisms are critically evaluated and the most suitable mechanisms chosen to implement protection.

The thesis defines an abstract component isolation API that allows a hosting process to create and manipulate component isolation domains, independent of protection mechanism, thus separating policy from mechanism.

Two concrete implementations of the above API are described, and its performance and effectiveness evaluated.

1.3 RESEARCH METHODOLOGY

The main research question involves devising methods for protecting a host application from components loaded into its address space. Hence, the main hypothesis is that host applications can be supported to create protection domains for

individual components and thereby prevent a faulty component from adversely affecting its host process. The "scientific method" was employed in answering this question [12, 81], and the individual research questions in Section 1.2.2 were empirically tested through objective quantitative inquiries, aided by the traditional method of developing a prototype implementation of the protection mechanisms. The methods used are in line with empirical research approaches detailed elsewhere [71, 72].

The overall process was sub-divided into several smaller tasks, all of which incrementally met the research aims. While the process as a whole was iterative in nature, the initial step was a review of the background literature, which consisted of the identification of the causes and symptoms of failure, followed by a review of existing mechanisms for dealing with such failure as well as the drawbacks in existing protection systems which utilised those mechanisms. The overall architecture of the system was defined, including the functional capabilities and guarantees that had to be provided by an isolation container, and the conditions under which those guarantees could be made. Subsequently, protection mechanisms had to be implemented which met the stated functional specification, and the abstract API was implemented. The development of the two prototype implementations followed standard software engineering practice and adopted an iterative approach to development. Finally, in order to determine the validity of the resultant software, the solution was evaluated against its stated goals.

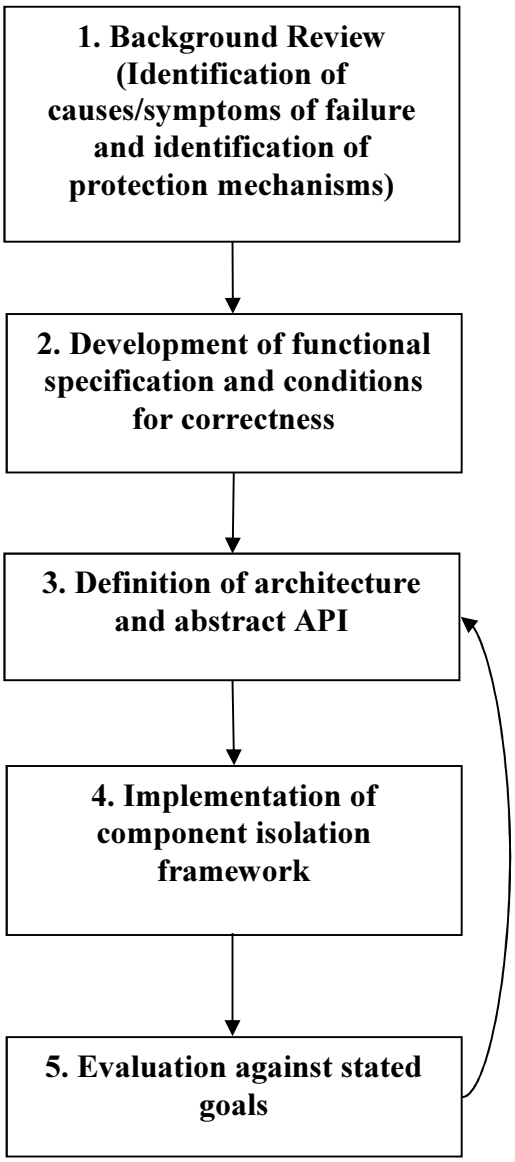


Figure 1.1: Overall research methodology

The overall methodology employed in this research is depicted in Figure 1.1.

The next sections discuss these overall steps in greater detail, and point to the relevant sections in which more detailed procedures can be found.

1.3.1 Background Review

1.3.1.1 Goal definition

This stage seeks to identify the causes and symptoms of failure as well as the protection mechanisms available for isolating components, leading to a shortlist of the most suitable mechanisms for the purpose.

1.3.1.2 Motivation

This stage will lay the groundwork for answering all major research questions identified in Section 1.2.2. Identifying the causes and symptoms of failure is necessary in devising methods for isolating those failures. Identification and classification of available protection mechanisms is necessary to determine the most relevant isolation mechanisms for implementing isolation domains. Finally, these mechanisms must be critically evaluated and the most suitable mechanisms must be chosen.

1.3.1.3 Goals

- Identification of causes and symptoms of failure
- Identification and classification of available protection mechanisms
- Identification of suitable mechanisms for implementing isolation domains

1.3.1.4 Design

The stage will be mainly an exploratory study aimed at laying the groundwork for understanding the available mechanisms, and will be in the form of a literature review. The process is detailed in Chapter 2.

1.3.2 Development of Functional Specification

1.3.2.1 Goal definition

This goal defines the isolation properties as well as the conditions that must be satisfied in order to have confidence that these properties remain present.

1.3.2.2 Motivation

In order to isolate components, the specific definition of what isolation means must be developed. The functions that must be provided to implement this isolation must

also be identified. Finally, the conditions which must hold in order to have confidence that the isolation can function properly, must be identified.

1.3.2.3 Goals

- Identify threat model
- Identify the functionality that an isolation framework must provide
- Develop a specification and the conditions under which the specification holds

1.3.2.4 Design

The isolation properties and relevant conditions are developed mainly through rigorous argument. The details, including the methodology for developing the conditions, are presented in Section 3.4.

1.3.3 Definition of Architecture and Abstract API

1.3.3.1 Goal definition

This stage defines the architecture and API which can be used to realize the functional specification previously developed in Stage 2.

1.3.3.2 Motivation

This stage will use the outcomes of the previous stages to define an architecture for the isolation container and subsequently develop and implementation agnostic API. The architecture will also need to take into account practical development concerns and the API will have to be developed accordingly.

1.3.3.3 Goals

- Identify development concerns
- Define overall architecture
- Define abstract API to implement architecture
- Highlight implementation concerns

1.3.3.4 Design

This stage will mainly be carried out through a rigorous analysis of the outcomes of Stages 1 and 2. This is detailed in Chapter 3.

1.3.4 Implementation of Component Isolation Framework

1.3.4.1 Goal definition

This stage is an empirical evaluation of whether the hypothesised framework can be implemented in practice.

1.3.4.2 Motivation

The purpose of this stage is to determine whether the architecture and design of the previous stage can be realized in practice, via a working prototype. In addition, it must demonstrate that the design is indeed abstract enough for multiple implementations to be possible.

1.3.4.3 Goals

- Provide proof that architecture is viable
- Provide evidence that API is implementation agnostic by creating at least two implementations
- Provide sufficient detail for recreating/duplicating this system

1.3.4.4 Design

This stage will be an empirical test of whether the architecture and design of previous stages can be implemented in practice. The development process will follow an iterative process, using standard software engineering practices. Relevant design patterns will be applied where appropriate. Premature optimization will be avoided, in favour of clarity of implementation. Testing will be done at the API level, as this represents the most important functionality in the system (black box testing as opposed to white box testing). By providing two separate implementations employing different isolation mechanisms, it will be possible to provide objective evidence that the API is implementation agnostic.

The detailed steps for recreating these implementations are provided in Chapter 4 and Chapter 5.

1.3.5 Evaluation

1.3.5.1 Goal definition

This stage analyses whether the implementations meet the security and performance expectations set for it.

1.3.5.2 Motivation

The goal of this stage is to analyse whether the prototype implementations are viable in practice, in terms of providing the security guarantees identified previously, as well as being able to provide the required isolation with reasonable overheads.

1.3.5.3 Goals

- Testing whether performance overheads are acceptable
- Analysis of whether implementation meets functional goals

1.3.5.4 Design

The performance overheads will be measured through a combination of micro and macro benchmarks. All benchmarks will follow the basic procedure below.

1. Running workload without isolation for a suitable number of iterations
2. Running workload within isolation container for the same number of iterations
3. Running workload against similar solutions for the same number of iterations
4. Repeating this process three times and averaging the values for comparison purposes

The micro benchmarks will measure performance characteristics testing specific aspects of functionality. The macro benchmarks will measure performance characteristics under general workloads. The industry standard SPEC benchmark [55] is particularly suited for measuring compute and IO heavy applications, and will be one standardised measure of performance. In addition, other relevant tests will be run an equal number of iterations, and the average execution times will be compared. These measurements, as well as results, are detailed in Chapter 5:

The analysis of whether the implementation meets its functional goals will be accomplished through the following methods.

1. Controlled tests which test specific aspects of the isolation mechanism (Detailed in Section 5.3.3)
2. Rigorous argument that analyses whether the conditions under which the security specification holds, is maintained. (Detailed in Chapter 6)
3. Using the Common Criteria [67] as a guiding framework (Detailed in Chapter 6)

1.4 THESIS STRUCTURE

Chapter 1 introduces the research problem, aims, methods and structure of the thesis. It gives a working definition of the term "component", some background to the importance of the problem, and the key contributions that this thesis aims to make.

Chapter 2 provides a detailed review of the literature. It provides an analysis of the causes and symptoms of failure as well as the mechanisms available to isolate components. The mechanisms as well as the work done by others in this area, are critically reviewed. Based on this review, the chapter describes a classification of isolation mechanisms, which resulted in a publication [46].

Chapter 3 provides an overview of the design and rationale behind the isolation API, termed LibVM, and lays the groundwork for all concrete implementations of the API. It provides an architectural overview as well as an analysis of implementation concerns, which are useful to those building a conformant implementation. The concepts and implementation described subsequently has resulted in an accepted publication [45].

Chapter 4 describes the first concrete implementation of the LibVM API, based on process tracing facilities [70] provided by the operating system. It is an in-depth discussion of the technicalities in realizing the API, the tradeoffs between various design decisions, and the implications on isolation.

Chapter 5 describes the second concrete implementation of the LibVM API, based on hardware virtualization support available in modern processors. It describes the architecture of the system, and provides an in-depth comparison against some

competing solutions, including an analysis of functional and performance tradeoffs. The process-tracing based implementation of LibVM is also compared against the hardware solution. The implementation described here has resulted in a publication which is in submission [44].

Chapter 6 provides a detailed functional evaluation of both LibVM implementations against a rigorous specification of its function. It identifies the strengths and weaknesses in each implementation. The ISO standard Common Criteria for Information Technology Security Evaluation was also used as a guiding framework in this evaluation.

Chapter 7 describes the final conclusions of this thesis, and describes future enhancements as well as directions for further research.

1.5 CONCLUSION

This chapter has provided an overview of the key research problem: that of isolating an untrusted component from its host application, such that the host application can shield itself from any harmful effects of such components. It has provided the background and motivation to this thesis, the general aims of the research, as well as the unique contributions made by this thesis, including the definition of an abstract API for isolating components, the realization of two concrete implementations of that API, and the preservation of address space transparency to preserve the benefits of single address space programming. The research methodology followed in the thesis has been outlined and the chapter has also described the structure of this thesis.

The next chapter provides a more detailed exploration of the problem and its background, as well as existing solutions, their drawbacks and lays the theoretical foundation for the rest of the thesis.

Chapter 2: Background and Literature Review

2.1 INTRODUCTION

This chapter puts the research problem into perspective by providing some working definitions, examining prior research in this area and highlighting the technology gap. It starts by providing a working definition of a "software component" that will be used for the rest of this thesis, as the term itself is somewhat vague and amorphous in general usage. This is followed by highlighting the importance of isolating software components, reviewing the key literature addressing the issue, and the various techniques available for the task. The existing techniques are divided into five main mechanisms, as published elsewhere [46] as a part of this research. Finally, a comparative analysis of these techniques is made, in order to highlight the gap that is being addressed by this research.

2.2 SOFTWARE COMPONENTS

2.2.1 What are Software Components?

The idea of reusable components was first proposed as far back as 1968 by McIlroy [91]. The idea was to create a library of reusable software units as a solution to the "software crisis", i.e., as a solution to the problem of unreliable, poorly performing software systems which practitioners were increasingly grappling with at the time [28], and for which no "silver bullet" exists to this day [35].

While components are an essential means of how any piece of software is constructed today, the definition of what constitutes a software component has evolved over time. Mendelsohn [93] provides an overview of this evolution, from the earliest form of reuse in the form of subroutines, to statically linked libraries, followed by dynamically linked libraries and culminating in component technologies such as Microsoft ActiveX/COM, and cross-platform portable components such as JavaBeans.

As a result, the definition of what constitutes a software component remains vague and amorphous. Szyperski [128] defines it as

"a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

Szyperski [128] also states that the need for independent deployment requires the component to be in binary form and therefore, any composable unit in binary form is a component, which includes a wide range from DLLs, Microsoft ActiveX/COM, XPCOM, CORBA components to Java classes and .NET assemblies or even entire applications.

Law and McCann [79] define a component as an "object encapsulating state and behaviour in order to achieve some specific task". Typical characteristics of a software component [79, 93] are:

1. Well defined interfaces defining the operations available on the component.
2. Use of object-orientated techniques by which components can be extended or specialized.
3. A binary calling standard for mapping between different implementations.
4. A means of instantiating and destroying components dynamically, and the ability to recursively nest or assemble multiple components in order to create richer components.
5. Machine readable meta-class descriptions to support tools.
6. Persistence mechanisms.
7. Standard packaging and registration mechanisms.

Despite the above identified features, much of current component based reuse continues to be in the form of code libraries. For example, Microsoft Windows utilises dynamic link libraries and Unix based operating systems utilise Shared Objects libraries [93]. These do not meet some of the typical characteristics highlighted by Law and McCann, such as the use of object-oriented techniques for

enabling extension, machine readable meta-class descriptions for tool support or even binary calling standards for mapping between different implementations.

Therefore, we limit a component to be any executable binary unit which is loaded by an application into its own address space, with communication taking place between the component and its host application via a well-defined interface, since both of the definitions given above are too expansive in their scope. Primarily, this will be in the form of dynamic link libraries (DLL)/shared object (SO) libraries, which form the primary means of composability in modern operating systems and applications. Furthermore, most other binary component standards such as COM and the Mozilla Foundation's XPCOM [99] are typically built on top of basic DLLs. Therefore, enforcing protection at a DLL/SO level would allow these higher level component models to utilise such protection mechanisms with relative ease.

2.2.2 The Need for Components

Traditionally, the basic abstraction offered by an operating system for protecting applications from each other is that of a process [130]. A process can be thought of as having its own virtual CPU and can be considered a container for grouping together related resources such as address spaces, threads, permissions, etc [130]. Processes can communicate with each other via an Inter-Process Communication (IPC) mechanism.

However, Mendelsohn [93] argues that as component based software is now the dominant means of assembling applications, the process as the main abstraction is incomplete and that future operating systems should support components as a fundamental abstraction.

The argument is bolstered by two major trends in software development as noted by Xu et al. [143]. The first is that of dynamic extensibility. This would typically involve a trusted host loading an untrusted extension dynamically into its address space. The second major trend is component based software development, where off-the-shelf components from multiple vendors are used to assemble a complete application.

Examples of such extensions are prevalent: OS kernels load device drivers from any number of disparate vendors, a web browser may load browser extensions and many applications support some form of extension components to provide or augment their

basic functionality [121]. Few operating systems provide any support for managing such components and better support for such extensions is a pressing need, particularly where trust in the component is a critical concern.

2.2.3 Problems with Components

Despite the need for component technology, the currently available mechanisms for providing any form of trust in those entities are often fraught with problems. In general, components are not protected from each other, and the failure of a single component can adversely affect the entire host application. This makes the process of creating dependable systems difficult, where dependability is defined by the "ability to deliver service that can be justifiably trusted" [5].

The statistics are quite revealing: over 85% of Windows XP crashes are due to faulty device drivers [125], and Linux drivers have 3 to 7 times the bug count of the kernel [20]. In Windows 2000, an analysis of support calls revealed that 27% of crashes were due to device driver failures, compared to 2% due to the kernel itself [127]. This indicates that the situation seems to have worsened since Windows XP suffers from an even greater percentage of device driver based issues.

Such failures are not limited to kernel drivers and user-level applications suffer from similar problems. Over 50% of CERT-reported security threats are due to buffer overflow vulnerabilities [77]. Zeigler [145] indicates that over 70% of crashes in the popular browser *Internet Explorer* are due to third party add-ons. One of the primary attack vectors used to compromise computers with internet access include applications such as Adobe PDF Reader, QuickTime and Adobe Flash [111], which typically run as embedded components within a browser. The Java Virtual Machine (JVM) has similar vulnerabilities, because any misbehaving Java Native Interface (JNI) component has the potential to overwrite critical memory regions of the JVM, bringing down the entire virtual machine [16].

As highlighted above, both accidental/unwitting errors and malicious exploitation of vulnerabilities can be at the heart of a component failure. However, the causes that give rise to such vulnerabilities are best understood through analysis of field failure studies which have been extensively published in previous research [21, 124, 142]. However, not all failure scenarios occur with equal prevalence, and a critical part

was determining the most relevant and common failure scenarios which needed to be addressed.

Failures have a cause and exhibit certain symptoms. Figure 2.1 shows this relationship.

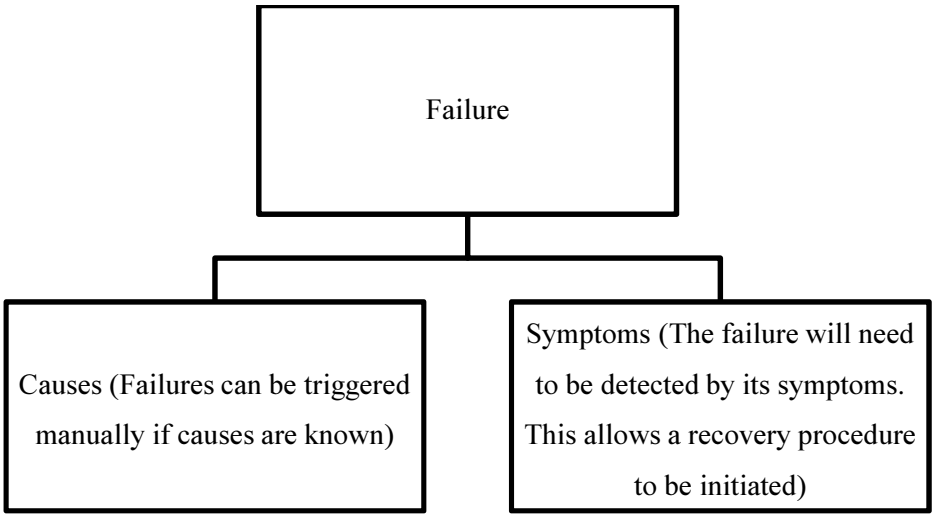


Figure 2.1: Failures, causes and symptoms

As identified by these previous studies [21, 124, 142], causes can be broadly categorised into overlay errors and regular errors [124]. Some typical overlay errors identified by various studies [20, 21, 30, 80, 124, 142] are given in Table 2.1.

Error	**Description**
Allocation management	A module deallocates a region of memory before it is completely finished with it. After the region is allocated, the original module continues to use it in its original capacity.
Copying overrun	The program copies bytes past the end of a buffer. Quite commonly, this is caused by off-by-one errors or an incorrect calculation on the required length of the buffer. Chieuh [17] points to techniques for preventing these errors by the use of machine debug registers.
Pointer management	A variable containing the address of data was corrupted.

	Code using this corrupted address caused the overlay.
Register reused	In assembly language code, a register is reused without saving and restoring its original contents.
Type mismatch	A field is added to a message format or a structure, but not all code using the structure is modified to reflect the change. Type mismatch errors also occur when the meaning of a bit in a bit field is redefined.
Uninitialised pointer	A variable containing the address of the data is not initialised.

Table 2.1: Common memory overlay errors

Error	**Description**
Data Error	An arithmetic miscalculation, use of incorrect constant or variable or other error in the code produces wrong data.
Statement logic	Statements were executed in the wrong order or omitted.
Synchronization	An error in locking code or synchronization between threads of control.
Unexpected situation	Unexpected parameter call from a process, or unexpected machine state or operation scenario.
Other	Difficult to classify.

Table 2.2: Common non-overlay/regular errors

As can be seen from these tables, the causes of errors are wide and varied, from rarely triggered edge conditions to complete failures in logic. A key observation is that memory overlay related errors form a significant portion of reported problems and thus is deserving of special consideration [124]. Approximately 34% of errors

were deemed to be related to such overlay errors [30]. However, the causes of such failure can be traced to bugs in the program code itself. Therefore, causes of failure, while important towards gaining contextual understanding, cannot directly be addressed by a component isolation framework.

A more useful way for a protection framework to deal with such problems is via the symptoms of the problem. By detecting the effects/symptoms of failure, appropriate action for recovery can be taken. For example, a wild pointer store could be detected and attempts to erroneously access a protected memory area could result in the component being terminated. Similarly, a deadlocked component should be terminated after the expiration of a timeout. Sullivan and Chillarege [124] identify the main symptoms characteristic of failure. In the context of a component, the symptoms could be explained as shown in Table 2.3.

Symptom	**Description**
Abnormal termination	The component attempts to execute an illegal instruction which results in a trap into the operating system. This may occur as a result of division by zero, for example. Typically, a signal will be raised notifying the application.
Addressing error	The component makes an attempt to address an illegal memory address. The memory may simply be corrupted if it is a valid address or result in a trap for an unmapped or invalid address.
Endless wait/infinite loop	The component enters an infinite loop or deadlocks, waiting for an event which never occurs.
Incorrect output	The component returns an invalid result to the parent.

Table 2.3: Symptoms of failure

If a component is properly constrained within an isolation domain, any of the symptoms in Table 2.3 would enable an isolation container to detect a failure in the component. Abnormal termination could be handled relatively easily by handling

the signal raised by the operating system. Similarly, attempts to access memory outside of the isolation domain can be detected and an appropriate notification made to the hosting application of the failure, allowing the host to unload the component. This is under the assumption that the isolation domain enforces memory protection and that the component cannot access memory beyond its allowed ranges. Finally, infinite loops can be dealt with through the use of a timer, after which the hosting component can be notified of failure, allowing the host to decide whether to continue execution or terminate the component. Since the upper bound on execution time may be better known to the host application, the host can implement a user-defined/application-specific policy which determines whether a component has hung (this can also include user intervention, such as displaying a dialog box which allows the user to terminate the offending component, or continue waiting). However, there may be cases where it is not possible to use any heuristics to determine whether the component has hung. Nevertheless, since the host component has a chance to execute and intervene, it can, in principle, terminate the component, which is a marked improvement over a situation where the component may not relinquish control at all, and the host never receives a chance to arbitrate.

The only situation that cannot be handled by an isolation container is that of incorrect output. The hosting application must take appropriate care to not use invalid output returned by the component, as there is no way for an isolation domain to know what the legal values are. Therefore, it is of critical importance that the hosting process is made fail-safe through the use of defensive programming techniques [47]. This means that all inputs, intermediate results, outputs and data structures should be checked as a matter of course. Any detected problem can be used to initiate recovery procedures.

Following from this, isolating a component in such a container is of critical importance, as emphasised by others [57, 59, 82, 126]. Indeed, microkernel-based operating systems take this concept to its ultimate manifestation [132]. Modern trends in browser architecture also emphasise the gravity of this issue, as both Microsoft's Internet Explorer [145] and Google's Chrome [8] browser have changed to multi-process architectures in which program components are isolated into several, disparate operating system processes.

In contrast to components, processes are typically protected from each other so that a buggy or malicious application has a hard time bringing down the entire system. Typically, protection is afforded by the following means in an OS process, as mentioned by Law and McCann [79].

a. Preventing applications from executing privileged instructions (i.e., disabling interrupts).

b. Preventing an application from accessing illegal memory locations (i.e., another application's memory)

Most operating systems today enable this isolation through use of processor modes and memory paging [130], while earlier computer designs such as memory segmentation and typing structures have been largely ignored. However, the more recent notion of an application-oriented, software component does not have the ability to take advantage of similar isolation schemes. Typically, an application will load a component into its existing address space, enabling the component to access any part of the host application's memory, thus leaving the host vulnerable to bad or misbehaving applications. Small and Seltzer [121] argue that if a component consistently crashes its host, the extra functionality is hardly worthwhile. Despite this, they note that few component extensions address the reliability issue.

Xu et al. [143] note that dynamically loaded extensions need to be verified as the hosting process needs to be assured that the untrusted component is safe and does not compromise the host's integrity. Similarly, components coming from different vendors and sources must ensure that they can survive without interfering with each other. Due to the low reliability of system components, there has been a renewal of interest in isolating components, justifying the need for this research [139].

Based on these requirements, the next section analyses current research on protection mechanisms for software components, and highlights the relative strengths and drawbacks of the various approaches.

2.3 PROTECTION MECHANISMS

2.3.1 Types of Protection

This section provides an overview of more current mechanisms available for component isolation. Three basic mechanisms for isolation are identified and categorised by Small and Seltzer [121]. As a part of this research, this was expanded to a more fine-grained categorization, to reflect the variety of isolation mechanisms available and has been published elsewhere [46]. The categorization, shown in Table 2.4, is based on the mechanism's level within the computer's architecture. It will be the basis for discussing those various isolation mechanisms. It should be noted however, that some mechanisms do not fit cleanly into a category and may be a mix of many techniques.

Isolation Mechanism	Description	Examples
Hardware isolation	This category deals with mechanisms that utilise some hardware support in enforcing isolation. The main drawback is that, due to reliance on specific hardware features, these techniques may not be portable across all computer architectures.	Privilege level change including protection ring hardware structures, Isolation using memory segmentation and/or typing support, Isolation using paging hardware, Isolation into separate processors, e.g. peripheral I/O processors, Hardware virtualization support.
Binary code level isolation	Protection afforded by modifying binary code.	Software Fault Isolation, Binary translation, Virtual Machine Monitors.
Integration into OS kernel isolation facilities	OS kernel protection mechanisms for isolating components.	Kernel Wrapping, Process containers.
Language support	Isolation provided through language level/compiler level support.	Type safe languages, Static analysis, Compilers.
Application level isolation	Isolation facilities implemented entirely in user space.	Interpreters, JIT Compilers, Application Virtualization

Table 2.4 - Categorization of isolation mechanisms and examples

2.3.2 Hardware Isolation

A variety of hardware-based techniques have been utilised for process/component isolation for almost 50 years, dating from the earliest second generation computer systems providing multiprogramming facilities, e.g., English Electric KDF-9 [50]. Some techniques, such as simple, dual-state privilege level change, based on OS-user separation, are available in most modern system architectures. Some other techniques, such as memory segmentation support, are now only widely available in certain architectures like the Intel "x86" line of processors. This section examines each of these mechanisms in turn, highlighting their usage and specific advantages and disadvantages.

2.3.2.1 Privilege level change

The idea of protection rings and segmentation was pioneered in the Multics system and implemented by the GE-645 machine architecture [10, 113], which utilised 8 rings of protection [115]. Earlier second and third generation mainframe computer architectures did, however, provide separate hardware mechanisms to assist with a form of isolation. These varied greatly from complete isolation into distinct and separate processor units, as in the peripheral processor concepts in the Control Data 6000, 7000, Cyber-70/170 computer systems and memory "tagging" used in the IBM System/360 series. Amdahl et al. [4] clearly pointed out in 1964 that program isolation was essential in the System/360 in terms of "tamper-proof storage protection" and a "protected supervisor program". The Burroughs B5000, introduced in 1961 was also an early system which featured segmentation and tagged memory [89], while the later VAX system included similar features, as well as support for virtualization of the operating system (OS) [74].

Most modern operating systems utilise at least a two level protection mechanism, which separates the operating system itself from application level programs [130]. The OS executes at a higher privilege level (ring), allowing it to execute any instruction. Applications have a lower privilege level and the hardware ensures that any attempt to execute a high privilege instruction causes a "trap" [131]. When a trap occurs, the operating system has the chance to intervene and arbitrate whether or not the process has sufficient privileges to execute the said instruction.

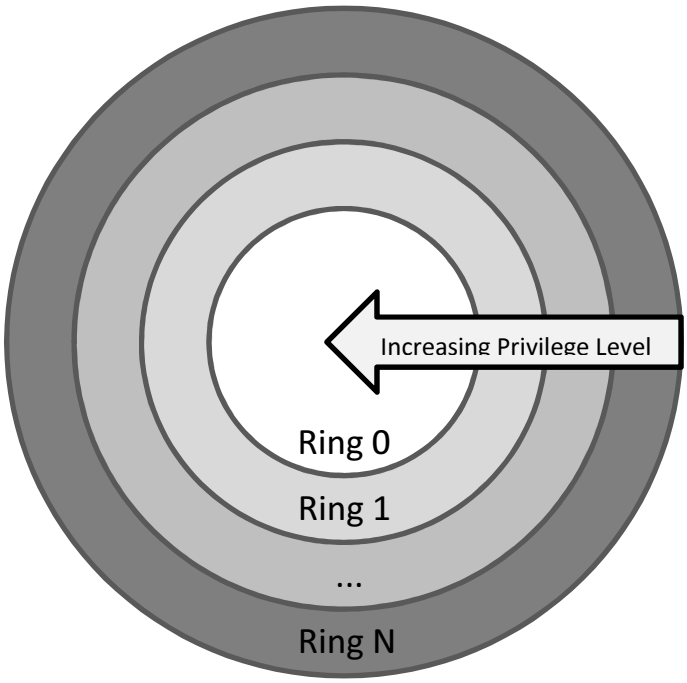


Figure 2.2: Protection rings

The details of this ring structure, as illustrated in Figure 2.2, may vary. For example, the x86 line actually defines 4 rings of protection, with the kernel running in ring 0 [130]. The VMX (Virtual Machine eXtensions) root mode introduced in some modern Intel processors, created an additional level of privileged execution [65], colloquially referred to as ring "–1". VMX root operation is described in greater detail in Section 2.3.2.4. The SMM (System Management Mode) mode [62], which operates at an even lower privilege level, is informally referred to as ring "–2", and is used typically by firmware.

Regardless of these differences, the basic kernel/user mode separation is the important mechanism in implementing protection. The kernel can pre-empt any application that does not respond in a timely fashion. Normally however, paging or segmentation hardware is also utilised on top in order to implement memory protection.

Windows XP and Linux execute most of the operating system in kernel mode, with applications executing in user mode. In combination with paging hardware, it is possible to protect Operating System level processes from interfering with each other. In a similar vein, Banerji et al. [6] utilise the kernel user mode separation along with paging and segmentation hardware, as discussed later, to protect shared

libraries. Effectively, the OS kernel is used as a "trampoline" to make sure that libraries can only be accessed at predefined entry points. By placing each library in a separate segment, libraries are prevented from directly accessing or corrupting each other's memory.

Czajkowski and Dayn's work [24] uses privilege level based protection for isolating Java Virtual Machine (JVM) components from Java Native Interface (JNI) extensions. Each JNI extension is isolated in a separate process and normal process protection applies in separating the JNI component from the rest of the JVM.

A more specific version of privilege level based protection is utilised in micro-kernel based operating systems [130]. In such operating systems, the kernel is extremely minimal, consisting of mechanisms for simply transferring control between applications [130]. This minimalist approach is an attempt to enforce the general security principle of having the least common mechanism and the principle of employing economy of mechanism [114]. At its extremes, even paging and scheduling may run as user mode applications [37, 132], which allows for a more modular approach with a greater degree of robustness and isolation of faults [29]. For example, in the Minix kernel [132], the kernel remains extremely minimal with most components running in user space. In the event of a failure in a user-space component, a "reincarnation server" is responsible for the automatic restart of the failed processes [56]. The reincarnation server periodically polls each process to see whether or not it is healthy. If the process is found to be defective, it is "reincarnated", by sending it a "kill" signal and restarting the process shortly thereafter [56]. A similar mechanism would be viable for restarting failed components.

The major drawback of the privilege level approach is the high cost of switching protection domains [69]. Estimates for the switching overhead over traditional monolithic kernels run as high as two orders of magnitude [106]. However, Leslie et al. [82] demonstrate that it is possible to drive a high speed gigabit Ethernet driver entirely in user space without significant performance loss, while work on the L4 microkernel also demonstrates that proper optimization can lead to very low overheads [86].

Overall, the level of protection afforded by this mechanism is mostly at the level of process level granularity. For components which are at much finer levels of

granularity, the protection domain switching overhead may be too prohibitive, depending mainly on the granularity of the component. This is sharply highlighted by the fact that many traditional monolithic kernels still continue to minimise privilege level changes specifically because of such perceived overheads [137]. This issue is explored further in this thesis when discussing the implementation issues of LibVM's p-trace based isolation mechanism in Chapter 3, which is reliant on process switching via privilege-level change.

2.3.2.2 Paging protection

"Paging protection" is a memory protection feature offered by most modern CPU hardware and works by dividing the address space typically into fixed-length pages, with the ability to set permissions per page [54]. Most modern operating systems utilise paging protection for enabling process isolation [130]. Mehnert et al. [92] point out that separate address spaces are beneficial even in real-time kernels, due to the higher level of protection afforded between subsystems. Attempts have been made to utilise paging protection for component level isolation as well.

One such attempt is the protection of the JVM heap through the use of paging support [16]. Unlike the more general approach adopted by Chiueh et al. [19], this approach focuses on protecting the various heap spaces only, under the assertion that many instabilities in the JVM are caused by heap corruption. While the approach itself is limited to heap protection, their technique, described below, has some interesting attributes. The basic idea behind their technique is to allow only certain threads to access different heap areas. The "HotSpot" JVM has two heap areas, one for Java data structures and another for dynamically compiled code. Each of these two heaps is placed in two separate "protection domains". A protection domain is, very simply, a separate memory page with read/write access permissions set. Whenever a thread switch occurs, the JVM changes the protection domains such that the thread may only access pages it is granted access to. For example, the compiler thread may access all heaps, but other threads cannot access the compiler heap. The technique relies on switching these protection domains at each thread switch. One major advantage of their method is that it does not rely on hardware segmentation support, unlike Chiueh et al.'s approach [18, 19], making the technique portable to most modern hardware.

2.3.2.3 Segmentation protection

The use of memory segmentation was also pioneered in the Multics system and implemented in the GE-645 architecture [10, 113]. Segmentation provides a hardware supported mechanism for neatly separating software components from each other. The basic idea is that system memory can be divided into variable sized regions (segments), and each segment can have four possible segment privilege levels (SPL), and two possible page privilege levels (PPL). The hardware ensures that lower privilege segments cannot access higher privilege segments, thus isolating memory segments from each other. The advantages of such hardware support for preserving high performance are stressed by Chiueh et al. [18]. They introduce an intra-address space component isolation scheme by using the paging and segmentation support in the Intel x86 hardware architecture, which is the most prevalent architecture for desktop machines. SPL support is utilised in isolating kernel extensions from the kernel itself, by placing all extensions in a separate segment of lower privilege than the kernel. A slightly different approach is utilised for user space components, due to problems with dynamic link libraries. Here PPLs are utilised for protection, to avoid the complexity of calculating relocation tables, as segment addresses start at zero. The result is a relatively low overhead approach for isolating regions of memory from each other.

An attempt to avoid the ring transition overhead by using segmentation support is reported by Vasudevan et al. [136], indicating that the technique has a wide degree of flexibility and is a viable alternative to privilege level changes.

However, a significant problem in using segmentation is the gradual dwindling of support for this hardware feature in the x86 hardware, as a flat memory model was conceptually simpler to program for and thus became the dominant paradigm. Current programming models on the x86 are based on the flat memory model where, in effect, segmentation support is completely disabled [60, 66]. In fact, the newer Intel 64 bit architectures do not support segmentation at all and is only available in backwards compatible modes [64].

2.3.2.4 Virtualization

Hardware virtualization is also an isolation mechanism which has been around for decades. The concept of virtual machines goes back to the 1950s and 60s, e.g., in early computer systems from the United Kingdom, as well as in the likes of the IBM System/360 Model 67 and System/370 series [49, 117]. The formal requirements for such fully virtualizable machines were later laid down by Popek and Goldberg [104], who established the essential characteristics for a system to be considered a Virtual Machine Monitor (VMM). System/370 featured hardware support for interpretive execution, making the development of VMM software much simpler [49].

Despite this, however, the popularity of VM technology waned somewhat over the years, but has gained a resurgence of interest with the invention and popularity of systems such as VMWare [1, 123]. VMWare provides a VMM for the original, popular Intel x86 architecture, despite the fact that the Intel x86 architecture itself had several non-virtualizable instructions [110], which did not meet Popek and Goldberg's virtualization requirements [104]. Many novel techniques have since been used to overcome such limitations, such as binary translation [1, 123] and para-virtualization [7, 13, 33, 138]. As a result, a variety of types of VMs exist, with Figure 2.3 providing an overview based on a taxonomy by Robin and Irvine [110].

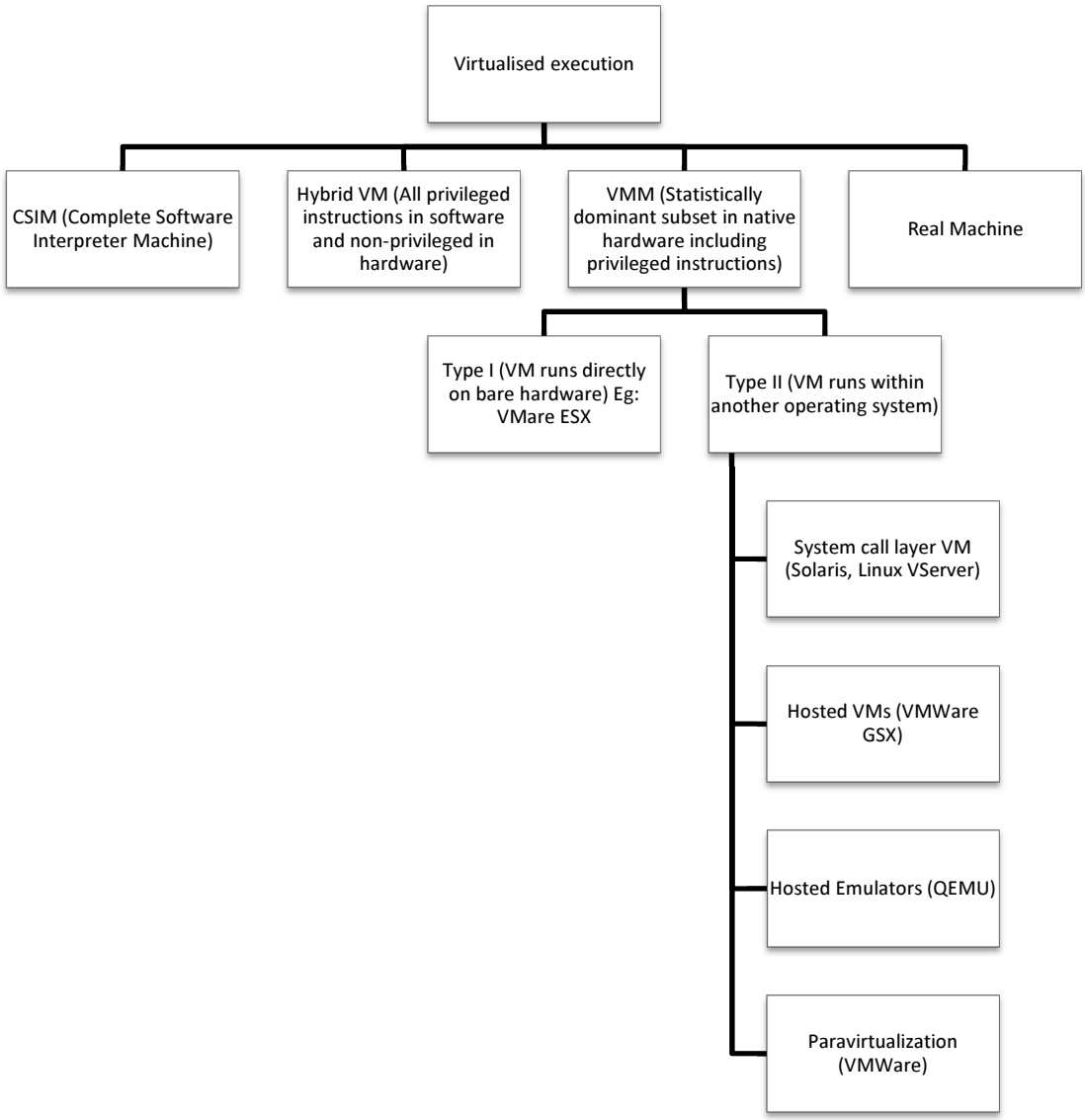


Figure 2.3: Graphical depiction based on Robin and Irvine's taxonomy [110]

In 2005, Intel and AMD introduced additional machine instructions to their respective architectures to improve virtualization support [3, 62, 64]. The machine instructions were similar in nature to the old System/370 and enabled the interpretive execution of code and additional hardware managed control blocks. The Intel and AMD extensions are extremely similar [1], which makes it easier to support either instruction set. Uhlig et al. [134] provide an overview of the architecture, with additional details being available elsewhere [62, 64]. Figure 2.4 depicts the basic operation of the Intel VT instruction set.

