LibVM: An Architecture For Shared Library Sandboxing

NUWAN GOONASEKERA, WILLIAM CAELLI AND COLIN FIDGE
Queensland University of Technology

Many software applications extend their functionality by dynamically loading libraries into their allocated address space. However, shared libraries are also often of unknown provenance and quality and may contain accidental bugs or, in some cases, deliberately malicious code. Most sandboxing techniques which address these issues require recompilation of the libraries using custom tool chains, require significant modifications to the libraries, do not retain the benefits of single address-space programming, do not completely isolate guest code, or incur substantial performance overheads.

In this paper we present LibVM, a sandboxing architecture for isolating libraries within a host application without requiring any modifications to the shared libraries themselves, while still retaining the benefits of a single address space and also introducing a system call inter-positioning layer that allows complete arbitration over a shared library's functionality. We show how to utilize contemporary hardware virtualization support towards this end with reasonable performance overheads and, in the absence of such hardware support, our model can also be implemented using a software-based mechanism.

We ensure that our implementation conforms as closely as possible to existing shared library manipulation functions, minimizing the amount of effort needed to apply such isolation to existing programs. Our experimental results show that it is easy to gain immediate benefits in scenarios where the goal is to guard the host application against unintentional programming errors when using shared libraries, as well as in more complex scenarios, where a shared library is suspected of being actively hostile. In both cases, no changes are required to the shared libraries themselves.

Categories and Subject Descriptors: D3.4 [**Software**]: Programming Languages—*Processors*; D4.6 [**Operating Systems**]: Security and Protection

General Terms: System Call Inter-positioning, Component Isolation, Dependable Systems

Additional Key Words and Phrases: Information Security, Hardware Virtualization, Component Software

1 INTRODUCTION

Most modern application programs consist of multiple shared libraries at an Operating System level, in the form of Shared Objects (.so) or Dynamic Link Libraries (DLLs). While such libraries may be statically linked, much more commonly they are also dynamically linked at program startup. In addition, such libraries may also be loaded on demand, in the case of "plug-ins" and software extensions for example, to augment application functionality at run time.

However, since these shared libraries reside within the same address space, an illegal memory access, an invalid instruction, or even penetration by malicious code, can compromise the containing application as a whole. This is amply demonstrated by the fact that over 50% of CERT-reported security threats are due to buffer-overflow vulnerabilities [1]. Zeigler reports that over 70% of crashes in the *Internet Explorer* browser are caused by third-party add-ons [2]. Therefore, in the case of both a trusted add-on shared library with a buffer overflow vulnerability, as well as a potentially untrustworthy add-on shared library from a third-party, the dangers are quite similar, with the distinct possibility of executing arbitrary code which can compromise the entire system. Even when there is no danger of malicious code, relatively common errors, such as an invalid memory reference by a shared library, could corrupt critical memory regions in the host application, since they reside in the same shared address space, leading to unstable applications. Therefore, there is a critical need to isolate applications from any extensions that they incorporate [3-6].

While several approaches have been used to address the overall problem of isolation [7], there are very few robust intra-address space isolation mechanisms which can run third-party shared libraries unaltered while preserving programming simplicity and maintaining low run-time overheads. In most previous work some modifications to the shared library, such as recompilation or linking against a framework, are necessary for the intra-address space protection mechanism to work [8]. However, when the source code of third-party libraries is unavailable, this may not always be viable anyway. In other previous work, the mechanisms provided do not guarantee the same level of safety as having separate address spaces, or impose restrictions on the range of allowed instructions, such as in the case of Software Fault Isolation [9] and its derivatives [10-12]. We are aware of only one other effort to isolate unmodified shared libraries [13], which we discuss in greater detail in Section 5.

In this paper, we present an intra-address space protection framework named LibVM, which usually utilizes Hardware Virtualization instructions to provide such guarantees. However, in the absence of such instructions, it can fall back on a slower but equally effective *ptrace*-based [14] approach to enable the isolation of entirely unmodified third-party shared libraries. The host application needs to be modified to utilize LibVM, but the changes required are relatively modest.

The basic idea behind LibVM is to isolate a shared library by "fooling" it into thinking that it is executing as normal, within an operating system process, as part of an application, and with full access to the usual machine instructions and system calls. The easiest way to achieve this is to execute the library within a "virtual process", a process which has all the trappings of a real process but is in fact a fully virtualized sandbox. By placing the library in a separate sandbox, which emulates a normal OS process, and by virtualizing all OS calls, we effectively control the world view available to the library as well as its effects outside of the sandbox. The application issuing a call to a function within such an isolated shared library must first execute a switch to the virtualized sandbox, so that the library is safely executed, and upon completion of the library function, execution must return back to the application. Most techniques for library, component or application isolation are a variant of this idea – they only differ in the strength of the illusion. The basic principle is also portable across operating systems. Some techniques (such as Software Fault Isolation (SFI) [9] or isolation of a library into a separate process via IPCs) are less transparent than others and have different trade-offs (discussed later). Our LibVM architecture provides a strong illusion of this virtualization in both our Hardware Virtualization based approach, as well as our software-based implementation.

The main contributions of this research are that it:

1. Demonstrates an implementation independent API for isolating individual shared libraries, as opposed to complete processes.
2. Demonstrates how shared libraries can be isolated using a hardware virtualization based technique. (This is in contrast to our previous work [15], in which we gauged the effectiveness of implementing System Call Interpositioning [16] on whole executables using hardware virtualization support.)
3. Demonstrates how shared libraries can additionally be isolated using a *ptrace*-based System Call Interpositioning technique, similar to earlier research in the field [13], that dynamically determines whether or not instructions about to be executed are safe.
4. Demonstrates a technique for isolating shared libraries without requiring any modifications to the libraries themselves, whereas all previous techniques require some modification or recompilation of the libraries. This also makes recovery from unmodified shared library faults possible, since they are isolated and self-contained within separate domains, in contrast to the address space of the whole process being accessible to a shared library.
5. Makes address space sharing between the library and application transparent within predefined bounds. This means that a memory address within the library and a memory

address within a host process refer to the same memory location, with no *"swizzling"* [17] or translation required. One caveat is that while the host application can access all of the shared library's memory, the shared library can only access memory allocated to it. (There is a slight but functionally non-impacting exception in our *ptrace*-based technique, detailed later)

The rest of this paper is organized as follows; Section 2 contains a high-level, implementation-independent overview of the LibVM architecture and Section 3 provides an implementation-independent description of the LibVM API. Sections 4 and 5 discuss two specific implementations of the API, a hardware virtualization based implementation and a shared memory/ptrace based system-call interpositioning implementation respectively. Section 6 provides an evaluation of our implementations, including micro and macro benchmarks of CPU and memory usage in contrast to a native executable. Section 7 reviews related work in this field and the paper concludes with Section 8.