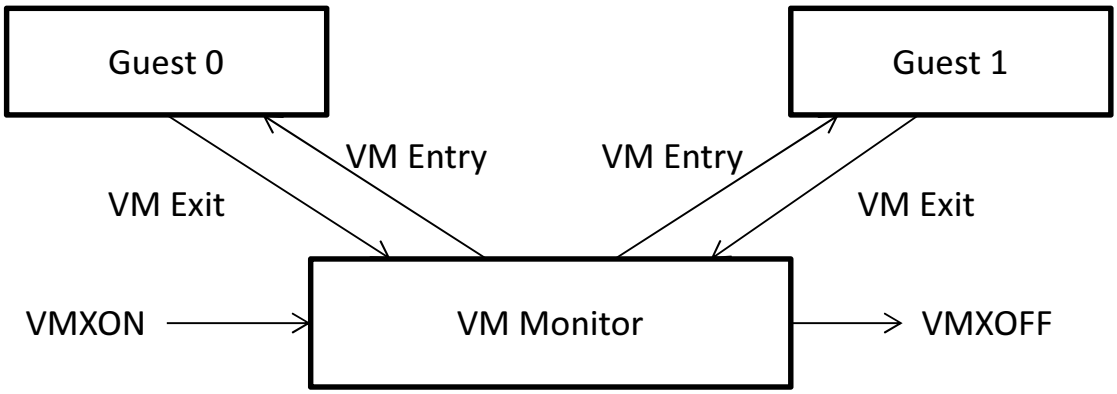


Figure 2.4: Intel VMX operation

As shown in Figure 2.4, processor support for virtualization is provided by a special form of processor operation named VMX operation [62, 64]. A Virtual Machine Monitor (VMM) is intended to run in a special processor mode known as VMX root, which is considered a higher privilege level, and colloquially referred to as "ring –1". The virtual machine monitor (VMM) may enable VM operation by executing the VMXON instruction, which changes the processor mode into VMX non-root operation. A VMM may then launch multiple guests by executing the VMEntry instruction. As the guest executes, any attempt to execute a privileged instruction will result in a trap, and the VMM will regain control The VMM can then safely emulate the privileged instruction and resume execution of the guest code. The VMM itself may exit via the VMXOFF instruction. Each logical processor in a virtual machine has an associated VMCS (Virtual Machine Control Structure) which maintains its state. By using these basic structures, it is possible for a VMM to execute a guest operating system with far fewer traps than traditional techniques such as binary translation [1].

However, as also noted by Adams and Agesen [1], early versions of Intel's and AMD's hardware virtualization did not necessarily result in better performance, due to the lack of support for Memory Management Unit (MMU) virtualization. To remedy this, AMD introduced Nested Page Tables (NPT) [3] and Intel has followed suit by adding support for Extended Page Tables (EPT) in their new "Nehalem" processor architecture, both of which add support for MMU virtualization [65].

While VMMs have achieved widespread use today [1, 7, 107] for emulation of complete machines, variations are seen of this, such as the work done by Rutkowska and Wojtczuk on the Qubes Operating System [68]. While Qubes builds an

Operating System on top of the Xen Hypervisor, it provides several child domains, called "AppVMs", which can be used to create a task-based separation of applications. For example, "banking", "work" and "personal" might be three separate tasks into which relevant applications can be isolated, with differing levels of security. It should be noted however, that these AppVMs do run full versions of the Operating System, in contrast to other schemes discussed later, in Section 2.3.3.2. The trade-off is between greater confidence in isolation versus greater performance.

This virtualization support has proven pivotal in implementing one of LibVMs own isolation mechanisms.

2.3.2.5 Non-standard protection schemes

This section reviews proposed protection schemes which are not provided as yet in readily available hardware. Foremost amongst these, Mondrian Memory Protection (MPP) [141] introduces a hardware architecture for protecting memory segments up to word level granularity, so that extremely fine grained control over individual memory segments is possible. MPP works by maintaining a permissions table, and works in a similar fashion to current page table architectures. MPP has a concept of a Protection Look-aside Buffer, which is analogous to a Translation Look-aside Buffer in conventional memory management units. The processor is responsible for scanning the permission table for each memory access. However, while the mechanisms suggested in MPP are extremely interesting, such hardware does not exist as yet [132].

A similar hardware based proposal for enforcing fine-grained memory protection and protected procedure calls was made by Wiggins et al. [139], but it suffers from the same problem as MPP. In addition, Chiueh [17] proposes a novel technique, which is the use of hardware debug registers to detect buffer overflow attempts. However, the solution deals exclusively with buffer overflow detection only, and can be considered a building block for higher level mechanisms.

2.3.3 Integration into OS Kernel Isolation Facilities

Attempts have been made to enforce isolation through integrating separate application level components with the OS kernel Application Programming Interface (API) layer.

2.3.3.1 Wrapping

One such example is the use of "wrapping" techniques. Wrapping involves the verification of all parameters passed between a containing host and its extensions [127]. In the Nooks architecture [127], an amalgamation of techniques such as hardware memory protection, software fault isolation and privilege lowering along with kernel wrapping are used to prevent device driver failures. Each device driver is carefully wrapped by a proxy which is responsible for fault isolation and recovery [127]. The wrapping process involves hand-crafting a software strait-jacket which proxies all calls made to the driver. The system has been further enhanced by Swift et al. [125] to also enable recovery of failed device drivers, by carefully playing back logged requests. However, Tanenbaum et al. [132] points out that attempting to write a wrapper around each device driver is an error-prone and painful process, hampering the adoption of the technique. Further, Erlingsson et al. [31] point out that the protection offered by Nooks can be easily circumvented by malicious code.

Peterson et al. [102] describe a generic operating system API for creating sandboxed programs, where each sandboxed process runs in a separate address space. Their work lends support to the need for making components an integral concept within the operating system, as argued by Mendelsohn [93]. An interesting implementation is also made in the "Go!" Operating system [79]. Instead of weaving in and out of kernel mode and user mode, the Go! component-based OS works entirely in kernel mode. Analogous to a traditional kernel, Go! has an Object Request Broker (ORB), which arbitrates communication between components [79]. To protect components from interfering with each other, and to prevent them from executing privileged instructions, the entire program is scanned for privileged instructions before being executed. The ORB is responsible for doing this and delegates the task to a helper component. Once the component is deemed to be safe, the ORB will allow its instantiation. Certain trusted components, such as interrupt service routines, are

exempt from this verification process. By eliminating protection domain switching, the Go! OS manages to achieve extremely high performance [79].

2.3.3.2 Process containers

Process Containers provide a means of executing a program in a sandbox, typically by building an isolation container on top of the Operating System's process isolation facilities. Such process containers provide restricted access to resources for the sandboxed application, for example, by providing a limited view of the file system.

Examples include such systems as chroot() [36], which provides a limited view of the file system to an untrusted application, FreeBSD Jails [73] and Solaris Zones [105], which provide an operating system level virtualization environment for applications. Linux Security Modules such as AppArmour [9] and SELinux [87] also enforce administrator defined security policies in applying fine-grained restrictions to the capabilities of processes.

A common approach to creating such a container is via system interpositioning [39, 40, 42, 101], which typically relies on a kernel level trace mechanism (e.g., *ptrace* [70]) to intercept the system calls executed by a process. It is based on the observation that an isolated process has no way to influence the system at large, other than through system calls [42]. Therefore, by restricting the execution of system calls, or modifying them before execution, an application can be made secure.

Some of these mechanisms required kernel level modifications [34, 39, 87, 102], with the advantage of being able to add all facilities as required but having the drawback of making widespread deployment difficult. Others were built on top of existing facilities such as *ptrace* and */proc* [70, 85], with the drawback of having to retrofit functionality onto available mechanisms. These systems sometimes resulted in timing and race related bugs such as TOCTOU (Time of Check To Time Of Use) [39], but workarounds have been implemented [101].

One of the chief drawbacks is high overhead due to the repeated context switches that are required [39]. While Kernel based mechanisms can reduce this overhead, they must also run an additional risk in doing so, as any exploitable bugs in the mechanism would result in a total compromise of the operating system. User space interception on the other hand, can attract even heavier overheads. In addition, user space mechanisms such as *ptrace* provide limited ways with which to manipulate the

confined application, making the technique more complicated to employ in library level isolation (as discussed in Chapter 3).

2.3.4 Binary Code Level Isolation

Binary code level isolation relies on modifying the application binary at load-time or run-time, in order to insert additional checks and guards for ensuring isolation.

2.3.4.1 Software Fault Isolation

This method was first described by Wahbe et al. [137] and the basic technique has been utilised in many forms. The authors make a strong case against placing software modules in their own private address space, as this would require a Remote Procedure Call (RPC) between them for communication, resulting in unacceptable context-switch overhead. Context-switching overhead is one of the chief reasons that such RPCs have unacceptable performance overheads [137]. For example, a single RPC from one process to another requires four context switches, two on making the call and two on return. However, considering the way in which operating systems are structured, the only way to protect two components from each other is to place them in two separate processes and to make an RPC call between them.

Wahbe et al. [137] describe an alternate approach. It enables you to place untrusted code within the same process and avoid the overhead of making an RPC. To ensure protection, the system uses software-based verification to ensure that no illegal memory accesses are made. The main technique used is to verify the object code of the distrusted module through static analysis, and inject code for double checking any potentially harmful instructions. A "sandboxed" code version is created so that memory references always fall within the sandboxed region, thus preventing a component from accessing memory outside of its bounds [137].

Software Fault Isolation (SFI), originally demonstrated by Wahbe et al. on a Reduced Instruction Set Computer (RISC) architecture, has also been demonstrated on Complex Instruction Set Computer (CISC) architectures [90]. Techniques such as binary translation [1] are offshoots of the ideas in SFI.

The ideas in SFI are directly utilised in the Google Native Client (NaCl) software system, which provides a software framework for safe execution of untrusted binary components [144]. NaCl aims to provide browser-based applications access to

increased computational performance through native binary components which have access to performance-oriented features such as Streaming SIMD Extensions (SSE) instructions, compiler intrinsics, hand-coded assembler, etc., without compromising on safety [144].

SFI techniques have also been used in Nooks in combination with hardware support, in order to create an architecture for device driver fault isolation and recovery [127]. Fraser et al. [34] describe similar protection mechanisms for "Commercial Off-the-Shelf (COTS)" systems. Kumar et al. [76] describe the use of SFI in embedded systems, where hardware support for protection domains is often absent. In addition, Small and Seltzer [121] have estimated the performance characteristics of various techniques, and conclude that SFI-based techniques offer good overall performance.

A strong example of such software-based techniques providing better performance than corresponding hardware protection comes from Adams and Agesen [1]. Through their experience in implementing the popular VMware virtual machine monitor, they provide performance measurements which indicate that hardware assisted techniques can be overshadowed by Binary Translation techniques.

Swift et al. [127] point out that it may be difficult to implement SFI when the range of addresses are not contiguous. Further, although it is relatively cheap to call into SFI code as opposed to a protection domain switch, the SFI code itself executes more slowly due to the additional checks.

2.3.4.2 Static analysis of binaries

While SFI and "Static Analysis" of binary code are closely related, we differentiate techniques which rely on a preventive approach, where the code is statically analysed to determine whether a program violates its safety contract, and not allowed to execute at all if it is found to do so. Typically, a verifier is used to perform a static analysis of the untrusted machine code program. The machine code itself can be directly passed to the verifier, requiring no modifications to current development methodologies.

The main advantages of this method, as identified by Xu et al. [143], are as follows.

1. It operates directly on binary code, allowing the freedom to choose any source language for program development

2. It provides the ability to extend the facilities offered by a host program at a very fine-grained level, in that the "foreign" code is allowed to manipulate the internal data structures of the host directly
3. It enforces a default collection of safety condition enforcement schemes to prevent array out-of-bounds violations, address-alignment violations, use of uninitialised variables, null-pointer dereferences and stack manipulation violations.

Xu et al. [143] utilise this technique to verify untrusted machine code. However, the program calling the extension is now required to send in type-state information and linear constraints in order for the verifier to work. The type-state information specifies the expected pre and post conditions of the calling program. The verifier uses this information in order to determine whether the machine code satisfies the given constraints, thus ensuring its safety.

In addition to passing in the type-state information, the caller may impose a custom security policy, which makes the method very flexible [143]. However, the requirement that the caller supply such type-state conditions requires a radical change in invocation semantics. Further, the verifier performs a rather complex static analysis of the program, which requires a significant amount of time per extension. In addition, compiler level modifications are required for the technique to work. However, Erlingsson et al. [31] have attempted to address this problem by using control flow analysis and a binary rewriter which ensures that all expected properties and guards continue to hold.

Overall, static analysis could be deemed a preventive measure and mainly be used in determining whether a given component obeys certain constraints before its execution. Thus, it could be used to prevent a faulty component from being loaded in the first place, but not to prevent failure during the execution of the component. In addition, static analysis is also prone to false positives.

2.3.5 Language Support

Language-based protection relies on the safety of a language's type system, where the operations that a program performs can only be operations that are deemed sensible for that type [52]. Typically, this will involve both a dynamic and a static access control mechanism to be available. Static checks can be made at compile time to verify that illegal operations on a type are not permitted and dynamic checks may be needed at runtime, for example to perform an array bounds check [52].

2.3.5.1 Type-safe code

The SPIN operating system utilises *"Modula-3"* as a type-safe programming language along with a trusted compiler to create type-safe extensions [11]. A more modern example of the use of type safe code for component protection is the *"Singularity"* operating system, a prototype operating system created by Microsoft Research [2]. The key philosophy behind Singularity is the concept of a Software Isolated Process (SIP), which, unlike traditional hardware-based process isolation, relies on static type checking and language safety rules to ensure protection between processes. The results indicate that compared to the 25-33% overheads that hardware-based isolation incurs, SIP only incurs as little as 5% overhead in the benchmark tests [2].

However, the major drawback in Singularity is that all software components would have to be rewritten in a type safe language (in this case a Microsoft ".NET" compatible language) in order for the scheme to work, making it unsuitable for the large base of existing applications [2].

In addition, Swift et al. [127] point out several difficulties in the adoption of type-safe languages. The major issue is the problem of rewriting all drivers in that type safe language and the significant overheads in copying data in and out of a driver. Further, they also mention that an elegant mechanism for accessing device drivers in a type-safe fashion is not available. Further, Dean [26] points out that building "bullet-proof" implementations remains a difficult problem.

2.3.5.2 Static analysis at language level

Static analysis of code allows the analysis and verification of a program prior to execution. The emphasis here is similar to the case of static binary analysis – ensuring that the program conforms to safety properties prior to execution.

For example, in order to deal with common buffer overrun situations, a combination of static analysis techniques and SFI can be used. Cowan et al. [23] provide an overview of common techniques used to combat buffer overflows. Apart from defensive programming techniques, of interest are tools like purify [51], which use object code insertion to instrument all memory accesses and StackGuard [22], which uses a canary value to detect whether a stack smashing attack has been attempted, for which a hardware based mechanism has also been mentioned previously [17].

Another approach is that of the Proof Carrying Code (PCC) technique, whereby an automated proof generator is used to analyse each program and attach a proof that the program will execute within its defined boundaries [100]. In essence, PCC involves statically checking program code and the automated generation of a proof that the program obeys a given safety policy. At runtime, the proof can be verified by the operating system. One major advantage of this technique is that no run-time checking is required, since a program passing the verification process is guaranteed to be safe.

However, writing a comprehensive proof generator which can deal with the complexities of optimised code remains a problem in this suggested solution and so far, the technique has not been demonstrated with non-trivial examples [132]. An additional difficulty is that the policy needs to cover any implied rules of the execution environment. Guaranteeing the completeness of the policy itself is also difficult [38].

2.3.6 Application Level Isolation

"Application Level Isolation" is a technique that involves isolation being enforced entirely in user address space and being managed by the application itself.

2.3.6.1 Interpretation and intermediate language compilation

Interpretation based isolation of application and like programs has been categorised under application level support, as it is usually performed entirely in user-space. Interpretation techniques have been used in a variety of scenarios. These include complete virtual machines such as the "Java Virtual Machine (JVM)", the earlier UCSD "p-System" [14], etc. or scripting languages such as AWK, Tcl, etc. Interpreted languages have shown excellent safety properties and can be made extremely secure [132]. For example, the JVM contains a built-in verifier that provides several safety checks to ensure that no forged pointers or pointer manipulations can be performed, effectively preventing code from accessing unauthorised memory locations [16, 132]. A security management policy can perform fine-grained control over each running thread.

The major drawback of interpretation techniques is speed. Although a great deal of optimization work has been performed, such as "Just-In-Time Compilation (JIT)", to dramatically boost the speed over simple interpretation, the overheads imposed by the use of constant checks continue to be very significant, and loss of performance still continues to be a concern over use of pure binary code [132].

A further problem is that not all programs can be written in an interpreted language. At some point, especially when high performance or low level device access is needed, it is necessary to fall back into lower level languages for software development. For example, the JVM uses the "Java Native Interface (JNI)" in order to access hardware specific features. Once the JNI barrier is crossed, the JVM is entirely at the mercy of the loaded C-language library [16] in the called component. A JNI environment pointer is passed into the JNI library in order for it to communicate. A badly written extension Dynamically Linked Library (DLL) element can easily crash the entire JVM, as they all reside within the same address space. To address this issue, Czajkowski and Dayn [24] propose isolating JNI components in separate address spaces. This however, incurs the usual context

switching and parameter marshalling overheads for inter-process communications (IPC) calls, incurring a significant penalty on the additional robustness provided.

Lastly, there is a large existing base of code that is in binary format. It is not feasible to rewrite all of these programs in an interpreted language [132], and as mentioned above, certain actions require low-level hardware access, posing a security risk to the integrity of the virtual machine (VM). However, since a large class of applications can be solved using this approach, there continues to be a massive surge in its popularity, particularly in the commercial and business areas.

Further, Small and Seltzer [121] argue that interpreted technologies are not suitable for building kernel extensions, since the timing granularity in systems events are extremely fine and the performance requirements more exact.

Another component isolation model and technique scheme is known as the "multi-process application architecture". This model is becoming increasingly popular in web browsers [8, 109]. The basic idea is to isolate individual components into private address spaces through disparate OS processes and use the operating system's IPC mechanisms to communicate between them. In Google's *Chrome* browser, a single browser coordinating process spawns additional processes to perform sub tasks [133]. These additional processes run at a lower privilege level and access is tightly arbitrated by the coordinating browser process. In effect, different components are loaded into different processes and communication takes place using OS supplied IPC mechanisms. This isolation into separate processes allows the browser to survive component crashes. Microsoft's *Internet Explorer 8* follows a similar model [145]. There is however, an increase in complexity as coordination between several processes is required. Also, Wahbe et al. [137] make a strong case against placing software modules in their own address space, as this requires IPC between them for communication, resulting in unacceptable context-switch overheads [137], so a trade-off is made between performance and reliability [133].

2.4 CONCLUSION

This literature review in this chapter provides a working definition for the term "component", as suited to the purposes of this thesis, as a dynamically loaded shared object library (DLL or SO). The pressing need for such library level isolation was

also highlighted, especially in view of the failures caused by third-party plugins. Common causes of failures as well as a taxonomy for isolation mechanisms were introduced. These were – hardware isolation, binary code level isolation, OS kernel level isolation, language level isolation and application level isolation. The currently available isolation mechanisms, and their strengths and drawbacks, were also discussed. In addition, existing efforts to isolating components from each other have also been analysed, and some common drawbacks are as follows.

- They do not maintain address space transparency.
- Require rewriting libraries as well as host applications.
- Require custom tool chains.
- Not general purpose.
- SFI based techniques require code verification and patching – large TCB.
- Restrictions on allowed machine instructions.
- False positives.
- Performance issues.
- Limited to certain hardware architectures.

The remainder of this thesis builds on this discussion and the next chapter introduces the isolation architecture utilised by our LibVM system.

Chapter 3: Isolation Architecture Design and Rationale

3.1 INTRODUCTION

Following from our analysis of the scientific, technological and engineering literature, and associated research outcomes and system offerings, in Chapter 2, this chapter provides an overview of the system architecture we developed to address the main research questions covered in this thesis. We begin by arguing for a generic Application Programming Interface, named LibVM, which encapsulates the functionality necessary for library isolation when used in an application program, and provides a specification for such an API. We then describe the design decisions that motivated the various architectural choices, and relate them to the overall problem of the need to isolate shared libraries, for improved robustness and security.

3.2 AIMS OF THE ARCHITECTURE

A key research aim in this thesis is to isolate untrusted shared libraries from their host program, so that, in particular, a badly written or misbehaving component will have minimal impact on its host, thus improving overall system robustness and resilience as well as guarding the host against potential security issues. However, untrusted libraries may include genuinely malicious code, masquerading as a useful library entry. For example, malicious web browser "plugins" would fall into such a category. Isolating libraries into sandboxed environments would thus minimise the threat posed by such a plugin, increasing safety. In addition to such malicious libraries, non-malicious yet potentially "buggy" libraries are also suitable candidates for secure isolation, since any unwitting programming errors in a non-critical component should not cause a wholesale crash of the entire application. For example, a buffer overflow vulnerability, which could enable a stack smashing attack, can be rendered ineffective because the component is confined to its sandbox. Ideally, failures in non-critical components should be recoverable, perhaps through a simple reload/replacement or reset of the faulty component, thus increasing overall system resilience.

Therefore, any isolation architecture should aim to:

1. Create protection domains for individual components, so that the interaction between the component and its environment can be tightly controlled.
2. Prevent an errant component from corrupting critical memory regions or having adverse effects on the hosting process.

3.3 THREAT MODEL

As explained above, isolation can be analysed from both resilience and safety/security perspectives. While resilience is the key reason for isolation in a trusted environment, safety becomes paramount in an untrusted one. This is best exemplified by Internet browsers which extend their functionality through plug-in components. In such an environment, both these factors are very important, as the extension code may be of unknown provenance and quality.

Our system, LibVM, is designed to deal with arbitrary binary components, from untrusted sources, which need to be executed in a constrained environment. Once a component is accepted for execution, it must have controlled access to resources, as determined by the host. Access to memory must be restricted to areas allowed by the host application and attempts to exceed these limits must be caught. The host must be able to constrain the component by preventing arbitrary access to the full operating system call interface. Such access must be mediated and the host must be allowed to set resource limits on memory usage or disk / storage access.

3.4 LIBVM – A FUNCTIONAL SPECIFICATION

In order to demonstrate that an implementation of LibVM is correct in achieving its functional intent, we must first establish the following.

a. A clear specification of its functional requirements.
b. A methodology for extracting the conditions under which an implementation can be considered correct with respect to that specification

We discuss each of these in turn.

3.4.1 Overview of Functional Specification

The stated goal of LibVM is to isolate a library from its host application, so as to ensure that the host's (and the underlying system's) security and integrity cannot be violated. However, since security and integrity are broad terms, we narrow them down to a very specific meaning. We require isolation such that a library component cannot affect its host (including the host's underlying environment), without the host's explicit knowledge. More specifically, by "affect", what is meant is that the library component cannot change the data or control flow in the host, nor its environment, without the host being explicitly party to the fact, and thus in a position to permit or reject that effect (This is analogous to a "non-interference" security policy model [122] or other multi-level security policies such as the Bell-LaPadula model [25], although we do not use the related nomenclature as we do not such strict information flow policies. However, stronger policies could be implemented by restricting the conditions that are identified even further, such that they are conformant with such multi-level security policies).

If the library can have no effect on its host, without the host's knowledge, we contend that it is sufficiently isolated. Since the host is aware of any effect that can potentially alter its control/data flow or environment, it can determine its own security policy. This is the desired improvement over the prevailing norm for libraries, which is that library components are free to execute unfettered within the host's address space.

Therefore, what LibVM provides is a security mechanism, not a security policy. The LibVM isolation container cannot guard against incorrectly implemented security policies in the host. Nor can it guard against bugs in the host application. For example, if the host application allows a system call made by a library component in order to access the network, and the host does not adequately check whether the library component should be performing network communications in the first place, it is a failure in the host's policy. Similarly, if the host suffers from a bug where it unwittingly performs a jump into a location within the isolation container's boundaries, LibVM has no ability to guard against that situation. It is only the host's own correctness that must guard against this. However, in Chapter 7, we discuss techniques for reducing the burden placed on the host, so that it can enforce its security policies with greater ease.

In addition, it is also possible for the component to starve the host of resources. For example, it could sit in a tight loop eating up CPU cycles, or, if so allowed by the host, allocate memory or disk space without bound. However, we assume that the Operating System enforces resource limits to guard against such possibilities, in addition to the host itself being prudent in allocating resources to the component. Furthermore, covert channels for leaking information may exist [78]. However, we contend that these issues are orthogonal to our main goal of preventing an errant component from compromising the host's integrity.

3.4.2 Methodology for developing correctness conditions

The methodology we use for extracting the conditions under which a LibVM implementation can be valid, is as follows. The strategy is to treat the security specification as an invariant that must be preserved by each feature of the implementation.

a. Start with a minimal, totally isolated container which does nothing, and therefore can have no effect on its host, which conforms to our security specification by definition.

b. Progressively add new features which are necessary, using key security principles as guidelines.

c. Ensure that each added feature continues to conform to our specification (the invariant).

d. If it does not, add a new "security requirement/condition" which will guarantee that the invariant will be preserved.

e. Repeat till all essential features have been added.

During this process, since the identified conditions ensure that the specification is met at each step, we know that the final set of conditions will meet the specification in aggregate. Therefore, we present these final set of conditions as those that must explicitly be met in order to be conformant to the specification.

While the method outlined above works well for conceptual features, it becomes far more difficult to follow as features become embroiled in the complexity of actual implementation details, rendering the process itself very tedious and time consuming.

As a remedy to this issue, we utilise principles from the Common Criteria as a guiding framework by which confidence can be gained in a concrete implementation.

3.4.3 Requirements for Correctness of LibVM

In order to provide the assurances of correctness that the specification above defines, we show that five propositions must hold. We establish these five propositions by following the methodology outlined above. It should be noted that some of these propositions are reliant on conditions outside of the control of a LibVM implementation. We flag these external conditions as appropriate, but include them in our overall list of conditions, as all of these must be satisfied for the specification to hold. In the process of doing this, we always enforce the well-known principle of "granting least privilege" [114], giving the isolation container only the minimal privileges it needs to accomplish its tasks.

We start with a hypothetical isolation container "C", which is totally devoid of a CPU or memory. In other words, C can do nothing, as it can perform no computation. Such an isolation container is the epitome of total isolation, and implements the principle of least privilege best, since it cannot affect the host or even itself in anyway. Therefore, it conforms to our specification, by definition. However, such an isolation container is also of no utility whatsoever.

3.4.3.1 Deriving Condition C1

In order to remedy the uselessness of the container described above, we introduce a Central Processing Unit (CPU) to this container. This CPU may contain some internal, volatile registers, but no other memory apart from this. When we introduce this CPU, we must ensure that it remains separate from the host's CPU in order to arrive once again at total isolation. If it is totally isolated, it once again meets our specification by definition. For this total separation to occur, two conditions must clearly hold.

Lemma A: The CPU (including registers) must be fully virtualised, so that C will continue to be unable to affect its host H.

Lemma B: Conversely, the host H cannot access the registers of the isolation container's CPU.

If either of the two conditions is not met, information leakage clearly occurs between the host and the isolation container.

In order to clarify the importance of the above two conditions further, let us assume the inverse of these conditions, such that a thread of execution T within a LibVM container is not virtualised so that it can affect the CPU state S in a host process H. It is possible that T could then succeed in either of the following.

a. Affecting a segment register and thereby causing the host H to access an erroneous memory location. (For example, by changing the CS (Code Segment) register, the thread T could offset the host's instruction pointer by a desired amount, thus causing it to execute arbitrary code.)

b. If T leaves a residual value in a register S, and if the host unknowingly uses this value, it is possible for T to affect the execution of the host

These are contradictions to our specification that our host H cannot be adversely affected by T. While it is not the case that allowing access from T to host CPU state S will always result in a vulnerability, we nevertheless adopt the more restrictive condition of completely disallowing access to the host CPU's state, erring on the side of caution and safety. This is in keeping with the principle of "using the least common mechanism" [114], which aims to minimise shared information paths.

Thus far, we have defined an isolation container C which is far more restricted than the one defined by LibVM. Clearly therefore, this isolation container is also inherently more secure than LibVM, due to the reduction in the size of the "attack surface" [58, 88]. We therefore utilise this intermediate result of Lemma A, as a suitable invariant that must hold from now onwards till we get to the functionality level of LibVM, while Lemma B will be relaxed (while preserving our desired properties) for reasons explained later.

Therefore, we promote Lemma A to our first condition:

The CPU utilised by the LibVM container must be virtualised such that it is isolated from the host application.

3.4.3.2 Deriving Condition C2

Although the isolation container described above is marginally more useful than one that achieves nothing at all, as it can perform some limited computations, any useful

result is still inaccessible to the host, due to Lemma B. Furthermore, the host is also unable to pass parameters to this CPU to invoke a desired computation, since the registers are inaccessible. Therefore, we are required to relax this restriction, while ensuring that our specification continues to be met.

Therefore, we allow the host access to the isolation container's CPU registers, and observe that there are only two possibilities that emerge.

a. The host writes data to the registers

b. The host reads and subsequently uses data from these registers

In order to deal with these two possibilities in a way that does not affect our invariant, we qualify that access with two more conditions.

Lemma C: Any information placed in the CPU's registers must be guaranteed to be non-sensitive data, so as to prevent data leakage.

Lemma D: Any information retrieved from these registers must be treated with the utmost suspicion, and always checked for validity before use.

By stipulating the above two conditions, we make access to the CPU registers possible, while preserving our desired safety properties, although it should be noted that data flow safety is entirely in the hands of the host. This is in keeping with our originally stated specification.

However, we argue that Lemma C is not essential to our purpose, since placing data into the CPU registers of the isolation container is a deliberate decision, and while such data can affect the execution flow within the container, they cannot affect execution flow, data flow or resources external to the container. Therefore, Lemma C is outside the scope of our specification, although it would be important if even more secure data flow semantics are desired.

On the other hand, the absence of Lemma D can affect the conditions outside of the container and we provide a practical example of how the integrity of the host could be potentially affected. Suppose that we utilise a value obtained from one of these registers without adequate circumspection. For example, a value returned by a thread of execution within the isolation container is used to index into a memory array. Should that return value be illegal, it would be possible to overwrite or corrupt host memory, including triggering a segmentation fault which could crash the host.

Or consider an even more careless situation in which this value is used to index into a jump target. In such a scenario, the execution flow of the host could be altered through this illegal value, forcing for example, a return-to-libc attack [23].

Therefore, Lemma D gives rise to a more general condition, C2:

A host must never utilise a value obtained from within the isolation container without ensuring that it is adequately validated.

However, it must be explicitly noted that validating these values is a responsibility which is delegated to the host application itself, and cannot be implemented by a LibVM implementation, as the legal range of values are meaningful in the context of the host application only. However, this can also be considered a separation of policy and mechanism, as the security policy must be enforced in a meaningful way by the host application, and LibVM only provides the mechanisms with which to do so. Therefore, as with any security mechanism, an incorrect security policy could result in exploitable vulnerabilities.

It should also be noted that this places a considerable burden on the host application as careful validation of values is required, especially when dealing with potentially malicious components. We discuss ways of mitigating this burden through techniques such as compiler support and predefined security templates, in Section 7.3.

We also observe that this expansion of rights to include host access to CPU registers, was done in full accordance with the principle of granting least privilege [114], and that the additional privileges proffered to the isolation container are indeed minimal. We also continue to preserve our invariant, in concordance with our specification.

3.4.3.3 Deriving Condition C3

The isolation container discussed thus far can perform rudimentary computations, but is still unable to perform extensive ones, due to the absence of any sizeable volatile memory, apart from the CPU registers. We now introduce this memory. Once again, to preserve total isolation, we would have to ensure that this memory is entirely separate from the host's memory with the following conditions.

Lemma E: The CPU within the isolation container cannot access memory outside of its allocated regions, in accordance with condition C1.

Lemma F: The Host cannot access memory within the isolation container.

The above two conditions would continue to keep all our conditions intact, since the introduction of the above disallows interaction between host and container.

Once again, it is possible to show concrete examples of how the absence of the above conditions could affect the host. If a thread of execution T within a LibVM container, can access a memory location M outside of its boundary, it trivially follows that it can compromise the host's integrity, since it could potentially write to memory locations used or accessed by the host. In the most innocuous scenario, it can result in data leakage, or perhaps an attempt to access the host's memory could result in accessing an unmapped region of memory, causing a page fault and causing thread T to crash with no significant effect on the host. However, if it is a mapped page, it could cause data corruption in the host's memory. In the most dangerous scenario, it could trigger a buffer overflow [23] or otherwise induce the host to execute arbitrary code.

Lemma F could be similarly demonstrated in practice by the same logic used in demonstrating Lemma B previously.

We can therefore readily promote Lemma E to a required condition, since allowing the isolation container to access data outside of its regions is far too dangerous, as shown above. Consequently, our next condition C3 is:

A thread of execution within a LibVM container cannot access memory regions outside of its allocated regions.

However, Lemma F is too restrictive for the same reasons that Lemma B was too restrictive, which is that the host cannot access useful results of computations made by the container. Being able to access and share this memory with the host is therefore a desired characteristic of LibVM. However, allowing access to this memory results in the same weaknesses as those caused by allowing access to registers, since registers are also a form of memory. Therefore, it trivially follows that lemmas C and D apply in this situation, and by the same process of argumentation, gives rise to the same condition C2. Condition C2 has already been sufficiently generalised to apply to both memory and registers. In other words, Lemma F can be discarded in favour of Condition C2, which gives us greater flexibility while preserving our desired properties.

3.4.3.4 Deriving Conditions C4 and C5

Finally, we add one more necessary capability to our hypothetical isolation container - the ability for the isolation container to trigger a domain transition in order to request additional required (and allowed) resources from its host. We refer to this transition as a "host call", analogous to a system call made by the host process itself.

Several steps are involved in this process

a. The isolation container must trigger a domain transition indicating that it wishes to avail itself of a particular host call.

b. The host must resume execution from a predefined, safe call gate (the isolation container cannot be allowed to make the host resume from an arbitrary location).

c. The host must obtain the parameters required for the requested host call from the isolation container's memory (or registers).

d. The host must execute this request, once it has determined that the requested action is "safe".

e. The host must place the results in the isolation container's memory or registers.

f. The host must return control to the isolation container so that it can resume execution by triggering a domain transition.

Step a above does not affect our isolation objectives. It is merely a step in the mechanism of a host call. In contrast, step b has implications for our isolation objectives, as the host must resume from a safe location, and not an arbitrary one triggered by a thread of execution within the Isolation Container. However, we assume that this too is an integral part of the transition mechanism.

Step c accesses the isolation container's memory. Such accesses are already covered under Condition C3, which states that all such accesses must be thoroughly validated.

Step d involves determining whether an action is "safe" before execution. At this point however, complexity erupts. Of the resources discussed so far (CPU and memory), the isolation container has always been provided private "copies", thus maintaining isolation. However, any new resources requested at this point may well

be shared resources. For example, a library executing within the isolation container may request to write to a file. This file may potentially be accessed by the host itself or some other process in the system. The library execution within LibVM may proceed to corrupt this file, affecting the host or the rest of the system.

Therefore, our isolation guarantees are most subject to vulnerablity during the execution of this step, due to the sheer number of possible actions. LibVM itself cannot determine a priori what actions are safe, since it is the host that determines what the host call interface is. As a result, this decision must necessarily be deferred to the host. Therefore, the same caveats which applied to condition C2 apply here.

However, we observe that "an application can do little damage if its access to the underlying operating system is restricted", which is the core assumption behind system call interpositioning based isolation mechanisms [42]. Since a component is a subset of an application, it follows that if the components access to both the host and the underlying operating system can be restricted, the damage it can cause will be minimal. Therefore, at the minimum, LibVM provides defence in depth, by adding an extra layer in which the principle of complete mediation can be exercised [114].

By looking at the above requirements, we formulate two more conditions, which are

C4: The domain transition mechanism provided by LibVM must not compromise the host's integrity.

C5: An action executed by the host on behalf of a library isolated within LibVM must not compromise the host's own integrity.

These two conditions encompass all the steps above.

We consider both these lemmas to be full requirements in order to maintain our invariant that the host's integrity cannot be compromised by a library executing within a LibVM container.

3.4.3.5 Final Conditions

Through a process of incrementally expanding the abilities of a LibVM container, as outlined in our methodology, we have carefully derived the conditions under which a library executing within that container can be guaranteed to be unable to compromise its host. This incremental process relied on many security principles, including the

principle of least privilege [27, 114], where at each step, we provided the minimal abilities necessary to accomplish the actions executing within that step and added conditions which ensured that our target invariant would be preserved. All of these conditions taken together therefore, present the necessary and sufficient conditions to guarantee that a library executing within a LibVM container cannot compromise the integrity of its host.

The five propositions that were derived are.

Condition 1 (C1): The CPU utilised by the LibVM container must be virtualised such that it is isolated from the host application.

Condition 2 (C2): A host can never utilise a value obtained from within the isolation container without ensuring that it is adequately validated.

Condition 3 (C3): A thread of execution within a LibVM container cannot access memory regions outside of its allocated regions.

Condition 4 (C4): The domain transition mechanism provided by LibVM must not compromise the host's integrity.

Condition 5 (C5): An action executed by the host on behalf of a library isolated within LibVM must not compromise the host's own integrity.

It should be noted that conditions C2 and C5 are outside of the control of the security mechanisms provided by LibVM, and must be guaranteed by correct implementation of security policies by the host, as discussed in Section 3.4.3.2. Nevertheless, both of these conditions must be fulfilled for assurance of correctness.

Next, we discuss the architecture which can be used to realise this .functional specification.

3.5 LIBVM ARCHITECTURE

The basic architecture of LibVM consists of shared library isolation domains, each of which can contain multiple shared libraries, as depicted in Figure 3.1.

This model enables individual domains to have different isolation policies, depending on the level of trust awarded to the shared library. Domains can also contain multiple shared libraries, allowing an isolation policy to be shared.

From an application developer's perspective, LibVM is a sandboxing library that can be used to define such an isolation domain and to load additional shared libraries into it. One significant assumption in LibVM is that it provides address space transparency, in that pointer values can be freely passed from the containing host to the isolation domain and vice versa. (The rationale for this design decision is laid down in Section 3.6.3.) Therefore, the developer must specify the size of the reserved address space range in advance. Although this specified address space is initially only reserved and not actually used yet, it does have the restriction that it cannot subsequently be relocated, since all pointers within this address space would need to be adjusted.