2 LIBVM ARCHITECTURE

In this section we describe the overall principles behind and requirements for our LibVM isolation architecture in an implementation-independent way.

2.1 Overview

For the purpose of discussing the architecture, we utilize a Linux based process model, as this was the operating system on which our two implementations were developed. However, the general idea remains portable by substituting the conceptual equivalents in another operating system.

Figure 1Figure 2 shows a typical layout of a process in Linux memory. As depicted in the figure, a shared library executing within a process shares several things in common with the hosting process.

1. The heap.
2. The stack.
3. The C runtime.
4. The Virtual Dynamic Shared Object (used as a gateway when making Operating System Calls in Linux [18]).
5. The ELF Interpreter/Dynamic Linker (in order to load/resolve additional libraries the shared library itself may depend on).

Therefore, in order to fully isolate a shared library from its host application, all of the above must also be conceptually isolated, or virtualized in full. However, in order for the host application to access the result of a computation or share required data transparently with a shared library, the memory regions of the shared library must be accessible to the host (but not vice-versa, as this would violate isolation).

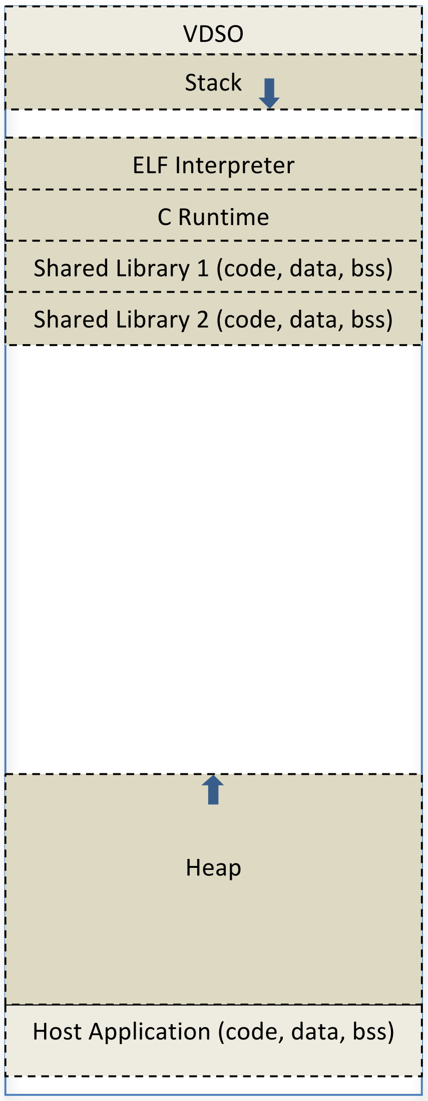
VDSO
Stack
ELF Interpreter
C Runtime
Shared Library 1 (code, data, bss)
Shared Library 2 (code, data, bss)
Heap
Host Application (code, data, bss)

Figure 1 - Process Layout in memory

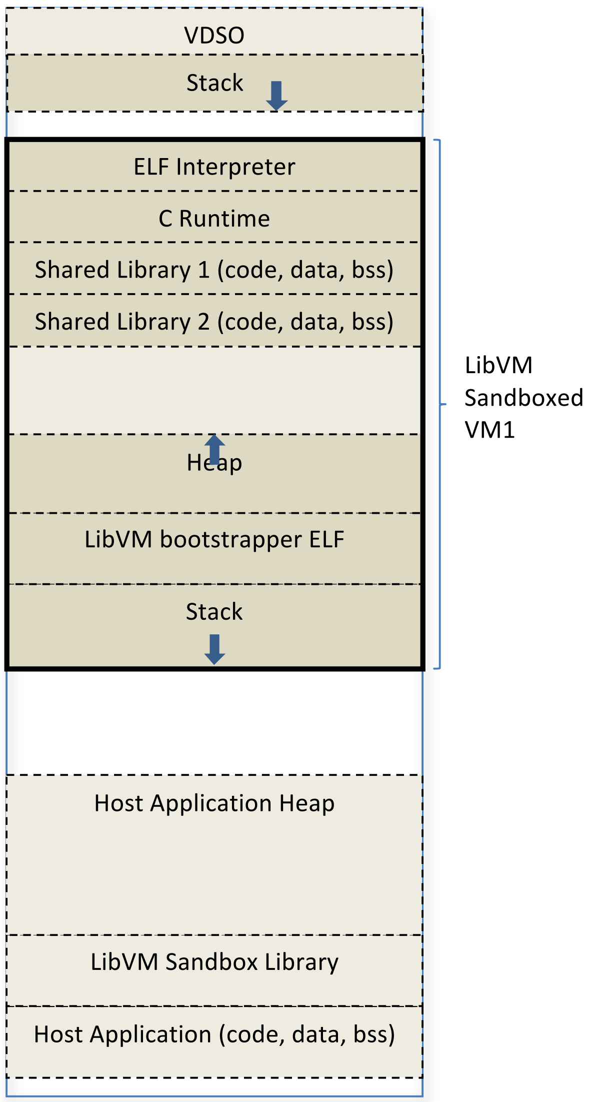

Figure 2 - LibVM Isolated Shared Library

Therefore, an isolated shared library in LibVM can be conceptually depicted as in Figure 2. As shown, the shared library gets its own heap, stack and C runtime, as well as a dynamic linker (to enable loading the library's dependencies in turn), which allows the shared library to execute within the virtualized sandbox, unaware that it is executing within a virtualized process.

However, the virtualized sandbox is assigned only a subset of memory within the entire application, so that the address space mappings remain synchronised. LibVM also assumes that a pointer in the sandbox and a pointer in the application space refer to the same memory address, so that pointers may be transparently passed between the host and the shared library. The shared library cannot access any memory locations outside of the sandbox's bounds.

The user can create multiple sandboxes, each with different security policies, or a single sandbox with multiple shared libraries. This is depicted in Figure 3.