Once a shared library is loaded, the methods in the library can be invoked in a manner analogous to the POSIX-based mechanism used in most UNIX systems. When a method is invoked, a controlled transition must be made into the isolation container, and the code then executed within it, with any return values returned to the host. The shared library is in turn free to make additional system calls, all of which can be intercepted by the host and proxied as desired by the programmer.

Importantly, LibVM's isolation domain separates "policy" from "mechanism" by defining an interface which abstracts away the details of the specific implementation. In order to test this, we have created two separate implementations, one based on hardware virtualization support and another based on shared memory and *ptrace*-based system call interpositioning. However, this chapter focuses on laying out the details of the interface in the abstract, with concrete implementations being discussed in chapters 4 and 5.

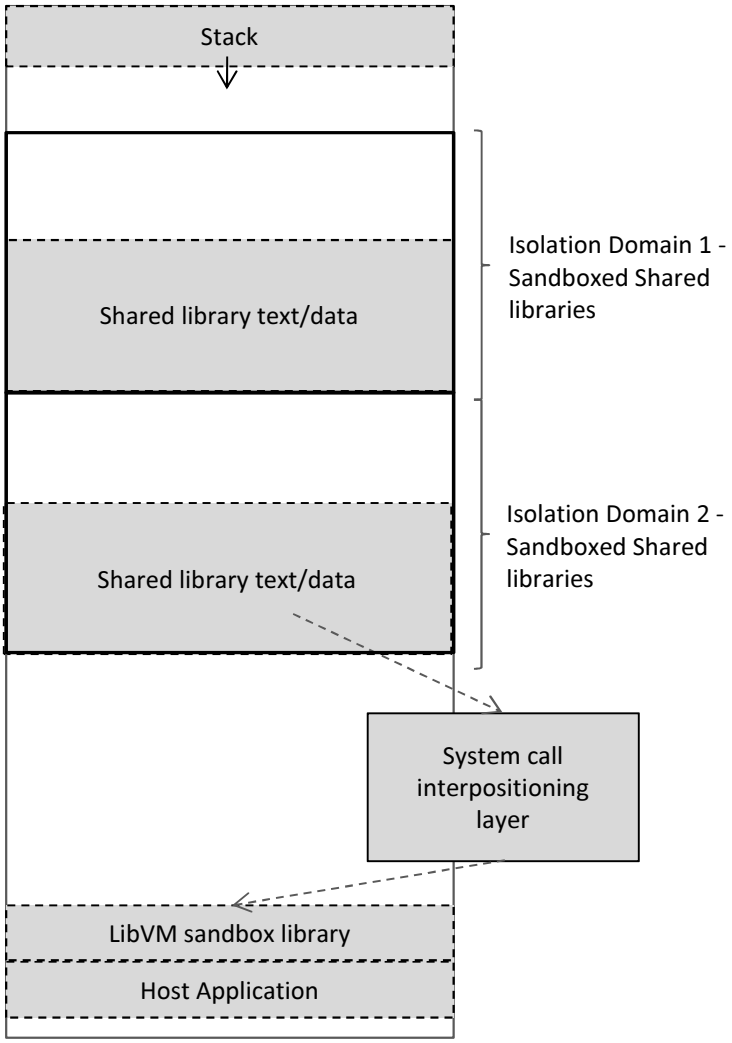


Figure 3.1: Process memory layout in a program which utilises LibVM

3.6 KEY ISSUES WITH A LIBRARY ISOLATION ARCHITECTURE

In this section we outline some of the rationale underlying the design of our LibVM architecture.

3.6.1 Changes to Existing Development Practice

In order for a library isolation architecture to be useful in practice, it is also necessary for it to be similar to existing shared library manipulation mechanisms, so that isolating shared libraries in separate domains does not require a significant re-engineering effort. Having to rewrite shared libraries in their entirety would be detrimental to the practicality of isolating libraries and could easily be rejected on economic and business grounds. Therefore, the library code should remain largely

"as is" and ignorant of the fact that its components are executing within an isolated domain.

The hosts too should require minimal change, and in keeping with this requirement, the API is designed to closely mimic the POSIX API for shared library manipulation, so that host applications can be ported to use the isolation API with less effort. In this way, the additional software effort is limited to the host program.

3.6.2 Implementation Agnosticism

The isolation API separates policy from mechanism by defining an interface which abstracts away the details of the specific implementation. In order to test this, we have created two separate implementations, one based on CPU hardware virtualization support and another based on shared memory and *ptrace*-based system call interpositioning (Other mechanisms are possible, such as an RPC-based mechanism which isolates components into different operating system processes). This frees the application developer from being tied to a specific implementation, allowing variation in the mechanism chosen, based upon security requirements.

3.6.3 Shared Address Space

Given that support for shared libraries is generally provided at an operating system level, it stands to reason that isolation of those libraries is also most easily implemented at an operating system level. Mendelsohn [93] argues that "components" should be a "first class" concept in operating system development and deployment. Unfortunately, this has not come to pass, for two key reasons.

a. Software components in shared libraries are usually designed to share the address space of their hosts, in order to make data sharing between the host and the library component simpler. However, due largely to the unavailability of a hardware mechanism to simultaneously share address space and constrain instruction execution within an isolation boundary, few attempts have been made to isolate libraries from the hosting application (Exceptions to this, such as Google's Native Client, Vx32 etc. have been reviewed in Chapter 2).

b. There are significant performance costs in switching between address spaces [86].

With regard to the first disadvantage, the main issue is that pointers have no meaning beyond the address space in which they were originally created, requiring that they be specially manipulated or readjusted before use. Consider a simple data structure with a pointer to another data structure, such as a linked list. Any attempt to pass this data structure unmodified between private address space boundaries would render the data structure invalid, as the memory address to which the structure is copied cannot be guaranteed to be the same. All pointers within the structure would have to be readjusted so that they remain valid in their new location. Therefore, some additional technique is needed to share data structures between address spaces, such as marshalling or pointer "swizzling" [140], adding overhead and complexity to the process and making data sharing between address spaces awkward. While this can be avoided in a classic segmented memory architecture, as all addressing is done relative to the base of the segment, it complicates inter-segment addressing, which led to it dwindling out of use in favour of a flat address space.

Secondly, address space switching has traditionally come at significant cost. While address space switching with schemes like hardware supported memory segmentation can reduce such costs, they have fallen out of favour due to the increase in overall software development complexity, as mentioned above. The currently available schemes, such as operating system supported context switches, incur significant overhead, and synchronization is required when crossing address spaces [19].

Therefore, the benefits of address space sharing have given rise to research into single address space operating systems, particularly in 64 bit environments, where address space is abundant [15, 53, 75].

Such address space sharing becomes even more important for fine-grained components, where it becomes prohibitive to bear the cost of complete context switches or to suffer the complexity of programming for different address spaces. Unfortunately, few viable mechanisms exist for isolating components within the same address space, as contemporary hardware isolation mechanisms were designed to isolate processes into different address spaces. While there have been attempts to commandeer existing paging and segmentation hardware for this purpose, they do so at the cost of reduced isolation guarantees [19].

Many other approaches to component isolation simply do not address this dimension at all, falling back to traditional RPC mechanisms and shared memory pages in order to pass data between a component and its host [8, 32, 144, 145]. This, of course, incurs a performance and complexity penalty in terms of parameter marshalling across process boundaries, or reduced functionality (the inability to use pointers) when using shared pages.

Thus, the very benefits of fine-grained components are lost in the process, as simple and efficient sharing of data structures is hampered. While shared memory pages can be conceivably made to emulate this functionality by mapping the same memory pages into different processes at the same virtual address, this must be done with great care to avoid potential conflicts, as synchronization between two processes is required. Reaching an agreement becomes proportionately more difficult when multiple components are involved [53].

```
 1 void    *handle;
 2 typedef void (*hello_func)(char * str);
 3
 4 /* open the needed object */
 5 handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL |
                 RTLD_LAZY);
 6
 7 /* find the address of function and data objects */
 8 hello_func fptr = (hello_func)dlsym(handle, "my_function");
 9
10 /* invoke function, passing value of string as a parameter */
11 (*fptr)("hello world");
```

Figure 3.2: POSIX example of shared library call

Thus, this problem remains largely unsolved or altogether neglected in practice, mainly due to the lack of feasible alternatives. Now however, hardware virtualization [134] offers a greater degree of flexibility that can be utilised in enabling secure address space sharing, and this forms the basis of one approach presented in Chapter 5, for isolating components with strong guarantees while preserving the benefits of a single address space.

Nevertheless, in designing a general purpose isolation architecture, no assumption can be made that the shared library must necessarily reside in the same address space as its host, in keeping with the previously stated goal of implementation agnosticism.

3.7 LIBVM - API SPECIFICATION

To solve these problems, we have designed an Application Programming Interface specification sufficient to achieve component isolation with minimal changes to existing coding practices. In this section, we present the API specification in a generic form. Specific implementations of this interface are presented in Chapters 4 and 5.

3.7.1 Overview of API Specification

As mentioned previously, the interfaces are largely implementation independent, with the only assumption being that the library executes within the same address space as the hosting container, although strictly speaking, even this assumption is only necessary for certain parts of the functionality requirements. The basic interface strives to emulate the standard POSIX interfaces, which we describe below.

3.7.2 Existing POSIX Interface

The POSIX standard [61] defines three basic calls for loading a component into its host address space. These are as follows.

1. `dlopen` – Loads the library
2. `dlsym` – Extracts contents
3. `dlclose` – Unloads the library

The above three system calls are utilised in most UNIX-based operating systems (Microsoft Windows utilises similar system calls – `LoadLibrary [96]`, `GetProcAddress [95]` and `FreeLibrary [94]`).

The code snippet in Figure 3.2 highlights the basic process of loading a shared library and invoking a function call within that library. As line 5 shows, the `dlopen` function is responsible for opening the shared library, giving its location, and returning a *handle* for future references to the library. This handle can be subsequently used to obtain a pointer to a symbol within the library (line 8). The

symbol may be a function or data pointer. If it is a function pointer, it can then be utilised to directly invoke the function (line 11).

Several observations can be made at this point.

1. It is trivial to obtain a pointer to a function and invoke it directly as in line 8, without the necessity for parameter marshalling, demonstrating programming simplicity within a single address space memory environment.
2. Preserving this same model is advantageous, as a lot of previously written code can be transferred to this model with minor changes.
3. The interface is suitably simple and easy to understand for something as important as dynamic shared libraries.

3.7.3 LibVM Interface

The LibVM interface was designed to closely mimic the functionality of the POSIX interface. This is in keeping with the idea of making it easier to port applications to be able to use LibVM isolation, yet general enough to support multiple implementations. Therefore, the POSIX interface can also be implemented as a "subset" of the LibVM interface, which however, would be equal to having no isolation at all.

The most important methods provided by LibVM's interface are outlined below.

1. `libvm_initialise` – Initialises the isolation subsystem
2. `libvm_open` – Loads a library
3. `libvm_sym` – Extracts contents
4. `libvm_close` – Unloads the library
5. `libvm_guest_malloc` – Allocates memory within the guest component's address space
6. `libvm_guest_free` – Frees memory allocated by a guest component
7. `libvm_destroy` – Frees resources used by the isolation sub-system

Predictably, the `libvm_open` function maps closely to `dlopen`, `libvm_sym` to `dlsym` and `libvm_close` to `dlclose`.

```
 1 void    *handle;
 2 typedef void (*hello_func)(char * str);
 3
 4 /* Initialise the isolation sub-system */
 5 struct libvm * libvm_ptr = libvm_initialise(argv);
 6
 7 /* open the needed object */
 8 handle = libvm_load(libvm_ptr, "/usr/home/me/libfoo.so");
 9
10 /* find the address of function and data objects */
11 hello_func fptr = (hello_func)libvm_sym(handle, "my_function");
12
13 /* invoke function, passing value of integer as a parameter */
14 (*fptr)("hello world");
```

Figure 3.3: LibVM example of shared library call

In order to demonstrate the functionality of this approach, the code snippet in Figure 3.3 can be contrasted with the POSIX implementation.

As demonstrated in Figure 3.3, the basic interface to the host is almost identical to the POSIX case. The differences are the addition of an extra initialisation function in line 5, which must be done once to initialise the isolation subsystem, followed by the subsequent destruction of it (not shown). It should be noted that no changes are required to the target shared library. The calling semantics of the function also remain identical as in lines 8, 11 and 14, meaning that, in simple scenarios, the application can be trivially ported to execute the library within an isolated environment.

In addition, this interface specification avoids being bound to a specific implementation of LibVM, in line with previously stated goals.

3.7.4 Interface Details

This section discusses each of the methods in the interface in greater detail, and describes their rationale and function. The section also tabulates the functions in a summarised form, and serves a point of reference throughout this thesis.

3.7.4.1 libvm_initialise

The purpose of this function is to initialise a LibVM isolation domain. It is provided so that an implementation has an opportunity to perform state initialisation, such as the allocation of initial resources required for the domain. It should be noted that multiple invocations of the method are possible, and therefore, multiple isolation domains can be created.

The method accepts a single argument in the form of an array of environmental values. This array can be used to provide various implementation specific properties so that the isolation domain can be fine-tuned if necessary. An array of arbitrary length can be supplied, with the end of the array indicated through a NULL terminator. The implementation specific parameters must be supplied as a character array with a name=value format, where name is the parameter name and value is the parameter's value.

The function returns a handle to the newly created isolation domain, which can be used to refer to and manipulate the newly created domain. All subsequent calls to a LibVM method must necessarily pass in the `libvm_context`, either directly or in an encapsulated form. Each invocation of the initialise method should return a unique `libvm_context`, each pointing to a unique isolation domain. All subsequent methods in the LibVM interface necessarily require this handle in order to identify, retrieve state from and modify the isolation domain. In the event of an error, a non-NULL context must nevertheless be returned, so that an implementation can determine the cause of the failure, through a call to `libvm_get_last_error (see` Table 3.8).

The `libvm_initialise` function is summarised in Table 3.1.

3.7.4.2 libvm_open

This method is analogous to the dlopen command in a POSIX implementation. It is provided so that shared libraries can be loaded into an isolation domain initialised previously.

Multiple shared libraries may be loaded into a single isolation domain. Each invocation therefore, returns a unique, opaque, implementation specific handle that can be subsequently used by the implementation to identify a library. Therefore, the

handle must not be interpreted by the caller in anyway. In addition, the implementation must ensure that the handle encapsulates the libvm_context returned by the initialise method as well, so that the parent isolation domain can be obtained from a library handle.

The filename of the library to be loaded must be passed into the method. Each implementation may have its own security policy and search paths for locating the library. Additional implementation specific parameters can also be passed through a flags parameter, although this parameter would ideally remain unused, to avoid such implementation specific dependencies.

This `libvm_open` function is summarised in Table 3.2.

3.7.4.3 libvm_sym

This method provides a means for resolving symbols within libraries loaded into a LibVM sandbox. It is analogous to the *dlsym* method provided in a POSIX implementation. Its primary use is in providing the caller with a reference to a function within the library, which can then be used to invoke that function.

Therefore, in the simplest possible implementation, it would return a direct function pointer to the library method, which is exactly what the *dlsym* method in a POSIX implementation does. However, such a function pointer would provide no isolation, since calling the function would cause it to be executed within the host's domain, effectively providing the untrusted function with the same privileges as the host.

Therefore, a more advanced implementation of LibVM should return a proxy function in place of the actual library function, which will also perform an isolation domain switch into the LibVM sandbox, before executing the actual library function, thus properly isolating the shared library. The proxy function would be responsible for copying parameter values safely from the caller's address space into the sandbox's address space.

This creates complications when passing values by reference, or when passing values which are pointers to values residing in the host's address space. The values would need to be serialised and copied over to the sandbox's shared address space, so that the library executing within the sandbox can access these values. Alternatively, the

values should be pre-allocated within the shared address space, so that the untrusted code can reference them without affecting the host.

This `libvm_sym` function is summarised in Table 3.3.

3.7.4.4 libvm_close

This method is provided in order to unload a shared library from within a LibVM sandbox, thus freeing up the memory and resourced used by that library. It is analogous to the *dlclose* method in a POSIX implementation. Once the dlclose method is called, previous handles issues through the libvm_open method should be invalidated. However, the isolation domain should not be destroyed, and it should be possible to reload the library afresh by calling the libvm_open method.

This `libvm_close` function is summarised in Table 3.6.

3.7.4.5 libvm_guest_malloc

This method's purpose is to provide a means through which to allocate memory within a guest's address space. While address space transparency is assumed between the guest and host, it is also essential that the guest code executing within the sandbox cannot access memory outside of the bounds of the sandbox. Therefore, this method provides a means by which a host application can negotiate a shared chunk of memory within the sandbox's allowed memory regions, which can then be addressed by both the untrusted guest code and by the host application itself.

While this memory region can be freely accessed by the guest, the host must take precautions when utilising a value obtained from this shared address space and treat is as untrusted, since untrusted code executing within the sandbox may asynchronously manipulate that memory. Therefore, validation must be performed before using such values, and TOCTOU [39] bugs should be avoided.

This `libvm_guest_malloc` function is summarised in Table 3.4.

3.7.4.6 libvm_guest_free

This method is the counterpart of the libvm_guest_malloc method and provides a means by which to release memory previously allocated with libvm_guest_malloc. It should be noted that the memory must be released from the guest's address space. Thus, an invalid `free` or a double `free` invocation would not result in an error in

the memory tables of the host's address space. An invalid `free` may corrupt the guest's address space, but this must be limited to the pre-defined address space of the guest, thus having no effect on the host.

This `libvm_guest_free` function is summarised in Table 3.5.

3.7.4.7 libvm_destroy

This method provides means by which an isolation domain can be completely destroyed and all resources used by it, released. Therefore, an implementation should suspend the execution of code within the sandbox, unload all libraries and release all resources used by it.

This `libvm_destroy` function is summarised in Table 3.7.

Method name:	`libvm_initialise`
Method description:	Initialises the isolation subsystem. Any state initialisation, global to an isolation domain, can be performed here.
Method signature:	`struct libvm_context * libvm_initialise(const char *const *argv)`
Parameter description:	`argv` – An array of strings containing environmental values, with the last element being equivalent to a `NULL` value. This can be used to pass in an array of arbitrary length, containing implementation specific parameters in name=value format. returns – A pointer to a `libvm_context`. A `libvm_context` is a handle that can be used to retrieve isolation domain state by the LibVM implementation. All subsequent calls to a LibVM method must necessarily pass in the `libvm_context`, either directly or in an encapsulated form. Each invocation of the initialise method should return a unique libvm_context, each

	pointing to a unique isolation domain. In the event of an initialisation error, a non-NULL `libvm_context` must nevertheless be returned. Callers can retrieve a detailed error code by passing in this invalid `libvm_context` to `libvm_get_last_error`.
Notes:	In the simplest implementation (POSIX case, with no isolation), this method would nevertheless return a basic `libvm_context` with the `error_code` field set to 1 (SUCCESS). In more advanced uses, the `libvm_context`'s `impl_data` field can be used to store additional implementation specific details.

Table 3.1: Initialising the isolation subsystem – libvm_initialise

Method name:	`libvm_open`
Method description:	Opens a shared library, given its path, and loads it into a previously defined isolation domain.
Method signature:	`void * libvm_load(struct libvm_context *libvm_ptr, char * filename, int flags)`
Parameter description:	`libvm_context` – A pointer to a specific instance of an isolation domain, obtained by calling `libvm_initialise`. `filename` – The filename/path of the library to be loaded. `flags` – Additional implementation specific parameters. Implementations must supply intelligent defaults if a `NULL` value is passed as the `flags` parameter. returns – A pointer to a handle which can be used by the implementation to identify the library in future. This

	handle must not be interpreted by the caller in anyway, and must be treated as opaque. In a typical implementation, the handle will also encapsulate the `libvm_context`. If an error occurs during `libvm_open`, the returned `libvm_context` can be used with the `get_last_error` function to obtain detailed error information.
Notes:	In the simplest implementation (POSIX case, with no isolation), this method would be equivalent to the `dlopen` command, with the `flags` parameter functioning identically. In a more advanced implementation, the `flags` parameter can be used to pass implementation specific details, but ideally, it would remain unused, shielding the caller from any implementation specific details.

Table 3.2: Loading a library - libvm_open

Method name:	`libvm_sym`
Method description:	Accepts a handle of a previous opened library and a symbol name, and returns an address via which the value of the symbol can be accessed.
Method signature:	`void * libvm_sym(void * handle, char * name)`
Parameter description:	`handle` – A handle to a library returned by a previous call to `libvm_open`. `name` – The name of the symbol that should be located. returns – An address via which the symbol can be

	accessed. If the address is a function, then it must be possible to invoke the function as any other. The function may be a proxy function that takes care of switching isolation domains before invoking the actual library function. If an error occurs during `libvm_sym`, a `NULL` value may be returned, after which the current `libvm_context` may be used with the `get_last_error` function to obtain detailed error information.
Notes:	In the simplest implementation (POSIX case, with no isolation), this method would be equivalent to the `dlsym` command. If the returned symbol is a function pointer, it should be possible to directly invoke the function as with invoking a function via any other function pointer. Advanced implementations may return a proxy function in place of the actual library function, with the proxy function being responsible for switching to and back from the isolation container, and copying parameters and return values back to appropriate locations so that the caller and callee can both access these values.

Table 3.3: Extracting contents - libvm_sym

Method name:	`libvm_guest_malloc`
Method description:	Allocates memory in the isolation container's address space, which should be accessible to both the host and the guest. Address transparency between guest and host is assumed.
Method signature:	`void * libvm_guest_malloc(void * handle, size_t size)`

Parameter description:	`handle` – A handle to a library returned by a previous call to `libvm_open`. `size` – The amount of memory to be allocated in bytes. returns – The address of the allocated memory region or `NULL` in the event of an error.
Notes:	This method has no equivalent in the POSIX case, but has been provided in order to enable the host and guest to negotiate a shared address space. The guest cannot access the host's memory, in order to maintain a strict isolation boundary, and any shared memory must be established within the guest's address space. The host should copy values back upon completion and double check the values in order to guard against timing attacks.

Table 3.4: Allocating memory within the guest component's address space - libvm_guest_malloc

Method name:	`libvm_guest_free`
Method description:	Frees memory previously allocated by `libvm_guest_malloc`.
Method signature:	`void libvm_guest_free(void * handle, void * ptr)`
Parameter description:	`handle` – A handle to a library returned by a previous call to `libvm_open`. `ptr` – The address of the memory in the guest's address space, to be freed. returns – n/a
Notes:	This method has no equivalent in the POSIX case, but has

	been provided in order to enable the host and guest to negotiate a shared address space. The `free` method must only be performed within the guest's address space. Thus, an invalid `free` or a double `free` invocation would not result in an error in the memory tables of the host's address space. An invalid `free` may corrupt the guest's address space, but this must be limited to the pre-defined address space of the guest.

Table 3.5: Freeing memory within the guest component's address space - libvm_guest_free

Method name:	`libvm_close`
Method description:	Unloads a library previously loaded by `libvm_open`.
Method signature:	`void libvm_close(void * handle)`
Parameter description:	`handle` – A handle to a library returned by a previous call to `libvm_open`. returns – n/a
Notes:	In the simplest implementation (POSIX case, with no isolation), this method would be analogous to the `dlclose` command. The implementation can unload the library from the isolation domain and free any memory utilised by it. However, the isolation domain must not be destroyed unless a close to `libvm_destroy` is made.

Table 3.6: Unloading a library - libvm_close

Method name:	`libvm_destroy`
Method description:	Unloads the entire isolation container and frees up any resources used by it.
Method signature:	`int libvm_destroy(struct libvm_context * libvm_ptr)`
Parameter description:	`libvm_context` – A handle to an isolation domain returned by a previous call to `libvm_initialise`. returns – 1 on success, 0 otherwise. Error codes can be obtained via a call to `libvm_get_last_error`.
Notes:	This method does not have an analogous POSIX command. The implementation should unload the entire isolation domain and free all memory utilised by it. Any libraries within the domain must also be unloaded.

Table 3.7: Freeing resources used by an isolation domain - libvm_destroy

Method name:	`libvm_get_last_error`
Method description:	Returns an error code by which the last error code returned by the API can be accessed.

Method signature:	`char * libvm_get_last_error (struct libvm_context * libvm_ptr)`
Parameter description:	`libvm_context` – A handle to an isolation domain returned by a previous call to `libvm_initialise`. returns – `NULL` if there are no errors, or a human readable string with a description of the error that occurred.
Notes:	This method is analogous to the `dlerror` command, and in a POSIX implementation, would be identical.

Table 3.8: Getting last error code - libvm_get_last_error

3.8 IMPLEMENTATION NOTES

While our LibVM interface strives to be implementation agnostic, and the basic methods required for its functionality do not expose many implementation specific details, thus enabling a variety of conforming implementations, all implementations do rely on a few assumptions, which are outlined below.

3.8.1 Requirements

The ideal implementation of LibVM relies on the availability of a CPU implemented virtual machine environment wherein the execution of arbitrary code is possible in a controlled fashion. The VM environment must match the architecture and implementation of the hosting application. Strictly speaking, LibVM requires a virtual machine library with the following specific features.

1. It must support full virtualization of the CPU features and memory.
2. It must be compatible with the host environment's architecture (e.g., 32-bit Intel x86).
3. Memory regions must be supported, with access attempts to unmapped regions resulting in trappable page faults.
4. It must provide full control over its address space layout (e.g., in a 32 bit system, it must provide access to all 4 GB of available address space).

5. Privileged instructions (e.g., interrupt invocations, `syscall` instructions) must be trappable.
6. The host operating system must have a clearly defined system call interface, so that system-call interpositioning is possible (currently unavailable on Microsoft's Windows operating system).

Therefore, the primary version of LibVM is built on *LibKVM* [107], which provides an abstraction layer over the hardware virtualization support built into newer Intel and AMD x86 processors, and is presented in Chapter 5. The combination of *LibKVM* and hardware virtualization support provides an extremely light-weight virtual machine facility which fulfils basic CPU and memory virtualization requirements well, without incurring the overhead of a full-blown virtual machine implementation. Performance measurements confirm that the overhead is low enough to offer competitive performance in comparison to other techniques, as shown in Chapter 5. However, a software-based implementation which relies on existing operating system isolation facilities, has also been implemented, and is presented in Chapter 4.

3.9 CONCLUSION

This chapter has defined the architecture and functional specification for LibVM, identifying the key conditions under which LibVM can be guaranteed to be correct with respect to isolating libraries from their host, such that the host is fully able to control the execution of the libraries. Five conditions were identified under which this guarantee could be provided, two of which are reliant on the correct implementation of security policies by the host. Each condition was analysed in depth. However, these five conditions are not designed to guarantee a non-interference policy [122] as LibVM was not build with the aim of enforcing a secure information flow policy. Nevertheless, these conditions can be expanded upon to provide such guarantees, if required. The section has also provided an overview of the functions underpinning our design of an isolation container, as well as the rationale behind each method in the Application Programming Interface. The API specification strives to be implementation independent, and as a result, there are three implementations — one degenerate case which is equivalent to the plain

POSIX API's structure with no isolation, and two other implementations which are described in Chapters 4 and 5.

Chapter 4: Software Solution – Process Tracing Based

4.1 INTRODUCTION

This chapter describes a reference implementation of the LibVM Application Programming Interface (API) as defined in Chapter 3. It is based on existing software facilities, and serves to highlight the general techniques that can be used to realise LibVM's API. It relies on existing Operating System isolation features, in contrast to the hardware virtualization supported implementation described in the next chapter. The purpose of this implementation is to assess the possibility of appropriating existing Operating System functionality for the purpose of intra-address space isolation, its effectiveness, as well as performance implications. This chapter delves into these aspects, and provides an in-depth discussion of the implementation details.

4.2 APPROACH

In selecting an appropriate software-based implementation for library component isolation, we evaluated several of the techniques discussed in Chapter 2. While Software Fault Isolation (SFI) has been successfully utilised to "sandbox" libraries, it places many restrictions on the origin and source of the shared libraries themselves, as well as the subset of machine instructions such libraries are allowed to execute. In order to enforce these restrictions, existing approaches require that the libraries be recompiled through custom tool chains [32, 144] and existing libraries in binary form be altogether rejected [144]. Our goal was to avoid these restrictions, allowing a binary component to take advantage of any new or existing machine instruction, as long as it was not a privileged one, as well as to avoid the burden of requiring that binary components obtained from various sources be recompiled according to the restrictions imposed by an SFI implementation. Given these constraints, we rejected SFI as a suitable approach to this problem.

A well-known technique used to execute untrusted programs in a sandboxed environment, thus restricting them from accessing system resources, is System Call Interpositioning [39, 40, 42, 70, 101]. It is based on the idea that "an application can do little harm if its access to the underlying operating system is appropriately restricted" [42] and therefore, denying or modifying a system call before the Operating System executes it, is an effective way to *jail* the application. Operating system provided tracing mechanisms such as *ptrace*, or kernel level drivers, can be used to intercept all system calls made by a jailed application, and to modify these system calls as needed, effectively confining the application to a restricted subset of resources.

We decided to investigate the possibility of utilising a similar mechanism, although most existing implementations were meant to confine entire processes, not libraries within a process. Nevertheless, the approach itself presented several advantages, such as the ability to utilise existing OS facilities such as *ptrace*, */proc* etc. To our knowledge, ours is the first attempt to utilise this method for jailing individual libraries used by a process.

There were several potential ways to set about this. One approach was to use a kernel module to intercept system calls. However, errors or faults in kernel drivers are likely to be more disastrous than at a user level. Therefore, we decided to reject this approach in favour of user level tracing mechanisms.

Of the user level tracing mechanisms, the most well-known are *ptrace* and */proc* [70], the latter of which provides tracing facilities on SUN's Solaris operating systems. We decided to utilise the *ptrace* interface since it is available in our chosen operating system - Linux.

Ptrace provides a fairly simple interface by which one process can trace the execution of another. As the traced process executes, the *ptrace* interface provides a callback on the execution of each system call, allowing the tracing process to observe and manipulate the system call as desired. In addition, a rudimentary interface is provided to write to the traced process's memory.

This interface was originally introduced to support debuggers and as a result, *ptrace* has several limitations. One of the main limitations is that it is an "all or nothing" interface – in that either all system calls need to be traced, or none can be. There is

no possibility of being selective in choosing which calls to trace, incurring the overhead of a callback on each system call. In contrast, the */proc* interface allows for more fine grained control. One of the biggest shortcomings in *ptrace* however, is that it does not provide a way to abort or ignore a system call. However, workarounds for these issues have been demonstrated [40, 101] and do not pose a significant impediment to its use.

4.3 IMPLEMENTATION OVERVIEW

Figure 4.1 depicts the basic idea behind enabling library isolation using process tracing. The host application is the parent, and the library executes in a different

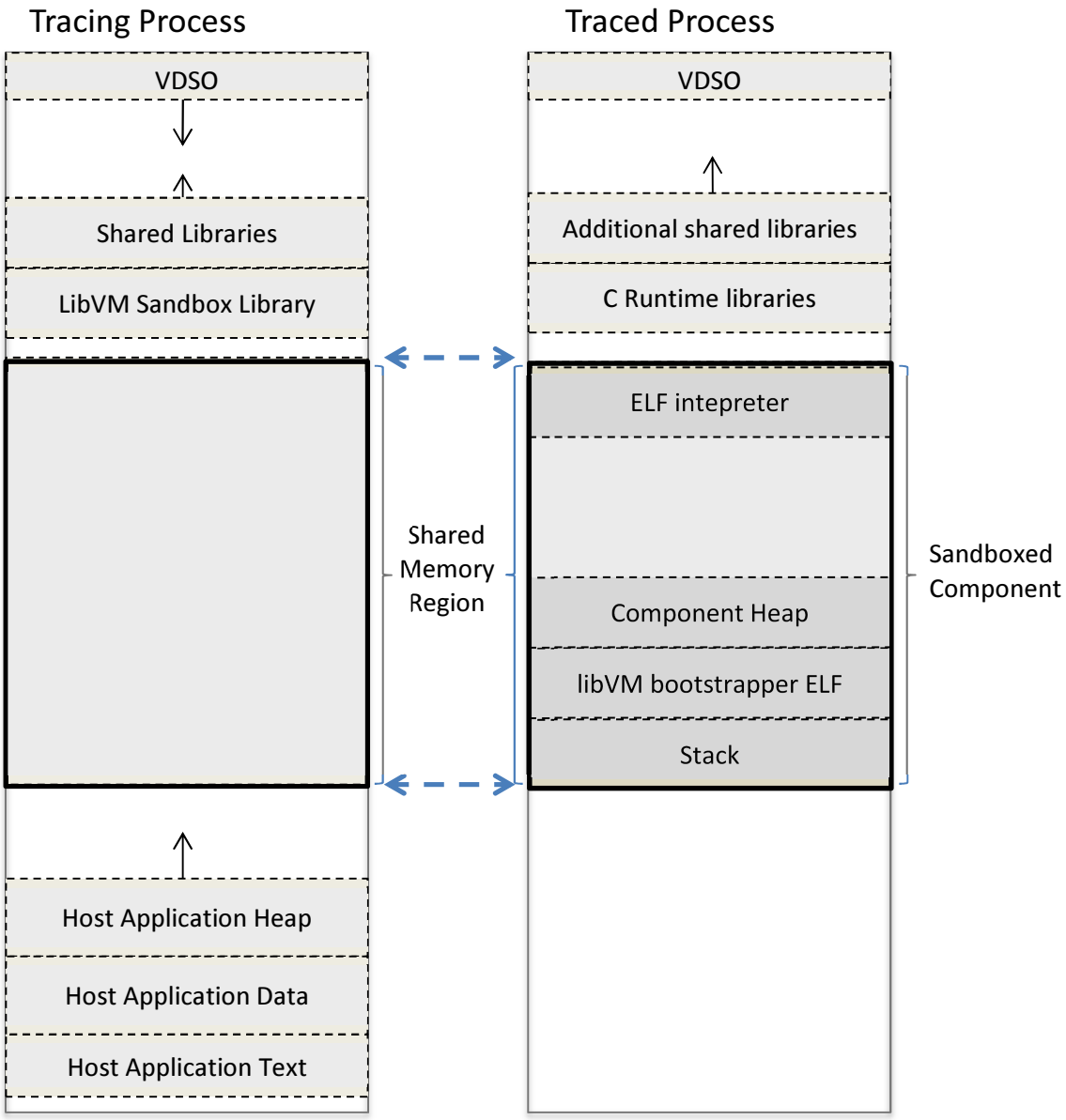


Figure 4.1: Relationship between parent and child processes

process, as a child of the host process. This enforces a strict separation between the parent and the child, using existing process isolation facilities.

The next task is to make sure that the child cannot execute any system calls without arbitration by the parent, which is achieved by the parent process tracing the child process. As explained previously, this enables the parent process to intercept all system calls made by the child, effectively controlling the worldview provided to it and constraining the child's execution with an isolation container.

The final task is to enable address space transparency between parent and child. This is achieved by establishing a shared memory segment between the two, and carefully ensuring that the child is limited to accessing memory only within that shared memory segment, in full view of the parent process. This enables the desired address transparency.

An important decision with regard to the implementation of this isolation mechanism was the portability factor. Since this was primarily a proof of concept to evaluate reference transparency in shared libraries, we decided to allow for platform specific assumptions to reduce the implementation burden. While the mechanics are specific to Linux, we show that the general ideas are applicable to any operating system, although they would need to be re-implemented to suit those systems.

Another trap we wanted to avoid was to place a new burden on those using LibVM, namely that they adapt existing libraries according to the constraints imposed by LibVM. This is in contrast to the method utilised in Google's Native Client [144] and Vx32 [32], as well as other SFI based isolation systems, which rely on the libraries being recompiled using a custom tool chain. This requirement is specified by SFI systems mainly because of the complexity in guaranteeing that arbitrary machine code is secure. By placing some restrictions on these machine instructions, such as disallowing unaligned jumps, this complexity is greatly reduced. However, we argue that this places an undue burden on developers, as it is not always feasible to expect that a third party library can be recompiled using a custom tool chain. Instead, our focus was on utilising existing libraries unaltered, and have the burden of adapting and isolating them placed on the host program only. We believe this is a more practical approach, as there is no escaping the fact that a host application which wishes to utilise LibVM's isolation facilities must be modified, whereas modifying shared libraries obtained from third parties is optional. This is also more desirable; as

such third-party libraries may not always come in an adaptable or recompilable form and would introduce a maintenance burden in addition.

The easiest method of achieving this end is to "fool" the library into thinking that it is executing as normal, within an OS process, as part of an application, and with full access to the usual machine instructions and system calls. However, by placing the application in a separate isolation container (in this case, a separate process), and restricting the library at a system call level, we effectively restrict the world view available to the library as well as its effect outside of the isolation container. This basic idea is portable across operating systems.

In the case of Linux, the simplest way to achieve this is to let the libraries be loaded by the existing dynamic linker/loader provided by the operating system. This removes a significant number of burdens and limitations imposed by other systems. For example, early versions of Google's Native Client system required that the libraries be statically linked [118, 144], with dynamic loading support introduced later, albeit in limited ways. Vx32 also imposes similar restrictions [32]. By allowing the system's standard runtime linker to perform all symbol resolution, we avoid all such restrictions, while simultaneously enabling the use of existing libraries with no modification.

Therefore, we opted to utilise the standard Linux dynamic linker to perform these tasks. However, we also wanted to avoid the complexity of adapting, recompiling or otherwise modifying the linker, which would create an unnecessary maintenance burden. Therefore, our aim was to use the linker unmodified as well, while constraining its worldview in a similar fashion, through interception of system calls.

The process of linking and loading is discussed in great detail by Levine [84]. The ELF (Executable and Linkable Format) interpreter utilised by Linux is basically a linker/loader which is executed by the operating system in response to a request to execute an actual program, typically through an *exec* system call. The operating system loads the ELF interpreter into a high address in memory, and populates the "AUXV" data structure, which points to where the actual executable lies. In addition, environmental variables are also placed just after the AUXV vector. The operating system will then switch to the process and jump to the interpreter's entry point.

The ELF interpreter is position independent and therefore, fully relocatable, which means that it can be placed anywhere within an address space and executed, in contrast to executables with fixed load addresses. When the ELF interpreter starts executing, it will read in the AUXV vector and environmental variables, in order to initialise itself. While this overall process is involved and described by Levine [84], the basic idea is that the interpreter will follow the AUXV vector to find the base address of the actual executable, read its ELF header, and start performing the dynamic linking routine. If it is a static executable, little else needs to be done, and the linker can transfer control over to the executable. However, if it is a dynamic executable, the linker must first load all dependent libraries, such as the C runtime library and all other dependencies recursively. This symbol resolution process is also quite involved. Once complete, execution can be transferred to the executable. However, the interpreter must remain in memory to perform any dynamic linking/symbol resolution needed by the running program.

As described above, rather than rewriting the ELF interpreter from scratch, we utilise it as it is for library loading and symbol resolution. By emulating the behavior of the operating system, we can effectively use the existing ELF interpreter with no modifications. Therefore, we utilise a "bootstrapper" executable, written to support LibVM's functionality, which we ask the ELF interpreter to execute. The ELF interpreter dutifully executes the executable program, within our isolated and fully traced and controlled process.

The bootstrapper executable is an executable which acts as a proxy for LibVM within the memory space of the child process. The idea is that we make the interpreter execute the bootstrapper executable, which in turn provides a set of services that we can use to control the child process, such as loading additional shared libraries. We consider the child process a sort of virtual machine for shared libraries, and hence, the bootstrapper is responsible for "booting" that VM, in collaboration with the ELF interpreter. This is also where the name "LibVM" comes from.

This bootstrapper executable must also be relocatable, since we wish to constrain all interaction with the shared memory region which is shared by the child with its parent. This is done by compiling the bootstrapper executable as a Position Independent Executable (PIE), via the –PIE flag in GCC. Normally, executables are

not position independent and are loaded at a fixed base address, whereas shared libraries are position independent, since they are loaded dynamically and it is difficult to predict in advance where in memory a space for it will be available. However, it is possible to compile standalone executables as being position

```
struct libvm_mem_region {
    void * base;
    size_t size;
};

struct libvm_mem_region * create_shared_region()
{
    void * mapped_address;

    struct libvm_mem_region * region_ptr = malloc(sizeof (struct
                                           libvm_mem_region));
    if (region_ptr == NULL)
        return NULL;

    if ((mapped_address = mmap(LIBVM_SHARED_BASE, LIBVM_SHARED_SIZE,
                        PROT_READ | PROT_WRITE | PROT_EXEC,
                        MAP_SHARED | MAP_ANONYMOUS, -1, 0)) ==
                        (void*) -1)
    {
        free(region_ptr);
        return NULL;
    }

    region_ptr ->base = mapped_address;
    region_ptr ->size = size;
    return region_ptr;
}
```

Figure 4.2: Allocation of shared memory region

independent, which is also how the ELF interpreter itself is compiled. This is also important in enabling Address Space Layout Randomization (ASLR), which is used by modern operating systems to reduce, or minimise return-to-libc attacks [119].

Next we discuss how the actual process of creating the isolation container in the software solution works.

4.3.1 Initialising the Library – libvm_initialise

As outlined in Chapter 3, the basic process of initialising a new isolation container is triggered through the *libvm_initialise* call. This creates a cascade of actions, which is presented in outline form below as a point of reference, but elaborated upon in detail subsequently.

1. A shared memory region is allocated;

2. The ELF interpreter and a "bootstrapper" are loaded into this shared memory region;
3. A child process is forked;
4. The parent immediately attaches itself to the child;
5. The child process then proceeds to clean up its address space;
6. The child jumps to the interpreter's start address. The interpreter executes, and in turn, executes the bootstrapper, with the parent carefully ensuring that the child's access is constrained within the shared memory region;
7. The bootstrapper heralds completion by invoking a special syscall;
8. The dlsym function in the child's C runtime is resolved; and
9. The host returns from the libvm_initialise call, with the child process suspended.

4.3.1.1 Setting up a shared address space between parent and child

When the *libvm_initialise* method is invoked, the first thing that LibVM does is to allocate a region of memory that will be shared between the host (parent process) and the library (child process). This is done by way of operating system provided shared memory facilities, in this case, POSIX shared memory [61].

The C code in Figure 4.2 highlights the process. The creation of the shared memory is done through the *mmap* operating system call. Function *mmap* is a POSIX-compliant UNIX system call that provides a mechanism to establish a mapping between a process's address space and a file or shared memory object. The key to ensuring that this memory is shared between parent and child, as per our requirement, is to create an anonymous region (MAP_ANONYMOUS) which is also shared (MAP_SHARED). An anonymous mapping simply means that the mapped memory region is not backed by a file. When combined with the MAP_SHARED flag, the POSIX standard guarantees that when a child process is forked, it too will inherit and share this memory region with its parent. Since the POSIX standard is supported by most UNIX based operating systems, this method is also portable across these systems.

An alternative design choice that was available was to establish this shared memory region via the *shmat* system call, another POSIX compliant system call which provides a mechanism for attaching a named segment to a process's memory, which could subsequently be shared. However, *shmat* was unsuitable for several reasons. One reason was that we wanted to ensure that the parent and child would be able to attach themselves to identical virtual memory addresses, so as to meet our goal of maintaining pointer transparency between host and shared library. This meant that even in a multi-threaded host, which could have other memory mapping operations ongoing, we needed to make sure the host and child were kept in lock step. The easiest way to ensure this is to establish the memory region prior to forking the child process, thus ensuring that the child process will inherit the same memory segment. Using *shmat* would have caused additional complications, since the child would have had to negotiate a shared region with its parent after having launched itself, which may or may not be possible in a host which has many threads contending for memory, as has been mentioned by others [53]. Therefore, by opting for the former method, we avoided this possible race condition.

4.3.1.2 Loading the ELF interpreter and bootstrapper into the shared region

Once the shared memory region is established as shown in Figure 4.2, we then proceed to read in and parse the ELF files that represent the ELF interpreter and bootstrapper respectively. They are laid out in memory in the manner shown in Figure 4.1. We do not perform extensive validation in our ELF parser, as both the interpreter and bootstrapper come from trusted sources, and will execute prior to the execution of any potentially malicious code. It should be noted however, that neither of these are part of the system's TCB (Trusted Computing Base), since they too are executed within the isolation container.

Once the two executables are loaded into memory, the AUXV vector which was described earlier, is laid out in the format required by the ELF interpreter. We also ensure that the AUXV vector contains a pointer to the bootstrapper's entry point, so that the interpreter may find and execute it.

4.3.1.3 Forking the child process

After the executables are correctly set up within the shared memory, the next step in the initialisation process is to perform the actual fork. A fork basically splits a

running process into two separate processes, with an identical state of execution, with the original process being marked the parent and the forked process being marked as the child. Typically, a fork is followed by an *exec*, in order to launch another process. However, we utilise fork as the mechanism for establishing the parent child relationship while retaining an anonymous shared memory mapping between parent and child, as described in Section 4.3.1.1.

4.3.1.4 Attaching the parent process to the child

At this point, we make the parent immediately attach itself to the child process, via

```
child = fork();
if (child == 0)
{
    // child's code omitted for brevity
}
else
{
    ptrace(PTRACE_ATTACH, child, NULL, NULL);
    libvm_ptr->child = child;
    if (ptracevm_loop(libvm_ptr) == 0)
        return libvm_ptr;
}
```

Figure 4.3: Point at which parent and child fork off

ptrace, using the code shown in Figure 4.3. The parent then goes into a loop (*ptracevm_loop*), in which it traces the child process till bootstrap completion is signaled.

4.3.1.5 Cleaning up the child's address space

However, since we forked from a parent process, the child now also inherits all memory mappings from the parent, and the runtime state is also identical, including the state of the C runtime libraries. This is unacceptable, for two main reasons. The first is that the child process can now see the state of the parent, albeit a stale state since the memory mappings are inherited as Copy on Write (COW) However, from an isolation perspective, this needs to be avoided, as it presents an information leak from parent to child. Secondly, the C runtime itself maintains state, such as all memory allocations made by *malloc*, thread state information etc., all of which are now invalid, since the child is not expected to be able to inspect or modify the parent's state. Therefore, our next step should be to cleanup this inherited address

space (apart from the shared segment itself) as well as reset/unload the C runtime code.

This is achieved by unmapping the memory segments used, apart from the shared memory segment itself, based on a technique detailed elsewhere [48]. The existing mapped segments are obtained through reference to */proc/self/maps*. An additional complication here is that care must be taken not to unmap the LibVM code itself, as it is the code which is executing to perform the unmapping in the first place. Unmapping the running code would result in an immediate segmentation fault, terminating the child application.

Similarly, all other resources used by the parent, such as open files, must be released, in order to prevent them being inadvertently shared with the child process. However, as we describe later, this is not a major concern, as we perform a check at the system call layer to ensure that the child is only capable of accessing files and sockets it has opened itself.

4.3.1.6 Transferring control to the ELF interpreter

Once this process is complete, the thread local storage register is cleared, the stack is set to our custom stack as depicted in Figure 4.1, and a jump is performed to the entry point of the interpreter.

The interpreter in turn executes the bootstrapper executable, and proceeds to resolve its dependencies, such as the C runtime library. Since the C runtime library was unmapped, this ensures that it will be reloaded afresh, and the bootstrapper starts with that fresh environment.

However, since the parent is tracing the child executable, all system calls made by the child are intercepted by the parent. The interpreter will typically perform actions like checking ld.so.conf (the library preload locations), find the dependent libraries such as the C runtime library (libc.so), and map these libraries into memory. However, since these are all trusted libraries, and no untrusted code has begun execution, the parent process does not have to impose any restrictions, other than making sure that any mapped files are made within memory regions endorsed by the parent, so as to keep the parent's and child's memory maps in lock step.

4.3.1.7 Bootstrap completion

Once the bootstrapper finishes executing, it heralds completion to the parent, as

```
asm volatile("movl %%eax, %%edx\n\t" // save current eax - potential return value
             "movl $0xCAFE, %%eax\n\t"  // special syscall value indicating bootstrap
                                          completion
            "movl %0, %%ebx \n\t" // ebx will contain address of dlsym - for all future
                                     symbol resolution
            "int $0x80\n\t"
            : : "c"(&dlsym)
            : "memory", "cc");
```

Figure 4.4: Bootstrapper completion

shown in Figure 4.4. This snippet performs four main tasks. It sets a "magic" value in the *eax* register (a magic value is a unique value of significance to the recipient), the receipt of which will confirm to the parent process, which is tracing these events, that the bootstrapper has completed execution. It also stores the address of *dlsym*, the symbol resolution routine provided by the C runtime, within the *ebx* register. Finally, it invokes interrupt *0x80*, which is the traditional system call interrupt.

Since a system call has been made, the operating system promptly suspends execution of the child and notifies the tracing process of the fact. The tracing process (LibVM), then reads in this value and ensures that it is indeed the magic value agreed upon previously. It also saves the instruction pointer for future use, as well as the *dlsym* address sent in via the *ebx* register, within local variables. Finally LibVM's state is marked as initialised, and LibVM goes into *lockdown* mode as explained below.

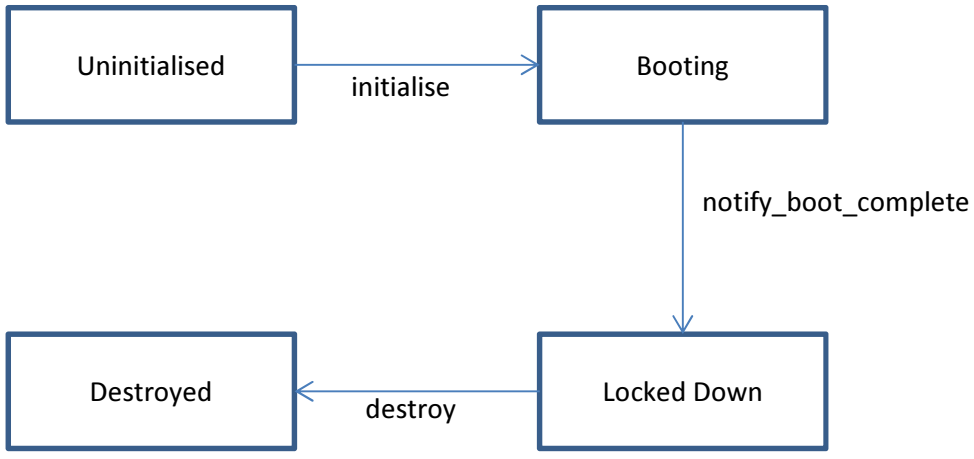


Figure 4.5: LibVM-ptrace - State transition diagram

Lock-down mode refers to a state transition in LibVM as depicted in Figure 4.5. It is triggered on completion of the boot process. This state is utilised to track the fact that, from this point onwards, potentially untrusted code can be loaded into the isolation container, and therefore, all system calls must be strictly fielded.

LibVM then saves the child's register state (recalling that the child's execution was suspended at the point of lockdown) within the LibVM context data structure. This saved register state is used to resume the child process at a later point. Essentially, what remains now is a full controllable execution container in the form of a child process. This child process can be used to load additional libraries or perform arbitrary actions under the supervision of the parent process. This is precisely the characteristic that is needed in loading and isolating libraries.

4.3.1.8 Resolving symbols in the child's address space

The main reason that the C runtime and bootstrapper was loaded into the child process's address space was so that these could in turn load additional shared libraries on behalf of the parent process. However, since they are running within a different, fully isolated and restricted process, the parent process is effectively isolated from the shared libraries loaded into the child process's address spaces

As discussed previously, the best way to load these libraries is to use the existing C runtime to do this, since writing a specialised "bespoke" dynamic linker is a time consuming process. We also gain the advantage of being able to utilise existing libraries as is. Libraries are typically loaded by invoking the dlopen function, as described in Chapter 3, and symbol resolution is performed by using dlsym.

The key to using the C runtime code executing within the child process is then, the *dlsym* function, since it can be used to obtain a pointer to any function or symbol within the child process. This is the reason that the function address of dlsym is passed onto LibVM on completion of the child's bootstrap process. By restarting the child process with the instruction pointer set to dlsym's address, and with the

```
 1. int libvm_dlsym_wrapper(struct libvm_context * libvm_ptr,
                            long dll_handle, char * name)
 2. {
 3.     void * temp_str = libvm_stack_push_string(libvm_ptr,
                                                  name);
 4.     libvm_stack_push_word(libvm_ptr, (uintptr_t)test_str);
 5.     libvm_stack_push_word(libvm_ptr, dll_handle);
 6.     libvm_stack_push_word(libvm_ptr,
            libvm_ptr->springboard_adress); // force return
                              address to springboard instruction
 7.     libvm_ptr->regs.eip = libvm_ptr->dlsym_address;
 8.     libvm_run(libvm_ptr);

 9.     libvm_stack_pop_word(libvm_ptr);
10.     libvm_stack_pop_word(libvm_ptr);
11.     libvm_stack_pop_bytes(libvm_ptr, strlen(name) + 1);

13.     return libvm_ptr->regs.eax;
14. }
```

Figure 4.6: Symbol resolution by wrapping dlsym

appropriate parameters passed through the stack, we can resolve any symbol in the child's C runtime, including the dlopen symbol.

Figure 4.6 shows this basic idea in action. Recall from the previous section that the child process was suspended, and its registers saved upon completion of the bootstrap process. This function effectively manipulates the child process, by pushing the values needed to invoke the dlsym function into the child processes's stack. These are basically the name of the symbol to be resolved, the handle to a *dll* if available (NULL in this case, since we are resolving a global symbol) and the return address upon completion of the function, as shown in lines 4, 5 and 6.

In normal function execution, the function epilogue would pop the return-address off the stack and jump to that return address, effectively returning control to the calling function (assuming *cdecl* calling conventions, which is the default calling convention used). However, we carefully change this return address to point to a location within our bootstrapper code, known as the spring board, a shown in line 6. The spring

board effectively performs a domain transition, suspending the execution of the child and returning control to the parent, in the exact same manner depicted in Figure 4.4.

In line 7, we force the child process's instruction pointer (eip) to dlsym's address, and then line 8 performs the actual task of resuming the execution of the child, in the exact same manner shown in Figure 4.3. Since the instruction pointer has been changed to point to *dlsym*, the child will resume execution at that instruction, execute the *dlsym* function, resolve our desired symbol, and return to our spring board address.

The spring board will notify LibVM that the child has completed its function, and once again, the child's execution will be suspended, registers saved and control will return to line 8 above.

Subsequently, we perform a cleanup of the child's stack, since the parameters must be popped off the stack upon completion of the function call, an action that was deliberately left incomplete by our spring board. At the end of this process, the *eax* register contains the return value from *dlsym*, which is the address of the symbol that was resolved.

Thereafter, the LibVM runtime proceeds to invoke dlsym again, caching frequently used symbols such as *dlopen*, *malloc* and *free* in the child's runtime. These can now be invoked to load libraries into the child process, allocate additional memory in the child and free any allocated memory, respectively. In short, the parent process can completely control the runtime of the child process as desired.

Finally, control is returned from the *libvm_initialise* method, with the caller receiving a pointer to the LibVM context data structure.

This completes the initialisation process.

4.3.2 Loading a Library – libvm_open

Once a LibVM context has been obtained, it is possible to load libraries into the guest's address space. This is accomplished by invoking the *libvm_open* function, as elaborated in Chapter 3. The desired functionality at this point is to simply have the child's C runtime load the library on our behalf, into its address space. Since the library is loaded into a different process, it is completely isolated within its confines, and is unable to access or modify the host application in anyway. This is the exact

characteristic which we desire, in order to prevent a faulty library from otherwise accidentally trampling over the memory of the parent.

With some detail omitted for brevity, Figure 4.7 helps to highlight the essence of how this procedure works. Recall from the previous section that the address of the *dlopen* function in the guest's C runtime was cached within LibVM context. We utilise this cached copy to forge a request to the guest, in a manner analogous to that described in prior sections.

```
void * libvm_dlopen_wrapper(struct libvm_context * libvm_ptr,
                            char * filename)
{
   void * guest_handle =
      (void*)libvm_stack_push_string(libvm_ptr, filename);
   void * dll_handle =
      libvm_ptr->guest_cache.guest_dlopen(guest_handle,
                                    RTLD_LAZY | RTLD_GLOBAL);

   libvm_stack_pop_bytes(libvm_ptr, strlen(filename) + 1);

   return dll_handle;
}
```

Figure 4.7: Library open implementation

What should be noted is that the method that is being invoked, *guest_dlopen*, has the exact same signature as that in the actual *dlopen* method in the C runtime (described in Chapter 3). In a traditional IPC setting, this would require that the parameters be marshaled across process boundaries, and copied back on completion. However, we achieve this same effect without these marshalling overheads, by taking advantage of our shared address space.

By allowing the host application to access and manipulate memory in the guest in a transparent fashion, it makes the process of manipulating the guest's stack, allocating memory in the guest's address space etc., far easier than the alternative of having to readjust pointers to match the guest's address space. This is where the true advantage of the transparent address space comes into play.

Secondly, in contrast to heavyweight approaches such as Remote Procedure Calls, which are designed to abstract away the actual address space and even the machine itself which is executing the function, we dispense with that complexity by making simplifying assumptions. Specifically, by assuming that the child is executing on an identical architecture with a fully transparent memory, which is a practical

assumption to make given that we are dealing with program libraries, unnecessary overheads are avoided. Through the demonstration above of the simplicity of transferring data between guest and host, we validate our hypothesis on the advantages of having a shared and transparent address space.

4.3.3 Resolving a Symbol in the Isolated Library – libvm_sym

```
1. void * libvm_sym(void * handle, char * name)
2. {
3.     struct libvm_handle * lib_ptr =
                        (struct libvm_handle *) handle;
4.     struct libvm_context * libvm_ptr = lib_ptr->libvm_ptr;

5.     int guest_symbol =
             libvm_dlsym_wrapper(libvm_ptr, lib_ptr->guest_handle,
                                 name);

6.     return libvm_install_trampoline(libvm_ptr, guest_symbol);
7. }
```

Figure 4.8: Dynamic proxy generation

The sequence of events, which transpire when a host application calls *libvm_sym* in order to obtain a pointer to a function within an isolated library, is depicted in Figure 4.8. The host must pass in a handle to the shared library itself, as described in Chapter 3, as well as the name of the function/symbol to be resolved. Since the library resides in the child process, we utilise *libvm_dlsym_wrapper* as shown in line 5. As explained in Section 4.3.1.8, *libvm_dlsym_wrapper* is used to resolve the symbol in the guest's address space and obtain a pointer to that function, which is still referring to the guest's address space.

However, simply invoking this obtained function would be disastrous, since the function would be executed in the host's address space, and not the guest's, which is counter to our purpose of isolating the library. What needs to happen then, is that a transition must be made to the child process, the function executed in the child process, and the result returned back to the host. In order to do this, we dynamically generate a proxy function, by calling the *libvm_install_trampoline* method, as shown in line 6.

This method, very simply, generates some binary code "on the fly" to perform the domain transition from the host to the child, so that the "guest_symbol" can be executed within the child's domain, as well as return the obtained result back to the

host. The binary code is generated and placed in a special page in the host's address space, and one trampoline is installed for each function resolved with *dlsym*.

```
1. IDENTIFIER(libvm_tramp):
2.   pushl   %esp // Argument 4 to cvm transfer is current esp
3.   pushl   %ebp // Argument 3 to cvm transfer is current ebp
4.   pushl   $0xDEADBEEF // Argument 2 to libvm_switch_domain is
                          guest ip
5.   pushl   $0xDEADBEEF  // Argument 1 to libvm_switch_domain is
      the LibVM pointer - patched at runtime to the correct value
6.   ljmp   $0xCAFE, $0xDEADBEEF
7. IDENTIFIER(libvm_tramp_end):
```

Figure 4.9: Trampoline for transitioning from host to guest

The snippet in Figure 4.9 shows the code which constitutes each trampoline. As shown in lines 4, 5 and 6, the values 0xDEADBEEF and 0xCAFE indicate placeholders which are binary patched to their correct values at runtime. Line 4 is patched with the value of the symbol in the child's address space, denoted by "guest_symbol" in Figure 4.8. Line 5 is patched to implicitly pass the LibVM context that is associated with the library that the symbol belongs to. Finally, line 6 is patched to point to the *libvm_switch_domain* function, which is responsible for the actual domain switch from parent to child.

A simplified version of *libvm_switch_domain* is shown in Figure 4.10. When control is received by *libvm_switch_domain*, note that the guest's target function pointer, which is essentially the instruction pointer at which the guest must be executed, as well as the current stack pointer and base pointer, are passed in by *libvm_tramp*.

```
 1. int libvm_switch_domain(struct libvm_context * libvm_ptr,
           const long guest_eip, const long ebp, const long esp)
 2. {
 3.    libvm_stack_push_bytes(libvm_ptr, esp, ebp - esp);
 4.    libvm_stack_pop_word(libvm_ptr); // pop current return
                                        address
 5.    libvm_stack_push_word(libvm_ptr,
           libvm_ptr->springboard_adress); // force return
                              address to springboard instruction
 6.    libvm_ptr->regs.eip = guest_eip;
 7.    libvm_run(libvm_ptr);

 8.    libvm_stack_pop_bytes(libvm_ptr, ebp - esp –
                      sizeof (long)); // restore original stack

 9.    return libvm_ptr->regs.eax;
10. }
```

Figure 4.10: Switching domains

Therefore, as shown in line 3, these same values are copied verbatim to the child's stack, with the calling conventions assumed to match. However, since the copied values will contain a return address in the host's address space, which the child cannot access, it must be popped and replaced with the "springboard_address", as shown in lines 4 and 5. The springboard_address is a helper function which was a part of the original bootstrapper coder, which helps the child to transition back from the child to the parent. By forcing the return address to the springboard_address, we trigger an automatic transition back to the parent at the end of function execution.

Line 6 shows how the guest's instruction pointer is forced to be the function we wish to invoke within the guest. In line 7, we execute the familiar *libvm_run* to resume execution of the suspended child, except that execution will now resume from the *guest_eip* which it was forced into.

Upon completion of the function, the springboard will trigger the transition back to the host, which in turn will return from the *libvm_run* function at line 7. Afterwards, line 8 shows how the guest's stack is restored to its original state and in line 9, the return value of the function, which is stored in the eax register is returned to the original function caller, which is oblivious to the machinations that occurred during invocation of the function.

Omitted from this description, is the process by which the caller's stack values are overwritten with the guest values, in order to support passing arguments by reference. However, the guest is not able to manipulate the return address in any

way, since it is always overwritten to ensure that no stack smashing attacks can occur. At this point, a few caveats are necessary, since some information leakage is occurring when copying values from the guest to host. However, this is not a critical issue, as it is an entirely optional process, and can be bypassed if more secure semantics are desired. This can be achieved by omitting the proxy generation altogether, and creating manual wrappers around the desired function, an example of which is the original *dlsym* function shown in Figure 4.6. However, the convenience of being able to bypass such manual proxy generation, thus improving transparency and usability to the user of LibVM, is a strong motivation for enabling this functionality. It is further mitigated by the low security risk.

A second caveat is with regard to passing values into the child's address space. Any pointer values passed into the child which are outside of its bounds, cannot be accessed by the child, thus ensuring the security of the host application. However, any values returned by the child, which contain pointer values, must be treated as suspect, since they may be forged to point to locations in the host's address space. This issue has been discussed previously under section 3.4, and was stated as Lemma D. If the host application blindly dereferences such a return value from the guest, and/or writes values to such locations, there is every possibility for the child to launch an attack on the host application. As such, these values must be treated as suspect.

However, how strict a stance should be taken depends entirely on the nature of the library which is being isolated. If the library is being isolated in order to localise errors, and the library is not expected to be actively hostile, there is relatively little security risk in accessing these values, only a risk in terms of errors propagating out of the isolation container. Nevertheless, advantages are still gained because the library's execution is isolated.

On the other hand, if the library is expected to be actively hostile, there is significant risk in using pointer values returned by the untrusted library. As such, great care must be taken by clients to validate that all values fall within the boundaries of the isolated container, before they are accessed. In later chapters, we show how future work could lessen the burden on clients through language level support for LibVM in the compiler. However, in the absence of such support, it is necessary that the client

treat all values received from the untrusted library, as well as all other actions taken, as highly suspect in the interests of security.

4.3.4 Execution of System Calls

Due to the nature of *ptrace*, each system call is intercepted twice,

a. before it is executed by the OS; and

b. immediately after it is executed but before returning to the calling process.

This provides the ability to modify both the actual call executed by the OS as well as the return value sent back to the child process. This process is illustrated in Figure 4.11.

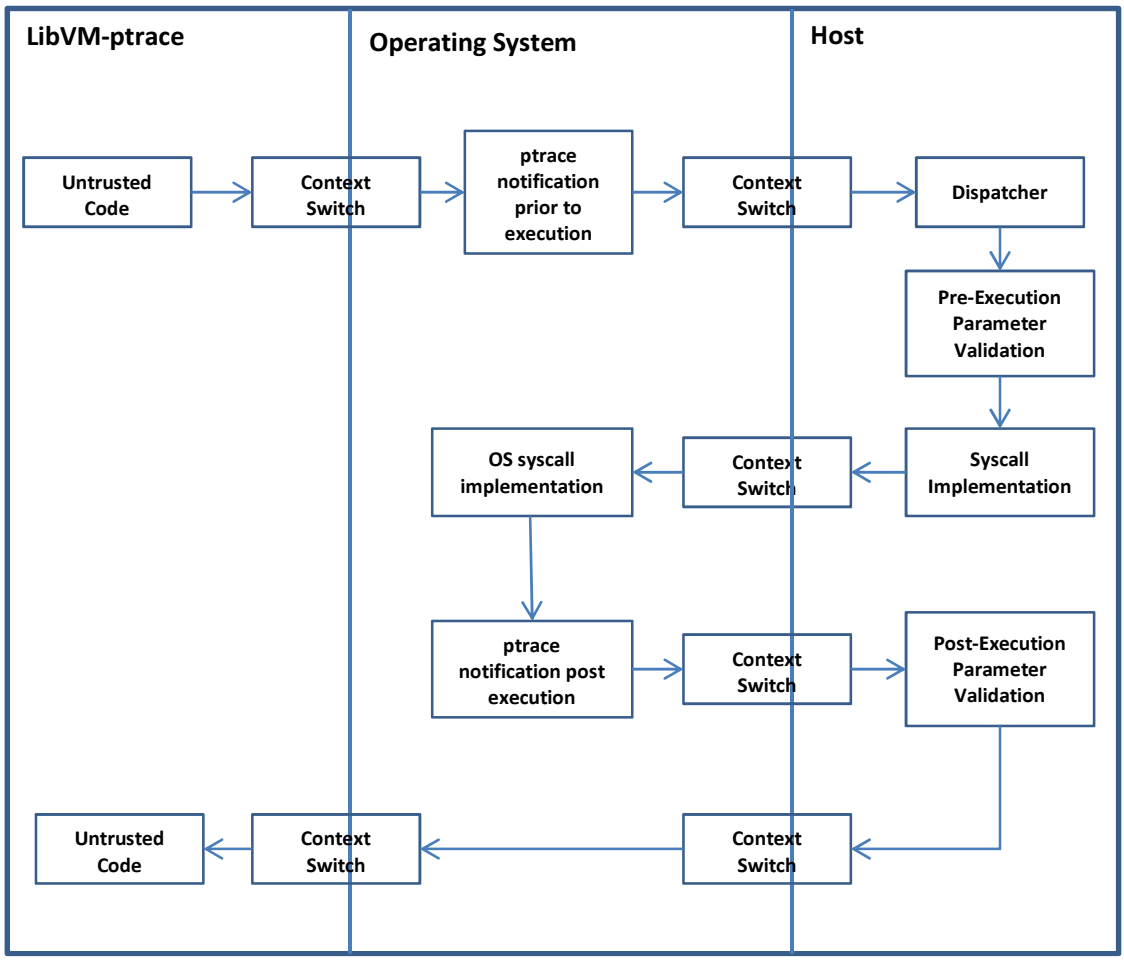


Figure 4.11: LibVM-ptrace – system call execution sequence

As shown in Figure 4.11, the number of context switches required is one of the main reasons for performance penalties in this architecture. The domain transitions occur in the following order: child → OS → host → OS → host → OS → child, a total of 6 transitions per system call, instead of the two transitions that occur in a typical,

untraced system call (process → OS → process), which is a threefold increase in overhead. However, as our performance measurements show, these costs are ameliorated over the lifetime of the program, as other factors like waiting for IO or the CPU tend to dominate the transition costs, leading to far more acceptable performance overheads. We discuss performance characteristics in detail in the next chapter.

In addition, due to the very nature of this mechanism, there are several potential security vulnerabilities that emerge, which are discussed extensively by Garfinkel [39]. However, solutions have also been proposed to many of these deficiencies [40, 101]. We discuss our own approach to these problems extensively in Chapter 6.

We default to a delegating approach wherever a security risk is possible, by executing the system call in the host, on behalf of the child. This helps to eliminate race conditions and timing attacks caused by TOCTOU (Time of Check To Time Of Use) bugs [40]. However, more innocuous system calls are simply passed through for the Operating System to execute. The decision on which system calls are innocuous must be made on a case by case basis, and is determined by whether the system call is read-only (such as getting the system time), or whether it can affect the execution state of the host. Allowing read-only system calls is safe, as discussed in Section 3.4, since information flow security is not amongst our security objectives. However, if a more secure implementation is desired, even read-only system calls may have to be vetted based on a user-defined policy.

The greatest difficulty lies however, in ensuring that the child's actions are confined to its shared memory areas, as well as ensuring that the host and the child continue to share state. Some of the most problematic calls are the memory mapping related calls, as well as file handling calls.

Memory mapping related calls, such as *mmap*, *mprotect* and *munmap*, are handled by carefully modifying the parameters in such a way that a file being mapped into memory is mapped into both the parent and the child at the exact same memory locations, thus keeping the two processes' memory maps in synchronization.

File descriptors are handled by the technique of passing them to the child via domain sockets, as described by Noordende et al. [101].

4.3.5 Handling of Unsafe Instructions

It should be noted at this point that the loaded library can execute any machine instruction it desires. In contrast to techniques that explicitly rely on filtering out potentially dangerous instructions through static analysis, we focus on a runtime approach. This helps to:

a. Prevent falsely flagging valid executables as malicious;

b. Provide maximum flexibility for compiler optimizations as well as developers to hand tune their libraries as desired; and

c. Make the technique immune to processor architectural changes, allowing the latest and greatest features to be used, such as newer SSE instructions, with no changes to our isolation container.

These represent a tremendous advantage and simplicity over other approaches.

The reason that this is possible is because we rely on the natural split between privileged and unprivileged instructions provided by hardware protection rings, and focus on the transitions between these rings. Therefore, any instruction that executes in the child's privilege ring is contained within its address space. It is at the border that malicious code must be caught, since this is the only point at which it can affect the outside world. We achieve this through the system call filtering mechanisms that have been described thus far.

4.3.6 Threading

This implementation of LibVM currently does not support threads executing within the library due to time constraints, although our hardware supported implementation, described in Chapter 5, does. However, since we rely on system call interpositioning mechanisms, it is possible to add on threading support, in a manner analogous to that followed by others [101].

4.4 PERFORMANCE OF THE ISOLATION ARCHITECTURE

We provide a comparative performance analysis of this implementation in the next chapter, along with our hardware virtualization based implementation.

4.5 CONCLUSION

This chapter has described the implementation details of the system call interpositioning-based implementation of LibVM. It has shown how a practical library isolation mechanism can be built based on our defined interface, proving the validity of the design. It has also shown how such an interface can be implemented by simply relying on existing process isolation facilities provided by the operating system, in concert with hardware mechanisms. To our knowledge, this is the first approach that uses these mechanisms for the purpose of isolating individual shared libraries. Although the described implementation has been limited to Linux, the general techniques and ideas can be easily transferred to other operating systems.

The main drawbacks of this system are the performance penalties induced due to the inherent overheads of the *ptrace* mechanism, as well as the nature of the *ptrace* mechanism itself being prone to introducing race conditions. However, we have pointed out various techniques available to mitigate these drawbacks and have shown that overall, this is a viable mechanism for isolating libraries. We propose enhancements to this mechanism in Chapter 7.

Chapter 5: Hardware Solution – Virtualization Based

5.1 INTRODUCTION

As described in the previous chapter, the process tracing based implementation of LibVM had several drawbacks, including performance penalties, which were difficult to mitigate through available mechanisms. This motivated us to explore other options that could provide a mechanism for intra-address space isolation. As elaborated in the literature review (Chapter 2), a relatively newer mechanism is the hardware virtualization support provided by modern Intel and AMD processors (although the original idea dates back to the 1960s and was implemented in the likes of the IBM System/360 Model 67 and later in the System/370 [49, 117]). Based on our literature review, we considered this the most promising unexplored candidate for the task. In order to test this hypothesis, we followed a staggered approach, by first testing the solution on an existing platform, and then migrating it to a standalone solution implementing the LibVM interface. This chapter describes the approach in detail.

5.2 APPROACH

The hardware virtualization based solution comprised of design, development and testing carried out in two separate stages. The first stage consisted of testing the viability and performance characteristics of the overall concept by integrating hardware virtualization support into an existing system of a similar nature. A key motivator for this stage was the potential overhead in the approach described by Adams and Agesen [1]. Once we had validated that the performance characteristics were indeed satisfactory, we went ahead with creating a standalone solution, based on the LibVM interface.

5.3 STAGE 1 - GOOGLE NATIVE CLIENT BASED IMPLEMENTATION

This section provides an in-depth discussion of the implementation, the performance figures of the implementation compared to software based solutions and analyses its overall suitability to LibVM's purposes.

5.3.1 Implementation Overview

In order to test the hypothesis that virtualised technology would be a suitable mechanism for isolating applications, we chose an existing solution as a base on which to add on this hardware-assisted mechanism. We chose Google's Native Client (NaCl) [144] sandbox as the basis, given its support by a large software enterprise, its open source nature and the fact that it is actively maintained. In order to differentiate our hardware virtualization based isolation container from the Software Fault Isolation (SFI) based Native Client sandbox structure, we will refer to our container as the "VT Sandbox".

The VT Sandbox was added onto NaCl in such a way that it supported both standard ELF (Executable and Linkable Format) executables as well as the modified native executables supported by Google Native Client. An additional advantage of this is that we can directly execute standard GCC [41] compiled ELF executables, whereas NaCl requires a custom tool chain, which is a modified version of GCC generating non-standard executables [144]. We utilise the ELF loader provided in the NaCl implementation to create the in-memory layout of the ELF executable. In this section, we describe the loading and execution process of a typical component that may be integrated into a full application.

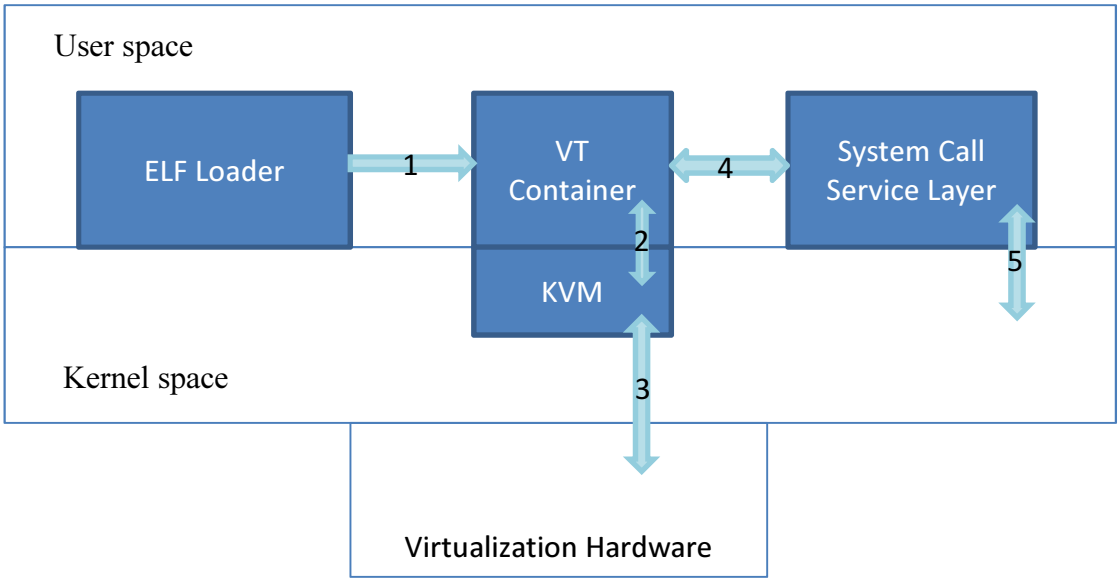


Figure 5.1: Implementation of the VT sandbox

Figure 5.1 shows the VT Sandbox's components at an implementation level. We read and parse an Executable and Linkable Format (ELF) executable program. We then load the executable into our VT Sandbox, which is where the bulk of our implementation lies. The VT Sandbox is responsible for the safe execution of the component and the mediation of any potentially dangerous instructions or activity. The VT Sandbox is built on top of KVM (Kernel Virtual Machine), which provides a layer of abstraction over the lower level hardware virtualization instructions, in the form of a device driver [107].

If a component makes a system call, the VT Sandbox carries it out on behalf of that component. Here too, our prototype saved a significant amount of work by building on top of NaCl's system call layer. The NaCl implementation provides a highly restricted system-call layer which we have modified to suit our needs.

The typical process, as per the sequence numbers in Figure 5.1, is as follows:

1. An ELF executable is loaded into the VT Sandbox.
2. The VT Sandbox uses the KVM device driver to execute the component.
3. The KVM module interfaces with the virtualization hardware and shields the layers above from the specific processor in use (Intel or AMD).
4. System calls by a component are intercepted and passed on to the modified system call layer.

5. The system call layer may invoke the operating system to carry out the actual system call and return results to the component.

The following sections give a detailed, step by step description of the tasks carried out by the VT Sandbox, during various stages of execution.

5.3.1.1 Initialisation of a component

This section describes the step-wise process for initialising the VT Sandbox and loading an executable into it for execution.

1. When an ELF executable is launched, the module is initially read in and the ELF header parsed. Our implementation supports standard ELF files as well as Google's customised ELF format.
2. The ELF executable is mapped into a contiguous block of memory, which is 256MB in size by default. This 256MB block serves as the initial physical memory for the virtual machine. Thus, there is a 1:1 correspondence between this memory block and the physical memory map seen by the virtual machine. If the ELF executable is in NaCl format, the first 64KB of this memory is padded with nulls, similar to NaCl's default implementation, which helps in the detection of null pointer exceptions. We also use the "trampoline" code used in NaCl. This trampoline code is used to exit the VT Sandbox in NaCl executables and carry out system calls and is described in detail below. The executable's text and data sections come afterwards. The rest of the memory is uninitialised.
3. Once the executable is mapped in and the memory area initialised, we initialise the Virtual Machine Control Block (VMCB) needed by the processor, specifying the aforementioned memory area as the physical memory block used by the virtual machine. We use KVM's abstraction layer to initialise the VMCB.
4. Once the virtual machine's processor control block is defined, we then initialise the processor registers and switch the machine directly into 32 bit mode. In this way, we avoid having to write a bootstrap loader which would switch the processor from 16-bit real mode to 32-bit protected mode.

Protected mode is set by setting the Protection Enable (PE) bit in the CR0 register [63].