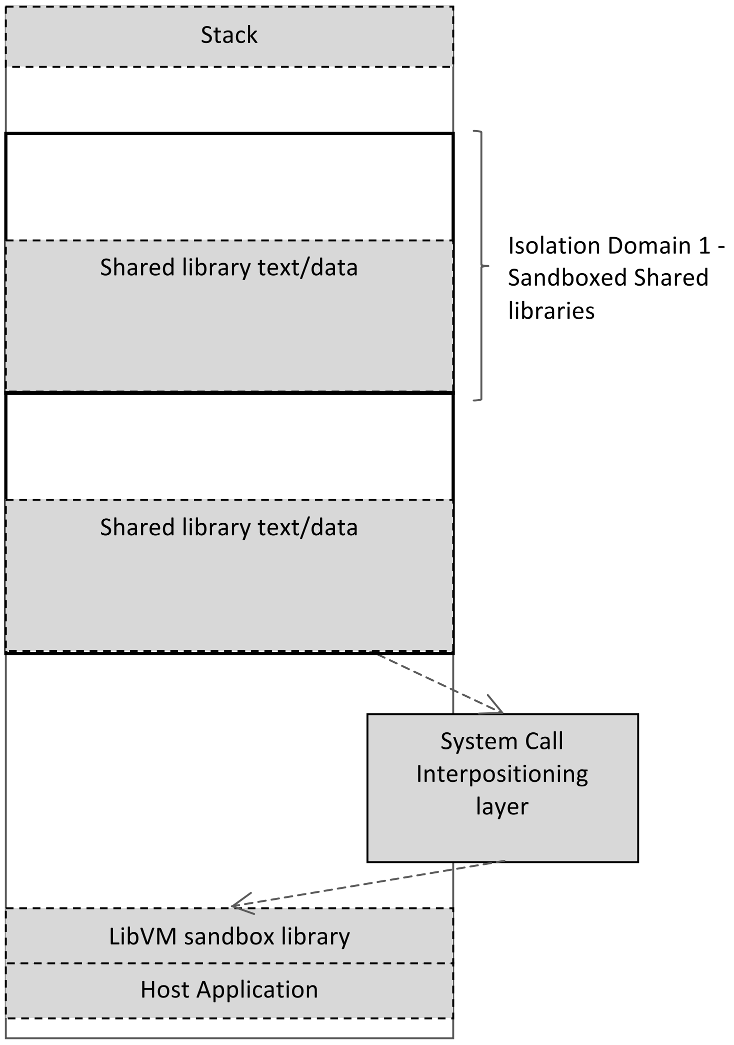

Figure 3 - Process memory layout in a program which utilizes LibVM

This model enables individual sandboxes to have different isolation policies, depending on the level of trust awarded to the shared library. Sandboxes can also contain multiple shared libraries, allowing an isolation policy to be shared. (LibVM does not at present support direct communication between two sandboxes. Such communication would have to be mediated by the parent or both libraries would have to be loaded into a single sandbox.)

From an application developer's perspective, LibVM is a sandboxing library that can be used to define such a sandbox and to load additional shared libraries into it. Since we provide address space transparency, we require the developer to specify the size of the reserved address space range in advance. This specified address space is initially only reserved and not actually used yet. It also has 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, we perform a controlled transition to the sandbox, execute the code within the sandbox, and return to the host. The shared library is free to make additional system calls, all of which can be intercepted by the programmer and proxied, thus separating security policy from mechanism.

In order to meet the requirements above, we need a virtual machine/sandbox environment with specific requirements, which are discussed next.

2.2 Requirements

As mentioned previously, our goal is to provide a fully virtualized "process" in which the shared library can execute. LibVM therefore relies on the availability of a virtualized sandbox environment wherein the execution of arbitrary code is possible in a controlled fashion. To provide such a fully virtualized process for a shared library, the VM environment must match the architecture and implementation of the hosting application. Strictly speaking, LibVM requires a process virtualizing library with the following specific features.

1. It must support full virtualization of the CPU and memory.
2. It must be compatible with the host environment's architecture (e.g., 64-bit x86).
3. Discontinuous memory regions must be supported, with access attempts to unmapped regions resulting in trappable page faults (so that the memory range allocated to the sandbox within the host process is identical in the virtual machine, which is required for maintaining pointer transparency).
4. It must provide full control over its address space layout (i.e., in a 32-bit system, it must provide access to all 4 GB of available address space). This is for the same reasons as those outlined above.
5. Privileged instructions (e.g. interrupt invocations, syscall instructions) must be trappable. This is so that the shared library's actions can be controlled.
6. The host operating system must have a clearly defined system call interface, so that system-call interpositioning is possible (currently unavailable on Microsoft Windows).

Based on these requirements, LibVM defines an abstract interface which hides away details of the specific implementation. In order to test this interface, we have created two separate implementations, one based on hardware virtualization support and another based on shared memory and *ptrace*-based system call interpositioning. As such, the following discussion is divided into two main parts, a description of the main interface and the two specific implementations.

3 LIBVM INTERFACE

This section describes how LibVM is used by a host application to execute components in a constrained environment. Our interfaces are largely implementation independent, but rely on the requirements outlined in Section 2.2. The basic interface strives to emulate the standard POSIX interfaces, which we describe below.

3.1 Existing POSIX interface

The POSIX standard 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 used in most UNIX-based operating systems. (Microsoft Windows uses similar system calls – LoadLibrary, GetProcAddress and FreeLibrary.)

```
 1 void    *handle;
 2 typedef void (*hello_func)(int x, int y);
 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 integer as a parameter */
11 (*fptr)(100, 200);
```

Figure 4 - POSIX example of shared library call

The code snippet in Figure 4 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 it 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 as in line 8. The symbol may be a function or a data pointer. If it is a function pointer, it can then be utilized to directly invoke the function (line 11).

Several observations can be made at this point.

a. It is trivial to obtain a pointer to a function and invoke it directly, as in line 8, without the necessity for parameter marshalling, which demonstrates programming simplicity within a single address space.

b. Preserving this same model is advantageous, as a lot of previously written code can be transferred to this model with minor changes.

c. The interface is suitably simple and easy to understand for something as important as dynamic shared libraries.

3.2 LibVM Interface

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

1. libvm_initialize – Initializes 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

In order to demonstrate the functionality of our approach, we start off with a code snippet, in Figure 5, which can be contrasted with the POSIX equivalent in Figure 4.

```
 1 void    *handle;
 2 typedef void (*hello_func)(int x, int y);
 3
 4 /* Initialize the isolation sub-system */
 5  struct libvm * libvm_ptr = libvm_initialize(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)(100, 200);
```

Figure 5 - LibVM example of shared library call

As can be seen in Figure 5, the basic interface to the host is almost identical to the POSIX case. The main difference is the addition of an extra initialization function in line 5, which must be done once to initialize the isolation subsystem and the subsequent destruction of it. It should be noted that no changes are required to the target shared library. LibVM functions are used in place of their POSIX counterparts on lines 8 and 11, but the calling semantics of the functions remain identical in lines 8, 11 and 14, meaning that, in simple scenarios, an application can be trivially ported to execute the library within an isolated environment.

While the methods to open, close and obtain a function pointer from the guest are analogous to their POSIX counterparts, the libvm_guest_malloc and libvm_guest_free functions exist specifically so that the host application can negotiate a shared chunk of memory within the guest 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 utilizing a value obtained from this shared address space and must treat it as untrusted, since untrusted code executing within the sandbox may asynchronously manipulate that memory. Therefore, validation must be performed before using such values, and Time Of Check To Time Of Use (TOCTOU) [16] bugs caused by asynchronous code are a potential source of danger which must be guarded against.

By using these allocation functions, and thanks to the address space's transparency, the host application can construct complex objects graphs within the guest's address space. However, in practice, the host may want to pass an object graph from its own address space onto the guest address space, in which it case it would have to be copied over to the shared space first. Alternatively, the guest may wish to return a complex object graph with lengthy pointer chains, in which case the whole graph would have to be validated before use, in order to prevent malicious or invalid pointer dereferences. This would quickly degenerate into a situation which is not different from marshalling objects between disparate address spaces, and would appear to limit the usefulness of transparent address spaces.

However, this problem can be addressed through language level or compiler level support for LibVM sandboxes. Since LibVM enables address space transparency, the compiler could trivially use type information to generate additional code to copy object graphs across, and to validate returned objects and their pointer chains. Any pointers obtained from a call to a sandboxed component could also be automatically validated by compiler generated code to ensure that they fall within the sandboxed region. 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 is a subject for future research. In simpler cases, as in the example shown in Figure 5, it would be possible to trivially pass in the required values to a function. Therefore, address space transparency support could both simplify and speed up data sharing with components.

Based on this interface, we have built two implementations. Our primary version of LibVM is built on KVM [19], which provides an abstraction layer over the hardware virtualization support built into newer Intel and AMD x86 processors. However, a completely software-based implementation is possible in principle, even though its performance will be slower (Section 5). The combination of KVM and hardware virtualization support provides an extremely lightweight virtual machine which fulfills our basic CPU and memory virtualization requirements well, without incurring the overhead of a full-blown virtual machine implementation. Our performance measurements have shown that the overhead is low enough to offer competitive performance in comparison to other techniques. Sections 4 and 5 describe these two distinct implementations.

4 HARDWARE VIRTUALIZATION BASED IMPLEMENTATION

Hardware Virtualization support in modern hardware can be used to easily create a virtualized sandbox which meets the requirements outlined in Section 2.2. This section describes both the background and the details of our implementation.

4.1 Use of Virtualization Hardware

Hardware virtualization is an isolation mechanism which has been used for decades. Although the popularity of Virtual Machine (VM) technology waned somewhat over the years, mainly in small systems and client-server environments, there has lately been a resurgence of interest in it with the development and marketing of software systems such as VMWare [20, 21], which provide a Virtual Machine Monitor (VMM) for the popular Intel x86 architecture. This has occurred even though the Intel x86 architecture itself has several non-virtualizable instructions which do not meet Popek and Goldberg's virtualization requirements [22]. Many novel techniques have been used to overcome these limitations, such as binary translation [20, 21] and para-virtualization [23-25].

In 2005, Intel and AMD introduced additional machine instructions to their respective architectures to remedy this problem [26, 27]. The machine instructions were similar in nature to those of the old IBM System/370 mainframe system and enabled the interpretive execution of code and additional hardware-managed control blocks. The Intel and AMD extensions are similar [20], which makes it easy to support either instruction set. Uhlig et al. [28] provide an overview of the architecture, with additional details being available elsewhere [26, 27]. However, as noted by Adams and Agesen [20], 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) [29] and Intel followed suit by introducing Extended Page Tables (EPT) in their *Nehalem* processor architecture, both of which add support for IO MMU virtualization [30]. These features enable the creation of lightweight hardware-assisted virtual machines with extremely high performance.

The KVM kernel module [19], an integral part of newer Linux distributions, builds on this hardware virtualization support, and provides an abstraction layer over the hardware instructions, which it exposes as a userspace library via LibKVM. While an operating system process would normally execute in kernel mode or user mode, KVM provides a third execution mode, "guest mode", which allows machine instructions to be executed in an interpreted fashion using hardware virtualization support. Due to the hardware virtualization support, instructions executing in guest mode incur no additional overheads, other than when executing privileged instructions, which could cause a trap into the KVM Virtual Machine Monitor. It is therefore possible to intercept such traps and respond appropriately, such as in attempts to invoke interrupts, and LibVM does so via the libKVM library. Therefore, KVM provides the basic support necessary for running a shared library in a virtualized execution environment. Later sections delve into the mechanics in detail.

4.2 Implementation Description

Figure 2Figure 1 and Figure 3 show the basic address space layout of a LibVM isolated shared library. LibVM 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 2. Therefore, there is complete parity between a guest address and a host address, which is the key to enabling simple, 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 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 utilize the system's existing ELF interpreter [31] for the purpose. The basic process we follow is to emulate the operating system's process initialization 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 virtualized 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 initialized "mini-process", along with a C runtime and dynamic linker, all of which reside within the execution container. We then utilize the dynamic linker to load additional shared libraries in turn, exactly as would occur within a standard process.

4.2.1 Initialization

The virtual machine bootstrap process is triggered when *libvm_initialize()* is invoked for the first time as shown in Figure 5. The process is as follows.

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

   We utilize the KVM library [19] 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 32-bit 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 established (as shown in Figure 2Figure 1).

   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 established 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) [18]. The Linux VDSO is a basic trampoline 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 uses a simple ELF loader which loads the Linux ELF interpreter [31] and our bootstrapper into memory at the top of LibVM'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, but extensive security checks are not necessary as the ELF loader is not a part of the system's vulnerable attack surface. This is because the loader is only used to load the system's ELF interpreter and our own bootstrapper, both of which are trusted libraries. The ELF loader also maps our bootstrapper executable into memory, and places it immediately after the ELF interpreter.

4. LibVM sets up the VM's stack and begin 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 then 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 GCC's –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 initializes 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 for all future symbol resolution within the execution container. This special system call also heralds the completion of the bootstrap process, and LibVM suspends execution of the virtual machine and returns from its main initialization routine. Up till this point, all system calls made by the execution container were trusted. However, from this point onwards, LibVM must treat all further system calls as potentially untrustworthy.