5. Before protected mode can be properly used, the machine's Global Descriptor Table (GDT) [63] must be initialised. In order to simplify our implementation, we disable paging hardware altogether and use segmentation hardware only. We use a flat memory model, and the GDT is initialised with a code segment which is the size of the text portion of the memory map. Therefore, the total amount of addressable memory on a 32-bit system is 4GB, while the actual memory is constrained by the memory available to the Operating System process within which the sandbox executes.
6. We make the data segment span the entire virtual memory and the stack segment and other segment registers such as FS, GS and ES are also set to use a flat memory model, by spanning the machine's allocated physical RAM (ES is an extra segment used by certain machine operations such as for far pointer addressing and FS and GS are general purpose segment registers which, while having no processor defined purpose, can be used for implementation specific purposes). We do not use Local Descriptor Tables and therefore do not need to initialise the relevant structures.
7. Thus, we directly bootstrap a minimal virtual machine with the processor already in 32-bit mode and assigned a flat memory model, greatly simplifying the programming model for a component. The memory map is shown in Figure 5.2.

5.3.1.2 Execution within a VT Sandbox

Execution of a component within our VT Sandbox begins as follows.

1. After initialisation of the virtual machine, we set the VM's Instruction Pointer to the component's entry point. In keeping with Native Client's implementation, only ELF executables which are statically linked and the relative locations zero based are supported, so that there is no need to perform any relocation.

2. An initial stack is set up for the program and the base pointer and frame pointer are initialised to point to the top of the stack. Any command line arguments used are pushed onto the stack area and the memory is adjusted as required.

3. The virtual machine is then launched via KVM, and execution begins from the program's entry point.

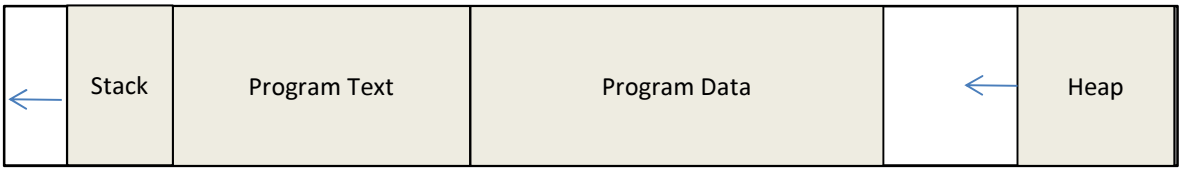


Figure 5.2: Component memory layout

5.3.1.3 Initialisation of the C runtime environment

The first task performed by the running program is to initialise the C programming language runtime environment, which is needed for basic input/output and for accessing system services. NaCl executables use a modified version of the *newlib* C runtime library [108], which is statically linked with the component. We support this same version of *newlib* so that direct binary compatibility with NaCl components can be enabled.

The *newlib* runtime initialises itself by allocating memory for the "Thread Control Block", which is a structure used by the runtime to maintain thread related information. Subsequently, it makes a system call to initialise the corresponding operating system thread. NaCl's default implementation additionally stores a pointer to the Thread Control Block in the GS segment register. In our implementation, we modify the virtual machine's GS segment register instead. This is an example of the kind of modification needed at the system call layer in order to make NaCl compatible with our implementation. Figure 5.3 shows the typical sequence of actions which take place in our VT sandbox during a system call.

1. A normal ELF executable will initiate a system call by invoking INT 0x80 or by using the fast system call instructions. In the case of Google's Native Client executables, it does this by jumping to the NaCl trampoline mechanism where each system call goes through a trusted code routine.

2. We modify this routine to suspend execution of the virtual machine by executing a sequence that triggers a *VMEXIT* event, which was described in Chapter 2. When this occurs, libkvm performs a call back to our VT Sandbox, allowing us to intercept the system call.

3. Upon interception of the system call, we use a modified version of the Native Client's dispatcher routine to figure out which system call was requested. We carry out the actual system call after verifying the parameters (e.g. checking whether the parameter values are in range, of the expected format, whether referenced handles are valid etc.). The component can only execute a subset of the available system calls and these are completely controllable, making the execution of arbitrary code secure.

4. Before the system call is executed, the parameters are validated to ensure that the values are within range and that only permitted system resources are accessed.

5. Once the system call is complete, the results are set in the virtual machine's registers and the stack and frame pointers adjusted to store the necessary return values.

6. Finally, execution resumes by changing the virtual machine's instruction pointer to resume execution from the return address stored on the stack.

7. The untrusted code resumes execution.

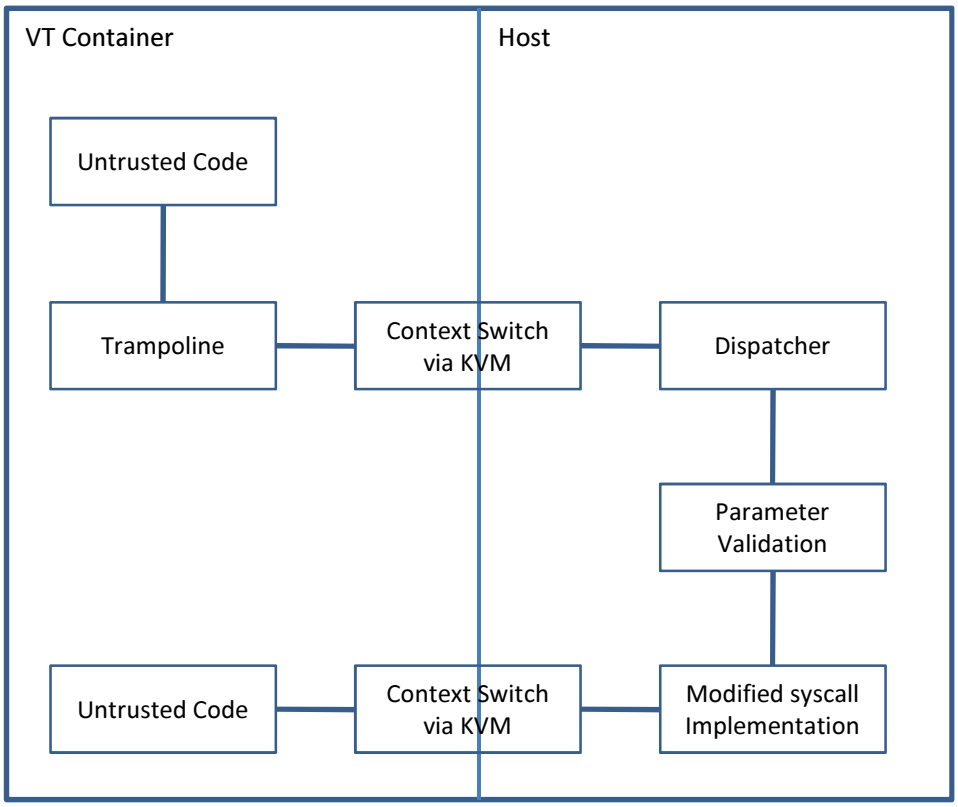


Figure 5.3: Execution sequence for a system call

5.3.1.4 Handling of unsafe instructions

Potentially unsafe machine instructions are handled by trapping the execution of sensitive instruction types. The process is described below.

1. The execution of instructions defined as sensitive causes the virtualization hardware to trigger an exit into the VT Sandbox. By default, all privileged ring 0 instructions are trapped. All other instructions are allowed to execute with no constraints within the virtual machine. The VMCB is configured to trap these sensitive instructions through the KVM layer.
2. If an attempt to execute a sensitive instruction is detected, our trap handlers are invoked. The trap handler then takes steps to terminate the offending component.

We also use this trapping functionality for our implementation of system call handling. However, in that instance, we execute the system call on the component's behalf and return the results via the virtual machine's stack.

5.3.1.5 Threading

We provide an extremely simplified implementation of threads, with one virtual processor per thread.

1. When a component requests the creation of a new thread, we create a new virtual machine, but map in the same memory belonging to the creator's thread. In other words, both virtual machines share the same physical memory.
2. Once the virtual machine is initialised, the virtual processor's instruction pointer is set to the thread's entry point.
3. The GS register must also be set to point to the Thread Control Block of the new thread.
4. The virtual machine execution is then started, thereby having two virtual processors executing the two different threads.

5.3.2 Comparative Analysis

This section directly compares our solution to the Google Native Client, which focuses on using static analysis, as well as Vx32 [32], which emphasises runtime binary translation.

Table 5.1 provides an overview of steps needed to load and execute a component. As can be seen, our approach saves significantly on load-time complexity by removing the code verification and patching steps altogether. Our approach thus eliminates an entire class of problems related to static analysis and code verification, as discussed below.

VT Sandbox	Google Native Client	Vx32
Load component Switch to sandbox Execute code Trap on exception or execute till end	Load component Verify component Patch unsafe instructions Switch to sandbox Execute code Trap on exception or execute till end	Load component Create sandbox Translate code fragment Execute code fragment Repeat Steps 3 and 4 till execution ends or an exception occurs

Table 5.1: Comparison of steps to load and execute a component

Our implementation saves significantly on execution overheads since no additional instructions need to be inserted. Our initial measurements show that code bloat for Google Native Client is significantly high, because it requires that jumps be aligned to 32-byte boundaries. Neither our implementation nor Vx32 have such an alignment requirement, so they can significantly lower the size of the executable code. In addition, while we suffer a heavier penalty for sensitive instructions since the trap handler may need to perform a context switch back to user space in order to handle the instruction, the complexity of our implementation is greatly reduced as no binary code patching needs to be done.

As shown in Table 5.2 our technique also differs fundamentally at a conceptual level. Google's NaCl relies on pre-execution checking, using static analysis, and rejects a component if it does not meet its defined criteria. It also inserts run-time checks into the code. Our technique simply starts executing the component and aborts if unsafe instructions are encountered. Therefore, our focus is on run-time checking as opposed to both load-time and run-time checking. We argue that our technique is faster, less complicated and more robust. In particular, our approach will execute components which contain potentially unsafe instructions only in dead code, whereas NaCl will not execute such components at all.

Notably, our implementation is immune to problems that may occur in static analysis and associated verification bugs. In contrast, a security contest conducted by Google to test for loopholes in NaCl revealed several bugs in the code verifier and patching

system, which allowed for arbitrary code execution vulnerabilities, enabling a malicious component to escape component isolation [43]. Our implementation is not vulnerable to such errors, since execution is entirely constrained to the virtual machine, making for a more secure implementation. We have tested this by executing similar classes of bugs reported in the Native Client security contest, and showed that the code is unable to break free from the confines of our container. A detailed example is discussed in Section 5.3.3.4.

Our technique also offers the advantage of being easily adaptable to 64-bit code, something that introduces much greater complexity to NaCl and Vx32. 64-bit support was added onto NaCl through a paging based mechanisms at a later stage [118], while the 32bit version uses segmentation hardware, which is no longer available on Intel's 64-bit architecture [144]. 64-bit support is altogether absent in Vx32 as it too relies on segmentation hardware.

However, we do share similar vulnerabilities as NaCl and Vx32 at the system call layer, since any loopholes at this level can be exploited in an identical way. (In both systems the problem can potentially be avoided by validating parameters before execution of system calls.)

	LibVM-VT	LibVM-ptrace	Google Native Client	Vx32
Approach	Hardware Virtualization	Process tracing	SFI	SFI
Technique	Minimal virtual machine container	OS provided ptrace mechanism	Static Analysis	Runtime binary translation
ISA	32bit (currently)	32bit (currently)	32bit/64bit	32bit (currently)
Specific Hardware features used	Intel VT/AMD SVM	None (standard OS facilities)	Segmentation	Segmentation
Compile time requirements	None	None	Customised tool chain with code alignment requirements	None
Advantages/ Disadvantages	• Smaller TCB • Simpler model • Hard isolation boundaries • No restrictions on allowed instructions • Compatible with 32/64-bit • Hardware dependent • Maintains address space transparency	• Smaller TCB • Simpler model • Hard isolation boundaries • Drawbacks in ptrace interface • High context switching overheads • Maintains address space transparency	• Large TCB • Complicated model • Restrictions on allowed instructions • More restrictions and overheads in 64-bit mode • No address space transparency as segments are 0 based. • Requires custom tool chains • Requires rewriting libraries as well as host application • Requires code verification and patching • Possible false positives	• Large TCB • Complicated model • 32-bit only • No address space transparency as segments are 0 based. • Requires custom tool chains • Requires rewriting libraries as well as host application • Requires code verification and patching

Table 5.2: Comparison of approaches

5.3.3 Effectiveness of the Isolation Architecture

In this section we evaluate the effectiveness of our method by demonstrating its use through examples, while comparing and contrasting the results with Google's Native Client and Vx32. These examples were incorporated into test routines and applied against both systems. We show that our solution provides stronger security guarantees than NaCl and Vx32, while eliminating the complexity of the code analysis and verification process and the need for runtime binary translation.

5.3.3.1 Example 1 – Handling illegal instructions

Figure 5.4 shows a C code fragment containing an illegal assembler instruction. It illustrates an attempt to directly access an I/O port through the *out* instruction. Typically, user level programs are disallowed from accessing I/O ports directly. This kind of problem could arise both from malicious code or a programming error such as an attempt to divide by zero.

```
#include <stdio.h>

void run_test() {
asm ("movl $32, %%eax; \
      out  %%eax, $0xf1"
      :
      :
      :"%eax"
    );
}

int main(int argc, char* argv[]) {
  run_test();
  return 0;
}
```

Figure 5.4: Example code containing an unsafe instruction

We can selectively forbid sensitive instructions that should not be executed, and our VT Sandbox adopts the policy of disabling such instructions by default. This is done by configuring the VMCB to intercept the specified instruction, in this case, the *out* instruction. The virtualization hardware will then automatically trigger a trap when an attempt is made to execute the instruction. We can then mediate and terminate the module gracefully or allow it to continue if the instruction is deemed innocuous.

When compared to NaCl, the protection offered is similar. NaCl would refuse to allow execution of the above component since the verification process would detect the presence of the disallowed instruction statically. Vx32 allows execution of the component but, because it dynamically translates the next 'fragment' of code to be executed, it may abort during runtime if it encounters the illegal instruction during its binary translation process, even when the instruction itself is only executed conditionally. We, however, trap only when an illegal instruction is actually reached, if ever. The advantage of this is better illustrated by the example below.

5.3.3.2 Example 2 – Reducing false positives

In this example we modify the previous program slightly to conditionally execute the illegal instruction by only calling the `run_test` method if condition '`argc < 0`' is true. In practice, `argc` will never be less than 0, which means that this will be dead code in the running program and the potentially harmful instruction can never be executed. NaCl however, would nevertheless generate a false positive and refuse to allow execution of the program and Vx32 would fail at runtime when it encounters that fragment of code. Our method entirely eliminates this class of false positive altogether.

Although the above example is somewhat contrived, it serves to illustrate that NaCl is always forced to err on the safe side, and disallow a range of instructions which are generally innocuous but potentially unsafe, such as all instructions that modify the x86 segment state, including `lds`, far calls, etc. Vx32 also suffers from similar constraints. Our method does not require such caution, as execution is entirely constrained to the virtual machine, and loading segment registers for example, only affects the virtual processor. Therefore, it highlights a strong advantage that our VT Sandbox has over NaCl.

5.3.3.3 Example 3 – Addressing errors

The program in Figure 5.5 highlights an extremely common programming mistake. It makes use of an uninitialised pointer which performs a 'wild store' into memory. When an attempt is made to access memory outside of the boundaries defined by the VMCS, the VT hardware can be configured to trap into our specific error handler.

```
#include <stdio.h>

void run_test() {
  int *test, offset = 1024*1024*10;
  test[offset] = 10;
}

int main(int argc, char* argv[]) {
  run_test();
  return 0;
}
```

Figure 5.5: Example code with an uninitialised pointer

In comparison to NaCl and Vx32, the protection performance is identical, since both NaCl and Vx32 use x86 segmentation hardware to enforce similar constraints.

5.3.3.4 Example 4 – Addressing exploits

This example demonstrates a situation where our technique is safer than NaCl. This example is an actual bug detected and submitted during the Native Client security competition, where several flaws in the verifier were identified [43]. Although the bug has been subsequently patched, it serves to illustrate the potential danger of instructions missed during the verification process, and that eliminating the verification process provides far greater security guarantees as well as flexibility.

The exploit took advantage of a miss in the verifier, where opcode prefixes for 2 byte instructions were not constrained. The code fragment in Figure 5.6 illustrates the key instructions used in the exploit. It works by pushing the value `0x10001` onto the stack, which points to the middle of the first `mov` instruction, which now represents the restricted instruction `int 3`.

```
cs:0x10000: mov eax, 0xCCCCCCCC
…
…
cs:0x10080: mov    $0x10001,%ebx
cs:0x10085: push   %ebx
cs:0x10086: xor    %eax,%eax
cs:0x10088: test   %eax,%eax
cs:0x1008a: data16 je 0x7f4f
cs:0x1008f: add    %al,(%eax)
```

Figure 5.6: Example code with an illegal jump

Normally, such an unaligned jump would be disallowed and detected by the verifier. However, the bug exploits the 16-bit data prefix to truncate the jump target, which the verifier miscomputed. As a result, the code jumps into a **`ret`** instruction in the trampoline code region, which results in a return to the address pushed onto the stack, in this case, the illegal **`int 3`** instruction. In our approach the illegal instruction is detected when it attempts to execute.

While this problem was patched in NaCl soon afterwards, it serves to illustrate the difficulty in writing a fool-proof static verifier. As a result, even legal instructions need to be severely restricted in order to prevent potentially harmful exploits. This same class of problems applies to Vx32, as the binary translation process is vulnerable to similar circumvention. In our method, since all execution occurs within the confines of a virtual machine, code execution can be allowed in an unrestrained fashion, as long as proper checking is done when switching between borders. This border crossing happens only during system calls, making our method far simpler and easier to verify as correct. Therefore, the above example executes but is unable to bypass the confines of the virtual machine (barring, of course, any actual errors in the hardware implementation).

5.3.3.5 Example 5 – General-purpose applications

In order to evaluate the technique in a more real-world situation, we create a modified version of the "bzip2" compression program with various bugs inserted to test isolation effectiveness. This included illegal instructions within dead code, accidental array bounds violations and other suspicious but harmless code. We ran this modified version under the Native Client, Vx32 and the VT Sandbox. We found that, while all three effectively prevented malicious code from executing, the pre-emptive approach of the Native Client resulted in increased false positives, even though the actual code did nothing harmful. Vx32's binary translation process triggered false positives only when the code fragment was encountered, although it would abort even if the instruction itself was never executed. Our solution delays this process to the last possible moment, when the instruction is actually executed, and therefore, does not result in false positives.

5.3.4 Performance of the Isolation Architecture

To ensure that our solution does not introduce unacceptable overheads, we executed some micro-benchmarks of illustrative cases as well as some large scale benchmarks, keeping in mind that our current VT Sandbox implementation is merely a proof-of-concept prototype. In all cases our VT Sandbox solution was compared with Google's Native Client, and certain benchmarks were also run against Vx32. Performance was tested in three cases—native execution as a linux executable, execution within Google's NaCl Sandbox and execution within our VT Sandbox.

5.3.4.1 Micro-benchmarks

The micro-benchmarks were chosen to test performance under highly specific circumstances. These help to establish the upper and lower bounds that can be expected in best and worst case scenarios respectively.

We performed empirical performance measurements on four main workloads:

1. Execution of a simple loop based calculation.
2. Execution of a "null" system call.
3. Execution of I/O instructions (which require an operating system call and therefore, at least one context switch).

The measurements were repeated thrice, and an average value was obtained. The results show that the overheads of our approach in compute-bound scenarios are comparable to those of NaCl with no significant differences in performance (keeping in mind that our prototype implementation utilises the NaCl system call layer for NaCl compatible executables).

In the second experiment, a simple transition in and out of the VM was performed in a tight loop, and added about 20% overhead in comparison to NaCl's performance.

In the third experiment, the system call execution overhead varied, with 10% being typical with an outlier case of 400%. The difference between the typical case and the outlier demonstrates that attendant circumstances of the environment, such as competition with other system processes, internal kernel buffering etc. can be dominating factors in determining overall overheads.

5.3.4.2 Large-scale benchmarks

In order to empirically measure how these approaches perform under more realistic workloads, we have tested their performance in several scenarios.

1. Execution of three compute-bound graphics performance tests provided with Native Client's test suite;
2. Earth: a ray-tracing workload, projecting a flat image of the earth onto a spinning globe;
3. Voronoi: a brute force Voronoi tessellation;
4. Life: a cellular automata simulation of Conway's Game of Life;
5. Quake; and
6. The SPEC2006 benchmark suite.

In all of the above cases, we disabled VSYNC (Vertical Synchronization) so that the rendering thread would not be put on hold until the display's vertical refresh had completed.

5.3.4.3 Graphics performance tests

The samples were built with nacl-g++ version 4.2.2 with compiler parameters `-O3 -mfpmath=sse -msse -fomit-frame-pointer`. The Linux `time` command was used to measure the execution time in all 3 cases.

Both Earth and Voronoi were executed with four worker threads for 1000 frames, averaged over three runs. Life was run as a single thread for 5000 frames. The results are summarised in Table 5.3.

Sample	**Linux Executable**	**Native Client**		**VT Sandbox**	
		Execution Time	**Overhead**	**Execution Time**	**Overhead**
Voronoi	34.13	19.18	–43.8%	19.12	–43.98%
Earth	11.38	11.64	2.28%	12.48	9.66%
Life	14.88	17.47	17%	17.88	20.16%

Table 5.3: Compute/graphics performance tests. Times are elapsed time in seconds. Lower is better.

Somewhat surprisingly, we found that both the Native Client and the VT Sandbox significantly out-performed the native executable in the Voronoi test. However, the results are consistent with those reported by Yee et al. [144].

In the other two instances, the results were as expected, with the native Linux executable having the best performance. The Native Client and the VT Sandbox had fairly similar performance in all 3 cases, with the VT Sandbox having a slight edge on the Voronoi example and a slight loss in the other two tests.

5.3.4.4 Quake

Quake was executed on Suse Linux 11.3, with kernel mode-setting switched off, at 1024x768 graphics resolution. Quake was built using `-O3` optimization. The version used was sdlquake-1.0.9 from www.libsdl.org. The results are shown in Table 5.4.

Run #	Linux Executable	Native Client	VT Sandbox
1	137.1	123.0	122.1
2	136.9	124.0	121.5
3	136.0	124.1	121.3
Average	136.67	123.7	121.63

Table 5.4: Quake performance comparison. Numbers are in frames per second. Higher is better.

While performance differences between nulls were minimal and almost negligible, we found that the native Linux executable performed best overall. The difference between the Native Client and the VT Sandbox were extremely small, with the VT Sandbox incurring a slight overhead of about 1.7%.

5.3.4.5 SPEC2006 results

The performance of our approach was tested primarily by executing the SPEC2006 benchmark suite. We compared our approach against

a. Native execution of the binary executable with no modifications;

b. Google's NaCl implementation; and

c. Vx32.

It should be noted that only the C integer benchmarks are supported by the Vx32 runtime at the time of writing [32].

The tests were run on two machine configurations. Figure 5.7 shows the results on a Core-i5 540M processor dual core CPU with 4GB of RAM, running on OpenSuse 11.3 with kernel version 2.6.34.07. The executables were compiled with the `-O3` gcc flag in all three cases. The vertical axis is the ratio of each tests' execution time against a reference execution time provided by SPEC. Higher values are better.

As expected, in all cases, native execution of the unmodified binary provided the best results. In all except one case, the NaCl execution time was slightly better than the VT Sandbox execution. This was not unanticipated, since the context switch overhead takes a toll on execution times. However, in all cases, the performance of the VT Sandbox was extremely competitive, with the overhead being less than 1% in all cases, except for the mcf benchmark, which peaked at 4%. In contrast, Vx32's results were slower, with overheads increased up to 4%.

The tests were rerun on a Core-i7 920 quad core processor with 4GB Ram as shown in Figure 5.8. The configuration was identical to the previous machine, with both kernel versions and executable compilation flags matching.

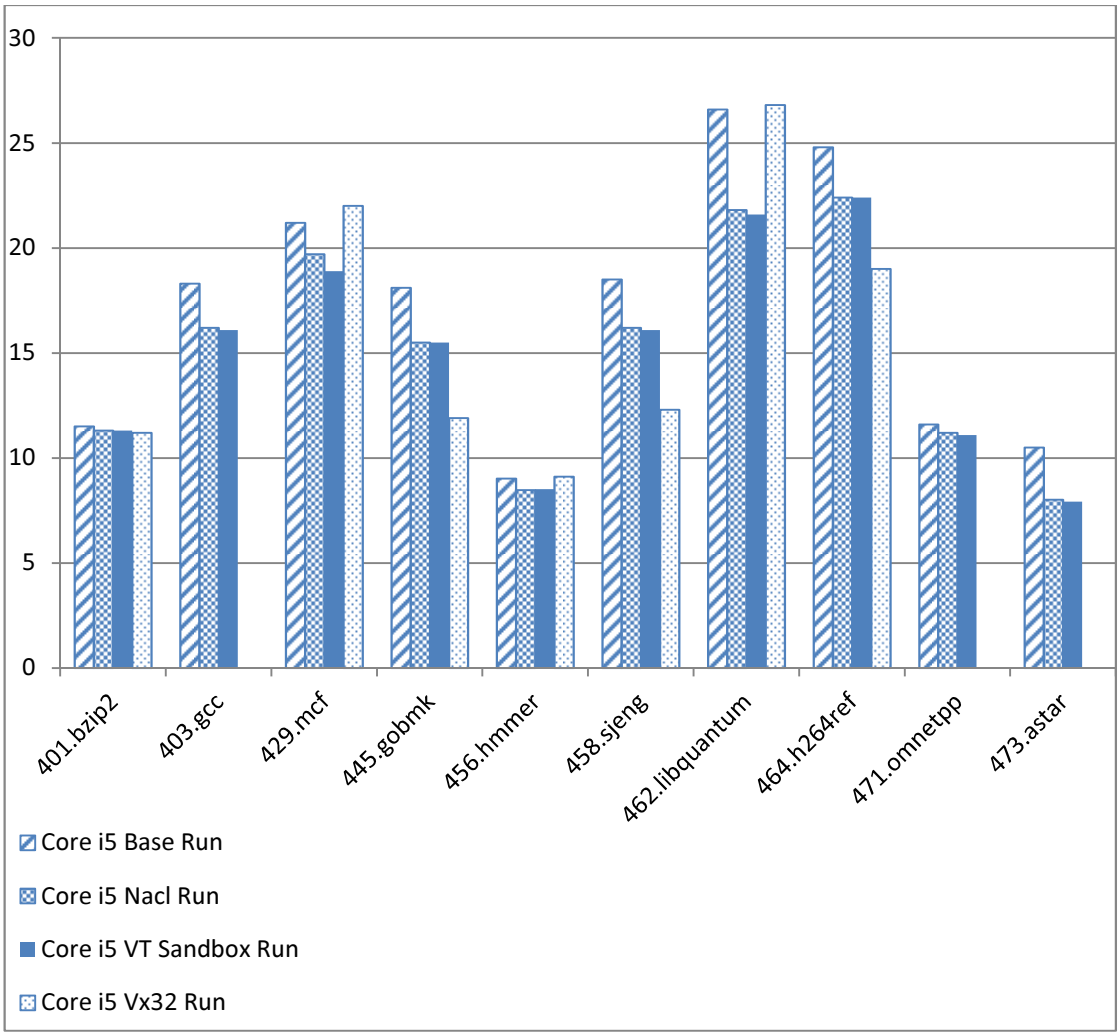


Figure 5.7: SPEC2006 on Core i5-540M processor

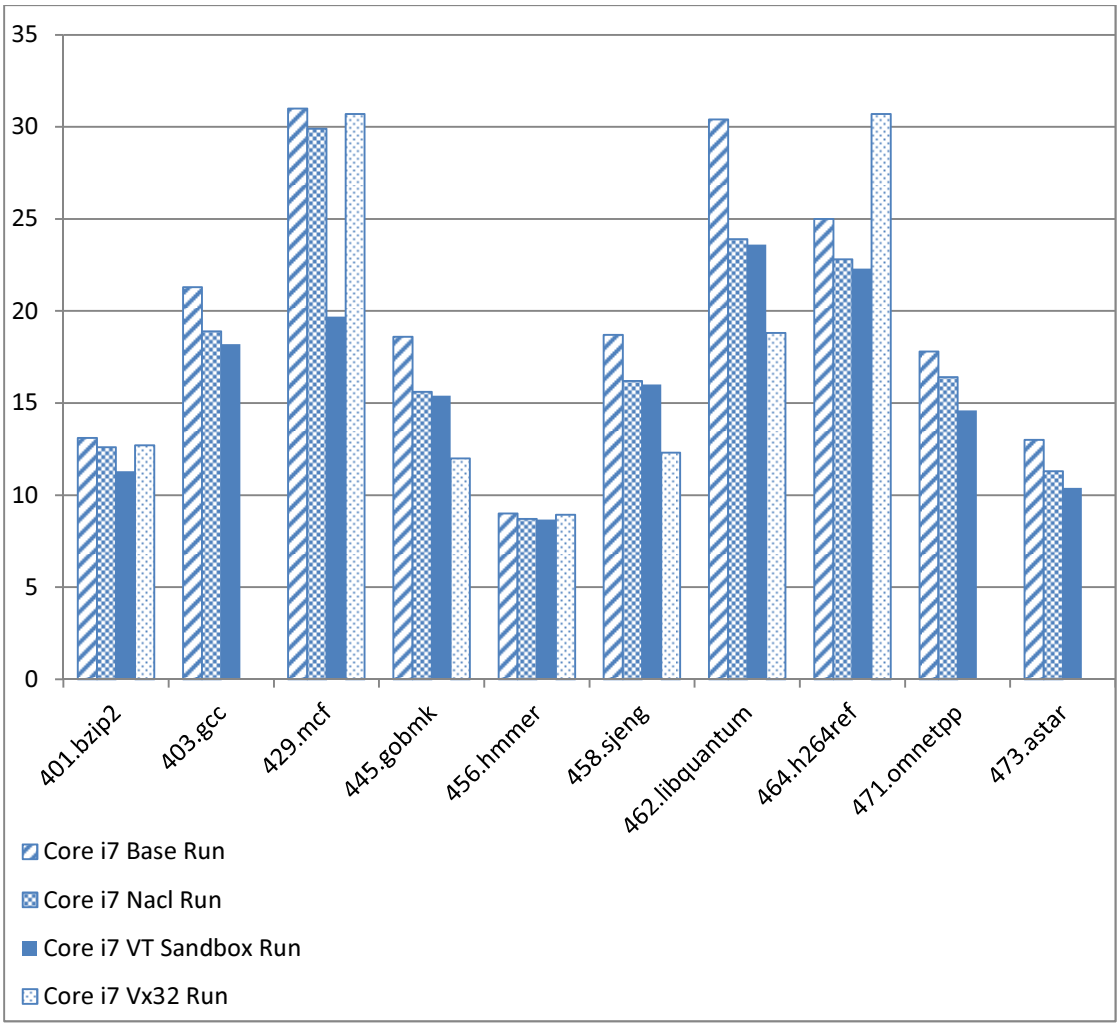


Figure 5.8: SPEC2006 on Core i7-920 processor

The results were similar, although the differences were more pronounced this time. The overheads ran as high as 10%, although the mcf overheads were far more pronounced at 34%. The difference is mainly attributable to cache locality and context switching overheads. However, since this was the only anomolous case, we do not consider it to be representative of average case performance.

Overall, we found that the performance of the NaCl production code, current Vx32 implementation and our initial VT Sandbox prototype were competitive with each other. This is despite the fact that our prototype currently suffers from excessive context switching due to its reliance on the KVM driver.

5.4 STAGE 2 – STANDALONE LIBVM IMPLEMENTATION

Once we had ascertained that the performance of the hardware virtualization based solution was satisfactory, the second stage involved creating a standalone implementation based on the LibVM interface. We refer to this particular implementation as LibVM-VT, in order to distinguish it from the software based implementation, which was referred to as LibVM-ptrace.

Figure 5.9 depicts the basic address space layout of a LibVM-VT isolated shared library.

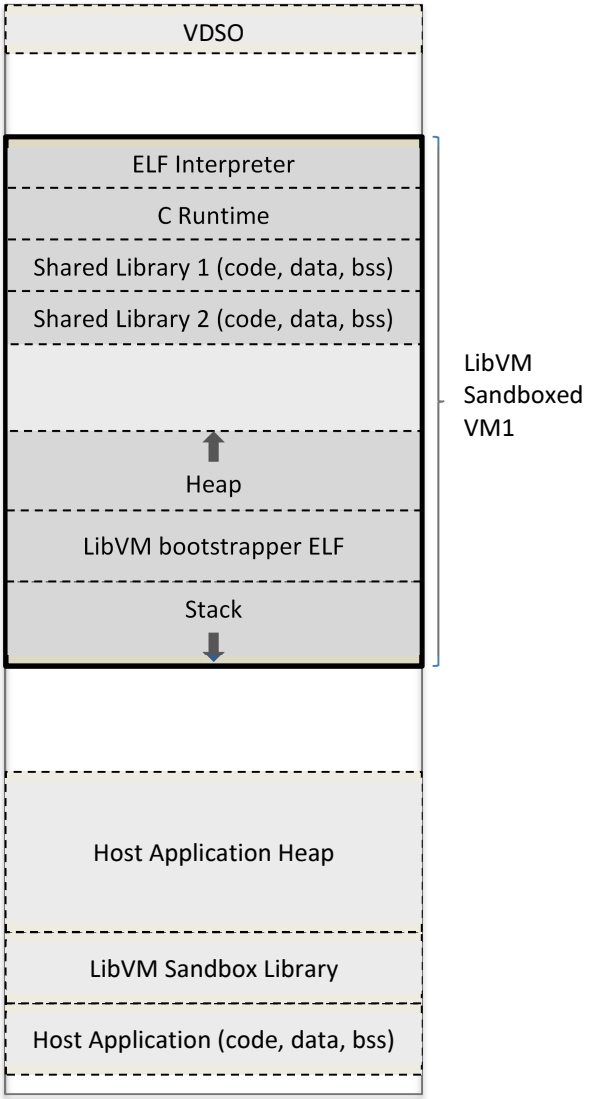


Figure 5.9: LibVM-VT address space structure of guest and host

LibVM-VT works by partitioning the host address space into several sandboxed regions. Each sandboxed region is a lightweight virtual machine. The virtual machine

and its host container share their address spaces as shown in Figure 5.9. Therefore, there is complete parity between a guest address and a host address, which is the key to enabling transparent sharing of data between the shared library and its host.

Each virtual machine is a simple execution container, in which arbitrary code can be executed safely. Code executing within the virtual machine cannot exceed its defined boundaries and it cannot affect the rest of the system other than through system calls. The LibVM-VT runtime intercepts all system calls, thus ensuring that code executing within the container cannot bypass security measures.

In order to load an existing shared object unmodified into this execution container, we must use a dynamic linker, which loads the shared object and its dependencies, as well as carrying out relocation of the executable image. To avoid the complexity of writing our own dynamic linker, we utilise the system's existing ELF interpreter [84] for the purpose. The basic process we follow is to emulate the operating system's process initialisation routine, by first loading the system's ELF interpreter into the address space, and then executing the interpreter, which in turn is requested to load a small bootstrap executable which we provide. The interpreter dutifully performs these tasks, unaware that it is executing within a virtualised container. We intercept all system calls made by the interpreter, and provide appropriate emulations which confine all operations to the isolation container's address space. Once the interpreter executes our bootstrap code, we have a fully initialised "mini-process", along with a C runtime and dynamic linker, all of which reside within the execution container. We then utilise the dynamic linker to load additional shared libraries in turn, exactly as would occur within a standard process.

5.4.1 Initialisation

The virtual machine bootstrap process is triggered when *libvm_initialise()* is invoked for the first time as shown in Figure 3.3. The process is as follows.

1. The LibVM-VT runtime first creates an instance of a light-weight virtual machine.

We utilise the KVM library [107] to create a simple virtual machine. The virtual machine consists of a single CPU and is set to the same architecture as the hosting process, in this case, a 32bit Intel x86 machine. We emulate the CPUID instruction to match the host, create a flat memory model, enable all instructions including SSE and create a basic virtual machine which serves as our isolation container. It should be noted that this is not as expensive a process as it may sound at first, as KVM simply creates the processor data structures used by Intel VT/AMD SVM as well as page tables used by the virtual machine, and there is no need for the emulation of complex devices. Furthermore, this virtual machine/isolation container can be used to load multiple libraries and is therefore a one-off cost that is amortized over the lifetime of the program.

2. The virtual machine memory layout is then created and a shared memory region between the host application and the virtual machine is defined (as shown in Figure 5.9).

   This involves allocating a region of memory which is shared between the guest virtual machine and its host process. The memory within the guest virtual machine is defined at the exact same addresses as the host, thus achieving parity in the memory layouts, which serves our goal of transparently passing memory references between guest and host.

   At the very top of the VM's address space, we map-in the Linux VDSO (Virtual Dynamic Shared Object) [103]. The Linux VDSO is a springboard that is used by the Gnu C library to make system calls and therefore must be mapped into a fixed location in memory.

3. LibVM-VT uses a simple ELF loader which loads the Linux ELF interpreter [84] and our bootstrapper into memory at the top of LibVM-VT's allocated address space.

Our ELF loader performs a few basic integrity checks, such as ensuring that the ELF program segments fall into valid memory regions. However, extensive security checks are not necessary as the ELF loader is not a part of the attack surface. This is because the loader is only used to load the system's ELF interpreter and our own bootstrapper, and not untrusted libraries. The ELF loader also maps our bootstrapper executable into memory, and places it immediately after the ELF interpreter.

4. LibVM-VT sets up the VM's stack and begins executing the VM.

   We create the data structures necessary for the ELF interpreter, such as the AUXV vector specifying system parameters and copy all system environment variables onto the VM's stack. The AUXV vectors instruct the ELF interpreter, where in memory our bootstrapper executable can be found, and the ELF interpreter will load and execute the bootstrapper as well as its dependencies, such as the C runtime library.

   Our bootstrapper executable is compiled as a position independent executable, with the -PIE flag, so that it can be placed anywhere in memory. This is in contrast to standard executables which have a fixed load address. This again helps us to ensure that there are no memory overlaps between the virtual machine and the host machine.

   We also copy environment variables and command line arguments onto the stack. Following this, we set the VM's program counter to the ELF interpreter's entry point and launch the VM.

5. The ELF interpreter initialises our bootstrapper.

   The ELF interpreter starts its boot up process, unaware that it is executing within a virtual machine, and carries out the same sequence of actions which it normally would during the execution of a process. This includes mapping our executable into memory, loading its dependencies, such as the C runtime library, and jumping to the entry point of our bootstrapper executable.

   We intercept all system calls made by the ELF interpreter during this process, and proxy all the operating system functionality, forcing the memory mappings for example, to fall within the allocated boundaries of the execution container.

6. Bootstrap completion.

   Once the bootstrapper's *main()* function executes, we make a standard system call with an unused system call number, which our interception layer recognizes as special. As parameters to our custom/special system call, the bootstrapper passes the address of the dynamic linker's symbol resolution routine – *dlsym*. This value is cached by LibVM-VT for all future symbol resolution within the execution container. This special system call also heralds the completion of the bootstrap process, and LibVM-VT suspends execution of the virtual machine and returns from its main initialisation routine.

5.4.2 Library Function Calling Sequence

We now describe the sequence that transpires when a function call is made from the host to a shared library. Whereas a direct function call can be made in standard POSIX, LibVM-VT must maintain the illusion that the same process is occurring, while in reality, ensuring that the library executes within an isolated environment.

Figure 5.10 highlights the actual process that takes place when a function call occurs.