4.2.2 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 must maintain the illusion that the same thing is occurring, while in reality, ensuring that the library executes within an isolated environment.

Figure 6 highlights the actual process that takes place when a function call occurs.

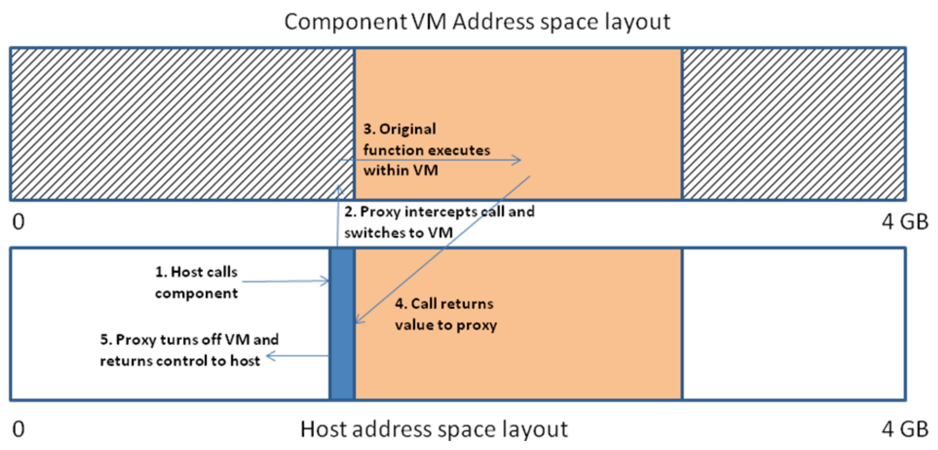
Component VM Address space layout
3. Original function executes within VM
0
4 GB
2. Proxy intercepts call and switches to VM
1. Host calls component
4. Call returns value to proxy
5. Proxy turns off VM and returns control to host
0
Host address space layout
4 GB

Figure 6 - 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 6, whereas the host can access any area. In this way, passing pointers back and forth can be done without any pointer swizzling 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 utilizes 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 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, and the default system calling convention is assumed. This function is binary patched to implicitly pass the *libvm_context* as well as the guest function pointer. (If calling conventions are different, it may be necessary to manually adjust the guest stack or write a wrapper library which does this, via LibVM provided functions.)

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 virtualized 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 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.

It should be noted that in the process outlined above, there are a total of 2 context switches for a single function call; a switch from a host application to the OS kernel, in which the KVM driver executes the guest, and a return context switch back to the host on completion. This compares favorably with a remote procedure call to another process, which would double the number of context switches, as well as incurring the overheads of any cache invalidation due to process switching.

5 PTRACE-BASED SYSTEM CALL INTER-POSITIONING

The approach in Section 4.1 relies on hardware virtualization support. However, in the absence of such support, we provide an alternative implementation which uses *ptrace*-based system call jailing, which can nevertheless be used to create a virtualized sandbox that meets the requirements detailed in Section 2.2. System call interpositioning is well documented [16, 32, 33] and our implementation is similar to most mainstream implementations, with the key difference being that we use it towards the goal of isolating components, not processes. A similar approach has been adopted in an earlier software-only sandboxing implementation called CodeJail [13], and we compare and contrast that approach with ours in this section.

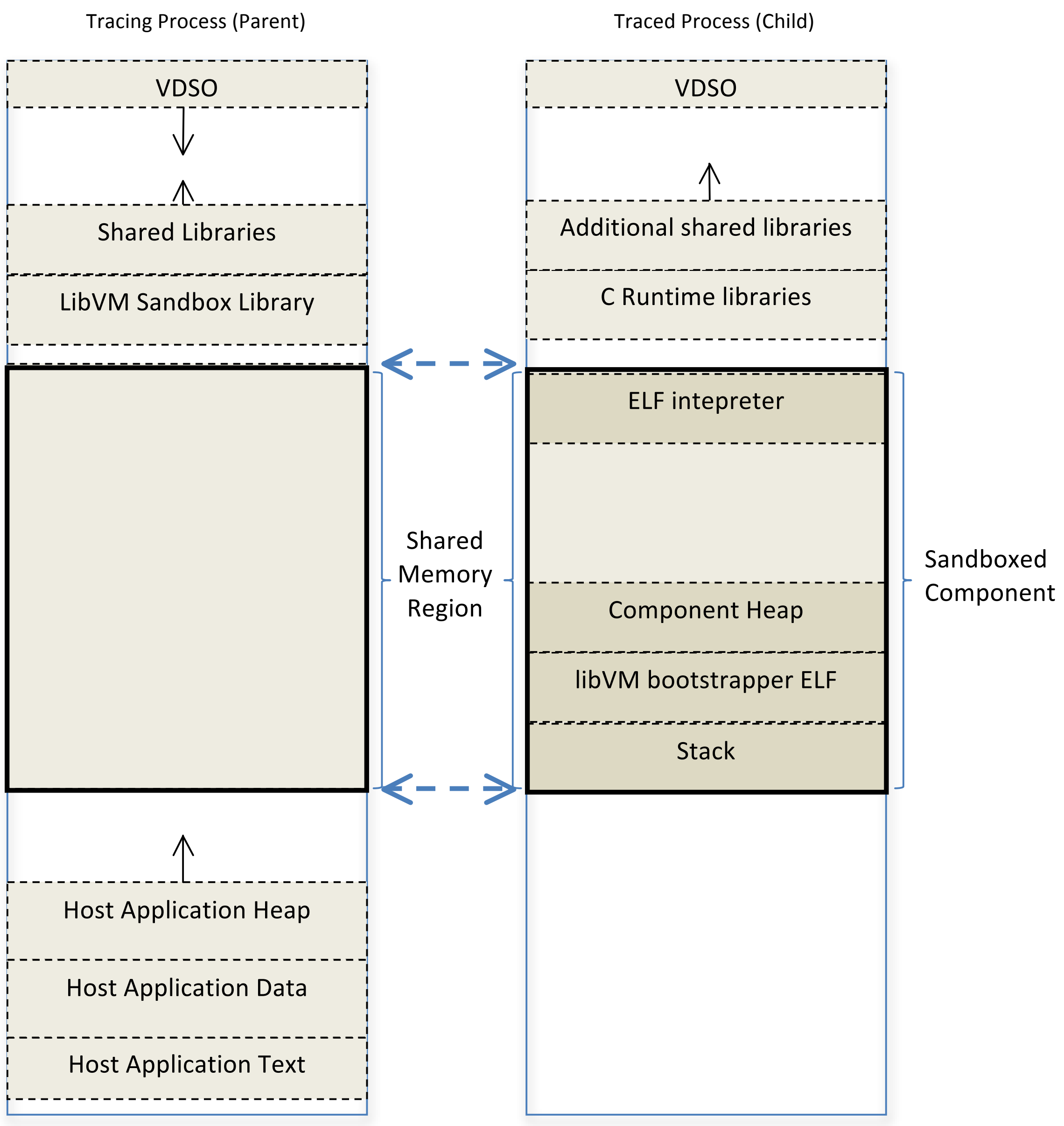

Figure 7 - LibVM *ptrace* based implementation - process layout

The basic steps in the software implementation are almost identical to the steps in the hardware implementation outlined in Section 4, except that instead of using a virtual machine as the isolation container, we utilize a standard system process as shown in Figure 7. Therefore, we load the ELF interpreter and bootstrapper as usual into a reserved memory area, but instead of creating a virtual machine, we fork off a child process, which serves as the isolation domain. The parent process also immediately attaches to the child process using the *ptrace* system call, and intercepts all the child's system calls, allowing the parent to completely isolate the child's actions. A shared memory region is established to enable communication, with both regions being mapped to identical virtual addresses, preserving address transparency.

The spawned child then proceeds to clean up its address space, thus ensuring that an isolated library cannot see any of the containing application's memory, preventing information leakage, and follows the same bootstrap process outlined in the previous section. Much of the system call interpositioning layer is also shared between the two implementations, except to account for differences mainly in memory management. All calls to *mmap* and any calls to open file descriptors must be duplicated in both the child process and the parent process, so that the two are kept in lockstep, which adds to implementation complexity over the virtualization based mechanism.

Inevitably, this implementation incurs significant overheads due to the number of context switches required and does not allow for as much control as the hardware virtualization based implementation, but it has the advantage of being hardware independent (on operating systems that support a *ptrace*-like mechanism).

This software-based implementation of the LibVM framework has much in common with CodeJail [13] which also uses *ptrace*-based mechanisms to achieve similar goals. The major difference between CodeJail and LibVM is that the LibVM framework defines an implementation independent container, with the *ptrace*-based mechanism being just one particular implementation. Codejail exclusively performs system call interpositioning through a *ptrace*-based mechanism (Codejail relies on the *etrace* library [14], which in turn relies on *ptrace*).

There are also slight variations in the two approaches which have interesting ramifications. The LibVM framework focusses on complete isolation of the untrusted library, with no ability for the untrusted library to view its parent's address space, unless specifically granted. This is ensured even in our *ptrace* based implementation, since LibVM makes sure all pages belonging to the parent are no longer mapped before loading the jailed library. By contrast, in Codejail the parent executable continues to reside in memory as a natural side-effect of a fork, and parent pages continue to be shared, although only in a copy-on-write format [13]. Codejail requires the parent and the jailed child to explicitly synchronize memory when needed. Since the child continues to see its parent's memory, it results in information leakage from parent to child, allowing the jailed library to steal information from its parent in a worst case, or glean sufficient clues to target attacks against it. Therefore, a fully separated model, as in LibVM, provides greater security.

In many other respects, however, the problems Codejail has to solve using *ptrace* are similar to those addressed by our LibVM *ptrace* implementation, such as keeping memory maps consistent, and duplicating file handles across processes, both of which are simpler and faster in our virtualization based approach.

6 EVALUATION

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. Finally, a memory benchmark was performed to determine LibVM's overheads. Both the hardware-based and software-based implementations of LibVM were evaluated.

6.1 Micro benchmarks

We carried out 4 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 basic validation of our implementation by simulating 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 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**	
	Time (secs)	**Ratio**	**Time (secs)**	**Ratio**	**Time (secs)**	**Ratio**	**Time (secs)**	**Ratio**
Null call	0.03	1	27.41	926	147.68	4989	146.88	4962
Fibonacci	21.11	1	21.9	1.04	22.85	1.08	22	1.04
Get-PID	0.54	1	47.58	88.12	278	516	150	278.2
File copy	66.1	1	66.3	1	66.8	1	66.6	1

Table 1: Micro-benchmark results – Core i5

Sample	**Linux Executable**		**LibVM – Hardware Virtualization**		**LibVM – *ptrace* Jail**		**Linux RPC**	
	Time (secs)	**Ratio**	**Time (secs)**	**Ratio**	**Time (secs)**	**Ratio**	**Time (secs)**	**Ratio**
Null call	0.02	1	25.92	1296	110.36	5518	106.08	5304
Fibonacci	22.1	1	23.6	1.07	23.7	1.07	24.25	1.1
Get-PID	0.56	1	51.12	91.28	222.5	397.37	179.3	320.26
File copy	65.1	1	66.4	1.02	67.7	1.04	66.4	1.02

Table 2: Micro-benchmark results – Core i7

Table 1 and Table 2 summarize 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, and therefore represent the slowdown factor. For example, the null call is 1296 times slower than a basic Linux executable in LibVM, but far better than the 5304 times slowdown in the RPC case.

In the null call measurement, which simply measures the overhead of transitioning in and out of the isolation container, we found that an RPC is 3 orders of magnitude slower than a local procedure call (no isolation), as is consistent with figures reported in the literature [34]. However, LibVM is about 5 times faster than an RPC, demonstrating a significant, but expected, boost in performance. The main reason for the 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 version 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, as context switching overheads pale into insignificance.

While these extreme kinds of computations 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.

6.2 Macro benchmarks

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

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

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

The benchmarks are compared against their raw execution times.

Sample	**Linux Executable (seconds)**	**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 3: Macro-benchmark results

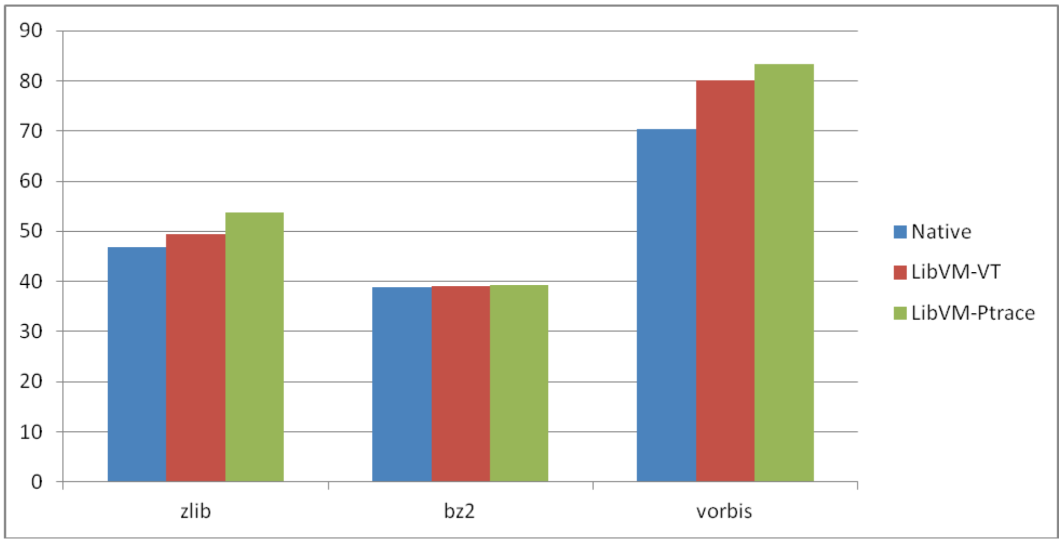

Figure 8 - Comparison of native, software-based and hardware-based implementations – Core i5

Our results in Table 3 and Figure 8 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 our expectations, as the VT hardware adds almost no overheads for normal execution of instructions. However, each domain transition/system call is intercepted by LibVM, which proxies it on the caller's 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' 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 (seconds)**	**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 4: Macro-benchmark results – Core i7

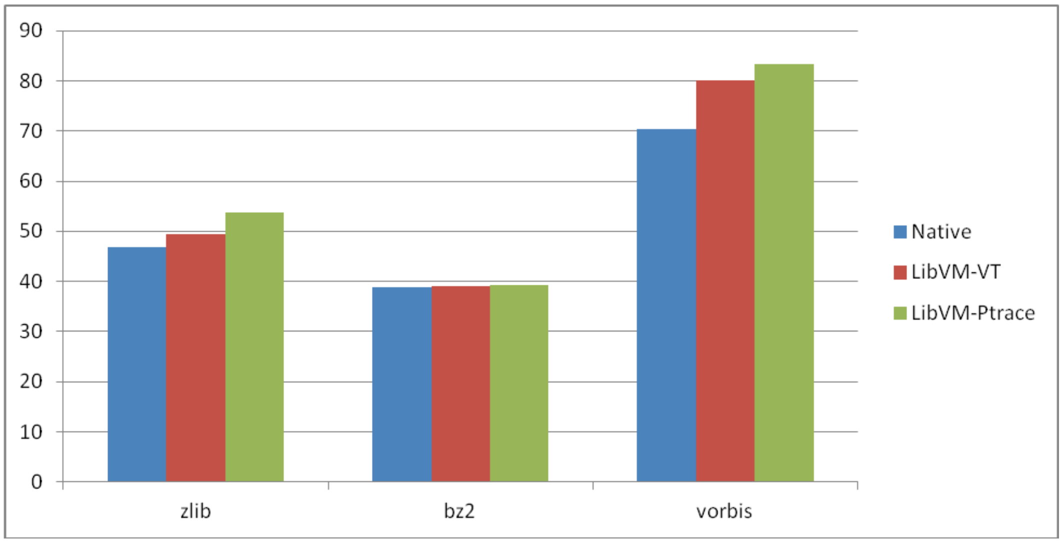

Figure 9 - Comparison of native, software-based and hardware-based implementations – Core i7

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

6.3 Memory Usage

As shown in Figure 2Figure 1, and briefly discussed in Section 2, the *potential* sources of memory overheads are as follows.

1. The LibVM library itself
2. The memory area reserved for the LibVM container
3. The LibVM bootstrapper executable
4. The additional heap within the container
5. The extra stack within the container
6. The additional C runtime within the container
7. The Virtual Dynamic Shared Object (VDSO)
8. The ELF Interpreter/Dynamic Linker

Out of these, the reserved memory area does not actually consume any physical memory, although it does render that address range unusable to the containing application. Similarly, many of the libraries loaded within the container, such as the C runtime, the VDSO and the dynamic linker, will share the bulk of their memory requirements with the rest of the system due to the fact that the Operating System shares library code segments across applications, and therefore, only the data segment of each library contributes to the actual overheads per LibVM container. The LibVM library itself exacts a fixed memory cost per LibVM application, while the bootstrapper executable exacts a fixed cost from each LibVM container. The heap and stack additions also exact a fixed cost at container initialization, but afterwards cause the same overheads as when used within a normal application. Therefore, the best way to determine the cost of LibVM is to calculate this fixed overhead caused by the container, in comparison to a vanilla executable.

We used the linux *pmap* command to determine these overheads. We created a simple stub application which initialises an empty LibVM container and compared it with a similarly compiled application without LibVM.

	Linux Executable			LibVM - Hardware Virtualization			LibVM-ptrace Jail					
	Writable Private	Read-only Private	Total (kb)	Writable Private	Read-only Private	Total (kb)	Writable Private		Read-only Private		Total (kb)	
							Parent	Sandbox	Parent	Sandbox	Parent	Sandbox
Empty Container	172	1584	1756	1904	1628	3532	1920	320	1624	1596	3544	1916
Container + Library	440	1624	2064	3104	1628	4732	2232	456	1624	1636	3856	2092
Container + Two Libraries	572	1664	2236	3276	1628	4904	2540	588	1624	1676	4164	2264

Table 5: Memory usage overheads associated with LibVM in kilobytes

Our experiments produced private memory usage as shown in Table 5. It reveals that an executable which contains an empty, hardware virtualization based LibVM container, compared with a plain linux executable which does nothing and simply returns 0, requires 1776 kb of extra memory. In the LibVM-ptrace implementation, we report two figures – one for the parent process's private memory usage, and one for the child/sandboxed process's memory usage. When measuring these two separately, we discovered a memory leak in the parent process, while the sandboxed process's memory increases were exactly as expected. Due to the presence of two forked processes, there is a higher overall overhead for the ptrace implementation (although a more highly calibrated implementation may be able to lower this requirement further).

After loading one library, the required memory increases to 4732 kb in the case of hardware virtualization. In comparison to a native linux executable which loads the same library, there is an overhead of 892 kb. This is partly due to the initialization of additional memory for the trampoline regions, additional heap memory usage by LibVM, and copy-on-write initialization of C runtime related pages. However, after the first library, it should be noted that the differences for additional libraries remain the same for the linux executable, the hardware virtualization based implementation and the ptrace based implementation and in this particular case, it is 172KB.

7 RELATED WORK

In a previous publication we reviewed a wide range of isolation techniques [7]. Here we focus on technical approaches close to our own.

As we have already seen, a basic facility provided by many UNIX based operating systems is *ptrace* based system call jailing [16]. This mechanism allows one process to monitor the system calls made by another process, and optionally, control the execution of that process. The mechanism is typically used by debuggers and has also been used to provide system call inter-positioning layers for whole processes. However, to our knowledge, this mechanism has not previously been used to provide isolation for individual shared libraries, which is an avenue we have explored in this paper, in the form of an alternative software-based isolation container that can be used when hardware virtualization support is absent.

Another key technique in binary code level isolation is Software Fault Isolation (SFI). This method was first described by Wahbe et al. [9] and has been applied in many forms. It allows untrusted code to be placed in the same operating system (OS) process as trusted code and avoids the overhead of Inter-process Communication (IPC) between processes. It uses software based static analysis of the untrusted component's object code to verify that no illegal memory accesses will be made and to inject code for double checking any potentially harmful instructions, effectively sandboxing the original component. The sandboxed code is generated in a way that ensures that the high bits of a memory address always fall within the sandboxed region, thus preventing components from accessing memory outside of their bounds [9]. Wahbe et al.'s [9] original idea has been improved and implemented in many forms. SFI, originally demonstrated by Wahbe et al. [9] on a Reduced Instruction Set Computer (RISC) architecture, has also been demonstrated on Complex Instruction Set Computer (CISC) architectures [12]. Techniques such as binary translation [20] are offshoots of the SFI concept.

However, a significant weakness of the SFI approach is that ensuring the correctness of the implementation is a difficult process. As Wahbe et al. [9] point out, modification of the executable binary is complicated and adds significant overhead to the code injection process because, for example, the difference between code and data can be difficult to identify. Therefore, safe execution of arbitrary binary components is difficult using SFI if the program was not compiled using an approved compiler.

The ideas in SFI are directly utilized in Google's Native Client (NaCl) system, which provides a software framework for safe execution of untrusted binary components [10]. NaCl aims to provide browser-based applications increased computational performance through native binary components which have access to performance-oriented features such as SSE instructions, compiler intrinsics, hand-coded assembler, etc., without compromising on safety [10]. In a previous paper [15], we compared Google's Native Client, as well as Vx32 [11], a similar SFI based system, with a hardware virtualization based jailing system. We concluded that both Native Client and Vx32 imposed restrictions in terms of what instructions could be safely executed, as well as required custom tool-chains and increased complexity due to their software based instruction verifiers.

Another component isolation technique is a multi-process application architecture. This model is becoming increasingly popular in web browsers [35, 36]. The basic idea is to isolate individual components in 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 [37]. 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 [2]. There is however, an increase in complexity as coordination between several processes is required. Also, Wahbe et al. [9] 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-switching overheads [9], so a trade off is made between performance and reliability [37].

Chiueh et al. [38] introduce an intra-address space component isolation scheme by using the paging and segmentation support in the Intel x86 hardware architecture, the most prevalent architecture for desktop machines. This support is used to isolate kernel extensions from the kernel itself, by placing all extensions in a separate segment of lower privilege than the kernel. They demonstrate that hardware solutions can provide high efficiency, although their technique is limited to isolating the application from all components; components themselves are not isolated from each other. Furthermore, this technique is designed for isolating trusted components from accidental attempts to violate their memory boundaries, and is not intended to isolate deliberately malicious components.

A somewhat similar approach is an application level library for isolating components using x86 segmentation hardware [11]. This approach is unique in that the entire library is implemented in user-mode, requiring no changes to the OS kernel. Google's Native Client also utilizes the above hardware segmentation technique for isolating components.

However, all of the above solutions offer only a partial remedy, as they require that applications be linked against custom C language runtimes and do not offer the benefits of a single address space, or offer reduced isolation guarantees.

Codejail [13] is the solution which is closest to our own, in that its defined goals and aims are similar, and its implementation has much in common with our ptrace based implementation. However, Codejail does not introduce an implementation independent API and container for the purpose, as LibVM does, and it is affected by the same performance issues and implementation complexities as our *ptrace*-based implementation, which are not present in our virtualization based implementation. On the other hand, Codejail's designers have emphasized hooking library loading as early as possible during program initialization, allowing more transparent library wrapping for complex implementations; while this is also possible in LibVM, we have not focussed on this issue. We provide a more in-depth technical comparison between our approach and Codejail in Section 5 when discussing our *ptrace*-based implementation.

8 CONCLUSION

This paper has described LibVM, a system for isolating shared libraries in separate isolation domains and interpositioning all system calls, while requiring no modifications to the shared libraries themselves, and retaining the ability to share pointers between address spaces, with acceptable performance overheads. We 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 software implementation for use in the absence of suitable 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. 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 software based mechanism offers more limited options. However, such limitations could be remedied with kernel modifications.

This system provides an attractive alternative to existing solutions because it requires less porting effort due to its similarity to existing POSIX interfaces, and it enables the possibility of a more natural programming metaphor due to pointer transparency (an avenue for further research) while maintaining comparable performance when hardware virtualization support is available, and good isolation guarantees even in its absence.

9 REFERENCES

[1] 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.

[2] 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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in *USENIX Annual Technical Conference*, Boston, MA, 2008, pp. 293–306.

[12] 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.

[13] Y. Wu, S. Sathyanarayan, R. C. Yap, and Z. Liang, "Codejail: Application-Transparent Isolation of Libraries with Tight Program Interactions," *Computer Security – ESORICS 2012,* vol. 7459, pp. 859-876, 2012.

[14] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," in *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 1999, pp. 19-34.

[15] 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.

[16] 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.

[17] 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.

[18] J. Petersson. (2005, Accessed: 2011, Jun. 18). *What is linux-gate.so.1?* [Online]. Available: http://www.trilithium.com/johan/2005/08/linux-gate/

[19] Redhat. (2010, Accessed: 2010, Jul. 20). *Kernel Based Virtual Machine* [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual* vol. 1: Basic Architecture: Intel Corporation, 2007.

[27] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* vol. 3B: System Programming Guide: Intel Corporation, 2007.

[28] 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.

[29] AMD. (2008, Accessed: 2009 Jan. 30). *AMD-V™ Nested Paging* [Online]. Available: http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf

[30] Intel. (2008, Accessed: 25th May, 2011). *Intel® Virtualization Technology* [Online]. Available: http://www.intel.com/technology/virtualization/index.htm

[31] J. R. Levine, *Linkers and Loaders*: Morgan Kaufmann Publishers Inc., 1999.

[32] 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.

[33] 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.

[34] 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.

[35] 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/

[36] 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.

[37] 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

[38] 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.