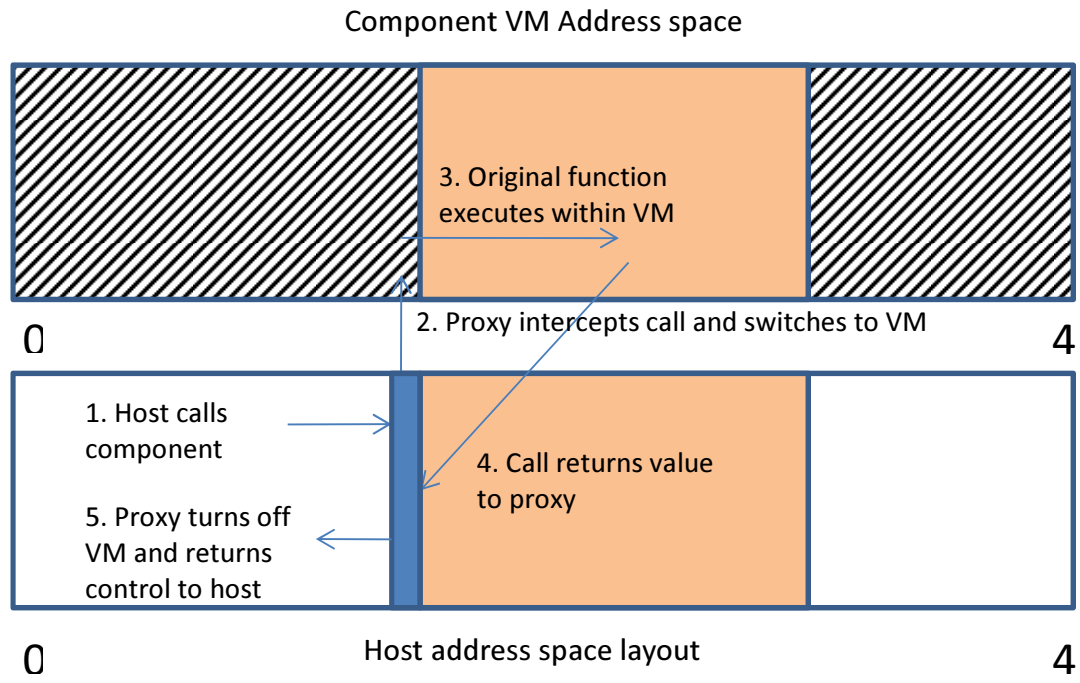


Figure 5.10: Invocation sequence of shared library call

Since the guest and component address space layouts are identical, a pointer in the host and a pointer in the VM refer to the same memory location. Therefore, everything is completely transparent to caller and "callee". However, the component can only access the solidly shaded memory areas in Figure 5.10, whereas the host can access any area. This way, passing pointers back and forth can be done without any pointer swizzling [140] or manipulation.

However, it should be noted that while addresses in the component address space can be freely accessed by the host, any additional memory areas must be specifically granted to the component. This means that only memory at page level granularity can be granted, since a page in the host address space must map into a page in the virtual machine's address space.

A host utilises the following steps in executing a function within a loaded component. The process is largely transparent to the host.

1. Host calls a function within the component.

   The function should have been obtained via *libvm_sym*, which returns a proxy function that shields the host from the details of the emulation layer. The proxy function is generated "on the fly" inside a specially allocated memory region. This function is binary patched to implicitly pass the *libvm_context* as well as the guest function pointer.

2. Proxy intercepts call and switches to VM.

   The LibVM proxy function copies the call parameters from the caller to the private stack of the VM. It then sets the instruction pointer in the VM to point to the actual function in the guest, and also sets the return address to our trampoline function. It then activates virtualised execution.

3. Original function executes within VM.

   As the function call executes within the VM, any system calls it makes are intercepted by LibVM through detection of privilege level changes, and channeled to the user-defined interceptor functions.

4. Call returns value to proxy.

Once the function has finished executing, it returns, but to the address of the proxy trampoline function that was originally passed to it by the LibVM-VT runtime. The trampoline function triggers an exit from the virtual machine. The stack values are then copied back to the caller's stack. Although, strictly speaking, there is information leakage at this point, this is entirely an optional process, and can be disabled if more secure semantics are desired. It is provided only to aid passing arguments by reference.

5.4.3 Effectiveness & Performance of the Isolation Architecture

We evaluated the functionality and performance of our LibVM system by carrying out several micro and macro benchmarks. The micro benchmarks were designed to measure several edge cases which can be used to glean the performance characteristics of LibVM, whereas the macro benchmarks provide a more holistic gauge. Both the hardware-based and software-based implementations of LibVM were evaluated.

5.4.3.1 Micro benchmarks

We carried out four main benchmarks.

1. Execution of a 'null' call which simply measures the overhead for transitioning in and out of the isolation container.

   This gives us a measure of pure isolation overhead, although such rapid switching would be unnatural in an actual program. Nevertheless, it is a useful measure of the most pathological case.

2. A highly-inefficient Fibonacci calculation in order to measure a compute-intensive workload.

   This provides a more realistic measurement through a compute intensive process, focusing on making execution within the container trump the number of transitions outside of the container.

3. A get-pid system call to measure raw system call performance.

   This scenario provides an estimate of the overhead incurred in a "plain vanilla" system call.

4. A file copy routine to measure raw I/O performance.

This scenario provides a measurement of situations where much of the time is spent waiting for I/O.

Sample	**Linux Executable**	**LibVM – Hardware Virtualization**	**LibVM – ptrace Jail**	**Linux RPC**
		Execution Time	**Execution Time**	**Execution Time**
Null call	1	926	4989	4962
Fibonacci	1	1.04	1.08	1.04
Get-PID	1	88.12	516	278.2
File copy	1	1	1	1

Table 5.5: Micro-benchmark results – Core i5

Sample	**Linux Executable**	**LibVM – Hardware Virtualization**	**LibVM – ptrace Jail**	**Linux RPC**
		Execution Time	**Execution Time**	**Execution Time**
Null call	1	1296	5518	5304
Fibonacci	1	1.07	1.07	1.1
Get-PID	1	91.28	397.37	320.26
File copy	1	1.02	1.04	1.02

Table 5.6: Micro-benchmark results – Core i7

Table 5.5 and Table 5.6 summarise the execution speeds on two different processors, a *Core i5* and *Core i7* respectively, both running identical SUSE Linux 11.3 installations. The figures are displayed as a proportion of the execution time of a basic Linux executable performing a local procedure call in a tight loop.

In the null call measurement, which simply measures the overhead of transitioning in and out of the isolation container, an RPC is 3 orders of magnitude slower than a local procedure call (no isolation), and is consistent with figures reported in literature [86]. However, LibVM is about 5 times faster than an RPC, demonstrating a significant, but expected, boost in performance. The main reason for increased performance is the reduction in context switching, as transitions are made only between the kernel and the process for each call. In the case of an RPC, a process switch is required, doubling the number of context switches, as well as reducing cache locality, thus incurring a commensurate performance penalty. The ptrace jail predictably has the highest performance penalty in such an extreme scenario, as it must perform an additional context switch due to the interpositioning layer.

The get-pid system call test measurements produced similar results. This time however, the native case also incurs a performance overhead due to a context switch, reducing the dramatic differences displayed in the null call case, where there were no context switches at all. The relative performance between LibVM, ptrace and RPC remain proportionate in both cases, as expected.

However, when the benchmark becomes IO or compute bound, the differences immediately vanish, as demonstrated by the Fibonacci and File copy benchmarks, since context switching overhead pales into insignificance.

While these measurements are unlikely to be found in real world applications, they serve to demonstrate that:

1. LibVM can perform, at best, up to 5 times faster than an RPC, when using hardware virtualization.
2. LibVM's ptrace-based isolation can be, at worst, 2 times slower than an RPC.

Real world performance differences however, are likely to be less dramatic, depending mainly on context switching and parameter marshalling overheads.

5.4.3.2 Macro benchmarks

The macro benchmarks are designed to measure both performance characteristics as well as porting effort needed to utilise LibVM. We execute the following benchmarks for this purpose.

1. Using the LibVorbis library to decode a Vorbis encoded audio file.
2. Using the BZip2 library to measure a compression algorithm.

   This example measures the passing of a large buffer to be decompressed in a compute-intensive run, followed by the return of the decompressed buffer to the caller.
3. Using LibJPG library to measure image compression/decompression performance.

The benchmarks are compared against their raw execution times.

Sample	**Linux Executable**	**LibVM – Hardware Virtualization**		**LibVM – ptrace Jail**	
		Execution Time	**Overhead**	**Execution Time**	**Overhead**
LibVorbis	46.8	49.5	5.77%	53.7	14.7%
LibBZ2	38.8	39.1	0.74%	39.2	1.1%
LibZip	70.3	80.2	14%	83.3	18.5%

Table 5.7: Macro-benchmark results

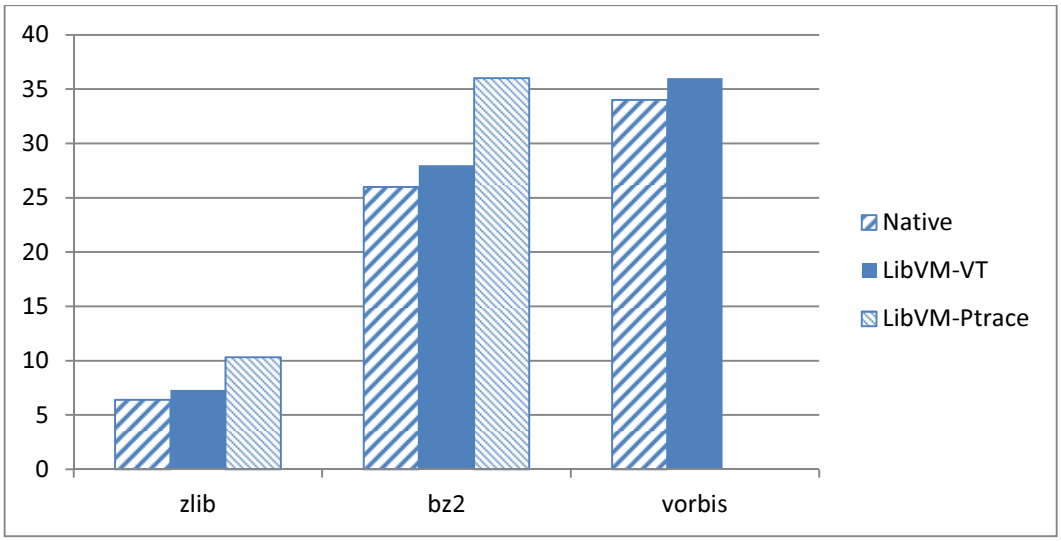


Figure 5.11: Comparison of native, software-based and hardware-based implementations – Core i5

Our results in Table 5.7 and Figure 5.11 show that the hardware virtualization based implementation of LibVM adds modest overheads ranging from 6% to 14%, depending largely on the number of domain transitions. This is entirely within expectation, as the VT hardware adds almost no overheads for normal execution of instructions. However, each domain transition/system call is intercepted by LibVM-VT, which proxies it on the callers behalf, including making additional security checks. This is the main source of overheads during execution.

Ptrace-based execution predictably suffers even worse overheads, as each system call results in at least 3 additional system calls – one to retrieve the processes's registers from the system, one to make the actual system call, and one to resume execution. In addition, security checks may end up causing additional system calls, worsening the performance as expected. However, when the total number of system calls are lower, for example in the BZ2 test, the performance differences become neglible. However, the LibZip benchmark, which contains a high volume of system calls, suffers fairly high overheads at around 20%.

Sample	Linux Executable	LibVM – Hardware Virtualization		LibVM – ptrace Jail	
		Execution Time	Overhead	Execution Time	Overhead
LibVorbis	39.6	41.8	5.55%	44.9	13.4%
LibBZ2	36.5	36.8	0.82%	36.9	1.09%
LibZip	64.1	73.2	14%	75.1	17.1%

Table 5.8: Macro-benchmark results – Core i7

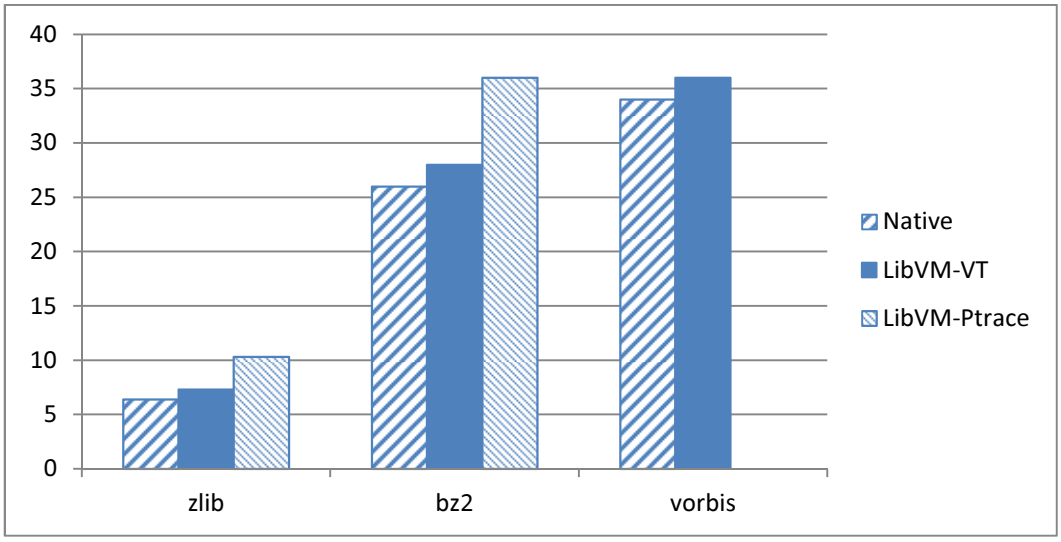


Figure 5.12: Comparison of native, software-based and hardware-based implementations – Core i7

The results when executed on a Core i7 machine are largely similar, with differences being accounted for by processor and disk speed differences.

5.5 CONCLUSION

This chapter provided an overview of the hardware based implementation of our isolation structure, as well as how the implementation evolved from initial prototypes to validate performance characteristics. The discussion was split into two parts, describing the initial validation of performance by adding hardware virtualization support to an existing solution, followed by the LibVM-based implementation. Performance measurements demonstrated that the hardware based solution was competitive with software only solutions, while improving confidence in the

robustness of the solution through clearly demarcated isolation boundaries, as well as by eliminating the need for machine code verification. These results validate one of the major hypotheses in this thesis, which is that hardware virtualization can be used to provide an effective mechanism for isolation of components in a simplified and more easily understood and verified manner.

Chapter 6: Evaluation of Implementations against Functional Specification

6.1 INTRODUCTION

This thesis has demonstrated, through the LibVM interface design and its software and hardware implementations, that component isolation is achievable with reasonable overheads, while preserving binary compatibility with existing libraries, in a manner comparable to or better than current practice. A performance evaluation has been detailed in Section 5.3.4. However, it is also necessary to provide assurance that the functions and mechanisms involved are effective in providing this isolation, and that they meet the functional specification defined in Section 3.4. This chapter provides an evaluation of the two LibVM implementations against that specification, identifying their key strengths and weaknesses. We evaluate LibVM through two methods. Firstly, by providing a rigorous argument that the isolation container encapsulated by the LibVM interface is sound in meeting its functional specification. Secondly, by using the Common Criteria for Information Technology Security Evaluation (CC), an international standard for trusted system evaluation [67], as a guiding framework for evaluating possible implementations. While a full CC evaluation is outside of the scope of this research, because it is a large-scale process intended for independent evaluation of commercial products with distinct roles for "developers", "evaluators" and "sponsors", the following sections are based on the processes and procedures outlined for security evaluation under the Common Criteria version 3.0. The main aim of such an evaluation is to gauge the effectiveness of our approach as well as to define security evaluation guidelines for the overall problem addressed in this thesis, namely that of isolating application libraries or components, from their host.

6.2 EVALUATING THE PROCESS TRACING BASED IMPLEMENTATION OF LIBVM

In this section, we evaluate the process tracing based implementation of LibVM, LibVM-ptrace, by analysing how well it conforms to LibVM's specification, as defined in Section 3.4. Since five conditions were identified under which an

implementation can be analysed for conformance, we take each of these conditions in turn and evaluate the process tracing based implementation against it, identifying its strengths and weaknesses.

6.2.1 Conformance to Condition C1

Condition C1 states that "The CPU utilised by the LibVM container must be virtualised such that it is isolated from the host application". As described in Chapter 3, LibVM-ptrace is implemented on top of existing operating system provided process isolation facilities. OS processes are isolated in typical operating systems by providing them with their own "virtual" CPU and registers, which are kept completely separate from the registers and state used by other processes, although the actual physical CPU is often shared, as has been discussed at greater length in Chapter 2. Furthermore, through the use of hardware protection rings, all sensitive instructions executed by a process (in which "sensitive" refers to those instructions which are liable to have an effect on state outside of the process itself) can be trapped and intercepted by the operating system. In addition, the *ptrace* mechanism also provides the ability to relay relevant traps to the host process. Overall, we contend that process isolation facilities are well understood facilities which provide entirely satisfactory guarantees that each process's CPU is adequately virtualised with respect to other processes running in the system.

Since the LibVM sandbox runs as a completely separate and dedicated OS process, it follows that it is completely isolated from the host process, and is unable to influence its CPU state or execution flow. This allows us to claim full conformance to Condition C1 for LibVM-ptrace.

6.2.2 Conformance to Condition C2

Condition C2 states that "A host can never utilise a value obtained from within the isolation container without ensuring that it is adequately validated". It should be noted here that this condition is one that is imposed largely on the host application, and not upon LibVM itself, as it is impracticable for LibVM to determine which use of data is valid or invalid. Therefore, that decision must be made in accordance to the host's security policy.

However, it is possible to highlight the points at which validation could be most easily implemented, at times by LibVM itself, so that the burden of implementing the host's security policy is greatly reduced. Towards this end, it is necessary to identify the key gateways through which information can flow between host and guest in LibVM-ptrace.

By going back to the LibVM interface, discussed in Chapter 3, it is possible to identify the key methods through which interaction with LibVM is made possible. Since these methods represent the only ways through which to instantiate a LibVM sandbox, all potential interaction paths also flow through these methods subsequently. The key methods are:

6.2.2.1 libvm_initialise

Since this method is responsible for initialisation of the libvm container, and no untrusted library has been loaded yet, it presents no security threat.

6.2.2.2 libvm_open

The method itself does not provide any untrusted data, since it returns an opaque, implementation specific handle, which will not be directly manipulated by the host. However, the method will cause an untrusted library to be loaded into the LibVM sandbox, and any initialisation routines in the library to be executed. At this point, the untrusted library may trigger additional host calls, at which point data will be passed from the sandbox out to the host. This is a critical point at which the parameters and information provided in these calls must be carefully vetted. We discuss the host call interface in greater detail under condition C4.

6.2.2.3 libvm_sym

This method provides a direct access point by which a function within a library residing in the LibVM sandbox, can be obtained. Once such a function pointer is obtained, it can be invoked as described in Chapter 4. While the mechanism provided for invocation is safe, as also explained in Chapter 4, the return values of the function are entirely suspect, since the library is untrusted.

For example, the invoked function could return a data structure, which contains a pointer referring to an illegal memory location outside of the sandbox address space.

If such a pointer were to be used, it can be potentially used to overwrite data in arbitrary locations of the host, potentially allowing an attacker to gain control. Therefore, it is up to the host application to perform adequate checks before using any such value.

However, we recognize that this procedure is likely to be error prone. While this may be less significant for libraries which are not malicious and are simply erroneous, it presents a far more significant threat when dealing with potentially malicious libraries.

In Chapter 7, we propose some methods to mitigate this issue, such as adding compiler support to natively recognize LibVM isolation containers, and to utilise type information available to the compiler to automatically insert validity checks, such as range checks on pointers, or to flag potential errors, before allowing the use of untrusted data. More advanced methods could include those such as taint analysis [116], which could perform more sophisticated forms of information flow analysis [112], both into and out of the isolation container, again flagging potentially dangerous use of untrusted data. These possibilities are also discussed in detail in Chapter 7.

6.2.2.4 libvm_guest_malloc

This method allocates a region of memory within the sandbox's address space. Since address space transparency is assumed, the host application can refer to this memory as if it were in its own address space. While LibVM-ptrace itself performs a range check to ensure that the return address falls within the sandbox's address space, once again, the onus falls on the host to use this memory region with caution, as it can potentially be altered by a library executing within the sandbox. The scenario becomes even more dangerous in a multi-threaded environment, which can give rise to TOCTOU (Time Of Check To Time Of Use) bugs [39]. Therefore, the same concerns as well as solutions discussed under the *dlsym* method, apply here.

6.2.2.5 libvm_guest_free

This method does not constitute a security threat because it is executed within the context of the sandbox, and no outbound information exchange occurs. However, it does call the *free* method in the C runtime within the sandbox to release memory.

Since this method can potentially be hijacked, and therefore, the library could invoke host calls, security being tight at the host call layer is essential as always.

6.2.2.6 libvm_close

This method unloads an existing library within the sandbox. While this method too is largely safe, the same caveats that applied to *libvm_guest_free* apply, since LibVM-ptrace invokes the *dlclose* method on the library executing within the sandbox, and that too could be hijacked to perform system calls.

6.2.2.7 libvm_destroy

This method is safe, as LibVM-ptrace simply terminates the child process, and any malicious code executing within the sandbox has no further opportunity to execute.

6.2.2.8 libvm_get_last_error

This method is used to return internal error codes in LibVM-ptrace, and therefore, has no interaction with untrusted code executing within the sandbox.

6.2.3 Conformance to Condition C3

Condition C3 states that "a thread of execution within a LibVM container cannot access memory regions outside of its allocated regions". There are two broad ways in which LibVM-ptrace ensures that this is satisfied. The first is by pre-allocating a shared memory area before forking the child process. Since the child process now runs in a completely separate address space, it is isolated from the host through the operating system's isolation facilities, which we assume can be relied upon. The only contact with the host is through the shared memory area, which the operating system guarantees is confined to the agreed upon region. Therefore, the shared memory area fully satisfies condition C3.

However, matters are complicated when additional pages are mapped into the sandbox's memory. The main method by which this is achieved is through the *mmap* (and related) system calls, which are POSIX compliant system calls [61]. These calls allow a process to map files or devices into memory, so that memory mapped I/O can be performed on them. Since these calls are vital to the function of many libraries, we provide a default implementation in LibVM-ptrace, although there can be libraries which do not require it, in which case it should be explicitly disabled.

The default implementation ensures that the parameters to these calls are validated, and that the memory regions are mapped into both the guest and the host. These memory regions are mapped such that they fall into unallocated memory regions available to both the host and the guest. (As described in Chapter 4, we ensure that the host and guest memory maps are kept in lock step) These newly mapped regions are also considered a part of the sandbox's address space, and therefore, must be checked by the host before use. It should also be noted therefore, that the sandbox address space is non-contiguous in the default LibVM-ptrace provided implementation, although this implementation should be overridden by the host if necessary.

6.2.4 Conformance to Condition C4

Condition C4 states that "the domain transition mechanism provided by LibVM must not compromise the host's integrity".

The domain transition mechanism used by LibVM-ptrace is the operating system provided *ptrace* facility, in conjuction with the operating system's own process switching facilities. Therefore, there is a reasonable expectation of correctness in the functions ability to inform LibVM of domain transitions, such as system calls. In addition, the Operating System utilises hardware protection rings to intercept the execution of privileged instructions by a sandboxed library. Once the *ptrace* mechanism informs LibVM-ptrace of the domain transition, it examines the reason for the domain transition, and terminates the library if an invalid action has been executed. If a host call has been requested, as described in Chapter 4, LibVM-ptrace's default implementation processes it, or delegates to the host if required. Therefore, we assert that this overall mechanism is tamper proof and reliant on the trustworthiness of the operating system itself, and that the real potential for vulnerabilities lies in the actual execution of the host calls, described below.

6.2.5 Conformance to Condition C5

Condition C5 states that "an action executed by the host on behalf of a library isolated within LibVM must not compromise the host's own integrity".

Recall from Chapter 4 that LibVM-ptrace enters lockdown mode soon after its initial bootstrapping process. Therefore, it screens all host calls which are deemed

dangerous before passing them onto the host. For example, if the untrusted library executes an *exit* system call, this system call should not be executed in the host's context at all, as it would cause the host to exit abruptly. Instead, we make the assumption that a library which executes this system call is either erroneous or malicious, and take steps to shutdown the isolation container immediately, as well as inform the host of the attempted transgression. Similarly, we have discussed under Condition C2, how memory mapping related calls are filtered.

Therefore, of the system calls for which default implementations are provided, the chief means by which confidence can be held in their safety is through the use of judicious programming techniques, which we have attempted to follow. These include careful checking of parameters and return values (while avoiding TOCTOU bugs), assertion of program invariants, and following the principle of least privilege. However, as noted in Chapter 4, LibVM-ptrace is more prone to TOCTOU bugs, as well as other implementation complexities, caused by the very nature of the *ptrace* mechanism.

In addition, given the plethora of system calls available, as well as the highly context-specific nature of host security policies, it is impracticable for LibVM to filter all system calls by itself. Therefore, in the current implementation, the host is responsible for implementing its own security policies, and it is therefore, prudent to revisit well known principles of security. As discussed above, minimising the size of the attack surface by disabling unneeded system calls, failing securely in the face of errors, and generally following the principle of granting least privilege at all times, are important principles by which the host's security policies should be guided.

In Chapter 7, we propose means by which the burden placed on the host can be eased, such as providing ready-made "security profiles", which are applicable to individual libraries. For example, a "compute-only" profile would allow an untrusted library the ability to execute a computation within the sandbox, but would disallow system calls altogether, whereas a "network" profile would allow an untrusted library to make network-related system calls.

6.3 EVALUATING THE HARDWARE BASED IMPLEMENTATION OF LIBVM

This section follows in the footsteps of the evaluation made for LibVM-ptrace, by considering each of the identified conformance criteria and evaluating our hardware based implementation, LibVM-VT, against it. Since there are many commonalities between the two implementations, we will compare and contrast whenever possible, as well as refer to the previous discussion for additional context.

6.3.1 Conformance to Condition C1

To restate condition C1, "the CPU utilised by the LibVM container must be virtualised such that it is isolated from the host application". As described in Chapter 4, LibVM-VT implementation follows a virtualization based approach for isolation. Popek and Goldberg [104] defined the main criteria that needed to be met for full virtualization of hardware, which the original Intel x86 architecture did not fully meet [110]. However, in 2005, both Intel and AMD introduced additional machine instructions to remedy this issue [3, 62, 64], and LibVM-VT is built on top of those machine instructions.

While we discovered that directly building on top of those hardware instructions is tedious, by utilising an abstraction layer provided by the Linux operating system, The Kernel-based Virtual Machine (KVM) [107], the drudgery was greatly simplified. It should be noted that LibVM-VT does not utilise the full KVM, which is a full-blown Virtual Machine Monitor (VMM), but instead, uses a minimal subset of it known as LibKVM, which is the portion that provides a layer of abstraction over the hardware instructions themselves. As a result, LibKVM is extremely lightweight, as it provides little more than virtualization of CPU and memory.

The hardware instructions themselves fully virtualise the CPU, including all registers, by running the CPU in an interpretive mode of execution. Each logical processor in the virtual machine has an associated VMCS (Virtual Machine Control Structure), which maintains its actual state. LibKVM therefore, provides a high-level wrapper over this functionality. In addition, LibKVM also virtualises memory page tables by using hardware support when available, and falls back to shadow page table based software techniques, in the absence of hardware support [135].

Therefore, the correctness of the LibVM-VT implementation is dependent on the correctness of LibKVM. However, since LibKVM is provided as a part of the operating system itself, and is a relatively thin layer of abstraction over the actual hardware virtualization support, there is a reasonable basis for high confidence in its implementation. Virtualization technology is fairly well-understood and mature, and therefore, we believe that there is a sound basis to have confidence that the CPU virtualization works effectively, making LibVM-VT conformant to condition C1.

6.3.2 Conformance to Condition C2

Condition C2 states that "a host can never utilise a value obtained from within the isolation container without ensuring that it is adequately validated". We note that most of the discussion carried out for LibVM-ptrace is applicable to LibVM-VT for this particular condition, and therefore, refer to Section 6.2.2 for a more in-depth discussion. Some of the minor differences are in the fact that the LibVM-VT implementation is less prone to TOCTOU bugs, since the system calls are carried out synchronously on the host, in contrast to the asynchronous nature of system call handling in LibVM-ptrace, as dictated by the *ptrace* system call. In addition, due to the absence of the two-stage interception of system calls brought on by the nature of *ptrace* (discussed in detail in Chapter 3), implementation complexity is also greatly reduced, increasing confidence in the LibVM-VT bases solution's conformance to Condition C2, due to the greater economy of mechanism [114].

6.3.3 Conformance to Condition C3

Condition C3 states that "a thread of execution within a LibVM container cannot access memory regions outside of its allocated regions". In order to assess this, it is necessary to delve briefly into how LibVM-VT handles memory allocated to the sandbox. As mentioned in the previous section, LibVM-VT relies on the layer of abstraction over memory and CPU provided by LibKVM. LibKVM will either use hardware NPT/EPT [3, 65] structures where available, or fallback to shadow page tables in their absence [135]. In either case, the end result is that the virtual machine can only access these assigned pages, and LibKVM ensures that pages outside of the assigned regions cannot be accessed. We, therefore, remap a region of memory within the host's address space, which is dedicated for use by the Virtual Machine used for the LibVM-VT sandbox. This region of memory is also shared with the

host however, which then becomes identical to the shared memory segment described in the LibVM-ptrace implementation. As discussed above, we believe that LibKVM is a relatively mature implementation, and distributed as a part of the Linux Kernel, and therefore, can be relied upon to ensure that a thread of execution within the sandbox cannot access regions outside of it.

In addition to this, any *mmap* calls made by the host are also easier to handle than in the case of LibVM-ptrace. This is because it is a simple matter of mapping the page in the host's address space, and reassigning that space to the guest virtual machine at the desired location. The asynchronous nature of the *ptrace* mechanism makes this far more difficult, once again increasing confidence in the correctness of the LibVM-VT implementation.

6.3.4 Conformance to Condition C4

Condition C4 states that "the domain transition mechanism provided by LibVM must not compromise the host's integrity". The domain transition mechanism for LibVM-VT is based on the virtualization hardware, which suspends interpretive execution and prompts a VMEXIT (described in Chapter 2), whenever a sensitive instruction is encountered [64], such as an interrupt invocation. Whenever such an exit based on a sensitive instruction occurs, LibKVM performs a callback to our userspace listener, which takes over in filtering the actual host calls. As discussed earlier, this mechanism is quite robust, and cannot be bypassed by a malicious executable.

6.3.5 Conformance to Condition C5

Condition C5 states that "an action executed by the host on behalf of a library isolated within LibVM must not compromise the host's own integrity". Once again, much of the discussion which applied to the *ptrace* implementation applies to this implementation, since both are largely identical except for the lesser complexity in the VT based implementation. Additionally, since the onus of managing system calls is largely placed on the host, the discussion in Chapter 7 on reducing that burden is also applicable to LibVM-VT. Therefore, full conformance to this condition is deferred to the host application itself, although future implementations of LibVM can potentially claim full conformance by adopting the strategies mentioned previously and discussed in detail in Chapter 7.

6.4 PART 5 – EVALUATING THROUGH THE COMMON CRITERIA

This section is based upon the processes and procedures outlined for security evaluation in the Common Criteria for Information Technology Security Evaluation version 3.0 [67]. The Common Criteria was chosen as a guiding framework as it is an international standard. While a full CC evaluation is not performed, we utilise relevant sections to ensure greater confidence in the processes and practices used in developing LibVM. Our basic approach has been to describe the security target aimed for, and to evaluate it against security functional requirements defined in the Common Criteria. The cross references below, of the form ASE_, are taken from the CC standard.

6.4.1 Related Security Targets and Experience

While there is no existing Protection Profile which matches our needs, we have relied on available security targets for operating systems as a basis on which to model our own CC-like Security Target, i.e. the object being evaluated, which in our case, is a LibVM implementation.

6.4.2 Introduction to the Security Target

6.4.2.1 ASE_INT.1.3C - TOE reference

The Target of Evaluation (TOE) is LibVM, a reference monitor/isolation container for dynamic shared libraries.

6.4.2.2 TOE description

ASE_INT.1.4C - The Target of Evaluation is a component isolation sub system, which is a container designed to protect its host from components of unknown provenance and quality. The TOE comes in the form of a sandboxing library that can be used to define such isolation domains and to load shared libraries into these domains. The TOE will be able to constrain faulty or malicious components through the following.

a. Preventing access to unauthorised host container memory regions
b. Disallowing access to privileged system resources
c. Allowing arbitration of all system calls

d. Preventing a component from causing a denial of service

e. Acting as a reference monitor for an untrusted component

The TOE will detect and flag attempts to bypass security constraints.

6.4.2.3 ASE_INT.1.8C - Logical scope

Our system is designed to deal with such arbitrary binary components, from untrusted sources, which need to be executed in a constrained environment. Once a component is accepted for execution, it must have controlled access to resources, as determined by the host. Access to memory must be restricted to areas allowed by the host application and attempts to exceed these limits must be caught. The host must be allowed to constrain the component by preventing arbitrary access to the operating system call interface. Such access must be mediated and the host must be allowed to set resource limits on memory usage or disk access. On the other hand, the component must be able to freely avail itself of all safe machine instructions.

Therefore, our system is based on the observations that a process cannot perform any actions harmful to the system as long as its system call interface, which is its window to the outside world, is strictly controlled. We utilise this principle for isolating components within a restricted address space, and provide strict arbitration over all system calls.

6.4.2.4 Definitions

A component is any binary unit which is loaded by an application into its own address space with communication taking place between the component and its host application via a well-defined interface. Primarily, this will be in the form of dynamic link libraries (DLL)/shared object (SO) libraries, which form the primary means of composability in modern operating systems (OS) and applications.

A malicious component is a component which intentionally attempts to subvert the hosting container and gain access to disallowed resources, including privilege elevation, information corruption or denial of service.

A faulty component is a component which unintentionally corrupts the host containers state or accidentally attempts to access memory or resources denied to it, causing failure or corruption of the containing host.

6.4.2.5 Conformance claims

6.4.3 Operating Environment

ASE_INT.1.6C -The TOE depends on

a. The Suse Linux 11.3 operating system
b. The LibKVM virtualization library provided with above operating system
c. The Gnu C Compiler

6.4.4 Objectives

The purpose of environmental objectives is to explicitly state the requirements which are expected of the operating environment. When these environmental objectives are met by the operating environment, the TOE itself meets its own security objectives satisfactorily.

The security objectives for the operations environment are as follows:

a. It must support full virtualization of CPU and memory.
b. Access attempts to unmapped memory regions must result in trappable page faults.
c. Privileged instructions (i.e. interrupt invocations, syscall instructions) must be trappable.
d. The host operating system must have a clearly defined system call interface, so that system-call interpositioning is possible.
e. The underlying Operating Environment is assumed to be free of exploitable vulnerabilities which reside outside of its system call interface.
f. It is assumed that the operating systems own security policies are sufficient to constrain a host application such that no component residing with that application can gain elevated privileges which are greater than the host application itself.
g. The host application must ensure that the conditions specified in Section 3.4.3.5 are met.

The security objectives for the TOE are as follows:

a. The TOE will detect and mitigate the effects of untrustworthy components.

b. The TOE will ensure that the subjects are only able to access resources according to a user-mediated policy object.

c. The TOE's reference monitor will operate in a separate domain from that of the untrusted library and allow full mediation over resources accessed and allocated by the untrusted library.

d. The reference monitor's mechanisms are tamper proof.

6.4.5 Special Requirements

None.

6.4.6 TOE Security Assurance

6.4.6.1 Assurance

Development

Sound software engineering practices were adopted, and key security principles adhered to, such as the principle of granting the least privilege, the principle of using the least common mechanism, and the principle of complete mediation. These practices have been outlined over the course of this document. In addition, automated testing has been performed through limited use of unit tests. These tests were designed to stress specific edge conditions, such as attempts to access memory outside of the bounds assigned to the TOE. In addition, defensive programming techniques have been utilised through the development of this software, and are detailed in Chapters 4 and 5.

Documentation

This thesis serves as the documentation for the TOE, and provides a complete description of its implementation aspects as well as a systematic basis for evaluation.

6.4.6.2 Platform assurance

Vulnerability assessment

We have conducted a systematic search for vulnerabilities by identifying all possible attack surfaces on the TOE as described in Sections 6.2 and 6.3. In addition to this systematic analysis of attack surface, we have conducted several targeted tests which

evaluated the robustness of the TOE to different kinds of attack, such as attempts to escape sandbox limits. These have been discussed at length in Chapter 4.

6.4.6.3 Summary

This section has provided a further evaluation of LibVM against the security guidelines defined in the internationally standardised Common Criteria for Information Security Evaluation. The Target of Evaluation, LibVM, was defined and its security objectives outlined. Secondly, the objectives were evaluated according to relevant assurance requirements in the Common Criteria (exluding those parts of the CC that are relevant only to mass-produced commercial products). References have been made to previous evaluations where relevant.

6.5 CONCLUSION

This chapter has provided an evaluation of both LibVM implementations, discussing the conditions under which LibVM can be guaranteed to be correct with respect to isolating libraries from their host, such that the host is fully able to control the execution of the libraries, as per the functional specification defined in Section 3.4. The five conditions under which this guarantee can be provided, have been analysed in depth, identifying the strengths and weaknesses of both implementations. Due to economy of mechanism, the hardware based implementation is deemed to be more secure than the software based implementation. Finally, the objectives and processes used in LibVM has been evaluated by using the Common Criteria as a guideline. While there is good confidence in the robustness of the mechanisms, potential pitfalls have been highlighted, and improvements outlined.

Chapter 7: Conclusions

7.1 INTRODUCTION

This chapter summarises the contributions and outcomes of this thesis. The chapter also outlines future enhancements as well as further research that can be undertaken in this area. Finally, it concludes the thesis with an analysis of the implications of this research.

7.2 SUMMARY OF RESEARCH AND ITS CONTRIBUTIONS

This thesis has addressed the problem of untrusted components potentially decreasing the reliability of host applications. The evidence for this position has been presented and discussed, including the prevalence of application crashes caused by component failures, as well as the security threats posed by insecure components. To solve this problem, we have investigated methods for creating protection domains for individual components, thus preventing an errant or untrusted component from adversely affecting the host. We also aimed to preserve the benefits of shared address spaces, a key advantage in library components, while doing so.

Therefore, the causes and symptoms of component failure were investigated, and the errors that are relevant to isolation domains were identified, which contribute to the literature on this subject. Subsequently, we have evaluated the methods available to create such isolation domains and have published our findings [46]. In addition, we have also identified some of the major shortcomings in present research in this area. Some of the key shortcomings include the inability to preserve the benefits of shared address spaces, as well as the requirement that components be recompiled through custom tool-chains, or that they adhere to a highly restricted subset of available machine instructions, in order to be safely executed.

To remedy these shortcomings, we have introduced an abstract API that defines the broad semantics of such isolation containers and described a hardware virtualization-based implementation for speed, and a less efficient *ptrace*-based software implementation for use in the absence of such hardware support.

Our solution retains the advantage of not requiring custom tool chains, and operates against standard linux binaries, providing an advantage over previous efforts in this area [32, 144]. The shared libraries themselves require no modifications before use. We found that the porting effort itself is proportional to the complexity of the host. While our hardware virtualization-based implementation provides simpler mechanisms for data sharing, the *ptrace* jail-based mechanism offers more limited options. However, such limitations could be remedied with kernel modifications.

This approach provides an attractive alternative to existing solutions because it requires less porting effort due to its similarity to existing POSIX interfaces, and provides a more natural programming metaphor due to pointer transparency, while maintaining comparable performance when hardware virtualization support is available, and good isolation guarantees even in its absence.

In addition, neither of these techniques had previously been explored for the purpose of isolating library components. Therefore, to our knowledge, this is the first documented research effort to do so.

We have also carried out extensive performance evaluations against competing solutions, using industry standard benchmarks, and show that our solutions offer competitive performance, while reducing the size of the required Trusted Computing Base, and eliminating entire classes of problems related to code verification and static analysis.

The two implementations have also been evaluated in-depth, and this thesis has identified five conditions that are necessary and sufficient to guarantee that a component can be isolated from its host. These conditions are:

1. The CPU utilised by the LibVM container must be virtualised such that it is isolated from the host application.
2. A host can never utilise a value obtained from within the isolation container without ensuring that it is adequately validated.
3. A thread of execution within a LibVM container cannot access memory regions outside of its allocated regions.
4. The domain transition mechanism provided by LibVM must not compromise the host's integrity.

5. An action executed by the host on behalf of a library isolated within LibVM must not compromise the host's own integrity.

As has been discussed, these conditions can be conformed to in various degrees, in accordance to the isolation needs of a particular application, while still being implemented through the LibVM API. For example, an application that supports components of unknown provenance could utilise a strictly conformant API with extremely strong isolation guarantees, whereas an application which simply wishes to guard itself against component failures, could utilise a less conformant API which nevertheless provides satisfactory fault isolation.

In summary, the contributions of this research are

1. A classification of isolation mechanisms.
2. An abstract API which encapsulates component isolation domains.
3. A set of conditions under which an isolation sandbox can correctly isolate a component.
4. A process-tracing based implementation of the LibVM API.
5. A hardware-virtualization based implementation of the LibVM API.
6. A comparative evaluation of the implementations.

Ultimately, we believe that the ideas presented in this thesis serve to validate our initial hypothesis that it is indeed possible to create lightweight isolation domains for libraries while maintaining heavyweight security guarantees. In the following sections, we note further work that can be undertaken to improve and build upon these ideas.

7.3 FUTURE WORK

The work presented in this thesis demonstrates how lightweight isolation domains for libraries can be created, while abstracting away the implementation details. The resultant LibVM sandbox has provided a base on which further work can be undertaken. This section includes not only enhancements which can be made to LibVM, but also to the general problem of component isolation, which, as we have

argued throughout this thesis, is a problem of critical importance when developing more dependable systems.

7.3.1 Support for Multiple Operating Systems

The current implementation of LibVM is built on the Linux operating system. While the general concepts and ideas presented in this thesis are not operating system dependent, the two implementations makes several operating system specific assumptions. An example would be the use of POSIX shared memory primitives in LibVM-ptrace. Some operating systems, such as Microsoft Windows, are not fully POSIX compliant [97] and therefore, LibVM would need to be ported to use the corresponding OS calls in Microsoft Windows. In addition, both LibVM-ptrace and LibVM-VT directly use the GNU/Linux loader/linker for loading additional libraries into the sandbox. Such issues too can be solved by utilising the operating system specific linking/loading mechanism.

In all cases, since the LibVM API itself is abstract, it is possible to implement these mechanisms in a highly operating system dependent manner, yet leave the host program itself largely unaware of those underlying mechanisms (although recompilation for each operating system is required). However, a more difficult issue lies with the operating system call interface itself, as the current version of LibVM requires that this interface be intercepted and mediated by the host application. Since this call interface is highly operating system dependent, the host application too will be tied down to that particular operating system interface.

Several solutions to this problem are possible. One possibility is to absorb the operating system dependent portions into LibVM itself, and to expose a more high-level security policy abstraction to the host application, based on pre-defined templates. For example, if a component is known to require only CPU and internet access, the host application could simply inform LibVM to activate a “CPU only” and “Internet only” security policy. This would then allow system calls related to those policies only, with all other system calls being barred. Such functionality-based confinement policies have already been suggested and implemented in application-oriented access control mechanisms [120], and can be readily adapted to LibVM.

Yet another approach is to define an operating system independent system call interface, which require that components use only these system calls. This approach

is utilised in Google's Native Client [144]. However, such a policy forces all components to be recompiled to support the new interface, which is disadvantageous when using pre-existing library components.

A third approach is for LibVM to abstract away the operating system interface, yet provide the host with an OS independent mechanism for making policy decisions. Combined with the first approach of policy templates, this method could be used on existing library components while allowing the host application to remain independent of operating system details.

7.3.2 64-bit Support

The current implementation of LibVM has been developed on a 32-bit architecture, and therefore, a port to a 64-bit architecture would be useful. It should be noted however, that neither implementation of LibVM has any specific dependency on a 32-bit architecture, and consequently, can be easily ported to 64-bit systems. For example, LibVM-ptrace is built on existing process isolation facilities and *ptrace*, both of which are supported in 64-bit Linux systems. LibVM-VT is reliant on hardware virtualization, and this too is easily ported to support 64-bit instruction sets by simply setting the virtual processor to 64-bit mode.

This should be contrasted with Vx32, which relies on hardware segmentation [32], and therefore is dependent on 32-bit systems, as well as Google's Native Client, which uses segmentation on 32-bit systems and has a paging-based mechanism to support 64-bit systems [118].

7.3.3 Improved Debugging Support

Debugging support for components executing within a LibVM sandbox is also an important consideration. We have implemented rudimentary debugging capabilities in our current version of LibVM-VT, such as the ability to set breakpoints as well as step-through instructions individually. However, better debugging support, such as the ability to inspect variable values, utilise debugging information embedded in binaries etc., is a clear necessity. Adding such support would be a tremendous improvement in supporting application development using LibVM.

The LibVM-ptrace implementation also has similarly rudimentary debugging capabilities, but would need to be enhanced to provide better external debugging

support. One particularly problematic issue is that, the *ptrace* interface supports attaching only one parent process to each child-process being traced. Since LibVM is already tracing the sandboxed process, an external debugger process cannot therefore, attach to this sandboxed process. Such limitations could be avoided by providing direct operating system support for LibVM sandboxes, which we discuss next.

7.3.4 Operating System Support for Component Sandboxes

In this thesis, we have argued that support for isolating components should be a fundamental abstraction in operating systems, and would lead to more robust applications. In the course of implementing this support, we have not directly modified the operating system Kernel, so as to maximise the portability of the solution. Based on our experience with LibVM-ptrace, we have shown that, while it is possible to implement component sandboxing independent of such operating system modifications, the mechanisms themselves would become more efficient as well as more economical, if the operating system recognizes components as a fundamental abstraction.

For example, in our current implementation of LibVM-ptrace, we have used the *ptrace* mechanism, which is a mechanism provided by the operating system mainly to support debuggers. While we have commandeered this mechanism for the purpose of component isolation, the *ptrace* mechanism has much inefficiency, as it was originally intended for a different purpose. Such inefficiencies include an excess of up-calls to the tracing process, which could have been avoided with direct operating system support, as well as limited support for manipulating the child process. This fact has also been demonstrated in application confinement [42].

Therefore, we believe that adding direct support in the operating system kernel, for component sandboxing, would be a useful area of further research.

7.3.5 Compiler Support for LibVM Sandboxes

In addition to the idea that component sandboxing should be an integral part of the operating system, we have also shown (in Chapter 6) that compiler supported sandboxed components could increase confidence in the security of the solution, as

well as make the process of using sandboxed components more transparent (Chapters 4 and 5).

For example, pointers obtained from a call to a sandboxed component, could be automatically validated by the compiler to ensure that they fall within the sandboxed region. This would provide a more natural programming metaphor for host programs, as they could be largely insulated from the details of validating information obtained from within the sandbox, a process which the host must do manually in the current implementation of LibVM.

This task would require data and control flow analysis to determine which data/methods in a sandboxed components are being accessed by the host, and additional instructions could be emitted to validate such accesses. If the validation cannot be automatically done, issuing a compilation warning would be sufficient to alert the user to potential security vulnerabilities.

This support could be built on top of the core infrastructure provided by LibVM, and provides an important avenue for further research.

7.3.6 Use of Hardware Virtualization for Driver Isolation

This thesis has focused on the isolation of user-level libraries, and we have utilised hardware virtualization support towards this end, in the LibVM-VT implementation. However, we have not explored the issue of isolating device drivers and other kernel modules, which constitute a frequent source of system failure too. For example, over 85% of Windows XP crashes are due to faulty device drivers [125], and Linux drivers have 3 to 7 times the bug count of the kernel itself [20]. Therefore, isolation of kernel modules is also a pressing requirement.

While others have explored the isolation of driver isolation using full virtual machines [83], lightweight driver isolation using hardware virtualization support has been described by Tan et al. [129], by using Intel's VT-x extensions. However, their research predated the introduction of Intel EPT and AMD Nested Paging support (described in Chapter 2), and therefore, this also presents opportunities for further performance and isolation improvement, as well as more general purpose abstraction.

7.4 CONCLUSION

This chapter has provided a summary of the research undertaken and contributions of this thesis, as well as future work required in this area. We have shown, through LibVM, that isolating library components with strong security guarantees, while preserving their key benefits such as shared address spaces, is possible. This has been validated through the development and evaluation of two concrete implementations of the LibVM API. The first implementation is based on operating system provided process tracing facilities. The second implementation is based on hardware virtualization support present in more recent processors. We have shown the relative merits and demerits of each approach, and conclude that the hardware virtualization based approach currently provides greater flexibility and performance in providing intra-address space isolation, while maintaining a relatively smaller TCB. However, we have also shown that, if components were to become a fundamental abstraction offered by the operating system, these isolation facilities could also be built on top of existing process isolation facilities while minimising the shortcomings in our process-tracing based approach. We have shown that our approach has the following key advantages.

1. Elimination of an entire class of problems related to code verification and patching.
2. A significantly smaller Trusted Computing Base and therefore, increased confidence in the safety of the system.
3. Our prototype implementation already provides competitive performance in comparison to NaCl and better performance than Vx32, with the promise of even better performance in an optimised implementation.
4. Since we perform checks at runtime, we minimise false positives which would prevent the execution of valid components.
5. The approach is easily extendable to support 64-bit code.
6. Our approach does not require the use of custom tool chains, and can isolate standard Linux binaries.

In addition, this thesis has also identified five conditions that are necessary and sufficient to guarantee that a component can be isolated from its host. This provides a

basis on which future isolation containers can be developed, with varying levels of assurance/conformance claims against those conditions.

Finally, the thesis concludes by presenting the argument that the ability to isolate untrusted components should be a fundamental abstraction supported by the operating environment, leading to more robust and fault-tolerant applications, which are also more resilient to security threats. By combining such an abstraction with compiler support at the programming language level, we believe it is possible to tremendously increase the ease of use of such isolation sandboxes, while having increased confidence in its security provisions.

Appendix A: Detailed SPEC Results

This section provides the detailed SPEC benchmark reports for the results summarised in Chapter 5. These reports provide more detailed information in comparing the individual performance of the solutions, as well as the flags and parameters used in their execution. The pages that follow include the SPEC reports in the following order.

1. Core i5 processor
    a. Native run (Pages 166-168)
    b. LibVM –VT (Pages 169-171)
    c. Google Native Client (Pages 172-174)
2. Core i7 processor
    a. Native run (Pages 175-177)
    b. LibVM-VT (Pages 178-180)
    c. Google Native Client (Pages 181-183)

SPEC® CINT2006 Result

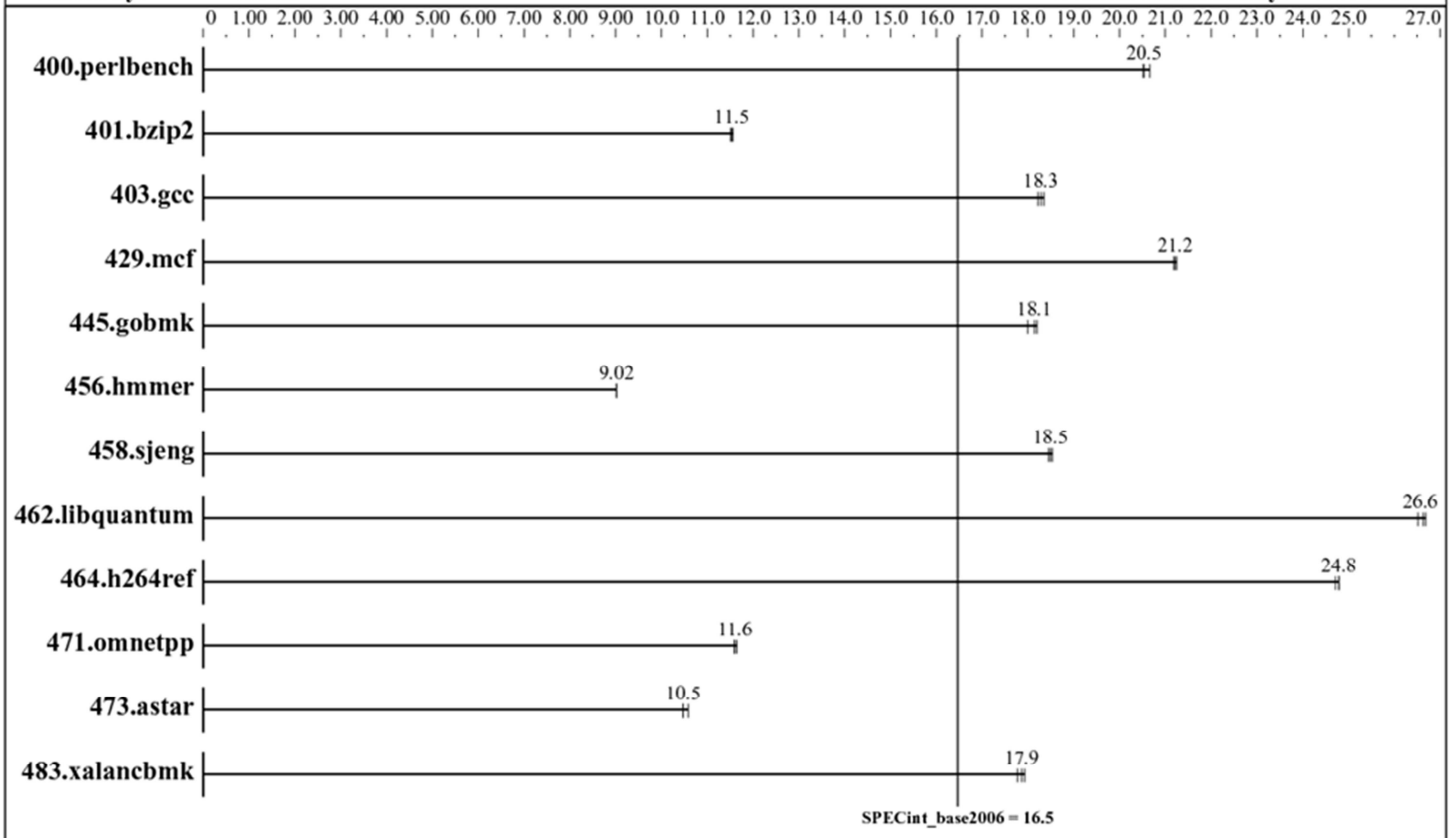
Intel
Lenovo X201

SPECint®2006 = Not Run

SPECint_base2006 = 16.5

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999



Hardware

CPU Name:	Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz
CPU Characteristics:	
CPU MHz:	2530
FPU:	Integrated
CPU(s) enabled:	2 cores, 1 chip, 1 core/chip, 2 threads/core
CPU(s) orderable:	1 chip
Primary Cache:	64 KB I + 64 KB D on chip per chip
Secondary Cache:	3072 KB I+D on chip per chip
L3 Cache:	None
Other Cache:	None
Memory:	4 GB (2 x 2GB DDR3 1066Mhz)
Disk Subsystem:	SATA
Other Hardware:	--

Software

Operating System:	SUSE Linux 11.3 (for i386)
Compiler:	nacl-gcc , nacl-g++ (for i386)
Auto Parallel:	No
File System:	ext4
System State:	runlevel 3
Base Pointers:	32-bit
Peak Pointers:	32-bit
Other Software:	None

SPEC CINT2006 Result

Intel
Lenovo X201

SPECint2006 = Not Run

SPECint_base2006 = 16.5

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench	**476**	**20.5**	473	20.7	476	20.5						
401.bzip2	835	11.6	**837**	**11.5**	838	11.5						
403.gcc	439	18.3	442	18.2	**440**	**18.3**						
429.mcf	**430**	**21.2**	430	21.2	429	21.2						
445.gobmk	**578**	**18.1**	576	18.2	583	18.0						
456.hmmer	1035	9.02	1034	9.02	**1034**	**9.02**						
458.sjeng	656	18.5	653	18.5	**655**	**18.5**						
462.libquantum	782	26.5	777	26.7	**778**	**26.6**						
464.h264ref	895	24.7	893	24.8	**893**	**24.8**						
471.omnetpp	537	11.6	539	11.6	**539**	**11.6**						
473.astar	663	10.6	670	10.5	**670**	**10.5**						
483.xalancbmk	**386**	**17.9**	385	17.9	388	17.8						

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

General Notes

```
400.perlbench: -DSPEC_CPU_LINUX_IA32
462.libquantum: -DSPEC_CPU_LINUX
C base flags: -O2
C++ base flags: -O2
Fortran base flags: -O2
```

Base Compiler Invocation

C benchmarks:
/usr/bin/gcc

C++ benchmarks:
/usr/bin/g++

Base Portability Flags

400.perlbench: -DSPEC_CPU_LINUX_IA32
462.libquantum: -DSPEC_CPU_LINUX
483.xalancbmk: -DSPEC_CPU_LINUX

Base Optimization Flags

C benchmarks:
-O3

C++ benchmarks:
-O3

For questions about this result, please contact the tester.
For other inquiries, please contact webmaster@spec.org.

Tested with SPEC CPU2006 v1.1.
Report generated on Sat Oct 16 15:37:58 2010 by SPEC CPU2006 PS/PDF formatter v6128.

SPEC® CINT2006 Result

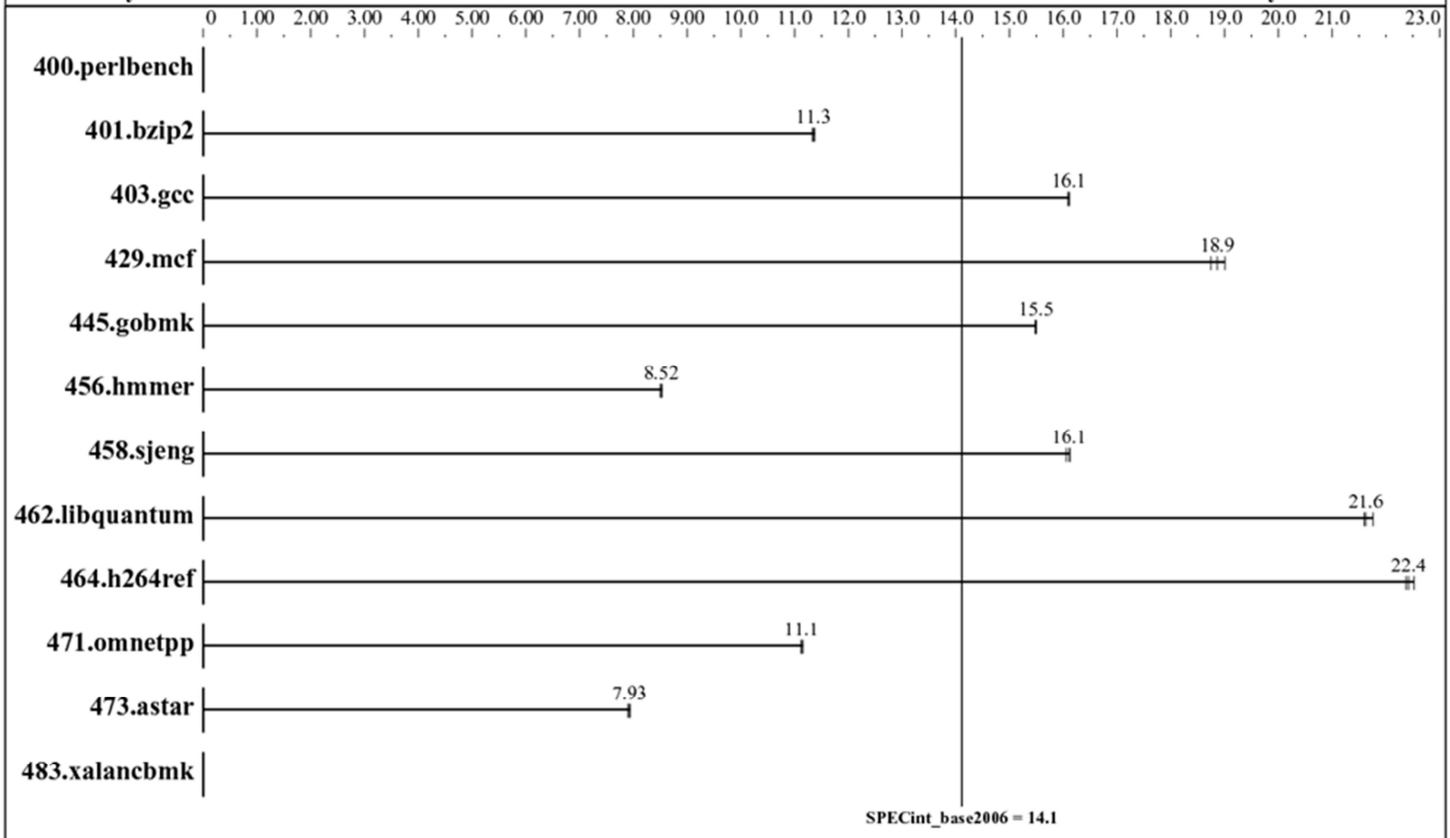
Intel
Lenovo X201

SPECint®2006 = Not Run

SPECint_base2006 = 14.1

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999



Hardware

CPU Name:	Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz
CPU Characteristics:	
CPU MHz:	2530
FPU:	Integrated
CPU(s) enabled:	2 cores, 1 chip, 1 core/chip, 2 threads/core
CPU(s) orderable:	1 chip
Primary Cache:	64 KB I + 64 KB D on chip per chip
Secondary Cache:	3072 KB I+D on chip per chip
L3 Cache:	None
Other Cache:	None
Memory:	4 GB (2 x 2GB DDR3 1066Mhz)
Disk Subsystem:	SATA
Other Hardware:	--

Software

Operating System:	SUSE Linux 11.3 (for i386)
Compiler:	nacl-gcc , nacl-g++ (for i386)
Auto Parallel:	No
File System:	ext4
System State:	runlevel 3
Base Pointers:	32-bit
Peak Pointers:	32-bit
Other Software:	None

SPEC CINT2006 Result

Intel
Lenovo X201

SPECint2006 = Not Run

SPECint_base2006 = 14.1

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench												
401.bzip2	852	11.3	849	11.4	**850**	**11.3**						
403.gcc	500	16.1	501	16.1	**500**	**16.1**						
429.mcf	480	19.0	**483**	**18.9**	487	18.7						
445.gobmk	677	15.5	**677**	**15.5**	678	15.5						
456.hmmer	1097	8.50	**1095**	**8.52**	1094	8.53						
458.sjeng	**751**	**16.1**	754	16.0	751	16.1						
462.libquantum	952	21.8	960	21.6	**958**	**21.6**						
464.h264ref	989	22.4	**987**	**22.4**	983	22.5						
471.omnetpp	560	11.2	**562**	**11.1**	562	11.1						
473.astar	884	7.94	889	7.90	**885**	**7.93**						
483.xalancbmk												

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

General Notes

```
462.libquantum: -DSPEC_CPU_LINUX
C base flags: -O2
C++ base flags: -O2
Fortran base flags: -O2
```

Base Compiler Invocation

C benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-gcc
 -lmycustom -lg(*)
```

C++ benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-g++
 -lmycustom -lg(*)
```

(*) Indicates a compiler flag that was found in a non-compiler variable.

Base Portability Flags

462.libquantum: -DSPEC_CPU_LINUX

Base Optimization Flags

C benchmarks:

401.bzip2: `-O3`

403.gcc: `Same as 401.bzip2`

429.mcf: `Same as 401.bzip2`

445.gobmk: `Same as 401.bzip2`

456.hmmer: `Same as 401.bzip2`

458.sjeng: `Same as 401.bzip2`

462.libquantum: `Same as 401.bzip2`

464.h264ref: `Same as 401.bzip2`

C++ benchmarks:

471.omnetpp: `-O3`

473.astar: `Same as 471.omnetpp`

For questions about this result, please contact the tester.
For other inquiries, please contact webmaster@spec.org.

Tested with SPEC CPU2006 v1.1.
Report generated on Wed Oct 20 11:52:23 2010 by SPEC CPU2006 PS/PDF formatter v6128.

SPEC® CINT2006 Result

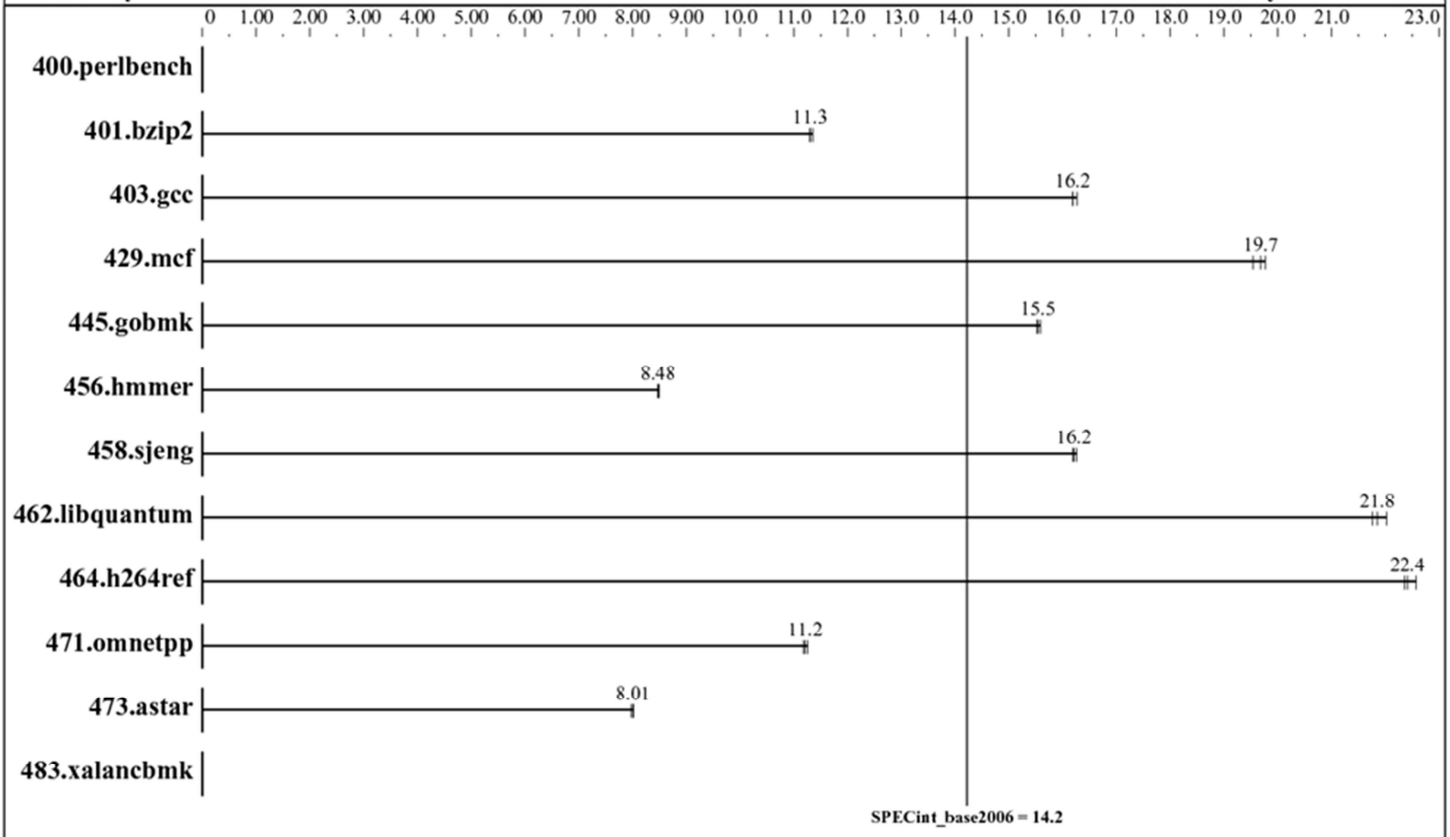
Intel
Lenovo X201

SPECint®2006 = Not Run

SPECint_base2006 = 14.2

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999



Hardware

CPU Name:	Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz
CPU Characteristics:	
CPU MHz:	2530
FPU:	Integrated
CPU(s) enabled:	2 cores, 1 chip, 1 core/chip, 2 threads/core
CPU(s) orderable:	1 chip
Primary Cache:	64 KB I + 64 KB D on chip per chip
Secondary Cache:	3072 KB I+D on chip per chip
L3 Cache:	None
Other Cache:	None
Memory:	4 GB (2 x 2GB DDR3 1066Mhz)
Disk Subsystem:	SATA
Other Hardware:	--

Software

Operating System:	SUSE Linux 11.3 (for i386)
Compiler:	nacl-gcc , nacl-g++ (for i386)
Auto Parallel:	No
File System:	ext4
System State:	runlevel 3
Base Pointers:	32-bit
Peak Pointers:	32-bit
Other Software:	None

SPEC CINT2006 Result

Intel
Lenovo X201

SPECint2006 = Not Run

SPECint_base2006 = 14.2

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench												
401.bzip2	854	11.3	850	11.4	**854**	**11.3**						
403.gcc	497	16.2	495	16.3	**497**	**16.2**						
429.mcf	461	19.8	467	19.5	**463**	**19.7**						
445.gobmk	676	15.5	673	15.6	**675**	**15.5**						
456.hmmer	1098	8.50	**1101**	**8.48**	1102	8.47						
458.sjeng	748	16.2	**747**	**16.2**	744	16.3						
462.libquantum	941	22.0	**949**	**21.8**	952	21.8						
464.h264ref	990	22.4	980	22.6	**987**	**22.4**						
471.omnetpp	555	11.3	**557**	**11.2**	559	11.2						
473.astar	**877**	**8.01**	876	8.01	880	7.98						
483.xalancbmk												

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

General Notes

```
462.libquantum: -DSPEC_CPU_LINUX
C base flags: -O2
C++ base flags: -O2
Fortran base flags: -O2
```

Base Compiler Invocation

C benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-gcc
 -lmycustom -lg(*)
```

C++ benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-g++
 -lmycustom -lg(*)
```

(*) Indicates a compiler flag that was found in a non-compiler variable.

Base Portability Flags

462.libquantum: -DSPEC_CPU_LINUX

Base Optimization Flags

C benchmarks:

401.bzip2: `-O3`

403.gcc: `Same as 401.bzip2`

429.mcf: `Same as 401.bzip2`

445.gobmk: `Same as 401.bzip2`

456.hmmer: `Same as 401.bzip2`

458.sjeng: `Same as 401.bzip2`

462.libquantum: `Same as 401.bzip2`

464.h264ref: `Same as 401.bzip2`

C++ benchmarks:

471.omnetpp: `-O3`

473.astar: `Same as 471.omnetpp`

For questions about this result, please contact the tester.
For other inquiries, please contact webmaster@spec.org.

Tested with SPEC CPU2006 v1.1.
Report generated on Mon Oct 18 04:52:23 2010 by SPEC CPU2006 PS/PDF formatter v6128.

SPEC® CINT2006 Result

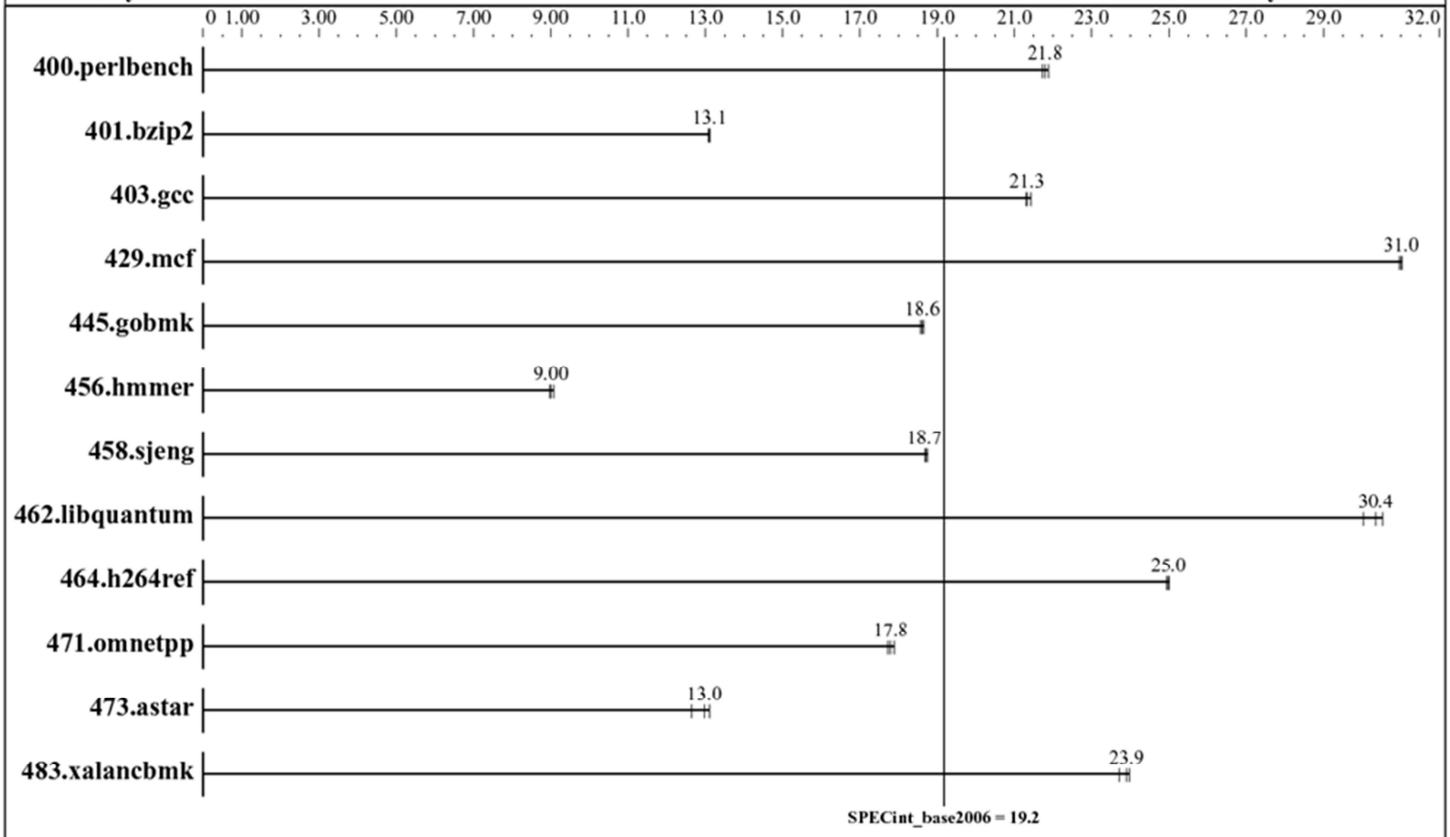
Intel
Lenovo X201

SPECint®2006 = Not Run

SPECint_base2006 = 19.2

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999



Hardware

CPU Name:	Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz
CPU Characteristics:	
CPU MHz:	2530
FPU:	Integrated
CPU(s) enabled:	2 cores, 1 chip, 1 core/chip, 2 threads/core
CPU(s) orderable:	1 chip
Primary Cache:	64 KB I + 64 KB D on chip per chip
Secondary Cache:	3072 KB I+D on chip per chip
L3 Cache:	None
Other Cache:	None
Memory:	4 GB (2 x 2GB DDR3 1066Mhz)
Disk Subsystem:	SATA
Other Hardware:	--

Software

Operating System:	SUSE Linux 11.3 (for i386)
Compiler:	nacl-gcc , nacl-g++ (for i386)
Auto Parallel:	No
File System:	ext4
System State:	runlevel 3
Base Pointers:	32-bit
Peak Pointers:	32-bit
Other Software:	None

SPEC CINT2006 Result

Intel
Lenovo X201

SPECint2006 = Not Run

SPECint_base2006 = 19.2

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench	447	21.9	450	21.7	**448**	**21.8**						
401.bzip2	736	13.1	738	13.1	**736**	**13.1**						
403.gcc	**378**	**21.3**	376	21.4	378	21.3						
429.mcf	**294**	**31.0**	294	31.0	295	31.0						
445.gobmk	**563**	**18.6**	562	18.7	565	18.6						
456.hmmer	1028	9.08	1039	8.98	**1036**	**9.00**						
458.sjeng	645	18.8	647	18.7	**647**	**18.7**						
462.libquantum	**683**	**30.4**	679	30.5	690	30.0						
464.h264ref	885	25.0	**886**	**25.0**	888	24.9						
471.omnetpp	353	17.7	**351**	**17.8**	349	17.9						
473.astar	**541**	**13.0**	555	12.6	536	13.1						
483.xalancbmk	291	23.7	**289**	**23.9**	288	24.0						

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

General Notes

```
400.perlbench: -DSPEC_CPU_LINUX_IA32
462.libquantum: -DSPEC_CPU_LINUX
C base flags: -O2
C++ base flags: -O2
Fortran base flags: -O2
```

Base Compiler Invocation

C benchmarks:
```
/usr/bin/gcc
```

C++ benchmarks:
```
/usr/bin/g++
```

Base Portability Flags

```
400.perlbench: -DSPEC_CPU_LINUX_IA32
462.libquantum: -DSPEC_CPU_LINUX
483.xalancbmk: -DSPEC_CPU_LINUX
```

Base Optimization Flags

C benchmarks:
-O3

C++ benchmarks:
-O3

For questions about this result, please contact the tester.
For other inquiries, please contact webmaster@spec.org.

Tested with SPEC CPU2006 v1.1.
Report generated on Thu Oct 21 06:58:30 2010 by SPEC CPU2006 PS/PDF formatter v6128.

SPEC® CINT2006 Result

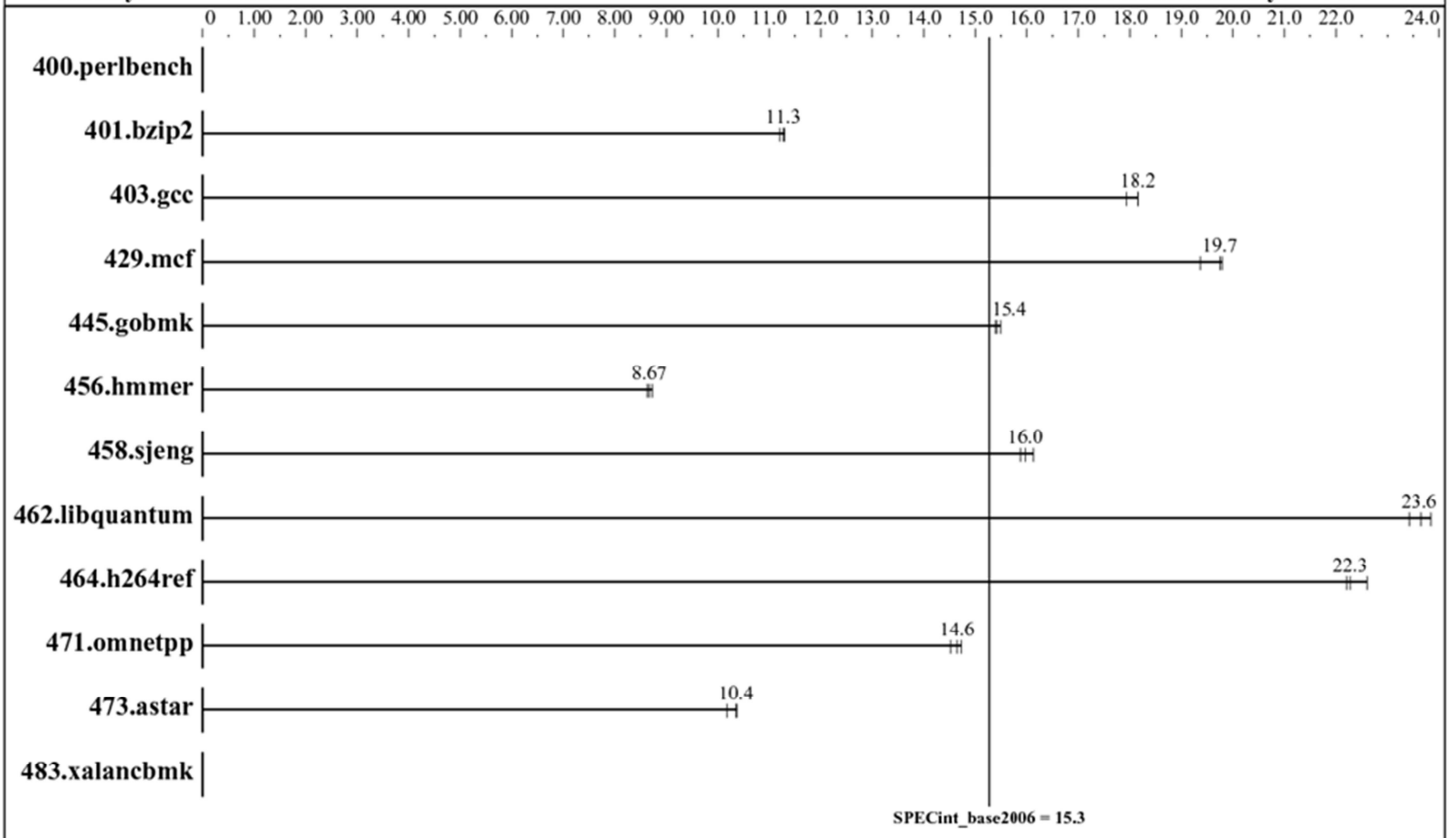
Intel
Lenovo X201

SPECint®2006 = Not Run

SPECint_base2006 = 15.3

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999



Hardware

CPU Name:	Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz
CPU Characteristics:	
CPU MHz:	2530
FPU:	Integrated
CPU(s) enabled:	2 cores, 1 chip, 1 core/chip, 2 threads/core
CPU(s) orderable:	1 chip
Primary Cache:	64 KB I + 64 KB D on chip per chip
Secondary Cache:	3072 KB I+D on chip per chip
L3 Cache:	None
Other Cache:	None
Memory:	4 GB (2 x 2GB DDR3 1066Mhz)
Disk Subsystem:	SATA
Other Hardware:	--

Software

Operating System:	SUSE Linux 11.3 (for i386)
Compiler:	nacl-gcc , nacl-g++ (for i386)
Auto Parallel:	No
File System:	ext4
System State:	runlevel 3
Base Pointers:	32-bit
Peak Pointers:	32-bit
Other Software:	None

SPEC CINT2006 Result

Intel
Lenovo X201

SPECint2006 = Not Run

SPECint_base2006 = 15.3

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench												
401.bzip2	861	11.2	854	11.3	**856**	**11.3**						
403.gcc	449	17.9	443	18.2	**443**	**18.2**						
429.mcf	471	19.4	**462**	**19.7**	461	19.8						
445.gobmk	682	15.4	677	15.5	**680**	**15.4**						
456.hmmer	1081	8.63	1069	8.73	**1076**	**8.67**						
458.sjeng	762	15.9	**758**	**16.0**	750	16.1						
462.libquantum	**876**	**23.6**	869	23.8	884	23.4						
464.h264ref	997	22.2	979	22.6	**994**	**22.3**						
471.omnetpp	431	14.5	**427**	**14.6**	424	14.7						
473.astar	689	10.2	**678**	**10.4**	677	10.4						
483.xalancbmk												

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

General Notes

```
462.libquantum: -DSPEC_CPU_LINUX
C base flags: -O2
C++ base flags: -O2
Fortran base flags: -O2
```

Base Compiler Invocation

C benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-gcc
 -lmycustom -lg(*)
```

C++ benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-g++
 -lmycustom -lg(*)
```

(*) Indicates a compiler flag that was found in a non-compiler variable.

Base Portability Flags

462.libquantum: -DSPEC_CPU_LINUX

Base Optimization Flags

C benchmarks:

401.bzip2: `-O3`

403.gcc: `Same as 401.bzip2`

429.mcf: `Same as 401.bzip2`

445.gobmk: `Same as 401.bzip2`

456.hmmer: `Same as 401.bzip2`

458.sjeng: `Same as 401.bzip2`

462.libquantum: `Same as 401.bzip2`

464.h264ref: `Same as 401.bzip2`

C++ benchmarks:

471.omnetpp: `-O3`

473.astar: `Same as 471.omnetpp`

For questions about this result, please contact the tester.
For other inquiries, please contact webmaster@spec.org.

Tested with SPEC CPU2006 v1.1.
Report generated on Wed Oct 20 20:32:26 2010 by SPEC CPU2006 PS/PDF formatter v6128.

SPEC® CINT2006 Result

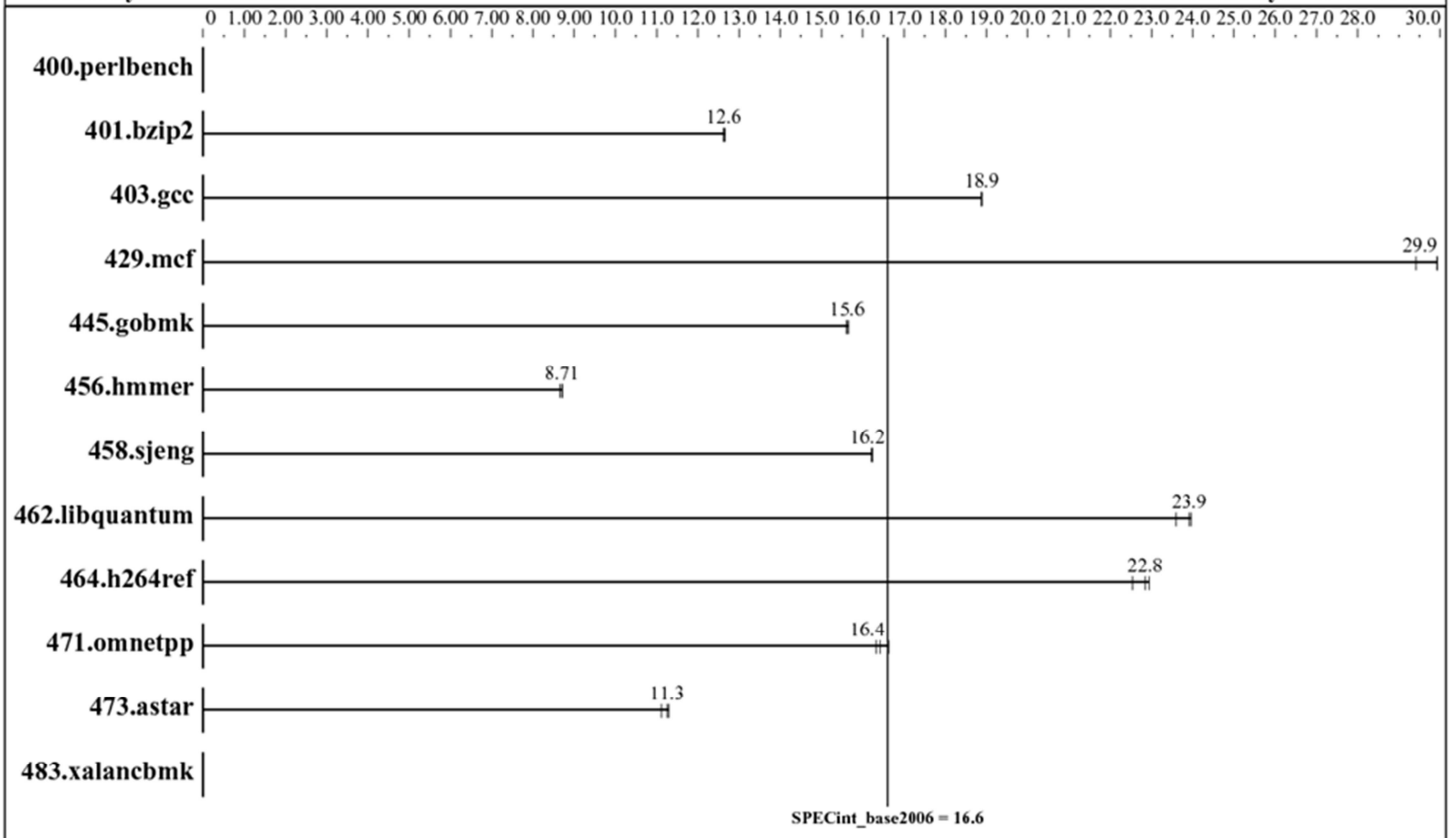
Intel
Lenovo X201

SPECint®2006 = Not Run

SPECint_base2006 = 16.6

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999



Hardware

CPU Name:	Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz
CPU Characteristics:	
CPU MHz:	2530
FPU:	Integrated
CPU(s) enabled:	2 cores, 1 chip, 1 core/chip, 2 threads/core
CPU(s) orderable:	1 chip
Primary Cache:	64 KB I + 64 KB D on chip per chip
Secondary Cache:	3072 KB I+D on chip per chip
L3 Cache:	None
Other Cache:	None
Memory:	4 GB (2 x 2GB DDR3 1066Mhz)
Disk Subsystem:	SATA
Other Hardware:	--

Software

Operating System:	SUSE Linux 11.3 (for i386)
Compiler:	nacl-gcc , nacl-g++ (for i386)
Auto Parallel:	No
File System:	ext4
System State:	runlevel 3
Base Pointers:	32-bit
Peak Pointers:	32-bit
Other Software:	None

SPEC CINT2006 Result

Intel
Lenovo X201

SPECint2006 = Not Run

SPECint_base2006 = 16.6

CPU2006 license: 0
Test sponsor: QUT
Tested by: Nuwan

Test date: Oct-2010
Hardware Availability: Dec-9999
Software Availability: Dec-9999

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench												
401.bzip2	**764**	**12.6**	764	12.6	762	12.7						
403.gcc	**426**	**18.9**	426	18.9	427	18.9						
429.mcf	**305**	**29.9**	305	29.9	310	29.4						
445.gobmk	**672**	**15.6**	670	15.7	672	15.6						
456.hmmer	1078	8.65	**1072**	**8.71**	1071	8.71						
458.sjeng	**746**	**16.2**	745	16.2	746	16.2						
462.libquantum	865	24.0	878	23.6	**866**	**23.9**						
464.h264ref	964	22.9	**969**	**22.8**	982	22.5						
471.omnetpp	383	16.3	**381**	**16.4**	376	16.6						
473.astar	631	11.1	621	11.3	**624**	**11.3**						
483.xalancbmk												

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

General Notes

```
462.libquantum: -DSPEC_CPU_LINUX
C base flags: -O2
C++ base flags: -O2
Fortran base flags: -O2
```

Base Compiler Invocation

C benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-gcc
 -lmycustom -lg(*)
```

C++ benchmarks (except as noted below):

```
/home/nuwan/Desktop/Kernel/nacl/build/native_client/src/third_party/nacl_sdk/linux/sdk/nacl-sdk/bin/nacl-g++
 -lmycustom -lg(*)
```

(*) Indicates a compiler flag that was found in a non-compiler variable.

Base Portability Flags

462.libquantum: -DSPEC_CPU_LINUX

Base Optimization Flags

C benchmarks:

401.bzip2: `-O3`

403.gcc: `Same as 401.bzip2`

429.mcf: `Same as 401.bzip2`

445.gobmk: `Same as 401.bzip2`

456.hmmer: `Same as 401.bzip2`

458.sjeng: `Same as 401.bzip2`

462.libquantum: `Same as 401.bzip2`

464.h264ref: `Same as 401.bzip2`

C++ benchmarks:

471.omnetpp: `-O3`

473.astar: `Same as 471.omnetpp`

For questions about this result, please contact the tester.
For other inquiries, please contact webmaster@spec.org.

Tested with SPEC CPU2006 v1.1.
Report generated on Thu Oct 21 04:26:23 2010 by SPEC CPU2006 PS/PDF formatter v6128.

Appendix B: Source Code Outline

This section provides an overview of the program code structure in the LibVM user space library, which can be used by clients to create library sandboxes via the LibVM interface. Additional files, such as testing code, examples and performance benchmarks have been omitted. Full source code can be obtained by e-mailing the author at nuwan.goonasekera@student.qut.edu.au or nuwan.ag@gmail.com.

LIBVM MODULE STRUCTURE

Figure B.1 depicts a simplified block diagram of the LibVM module structure. A description of each module follows.

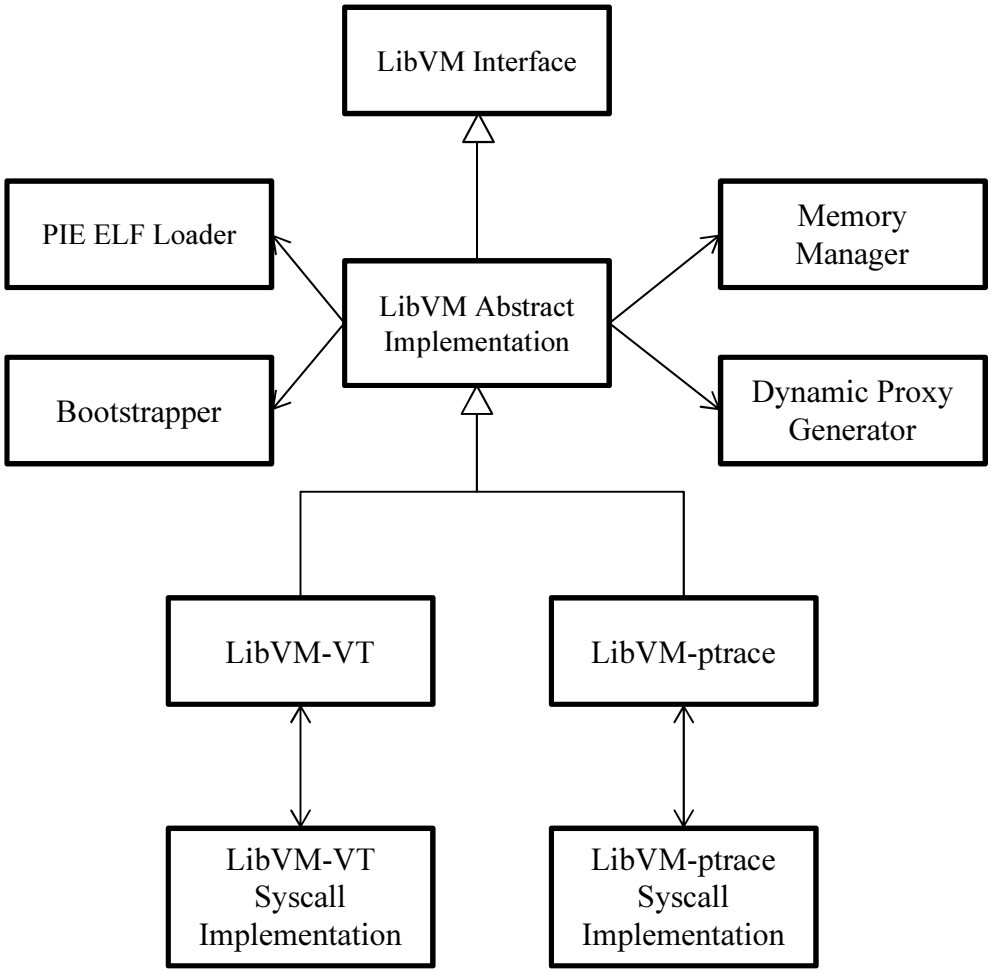


Figure B.1: LibVM module overview

1. **LibVM Interface** - The LibVM interface is the external façade of the LibVM user space library. The interface exposes the methods defined in Section 3.7.3. In addition, it exposes a `libvm_context` data structure, which is an opaque pointer supplied to clients, and is used to maintain LibVM state.

2. **LibVM Abstract Implementation** – This module provides an abstract implementation of the LibVM interface, based on the simplified virtual machine defined in Section 3.8. Concrete implementation details are deferred to the two implementations of LibVM, although other implementations conformant to the afore mentioned stack based virtual machine can also be plugged in.

3. **PIE ELF Loader** – The Position Independent Executable (PIE) ELF Loader module is responsible for loading and parsing position independent ELF modules. It is used by the LibVM implementation to load the LibVM bootstrapper code as well as the system ELF interpreter, which handle domain transitions and symbol resolution respectively, both of which are discussed in Section 4.3.

4. **Bootstrapper** – The Bootstrapper module is a PIE elf executable which provides domain transition support functions and executes within LibVM, and is discussed extensively in Section 4.3.

5. **Memory Manager** – The Manager Module handles basic memory allocation/de-allocation in the LibVM virtual machine, setting up the required memory regions and tracking memory regions which are in use by the LibVM virtual machine. This modules assists in determining whether memory references are valid, since all references obtained from untrusted code running within the virtual machine must fall within a region recognized and allowed by the memory manager.

6. **Dynamic Proxy Generator** – This module assists the `libvm_sym` function in the dynamic generation of proxy functions which are responsible for transparently switching execution from the executed function into the LibVM isolation domain for safe execution. This process is described in Section 4.3.3.

7. **LibVM-VT** – The module provides a concrete implementation of LibVM based on hardware virtualization support, and is described extensively in Section 5.4. It relies on libKVM [107] for the abstraction of lower level virtualization hardware. Some support modules have been omitted from the diagram for clarity, such as debugging support and CPUID emulation.
8. **LibVM-ptrace** – The module provides a process-tracing based implementation of LibVM, and is described in Section 4.3.
9. **LibVM-VT Syscall Implementation** – This module implements system calls for the LibVM-VT module. The implementation is discussed in Section 5.4. It shares code where possible with the LibVM-ptrace based implementation of system calls, but due to the two-stage nature of the *ptrace* mechanism is necessarily different from LibVM-VT.
10. **LibVM-ptrace Syscall Implementation** – This module implements system calls for the LibVM-ptrace module. Since *ptrace* is a two-stage process, the system calls are handled by a pre and post process. Where possible however, code is shared with the LibVM-VT based implementation. The implementation is discussed in Section 4.3.4.

LIBVM FOLDER STRUCTURE

The LibVM source code folder consists of a root folder which contains code common to both LibVM-ptrace and LibVM-VT. Implementation specific source code is contained in two subfolders.

LibVM root folder

- auxv.h – Header file for creating in-memory auxiliary vector.
- auxv.c – Implementation file for creating in-memory auxiliary vector.
- bootstrapper.c – LibVM bootstrapper executable.
- libvm.h – Main header file for LibVM interface.
- libvm.c – Abstract implementation of LibVM interface.
- libvm_tramp.h – Header file for LibVM domain transition proxy.

- libvm_tramp.S – Assembler routines for LibVM domain transition proxy, patched at runtime.
- memory_region.h – Header file for LibVM memory region management.
- memory_region.c – Implementation of LibVM memory region management.
- pie_elf_loader.h – Header file for Position Independent ELF executable loader.
- pie_elf_loader.c – Position Independent ELF loader, used to load LibVM bootstraper and ELF interpreter.
- syscall.h – Header file for LibVM recognized POSIX system calls.
- syscall.c – Abstract implementation of system calls.
- trampoline.c – Implementation of LibVM runtime trampoline patching for dynamically generated proxies.

LibVM-VT folder

- cpuid.h – Header file for LibVM CPUID emulation.
- cpuid.c – Implementation of LibVM CPUID emulation.
- debugger.h – Header file for debugging support in LibVM components.
- debugger.c – Implementation of debugging support for LibVM components such as breakpoints and step-through.
- kvm-impl.h – Header file for libKVM/VT based implementation of LibVM.
- kvm-impl.c – Implementation of VT based LibVM.
- libkvm.h – Header file for libKVM (from KVM source).
- syscall_proxy.h – Header file for LibVM-VT system calls.
- syscall_proxy.c – Implementation of LibVM-VT system calls.

LibVM-ptrace folder

- mem_clean.h – Header file for process memory cleanup routines.
- mem_clean.c – Implementation of process memory cleanup after fork, for LibVM-ptrace.
- ptrace_impl.h – Header file for *ptrace* based implementation of LibVM.
- trace_impl.c – Implementation of *ptrace* based LibVM.
- shared_mem.h – Header file for shared memory region management.
- shared_mem.c – Implementation file for shared memory region management.
- syscall_proxy.h – Header file for LibVM-ptrace system calls.
- syscall_proxy.c – Implementation of LibVM-ptrace system calls.

References

[1] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM SIGARCH Computer Architecture News,* vol. 34, pp. 2-13, 2006.

[2] M. Aiken, M. Fahndrich, C. Hawblitzel, G. Hunt, and J. Larus, "Deconstructing Process Isolation," presented at the Proceedings of the 2006 workshop on Memory system performance and correctness, San Jose, California, 2006.

[3] AMD. (2008, Accessed: 2009 Jan. 30). *AMD-V™ Nested Paging* [Online]. Available: http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf

[4] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development,* vol. 8, 1964.

[5] A. Avizienis, J. Laprie, and B. Randell, "Fundamental Concepts of Dependability," LAAS-CNRS, Toulouse, FranceApril 2001.

[6] A. Banerji, J. M. Tracey, and D. L. Cohn, "Protected shared libraries - a new approach to modularity and sharing," *Proceedings of the USENIX 1997 Annual Technical Conference,* pp. 59-75, 1997.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review,* vol. 37, pp. 164-177, 2003.

[8] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. (2008, Accessed: 2009 Jan. 30). *The Security Architecture of the Chromium Browser*. Available: http://crypto.stanford.edu/websec/chromium/

[9] M. Bauer, "Paranoid Penguin: An Introduction to Novell AppArmor," *Linux Journal,* vol. 2006, p. 13, 2006.

[10] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics virtual memory: concepts and design," *Commun. ACM,* vol. 15, pp. 308-318, 1972.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility Safety and Performance in the SPIN Operating System," presented at the Proceedings of the fifteenth ACM symposium on Operating systems principles, Copper Mountain, Colorado, United States, 1995.

[12] L. Blaxter, C. Hughes, and M. Tight, *How To Research*, 2nd ed.: Open University Press, 2001.

[13] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors," *ACM Transactions on Computer Systems (TOCS),* vol. 15, pp. 412-447, 1997.

[14] F. Campbell, "The portable UCSD p-System," *Microprocessors and Microsystems,* vol. 7, pp. 394-398, 1983.

[15] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems,* vol. 12, pp. 271-307, 1994.

[16] Y. Chiba, "Heap Protection for Java Virtual Machines," in *The 4th International Symposium on Principles and Practice of Programming in Java*, Mannheim, Germany, 2006.

[17] T. Chiueh, "Fast Bounds Checking Using Debug Register," in *High Performance Embedded Architectures and Compilers*. vol. 4917/2008, ed: Springer Berlin / Heidelberg, 2008, pp. 99-113.

[18] T. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," presented at the Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, Charleston, South Carolina, United States, 1999.

[19] T. Chiueh, G. Venkitachalam, and P. Pradhan, "Intra-address space protection using segmentation hardware," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, 1999, pp. 110-115.

[20] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," *ACM SIGOPS Operating Systems Review,* vol. 35, pp. 73-88, 2001.

[21] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Proceedings of the Annual Symposium on Fault Tolerant Computing*, 1996, pp. 304-313.

[22] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks," presented at the Proceedings of the 7th conference on USENIX Security Symposium - Volume 7, San Antonio, Texas, 1998.

[23] C. Cowan, F. Wagle, P. Calton, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *Proceedings of the DARPA Information Survivability Conference and Exposition*, 2000, pp. 119-129 vol.2.

[24] G. Czajkowski and L. Dayn, "Multitasking Without Compromise: A Virtual Machine Evolution," presented at the Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, Tampa Bay, FL, USA, 2001.

[25] D. Bell and L. LaPadula, "Secure Computer Systems: a Mathematical Model," MITRE Corp., Bedford, MA Technical Report MTR-2547 (Vol. II), 1973.

[26] D. Dean, "Secure Mobile Code: Where Do We Go From Here?," presented at the DARPA Workshop on Foundations for Secure Mobile Code, Monterey, CA, USA, 1997.

[27] P. J. Denning, "Fault Tolerant Operating Systems," *ACM Comput. Surv.,* vol. 8, pp. 359-389, 1976.

[28] E. W. Dijkstra, "The humble programmer," *Communications of the ACM,* vol. 15, pp. 859-866, 1972.

[29] K. Elphinstone, "Future Directions in the Evolution of the L4 Microkernel," in *NICTA workshop on OS verification*, Sydney, Australia, 2004.

[30] A. Endres, "An analysis of errors and their causes in system programs," presented at the Proceedings of the international conference on Reliable software, Los Angeles, California, 1975.

[31] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces " *Symposium on Operating System Design and Implementation,* pp. 75–88 2006.

[32] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in *USENIX Annual Technical Conference*, Boston, MA, 2008, pp. 293–306.

[33] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor," presented at the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure, Boston, MA, 2004.

[34] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," in *Foundations of Intrusion Tolerant Systems [Organically Assured and Survivable Information Systems]*, 2003, pp. 399-413.

[35] J. Frederick P. Brooks, *The Mythical Man-Month (anniversary ed.)*: Addison-Wesley Longman Publishing Co., Inc., 1995.

[36] S. Friedl. (2002, December 15th) Go Directly to Jail: Secure Untrusted Applications with chroot. *Linux Magazine*. Available: http://www.linux-mag.com/id/1230/

[37] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The Pebble Component-Based Operating System," in *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, 1999.

[38] I. Ganev, G. Eisenhauer, and K. Schwan, "Kernel plugins: when a VM is too much," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, San Jose, California, 2004.

[39] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *In Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, ed, 2003, pp. 163--176.

[40] T. Garfinkel, P. Ben, and R. Mendel, "Ostia: A Delegating Architecture for Secure System Call Interposition," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2004.

[41] GNU. (2011, Accessed: February 16th). *GCC, the GNU Compiler Collection*. Available: http://gcc.gnu.org/

[42] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the Wily Hacker," presented at the Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6, San Jose, California, 1996.

[43] Google. (2009, Accessed: 2010 Jul. 20). *Native Client Security Contest* [Online]. Available: http://code.google.com/contests/nativeclient-security/

[44] N. A. Goonasekera, W. J. Caelli, and C. J. Fidge, "A Hardware Virtualization Based Component Sandboxing Architecture," *Journal of Software,* vol. 7, pp. 2107-2118, 2012.

[45] N. A. Goonasekera, W. J. Caelli, and C. J. Fidge, "LibVM: An Architecture for Shared Library Sandboxing," *In Submission,* 2012.

[46] N. A. Goonasekera, W. J. Caelli, and T. Sahama, "50 Years of Isolation," in *Proceedings of the 2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, Brisbane, Australia, 2009, pp. 54-60.

[47] J. Gray, "Why do computers stop and what can be done about it?," *Technical Report 85.7, Tandem Computers,* June 1985.

[48] Grugq. (2004, Accessed: 6th January 2012). *Userland Exec*. Available: http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2004-01/0002.html

[49] P. H. Gum "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM Journal of Research and Development,* vol. 27, 1983.

[50] A. C. D. Haley, "The KDF.9 computer system," presented at the Proceedings of the December 4-6, 1962, fall joint computer conference, Philadelphia, Pennsylvania, 1962.
[51] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*, 1991, pp. 125-138.
[52] C. Hawblitzel and T. von Eicken, "A Case for Language-Based Protection," Cornell University Technical Report 98-1670, 1998.
[53] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke, "The Mungi single-address-space operating system," *Software: Practice and Experience,* vol. 28, pp. 901-928, 1998.
[54] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed.: Morgan Kaufmann Publishers Inc., 2006.
[55] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News,* vol. 34, pp. 1-17, 2006.
[56] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a Highly Dependable Operating System," in *Sixth European Dependable Computing Conference*, 2006, pp. 3-12.
[57] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault isolation for device drivers," in *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 33-42.
[58] M. Howard. (2004) Attack surface: Mitigate security risks by minimizing the code you expose to untrusted users. *Microsoft MSDN Magazine*.
[59] G. Hunt, M. Aiken, M. Fahndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber, "Sealing OS processes to improve dependability and safety," *ACM SIGOPS Operating Systems Review,* vol. 41, pp. 341-354, 2007.
[60] R. Hyde, *The Art of Assembly Language*: No Starch Press, 2003.
[61] IEEE, "1003.1-2008 Standard for Information Technology - Portable Operating System Interface (POSIX(R))," ed: Institute of Electrical and Electronics Engineers, 2008.
[62] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual* vol. 1: Basic Architecture: Intel Corporation, 2007.
[63] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A* vol. 3A: System Programming Guide: Intel Corporation, 2007.
[64] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* vol. 3B: System Programming Guide: Intel Corporation, 2007.
[65] Intel. (2008, Accessed: 25th May, 2011). *Intel® Virtualization Technology* [Online]. Available: http://www.intel.com/technology/virtualization/index.htm
[66] K. R. Irvine, *Assembly Language for Intel-Based Computers*, 4th ed.: Prentice Hall, 2002.
[67] ISO/IEC-15408, "Common Criteria v3.0," ed, 2005.
[68] J. Rutkowska and R. Wojtczuk, "The Qubes OS Architecture," Invisible Things Lab2010.
[69] T. Jaeger, J. Liedtke, and N. Islam, "Operating System Protection for Fine-Grained Programs," in *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, 1998, pp. 143–158.

[70] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," ed, 1999, pp. 19-34.
[71] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting Experiments in Software Engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. K. Sjøberg, Eds., ed: Springer London, 2008, pp. 201-228.
[72] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*: Springer Publishing Company, Incorporated, 2010.
[73] P. Kamp and R. N. M. Watson, "Jails: Confining the Omnipotent Root," in *Second International SANE Conference*, 2000.
[74] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A VMM security kernel for the VAX architecture," in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, 1990, pp. 2-19.
[75] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture support for single address space operating systems," *SIGPLAN Not.,* vol. 27, pp. 175-186, 1992.
[76] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: Software-based Memory Protection For Sensor Nodes," in *6th International Symposium on Information Processing in Sensor Networks*, 2007, pp. 340-349.
[77] L. Lam and T. Chiueh, "Checking array bound violation using segmentation hardware," in *The International Conference on Dependable Systems and Networks*, 2005, pp. 388-397.
[78] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM,* vol. 16, pp. 613-615, 1973.
[79] G. Law and J. McCann, "A new protection model for component-based operating systems," in *Proceedings of the IEEE International Performance, Computing, and Communications Conference*, 2000, pp. 537-543.
[80] I. Lee and R. K. Iyer, "Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 operating system," in *The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993, pp. 20-29.
[81] P. D. Leedy and J. E. Ormrod, *Practical Research: Planning and Design*, 8th ed.: Prentice Hall, 2004.
[82] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser, "User-Level Device Drivers: Achieved Performance " *Journal of Computer Science and Technology,* vol. 20, pp. 654-664, 2005.
[83] J. LeVasseur, V. Uhlig, J. Stoess, and S. G\\#246tz, "Unmodified device driver reuse and improved system dependability via virtual machines," presented at the Proceedings of the 6th Symposium on Operating Systems Design & Implementation, San Francisco, CA, 2004.
[84] J. R. Levine, *Linkers and Loaders*: Morgan Kaufmann Publishers Inc., 1999.
[85] Z. Liang, V. N. Venkatakrishnan, and R. Sekar, "Isolated program execution: an application transparent approach for executing untrusted programs," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, 2003, pp. 182-191.
[86] J. Liedtke, K. Elphinstone, S. Schonberg, H. Hartig, G. Heiser, N. Islam, and T. Jaeger, "Achieved IPC performance (still the foundation for extensibility)," in *The Sixth Workshop on Hot Topics in Operating Systems*, 1997, pp. 28-31.

[87] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," presented at the Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, 2001.
[88] P. K. Manadhata and J. M. Wing, "A Formal Model for a System's Attack Surface," in *Moving Target Defense*. vol. 54, S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Eds., ed: Springer New York, 2011, pp. 1-28.
[89] A. J. W. Mayer, "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?," *SIGARCH Computer Architecture News,* vol. 10, pp. 3-10, 1982.
[90] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," presented at the Proceedings of the 15th conference on USENIX Security Symposium, Vancouver, B.C., Canada, 2006.
[91] M. Mcllroy, "Mass Produced Software Components," presented at the Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 1968.
[92] F. Mehnert, M. Hohmuth, and H. Hartig, "Cost and benefit of separate address spaces in real-time operating systems," in *23rd IEEE Real-Time Systems Symposium*, 2002, pp. 124-133.
[93] N. Mendelsohn, "Operating systems for component software environments," in *The Sixth Workshop on Hot Topics in Operating Systems*, 1997, pp. 49-54.
[94] Microsoft. (2012, Accessed: February). *FreeLibrary function*. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms683152(v=vs.85).aspx
[95] Microsoft. (2012, Accessed: February). *GetProcAddress function*. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms683212(v=vs.85).aspx
[96] Microsoft. (2012, Accessed: February). *LoadLibrary function*. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx
[97] Microsoft. (2012, Accessed: February). *POSIX and OS/2 are not supported in Windows XP or in Windows Server 2003*. Available: http://support.microsoft.com/kb/308259
[98] Mozilla. (2011, Accessed: 25th May, 2011). *Multi-process architecture*. Available: https://developer.mozilla.org/en/Multi-Process_Architecture
[99] Mozilla. (2011, Accessed: 22/12/2011). *XPCOM*. Available: https://developer.mozilla.org/en/XPCOM
[100] G. C. Necula and P. Lee, "Safe Kernel Extensions Without Run-Time Checking," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, 1996.
[101] G. V. t. Noordende, B. Ádám, H. Rutger, M. T. B. Frances, and S. T. Andrew, "A secure jailing system for confining untrusted applications," in *In proceedings of the second International Conference on Security and Cryptography (SECRYPT)*, 2008, pp. 414--423.
[102] D. S. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," presented at the Proceedings of the 11th USENIX Security Symposium, 2002.
[103] J. Petersson. (2005, Accessed: 2011, Jun. 18). *What is linux-gate.so.1?* [Online]. Available: http://www.trilithium.com/johan/2005/08/linux-gate/

[104] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM,* vol. 17, pp. 412-421, 1974.
[105] D. Price and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," presented at the Proceedings of the 18th USENIX Conference on System Administration, Atlanta, GA, 2004.
[106] A. Purohit, C. P. Wright, J. Spadavecchia, and E. Zadok, "Cosy: Develop in User-Land, Run in Kernel-Mode," presented at the Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems, Lihue, Hawaii, USA, 2003.
[107] Redhat. (2010, Accessed: 2010, Jul. 20). *Kernel Based Virtual Machine* [Online]. Available: http://www.linux-kvm.org/page/Main_Page
[108] Redhat. (2012, Accessed: February). *Newlib homepage*. Available: http://sourceware.org/newlib/
[109] C. Reis, B. Bershad, S. D. Gribble, and H. M. Levy, "Using Processes to Improve the Reliability of Browser-based Applications," Department of Computer Science and Engineering, University of Washington, Technical Report UW-CSE-2007-12-01, 2007.
[110] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," in *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, 2000, p. 10.
[111] Rohit Dhamankar, Mike Dausin, Marc Eisenbarth, James King, Kandek W, Ullrich J, S. E, and L. R. (2009, The Top Cyber Security Risks. *http://www.sans.org/top-cyber-security-risks/summary.php*.
[112] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on,* vol. 21, pp. 5-19, 2003.
[113] J. H. Saltzer, "Protection and the control of information sharing in Multics," *Commun. ACM,* vol. 17, pp. 388-402, 1974.
[114] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE,* vol. 63, pp. 1278-1308, 1975.
[115] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," *Commun. ACM,* vol. 15, pp. 157-170, 1972.
[116] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010, pp. 317-331.
[117] L. Seawright and R. MacKinnon, "VM/370 - a study of multiplicity and usefulness," *IBM Systems Journal,* vol. 18, pp. 4-17, 1979.
[118] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," presented at the Proceedings of the 19th USENIX conference on Security, Washington, DC, 2010.
[119] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," presented at the Proceedings of the 11th ACM conference on Computer and communications security, Washington DC, USA, 2004.
[120] Z. C. Shreuders, "Functionality-Based Application Confinement," PhD Thesis, Murdoch University, 2011.

[121] C. Small and M. Seltzer, "A comparison of OS extension technologies," in *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, 1996, pp. 41-54.
[122] G. Smith, "Principles of Secure Information Flow Analysis," *Advances in Information Security,* vol. 27, pp. 291-307, 2007.
[123] J. Sugerman, G. Venkitachalam, and B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," presented at the Proceedings of the General Track: 2002 USENIX Annual Technical Conference, 2001.
[124] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability-a study of field failures in operating systems," in *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, 1991, pp. 2-9.
[125] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering Device Drivers," *Proceedings of the 6th Symposium on Operating Systems Design and Implementation,* 2004.
[126] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *The Nineteenth ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, USA, 2003, pp. 207-222.
[127] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: an architecture for reliable device drivers," in *The 10th ACM SIGOPS European Workshop: Beyond the PC*, Saint-Emilion, France, 2002.
[128] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, 2nd ed.: Addison-Wesley, 2002.
[129] L. Tan, E. M. Chan, R. Farivar, N. Mallick, J. C. Carlyle, F. M. David, and R. H. Campbell, "iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support," in *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, 2007, pp. 134-144.
[130] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed.: Prentice Hall, 2001.
[131] A. S. Tanenbaum, *Structured Computer Organization*, 5th ed.: Prentice Hall, 2005.
[132] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?," *Computer,* vol. 39, pp. 44-51, 2006.
[133] The Google Chrome Team. (2008, Accessed: 2009, Jan 30). *Chromium Developer Documentation: Multi-process Architecture* [Online]. Available: http://dev.chromium.org/developers/design-documents/multi-process-architecture
[134] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer,* vol. 38, pp. 48-56, 2005.
[135] Unascribed. (2010, Accessed: February, 2012). *How KVM deals with memory*. Available: http://www.linux-kvm.org/page/Memory
[136] A. Vasudevan, R. Yerraballi, and A. Chawla, "A high performance Kernel-Less Operating System architecture," presented at the Proceedings of the Twenty-eighth Australasian conference on Computer Science, Newcastle, Australia, 2005.
[137] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Operating Systems Review,* vol. 27, pp. 203-216, 1993.

[138] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the Denali isolation kernel," *ACM SIGOPS Operating Systems Review,* vol. 36, pp. 195-209, 2002.

[139] A. Wiggins, S. Winwood, H. Tuch, and G. Heiser, "Legba: Fast hardware support for fine-grained protection," in *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, 2003.

[140] P. R. Wilson, "Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware," *SIGARCH Computer Architecture News,* vol. 19, pp. 6-13, 1991.

[141] E. Witchel, J. Cates, and K. Asanovi, "Mondrian memory protection," *SIGARCH Computer Architecture News,* vol. 30, pp. 304-316, 2002.

[142] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Networked Windows NT system field failure data analysis," in *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, 1999, pp. 178-185.

[143] Z. Xu, B. P. Miller, and T. Reps, "Safety checking of machine code," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Vancouver, BC, Canada, 2000, pp. 70-82.

[144] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," *Communications of the ACM,* vol. 53, pp. 91-99, 2010.

[145] A. Zeigler. (2008, Accessed: 2009, Jan 30). *IE8 and Loosely-Coupled IE (LCIE)* [Online]. Available: http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx