Karlsruher Institut für Technologie

Address-Space Multiplexing revisited on AMD64/x86_64

Studienarbeit
von

Marco Kroll

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dipl.-Inform. Marius Hillenbrand

Bearbeitungszeit: 16. Januar 2013 – 30. April 2013

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Marco Kroll

Karlsruhe

April 30, 2013

Abstract

Address space multiplexing is a technique which enables multiple independent processes to share a single address space. This technique can be used to reduce the context switching time and the dependency on special hardware, such as tagged translation lookaside buffers.

The design consist of modifying the program to achieve isolation. Thereafter the program is loaded into a single address space where the necessary process-related structures are initialized and execution of the subprocess is started.

In this thesis we explore how LLVM intermediate representation (LLVM IR) can be used to achieve isolation of multiplexed processes. Using LLVM IR allows us to be independent of programming language and hardware. Our main focus is the protection of memory access, which we achieve by modifying the intermediate representation of the target program. The modification inserts additional instructions to limit the addressable memory of each process, thereby achieving protection from misbehaving processes. Further protection is realized by using mechanisms provided by the underlying operating system, such as the `clone` system call.

We developed a prototype that is able to load a modified LLVM IR program into its address space and start its execution in a subprocess. Using our prototype we were able to show that the process-related structures are isolated and that memory access can only occur within the assigned memory area, at the cost of performance. The performance impact varies depending on which type of memory access is limited and the total number of memory accesses performed by the target program. The overhead of our prototype for commonly used programs is expected to be between 14% and 50%. The maximum overhead we measured is less than 100%.

Applications can benefit from using a multiplexed address space if the number of context switches is high enough to compensate the performance loss caused by our modification. Future research has to be done to further optimize the instructions used to limit the addressable memory thereby improving isolation and

performance.

Contents

1. Introduction

All major operating systems currently use a dedicated address space per process. Loading each process in a separate address space is advantageous, because otherwise security breaches, such as modification to memory or file descriptors across process boundaries, would be possible.

Running multiple processes in the same address space is called address space multiplexing, see Figure 1.1. Operating systems can benefit from address space multiplexing in cases where a lot of context switches are done or where the creation of a process is more expensive than its execution. A context switch is the process of changing the current executing process. Additionally the technique we explored does not need any support from the hardware (e.g. virtual memory or segmentation).

One major cost of context switches, especially for microkernels [18], comes from invalidating the translation lookaside buffer (TLB), which is responsible for mapping virtual to physical memory. Before 2008 Intel processors used a TLB which had to be invalidated before loading a new process. For this reason tagged TLBs were introduced [17,19, p. 2]. A tagged TLB stores an address space identifier for each mapping, which prevents aliases and allows the TLB to keep its mapping. An alternative to tagged TLBs are multiplexed address spaces as they allow the reuse of all TLB entries regardless of TLB type. Furthermore by using a single address space we allow for an efficient implementation of interprocess communication. Additionally benefits are described in more detail in Section 1.1.

The main issue arising from using a single address space is that processes are able to access any memory location in that address space. This prevents the protection of individual processes from malicious or defective processes.

The solution is to partition the single address space. Each partition is then exclusively used by one process. The protection of each partition can be realized by the use of memory-safe languages [2, p. 24] or by special hardware features, such as segmentation.

We present a solution that relies on translation of intermediate code and just-in-

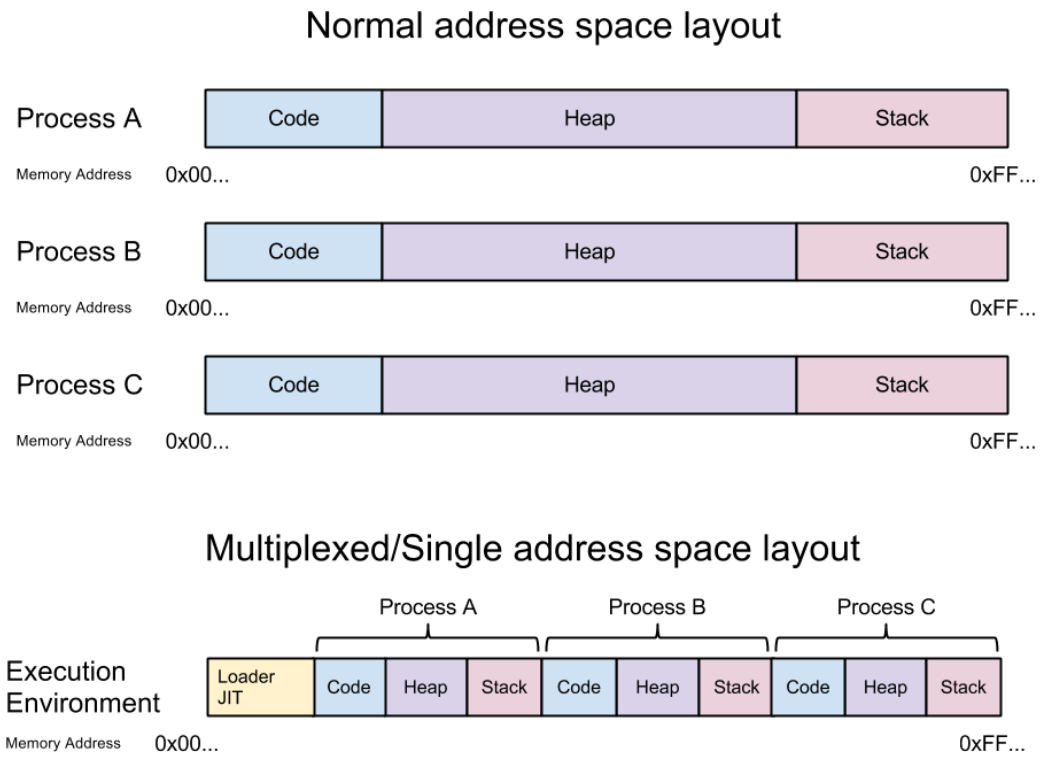

Figure 1.1: Layout comparison of dedicated (top) and multiplexed (bottom) address spaces.

time (JIT) compilation to secure the processes from each other. When a program is started we scan its intermediate representation (IR) for memory access instructions such as `load` and `store`. We then modify the address of these instructions in such a way that the most significant bits are set to an offset. Afterwards we hand the modified IR to our execution environment, which generates the necessary process structures and calculates the offset. Finally the JIT compiles the IR. This approach relies solely on software modifications.

In the next Chapter we elaborate on previous works on single address spaces and process multiplexing and provide some background information. In Chapter 3 we explain our design to create address-space multiplexing. The design consists of three parts, the loading, starting and protection of the process. Chapter 4 describes our prototype. Currently our prototype consists of two separate programs. The first modifies the IR and the second provides the single address space. In Chapter 5 we evaluate our solution with respect to security and performance. We demonstrate that our prototype has an acceptable overhead of less than 50% for

a commonly used program, with room for improvement. Finally, we summarize our conclusions and present ideas for future work in Chapter 6.

1.1 Expected Benefits

In the following we take a closer look on the expected merits of our solution. By using a single address space and an intermediate representation we can exercise more control over the isolation of the process than previous solutions.

Translation Lookaside Buffer One application field are untagged translation lookaside buffers (TLB). By multiplexing processes in a single virtual address space, it would no longer be necessary to flush the TLB entries. This leads to faster context switches and improved execution time for small application (commonly found in microkernels).

Interprocess Communication Another mechanism that we expect to benefit from a multiplexed address space is interprocess communication. Remote procedure calls (RPC) in particular can now be implemented via a normal function call. A possible implementation could map a page with RPC addresses into the subprocess address space. By using a normal function call to invoke a RPC we have a defined entry point into another address space. When switching the address space from the caller to the callee our technique automatically initiates a context switch.

Using the above mentioned implementation to invoke RPC, the resources such as CPU time are still accounted to the caller process, thereby allowing a better accountability of resources.

Kernel Space Our technique uses intermediate representation (IR) code to protect the address spaces and prevent modification of other subprocesses. Before execution we have to compile the IR code into native code. It is conceivable that before the compilation step a filter scans the code for privileged instruction and the use of privileged registers. If the code only contains unprivileged instructions we allow the program to run in kernel space, similar to Singularity [9].

2. Background & Related Work

The idea to allow multiple processes to run within the same address space has been around since 1992. Opal [6] was one of the very first operating systems to implement a single address space, followed by Vino [7], Mungi [8], L4 [12] and Singularity [9].

One of the main reasons for Opal and Mungi to use a single address space is to allow fast interprocess communication, via shared and public accessible memory. Both systems allow controlled access to addresses outside the address space of the process. The necessary protection is realized by using individual page tables for each process. Using a separate page table for each process forces the operating system to flush all TLB entries, if the TLB is untagged. One of our design goals is to avoid these flushes thereby allowing our solution to reduce the context switching time. Furthermore by restricting the addressable memory of each process to its dedicated address space our solution prevents the use of public accessible memory for interprocess communication. Shared pages can still be used.

The single address space implementation of L4, on the other hand, focuses on the context switching time of processes. The approach explored by Liedtke and others uses segmentation [12] to prevent malicious modification of memory. They were able to show that the execution time of certain applications improved by more than 60% [18]. However, there are two main drawbacks. First, this solution is not able to multiplex multiple large address spaces (>3 GiB). Secondly, with the introduction of the 64-bit architecture (AMD64/x86_64) segment register have been reduced in their functionality [10, 3.4.2.1] if used in 64-bit mode.

Singularity uses a single address space to increase the interprocess communication performance and to become independent from hardware realized protection mechanism. Process separation is ensured by only executing programs written in a memory-safe [2] languages (e.g. C#). This approach however prevents the reuse of programs written in a memory-unsafe language such as C/C++ and forces some developers to learn a new programming language.

Our solution is heavily inspired by the Vino [7] operating system. Vino is an ex-

tensible kernel which allows user programs to be loaded into the kernel address space. Memory protection is achieved by using a binary transformation to prevent illegal memory access of user programs, called MiSFIT [16]. This approach allows the use of memory-unsafe languages and does not rely on specialized hardware support. After the transformation MiSFIT optimizes the code and scans the symbol table for the address of each function. The function addresses are later used to verify indirect calls. The solution we have chosen expects the program to be compiled to an intermediate representation. The native code is generated after the transformation.

Another field that uses techniques similar to ours is securing memory access of C code. The CCured [13] type system analyzes C code to make it type safe. C pointers are classified into three categories: *Safe*, *Sequence* and *Dynamic*. Particularly the *Sequence pointers* are very similar to our approach. *Sequence pointers* store the base address, end address and current address. Before each memory access, the current address is cross-checked with the base and end address. If the current address is out of bounds CCured aborts the program. In contrast, our solution is far more coarse-grained. We verify every access of a process with the same base and end address. If the address is not within the specified range we automatically calculate an address which is within bounds. In addition, we abstract from the program language by using an intermediate representation of the code.

2.1 Background

Intermediate representation (IR) code allows programs to be stored in a language neutral and hardware independent way. IR code is mainly used by compilers to perform various analyzations and optimizations. IR code was first introduced with Pascal. One well-known example for IR is Java Bytecode.

Just-in-time compilers (JIT) translate a program just before or sometimes even during its execution into native machine code. JITs were introduced with Lisp and are used to speed up the execution of interpreted code, such as IR code or Java Bytecode.

3. Design

Address space multiplexing allows multiple processes to run within the same address space. Using a single address space allows for faster interprocess communication and reduced context switching time. Three design aspects have to be taken into account, to build a multiplexed address space:

- How to **load a program** into an already existing address space
- How to **start the execution** and provide the necessary resources, and finally
- How to **protect the program** and its resources from accidental or malicious modification from other programs in the same address space

One major goal of our design is to ensure the *isolation* of the processes. Further our design considers the reuse of translation lookaside buffer (TLB) entries and tries to avoid using specialized hardware such as tagged TLBs or segmentation.

We assume that the program was written in a non-memory-safe language. This adds the additional challenge that its instructions could in principle access any memory location.

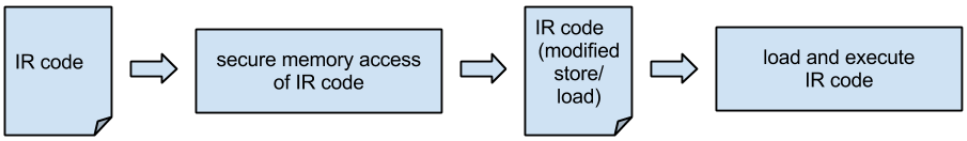

Figure 3.1: Final design. The program is compiled to LLVM IR. Thereafter the instructions that access memory are secured. Finally the IR is loaded into the single address space and executed.

3.1 Overview

To run multiple processes in a single address space we first translate the programs into a relocatable intermediate representation (IR). This enables us to exercise more control over the final compilation into native code and to load the code at any memory location (see Figure 3.1). We use the intermediate representation to set the n most significant bits of each memory address to an offset, thereby limiting the access of the program to a predefined memory area (with the size of $2^{(64-n)}$ bytes). The resolving of the offset has to be done by a modified loader because the memory area of the process is not known until the program has to be loaded into memory for execution. This loader is also used to prepare the memory and process-specific structures and to translate the intermediate representation of the program into native code. During the translation additional care has to be taken to handle indirect function calls as well as system- and library calls that do not consider the special memory layout. Finally, a call to the operating system is done to register a new process and start the execution of the program.

3.2 Code Preparation and Loading

The normal mechanism provided by the operating system (OS) to load a program into an address space is to call the loader which initializes various structures and loads the program code into memory. Two options exist to load a program into a multiplexed address space. Either each program gets always loaded at the same address or the native code can be loaded at arbitrary memory addresses. We have chosen the second approach since the first prevents the parallel execution of the same program. Two techniques are known to create appropriate code.

The first technique is to generate position independent code (PIC). Instead of hard-coded absolute addresses PIC uses relative addresses to access memory [11, p. 51].

The second technique transforms the program into relocatable code. In contrast to position independent code, relocatable code needs to be processed before execution to run at the desired memory location [11, p. 183 ff.]. During this process the program code and data addresses are adjusted with regard to the desired memory location.

Both techniques presented above are still not sufficient for our purpose, because they lack support for memory protection. Therefore, our solution modifies each memory access in such a way that reading and writing can only occur within a fixed memory area to achieve memory protection. Since we need to modify

the code anyway it was a natural choice to use relocatable code as technique to generate position independent code.

The relocatable code is stored in an intermediate representation (IR), to exercise more control over the code and to translate it to native code just before execution. Alternatively, a x86_64 disassembler could be used to scan and modify the code.

If the emitted code shall be reused it must be digitally signed, otherwise the original code must be processed before each execution. This requirement is necessary to prevent malicious modifications to the executed code. If the code is not verified before execution the protection of the memory can not be guaranteed, because the code could have subsequently been modified.

3.3 Program Execution

Various abstractions to execute program code are known, most noteable the process. Since a real process possesses an own address space it is not possible to run multiple processes within an existing process.

To circumvent this limitation and execute multiple tasks in parallel within a process threads are used. They exist within the same address space and can be used to execute code at arbitrary addresses. Unfortunately, they lack support for an own signal and file descriptor table as well as provide no protection from other threads.

A solution for this problem is a hybrid of a process and a thread. This hybrid would own the process-relevant structures without the need for an address space. We name this entity *subprocess*. Multithreaded programs are supported, if the OS permits threads to be associated with their corresponding subprocess. We call such threads *subthreads*.

If the OS does not provide a way to create subprocesses or subthreads an alternative would be to use normal threads. However, this workaround requires the replacement of system calls that interact with process-specific structures (such as `open`/`read`/`kill`) to emulate the correct behaviour. We do not explore this solution, because the platform used to create the prototype supported the creation of subprocesses and -threads.

3.4 Process Protection

As we already stated, protection is a core element of our design. We have to protect access to memory and to process-specific resources such as files from

other subprocesses. Our design allows us to load all processes in the same virtual address space, thereby allowing the reuse of the TLB entries.

3.4.1 Memory

As stated previously we force memory access into certain bounds by modifying the instructions. The technique presented here was successful deployed by the MiSFIT tool [16]. The 64-bit address of each `load` and `store` operation is changed in such a way that access is restricted to a certain area within memory. We achieve this by inserting additional instructions that first truncate the address to x bits (keeping the least significant bits) and then set the most significant $64 - x$ bits of the truncated address, by adding an offset. The modified address is then used in the `load`/`store` operation. This forces the subprocess to only access memory from its own address space and preventing attempts to access memory outside its boundaries, as illustrated in Figure 3.2. The offset is related to the location of the program within the multiplexed memory. Using this technique the usable address space of the program is 2^x bytes large. We call the address space of a subprocess *subaddress space*.

This modification has the advantage that no additional registers or (outdated) hardware support is needed and that the address space size can be adjusted dynamically. Since the transformation only operates on the native or intermediate representation (IR) of the code, non-memory-safe languages such as C/C++ can be used without modifications, which is another positive aspect of our solution.

Indirect function calls Particular attention should be given to function pointers to ensure that they only reference addresses that are function entry points. This can be achieved by a runtime comparison of the function pointer with all valid function addresses. The runtime overhead can be minimized by using an appropriate data structure, such as an open addressable hash table [16, 4.2].

3.4.2 Process-Specific Structures and System Calls

Since our solution relies on the operating system (OS) to create the subprocesses, most of the protection concerning the process-related structures and system calls is already handled by the OS, with one notable exception. The modification of the memory access bits of a process must be prohibited. Otherwise a program that generates or modifies code within its subaddress space would be able to overcome the access limitations imposed by our modification. For this reason all programs in the multiplexed address space must use the following access rights: read and

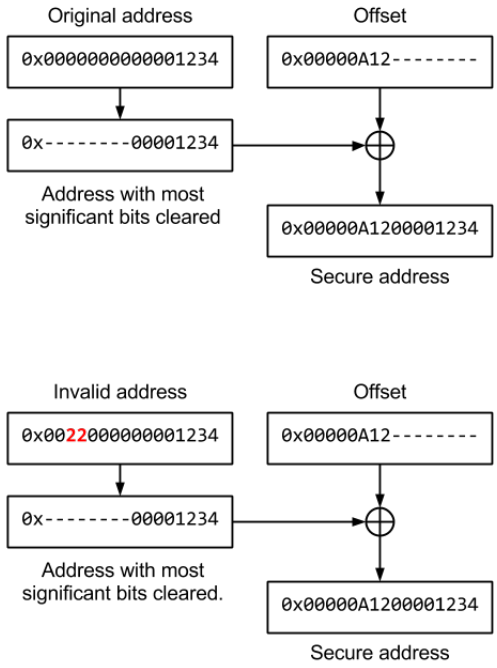

Figure 3.2: Example of address calculation. The first example (top) shows a valid address being truncated and joined with the offset. The second example (bottom) shows how an invalid address is recalculated to be within the bounds of the subaddress space.

execute for code, read and write for heap and stack. Some programs such as just-in-time (JIT) compilers violate this requirement. A JIT compiler for example, first writes code into the heap and then marks that memory as executable. If a program changes the memory permissions to executable we move that process into a dedicated address space. The transformation from a program that runs in a multiplexed address space to one that runs in its own address space can be achieved transparently by creating a new page table for that process in the OS.

3.4.3 External Libraries

Another problem occurs if a call to the system or external library is made by the subprocess. These external resources currently do not consider the usable memory area of the subprocess. This could lead to memory leaks and non-addressable data. This problem can be solved either by forbidding the use of such libraries

and system calls or by using wrappers to make them aware of the single address space.

4. Implementation

Based upon the design in Chapter 3 we implemented a single address space that is able to execute multiple processes. An implementation needs following functionality:

- the **loading of the target program** into an existing address space
- the **protection of the target program**
- the **execution of the target program**

The program loading is achieved by allocating the necessary resources in the single-address-space which is provided by the execution environment. The execution environment assigns a memory area of its own memory as subaddress space to the modified LLVM IR code. Thereafter the LLVM code is compiled into native code. The protection of the target program can be divided into memory protection and protection of process-related structures. The memory protection of the target program is realized by inserting additional instructions that recalculate the address of each memory access. The security of the process-related structures is implemented in the execution environment. It uses the mechanisms provided by the operating system. Finally, to start the execution of the target program the *main* method of the native compiled LLVM IR code is called by the subprocess.

Since our prototype is a proof of concept and the focus of this work is isolation, we have not implemented any special optimizations to improve the performance.

4.1 Representation of the Target Program

To ensure the correct operation of our prototype it is necessary that the target program is already compiled into an intermediate representation (IR), to be precise LLVM IR (Section 3.2). The technique presented here can also be applied to native compiled programs, by using a x86 to LLVM IR disassembler [3]. An additional advantage of using LLVM IR code is that we were able to use the LLVM just-in-time (JIT) compiler to relocate the target program (Section 3.4.1)).

4.2 Tools

We have been using Linux (Ubuntu 12.04 LTS) and the LLVM Compiler Framework (version 3.1) [1] to develop our prototype. We used the `clone` system call and the `pthread` library to create the subprocess. To modify the `load` and `store` instructions we implemented a LLVM FunctionPass. A FunctionPass is a shared library that can be instrumented by LLVM to analyze and transform LLVM IR code. The execution environment is a partial rewrite of the default LLVM interpreter (*lli*).

4.3 Instruction Modifier

The protection of the target program can be divided into two categories. The protection of the memory access and the protection of the process-related structures such as file descriptors.

We first implemented the memory protection of the process. We therefore created a LLVM FunctionPass (*muxpass*) which locates each `store` and `load` instruction in the IR of the target program and replaces it. The inserted instructions recalculate the memory address. First of all the address of the `store` or `load` instruction gets truncated to 32 bits by performing an `and` instruction. Afterwards an offset is added via an `or` instruction, which sets the most significant 32 bits, thereby limiting the addressable memory of the target program to 4 GiB (example in Table 4.1). The offset is stored in a global variable (`program_offset`). This global variable is then inserted into the target program by the FunctionPass. The `program_offset` variable is later set by the execution environment.

Original target program		Modified target program (after *muxpass*)	
		1	%orptr = ptrtoint i32* %1 to i64
		2	%rst = and i64 %orptr10, 4294967295
		3	%offset = load i64* @program_offset
		4	%fix = or i64 %rst, %offset
		5	%ofptr = inttoptr i64 %fix to i32*
1	store i32 0, i32* %1	6	store i32 0, i32* %ofptr

Table 4.1: Left: Unmodified target program. Right: Target program after beeing processed by *muxpass*. `store` instruction was replaced by a version which sets the most significant address bits to `program_offset`. The procedure is identical for `load` instructions.

The *muxpass* FunctionPass is written in C++ and contains only 144 lines of code

including comments. Our prototype allows us to separately switch the modification of `load` and `store` instruction on and off.

4.4 Execution Environment

The execution environment is responsible for loading the target program into its address space, to compile the target program into native code and to start its execution. Additionally, the remaining protection, namely the protection of the process-related structures, is ensured by the execution environment.

The execution environment (*muxlli*) is based upon the LLVM interpreter (*lli*). *lli* is used to execute programs compiled to LLVM IR. Contrary to its name the LLVM interpreter is also able to just-in-time compile the LLVM IR.

muxlli consists of 538 lines of C++ code. Compared to the original LLVM interpreter 233 lines were modified. Most of the modifications result from additional code to create the subprocess.

4.4.1 Overview

Our execution environment (*muxlli*) creates a subprocess with dedicated memory as illustrated in Figure 4.1. The current prototype reserves 4 GiB of memory for each subprocess, where the upper 1 GiB are used by the stack. Next our prototype creates a subprocess, which later executes the modified target program. The subprocess calculates its offset within the multiplexed memory. This offset is later used to ensure that the modified program can only access memory of its own subaddress space. Subsequently process-specific structures are initialized. Finally, the subprocess compiles the modified target program into native code and calls the `main`-method thereby starting the execution of the target program.

Currently, we do not load dynamic libraries into the subaddress space of the process. However, calling a dynamic library works since our prototype is not checking the memory address of function calls. We will cover the isolation of runtime dependent control flow, such as indirect function calls in future work.

4.4.2 Subprocess creation

We use the `clone` system call with `CLONE_VM`, to create a real subprocess with own signal and file descriptor table. The `CLONE_VM` flag ensures that the parent (*muxlli*) and the child (target program) run in the same address space. By using

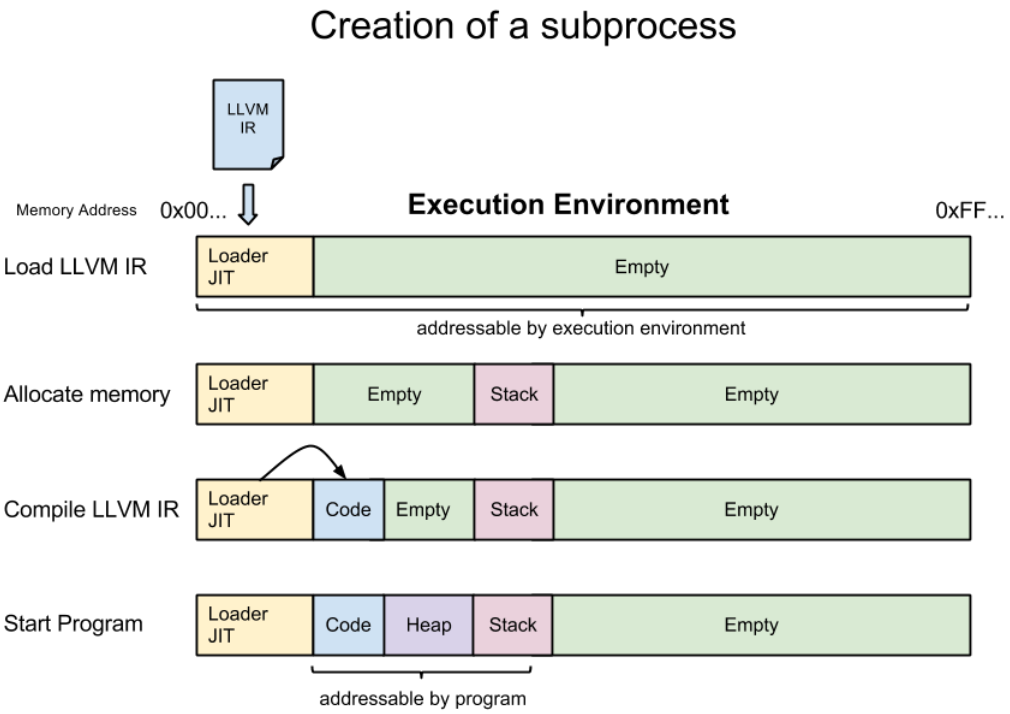

Figure 4.1: Steps to create a subprocess. First the target program LLVM IR code is loaded. Then the execution engine allocates memory and starts a subprocess (therefore the *stack*). Subsequently the subprocess uses the execution engine to compile the LLVM IR and store the native code into its memory. Finally the subprocess executes the compiled code.

the `clone` system call we delegate the protection of the process-related structures to the underlying operating system.

The current prototype does not create an own thread local storage (TLS) which is passed to `clone`. Therefore, immediately after the `clone` we call `pthread_create` from the `pthread`-library to get a proper initialized TLS. The TLS is used to store certain process-specific data, such as the value of `errno`.

After the subprocess is created a default signal handler for *SIGSEGV* is installed, which exits the subprocess if called. This prevents the termination of *muxlli* in case of a segmentation fault in the subprocess. Since *muxlli* and the subprocess share the same address space, Linux delivers the signal to *muxlli* if the subprocess has no segmentation fault handler.

Our prototype does not fully support dynamic libraries, therefore access to *std-*

in/stdout/stderr results in a memory access violation. In our prototype we copy the information of *stdin/stdout/stderr* into the subaddress space and rewrite the memory location during the compilation, to use standard in- and output.

4.4.3 Target Program Compilation

Before the modified target program is compiled the value of `program_offset` is calculated. This offset is stored in the target program and ensures that the subprocess can only address a limited amount of memory.

To force the compilation to emit the native code into a predefined memory area we wrote our own JIT memory manager. Based on the default LLVM memory manager we had to change 138 lines of code. The memory manager does not only enable us to specify the location of the native code, but it also allows us to replace function calls such as `malloc` and `free` with own implementations, which are aware of the subaddress space. For this reason we do not require the target programs to statically link with a modified *libc* which is aware of the multiplexed address space.

The only necessary modification to the LLVM source code was to enable argument passing via stack in the execution engine. This modification was needed because the default implementation uses the heap of the execution environment to store the arguments. Accessing the arguments in the subprocess resulted in an access violation error since the location of the arguments was outside the addressable memory of the subprocess.

4.5 Limitations

Although we limit the addressable memory (subaddress space) of the multiplexed process (subprocess), we currently do not check the address of functions. This allows us to use dynamic libraries outside the subaddress space. However, it also allows the program to make arbitrary calls to functions outside its address space, in a worst-case scenario a malicious program could use a modified function pointer to skip the address modification by directly jumping to the `load`/`store` instruction. This could be prevented if prior to an indirect function call the address is verified by comparing it to all valid function addresses of the program.

5. Evaluation

The evaluation of our approach is divided into two parts: isolation and performance. First we evaluate if our solution provides the same isolation mechanisms as a normal process. We tried to escape from the subaddress space (assigned area within the execution environment) and tried to manipulate process structures of the execution environment, *muxlli*. Thereafter, we show that the performance impact on a commonly used application is below 50%, with room for improvement. We measured the performance impact by comparing the execution time of our modified binaries to native compiled code.

5.1 Test Machine and Environment

All experiments were conducted on a Intel Core i7-2600 (with hyper-threading disabled) with 32 GiB of main memory and a SSD hard drive. We used Ubuntu 12.04 LTS (Kernel 3.2.0-38) as operating system and LLVM version 3.1 as basis for our own implementation. LLVM IR code was generated with clang version 3.0 [1].

5.2 Program Representation

For all our test we pre compiled the C programs to LLVM intermediate representation (LLVM IR) code. We used *clang* to compile each C file into a corresponding LLVM IR file. For the performance tests we also created an executable ELF file with *clang*.

Since our execution environment (*muxlli*) and the default execution environment of LLVM (*lli*) expect a single LLVM IR file, we used the LLVM linker *llvm-link* to link all LLVM IR files into a single one.

[1] compile options: *-S -emit-llvm*

Additionally, the LLVM IR files used by *muxlli* were modified to ensure that each access of a `store` and `load` instruction occurs only within the subaddress space. For our tests we have chosen to limit the addressable memory to 4 GiB.

The transformation was performed by the LLVM optimizer with our additional modification pass called *muxpass*. *muxpass* allows us to separately enable the modification of `store` or `load` instructions.

5.3 Isolation

As previously mentioned isolation is one of our main design goals. We therefore modify the LLVM intermediate representation (IR) code of the program to prevent memory access across subprocess boundaries. A subprocess is a process that we created in our execution environment, *muxlli*, with own file descriptors, memory, and signals. Although each subprocess runs in the same virtual memory modification of other subprocesses has to be prevented. In this section we demonstrate that our prototype isolates the subprocess to such an extent that it can be viewed as a full featured process, if indirect function calls are excluded.

5.3.1 Signals

We experimentally verified that signals sent to the subprocess are not routed to the execution environment. Furthermore we could demonstrate that modifying the signal handler in the subprocess does not affect the way signals are handled in the execution environment *muxlli*. We used the program in Listing 3 (page 34) to perform the tests. The program sets up a signal handler for *SIGUSR1* and then goes to sleep. If the signal handler is triggered a message is printed. Next we manually send a *SIGUSR1* signal to the sleeping process via the *kill* command. *muxlli* was modified to handle the same signal with a different output and to print a message before exiting. We consider the test successful when only the signal handler of the test program is called and the final message from the execution environment is displayed.

All our tests were successful. An example output can be seen below (Listing 5.1).

```
$ muxlli ./signal # kill -SIGSEGV PID expect no output
   from signal
muxlli: Bye
$ muxlli ./signal # kill -SIGUSR1 PID expect signal
   handler output
signal: Signal Handler
```

```
muxlli: Bye
```

Listing 5.1: Test if signals sent to the subprocess affect the execution environment.

5.3.2 File Descriptors

Each subprocess owns a dedicated set of file descriptors. Any modification to these file descriptors must not affect the execution environment. As with the signals we experimentally verified that our subprocess is not able to modify the file descriptors of our execution environment, *muxlli*. The conducted test is very simple (see Listing 4, page 34). We started a subprocess which closes the *stdout* file descriptor. Thereafter, the program prints a messages to *stdout*. The test was considered successful if the message of the subprocess after the close call is not shown and the final message of *muxlli* is displayed, since its *stdout* was not closed.

All performed tests were successful, see Listing 5.1 for a test output.

```
$ muxlli ./io # expect no output of io, but from muxlli
muxlli: Bye
```

Listing 5.2: Test if modifying the filedescriptor affects the execution environment.

5.3.3 Memory

Finally, we tested if the subprocess is able to gain access to the memory of the execution environment (*muxlli*). Our test program (Listing 5, page 34) allocates memory via *mmap* at a predefined location. *muxlli* ensures that this location is unused, so that the call succeeds. Since we did not modify the behaviour of *mmap* we enabled the subprocess to allocate memory outside its subaddress space. Of course *mmap* should not be allowed to handle memory operations outside the current subaddress space. This behaviour can be prevented by instrumenting our memory manager (see Section 4.4.3). Every attempt to use the memory of *mmap* has to fail, for the reason that it is outside its subaddress space. After the allocation, the subprocess tries to write into the memory and prints its content. We further installed a signal handler to show segmentation faults. We expected that the subprocess crashes due to an invalid memory access and the final message of *muxlli* is displayed.

As shown in Listing 5.3 all tests we ran were successful.

```
$ muxlli ./mem # expect segfault in mem
```

```
mem: Segmentation Fault
muxlli: Bye
```

Listing 5.3: Test if subprocess is able to modify memory of the execution environment.

5.4 Performance

We show that the overhead of our solution is less than 100% for the worst case and can be as low as 14% if we relax the isolation requirement. We analysed the performance impact introduced by replacing `load` and `store` instructions by measuring the execution time of three programs, with and without modification. All presented results are the mean of 30 runs. The maximal deviation from the mean was 1.6% in the unoptimized bzip2 run for the default LLVM interpreter. All other deviations were below 0.75%. The first two programs execute a loop wherein a load respectively store to memory is done. We used these programs as micro-benchmarks to evaluate the impact of our modification on a specific instruction (`load`/`store`). Finally, we measured *bzip2*, a commonly used compression program, to evaluate the impact of our modification on the execution time.

All programs are written in C and were precompiled to native code and to LLVM IR with the appropriate modifications. We compared the execution time of the test programs with:

- native compiled ELF binary (*clang*)
- just-in-time compiled LLVM IR executed by the default LLVM interpreter (*lli*[2])
- a just-in-time compiled LLVM IR, where only `store` instructions were modified, executed by our interpreter (*store-muxlli*[3])
- a just-in-time compiled LLVM IR, where `store` and `load` instructions were modified, executed by our interpreter (*muxlli*[4])

Since we observed that using optimizations reduces the number of `load`s and `store`s, we additionally compiled each program with optimizations enabled[5].

[2] version 3.1
[3] same interpreter as *muxlli* only the input file differs
[4] based on *lli* version 3.1
[5] compiler flags: *-O2 -fno-strict-aliasing*

5.4.1 Micro-Benchmarks

Our mechanism to secure the memory access of the subprocesses introduces an overhead during memory access. The overhead results from the replacement of the `load` and the `store` instructions in the LLVM IR. We measured the impact of our solution with two very simple programs. Both programs allocate memory and read from respectively write to that memory, thereby executing a significant number of `load`s and `store`s. The memory is allocated on the heap. We assume that these tests show the worst case behaviour for our modification.

load program

The *load* program (Listing 1, page 33) was used to examine the performance impact of our modification on the `load` instruction.

The program allocates an integer array on the heap. We chose an array size of 1 GiB. The program calculates the sum of the uninitialized array values. The sum is calculated multiple times ($32\times$) to simulate a larger array.

The results are presented in Figure 5.2. For the unoptimized code (red) our results show that the execution time of the full modification only increased by 20% (*muxlli*) and almost no penalty (only 3%) is recognizable for the `store`-only modification (*store-muxlli*) compared to the native compiled code (*clang*). The overhead of the *store-muxlli* can be accounted to the necessary process initialization performed by the execution engine. Although the default LLVM interpreter (*lli*) is faster than *clang*, this can be explained by the additional run-time optimizations performed by *lli*. Another reason for the difference in runtime between *lli*, *store-muxlli* and *muxlli* are the additional inserted instructions.

Using the optimized code (blue) *lli* and *store-muxlli* were almost as fast as *clang* (less than 0.1% overhead). However, the overhead of *muxlli* increased to almost 70%.

One reason for the faster execution of the optimized code is that certain optimizations reduce the number of memory instructions (see Table 5.1) for example by storing values in register instead of memory.

Our findings for the modification of read-only memory instructions are that the `store`-only modification has no impact on the performance and that 70% is the maximum overhead that is to be expected.

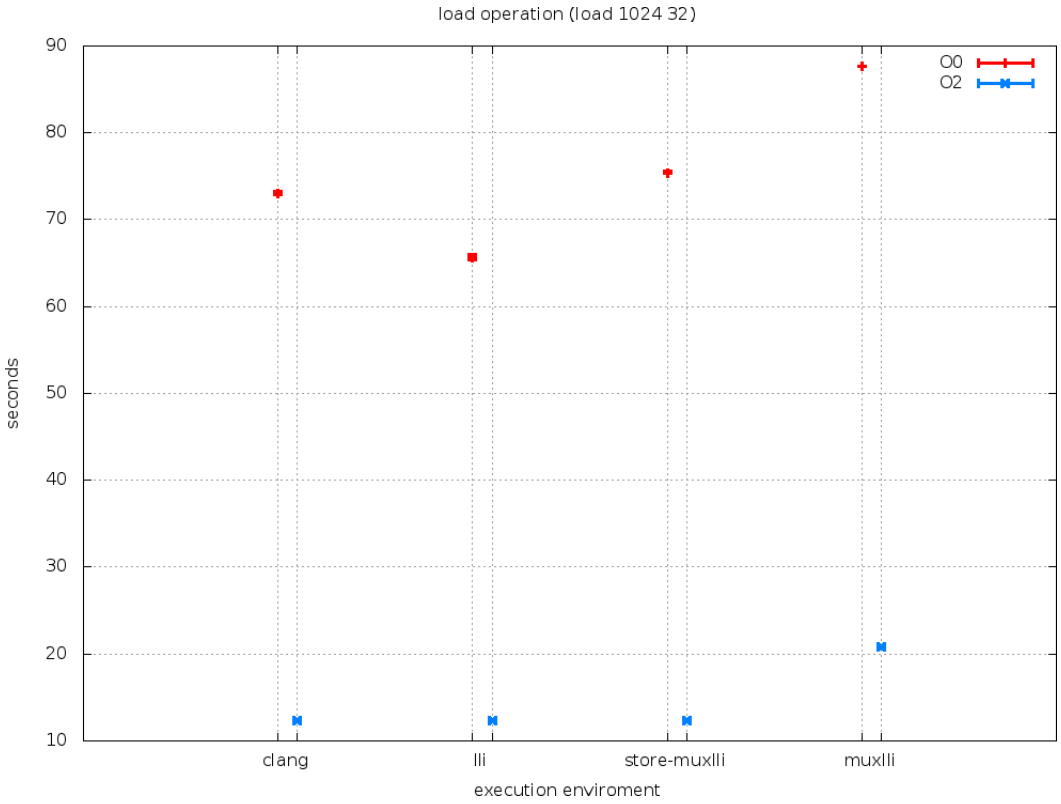

Figure 5.1: Execution times (min/avg/max) for reading 32 GiB of data. Enabling optimizations (blue) reduced the execution time as expected. The default LLVM interpreter (*lli*) performs additional optimizations and is therefore faster than the native compiled code (*clang*). Since only `load` instructions were used the `store`-only modification (*store-muxlli*) performed better than the full modification (*muxlli* uses modified `load`s and `store`s).

instruction	occurrences
`load`	23
`store`	16
`load` (*O2*)	4
`store` (*O2*)	1

Table 5.1: Total number of `load` and `store` instructions in the LLVM IR for the unoptimized and optimized (*O2*) *load*-program

store program

We used the *store* program (Listing 2, page 33) to examine the performance impact of our modification on the `store` instruction.

Similar to the *load* program *store* allocates 1 GiB of integer variables on the heap. *store* then calculates the sum of the uninitialized values. Again, a larger array is simulated by calculating the sum multiple times ($32\times$).

We present our measurements in Figure 5.1. The results for the unoptimized code show that the default LLVM interpreter (*lli*) performs even better (approx. 9%) than the native compiled code (*clang*). Again, the reason are the additional optimizations performed by *lli*. The overhead of the `store`-only and full modification is 6%.

When using the optimized code the overhead drastically increases to more than 95% for *store-muxlli* and *muxlli* compared to *clang*. In connection with the performance difference of the unoptimized code and the measurements from the *load test* the results suggest that optimizations are performed to the unmodified IR code, which cannot be applied to our modified IR code.

Some optimizations are able to reduce the number of memory instructions, without affecting the correctness of the program (see Table 5.2). Reducing the memory access allows the optimized code to execute almost four times faster than the unoptimized code.

Contrary to our expectations the `store`-only modification is inferior to the full modification in this test. We expected *store-muxlli* and *muxlli* to have the same performance. We verified that neither the array size nor the optimizations done by our execution environment caused this behaviour. We further reviewed the C and the modified LLVM IR code. The cause is currently unknown and will be investigated in a future work.

We conclude from the results that the maximum overhead of our solution is less than 100%.

instruction	occurrences
`load`	24
`store`	16
`load` (*O2*)	4
`store` (*O2*)	1

Table 5.2: Total number of `load` and `store` instructions in the LLVM IR for the unoptimized and optimized (*O2*) *store*-program

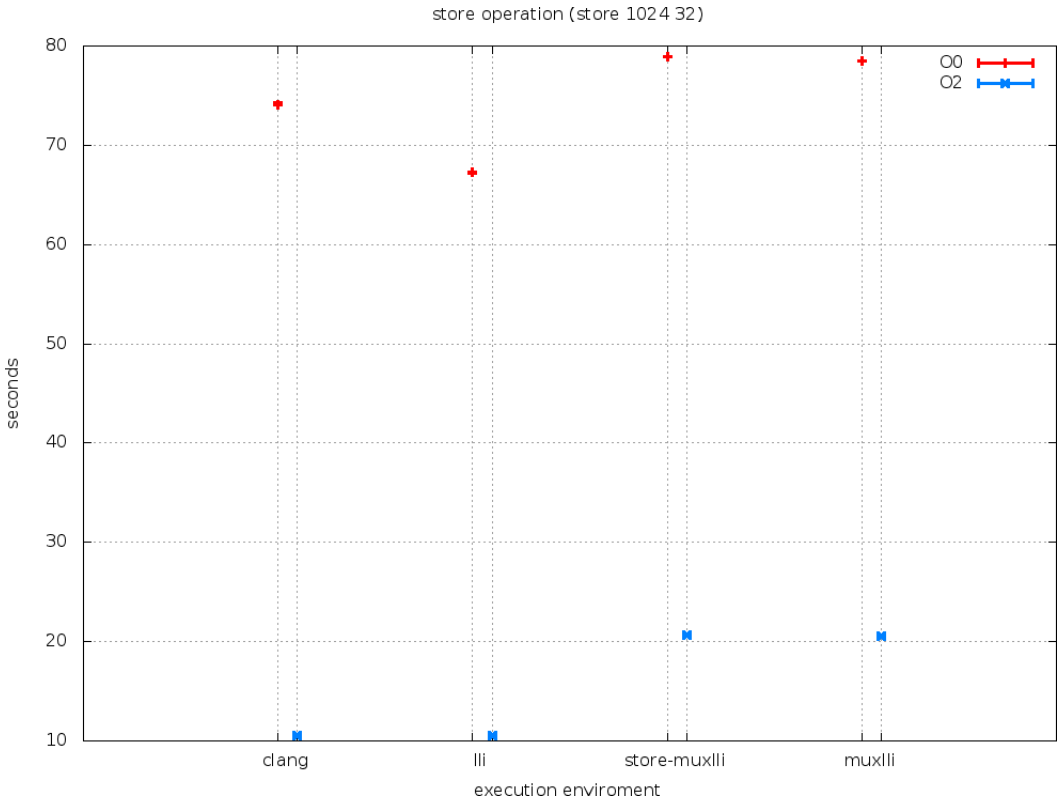

Figure 5.2: Execution times (min/avg/max) for writing 32 GiB of data. Enabling optimizations (blue) reduced the execution time as expected. The default LLVM interpreter (*lli*) performs additional optimizations and is therefore faster than the native compiled code (*clang*). Since almost exclusively `store` instructions were used the store-only modification (*store-muxlli*) and the full modification (*muxlli* uses modified `load`s and `store`s) are on par.

5.4.2 bzip2

We have chosen *bzip2*[6] [5,14], a commonly used compression program, to observe the impact of modifying `store` and `load` instructions on the execution time of a real world program. The code used for testing is the same as provided by the SPEC CPU2006 benchmark[7].

Figure 5.3 shows our results for the bzip2 benchmark.

The measurements clearly show a performance gain from using the optimization (blue). The unoptimized full modified code (red *muxlli*) is five times slower than

[6] version 1.0.3

[7] compiler flags *-DSPEC_CPU -DNDEBUG -DSPEC_CPU_LP64*

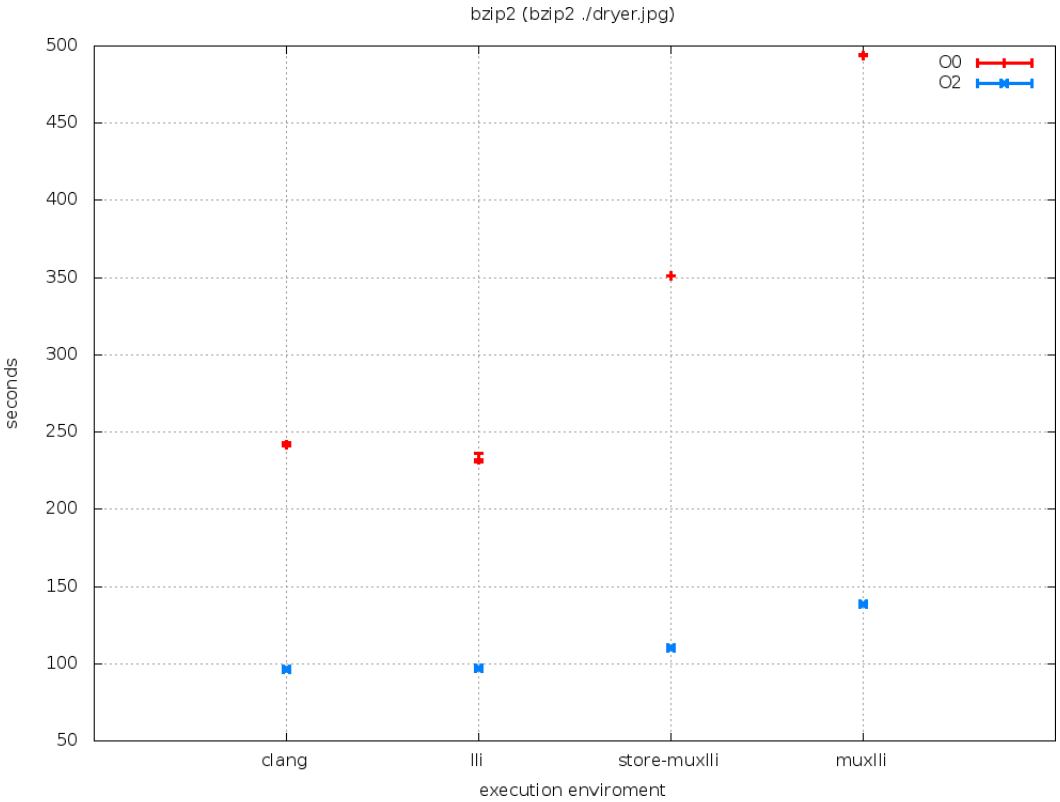

Figure 5.3: Execution times (min/avg/max) of compressing and decompressing 64MiB with bzip2 (from the SPEC CPU2006) with three different block sizes. Enabling optimizations (blue) reduced execution time down to 28%. The default LLVM interpreter (*lli*) performs additional optimizations and is therefore faster in the unoptimized case than the native compiled code (*clang*). As expected the `store`-only modification (*store-muxlli*) performed better than the full modification (*muxlli* uses modified `loads` and `stores`).

the optimized native compiled code and two times slower than the unoptimized native compiled code (*clang*). The performance gap for the full modified code (*muxlli*) compared to the native compiled code (*clang*) can be reduced from more than 100% down to 43% if the code gets optimized. The difference between the unoptimized and optimized code can be traced back to the increased number of necessary instruction modifications and the resulting increase in code size (see Table 5.4). Additionally, the full modified unoptimized code contains 2.75 times more memory instructions than the optimized code (Table 5.3).

Relaxing the isolation requirement to allow arbitrary read access (*store-muxlli*) reduces the performance impact of our solution by 30-50% depending on the optimization level. For the optimized case (red) the overhead is less than 14%

compared to the native compiled code (*clang*).

The measurements for *bzip2* differ from the values of the previous tests. This difference can partially be accounted to the fact that bzip2 is not as memory intensive as the *load* and *store* program.

instruction	occurrences
`load`	8945
`store`	2992
`load` (*O2*)	2615
`store` (*O2*)	1716

Table 5.3: Total number of `load` and `store` instructions in the LLVM IR for the unoptimized and optimized (*O2*) code

instruction modified	lines of LLVM IR	code size increase
none	30771	-
`store`	45698	49%
`store` & `load`	90201	193%
none (*O2*)	19911	-
`store` (*O2*)	28430	43%
`store` & `load` (*O2*)	41259	107%

Table 5.4: Comparison of LLVM IR code lines, after replacing the `store` instruction (`store`) and after replacing `store` and `load` instructions (`store` & `load`). The optimized code is marked with *O2*.

5.5 Summary

We have shown that our modification to the LLVM IR in connection with our execution environment is able to create a full featured and protected subprocess. A subprocess is not able to escape its subaddress space and modify the signal and file handler of another process. Our current prototype is able to run a bzip2 compression with an acceptable overhead of less than 50%. Using a slightly less restrictive modification (securing only the write access) allows us to further reduce the overhead down to 14%. The results from the *load* and *store* tests suggest that the maximum overhead introduced by our approach is less than 100%. This makes our solution practical for everyday use, when performance can be traded for isolation. The overhead of our solution can be compensated if the program makes use of the expected benefits described in Section 1.1.

6. Conclusion

In this paper we explored how to multiplex the address space between several subprocesses on a 64-bit architecture, namely AMD64 and x86_64. A single address space allows the reuse of the page table entries thereby improving interprocess communication and context switches.

Our design focused on isolation. A subprocess (process created within an existing address space) is not permitted to access memory outside its assigned memory area (subaddress space) and should not be able to modify the signal handlers and file descriptors of other subprocesses.

We approached this problem by modifying the compiled program code to limit the memory access to an area determined during the program start. For that purpose the program was compiled to LLVM intermediate representation (IR) code. Before execution we scan the IR for `load` and `store` instructions. We modify each memory address of a `load` and `store` by setting the most significant bits to an offset. The next step is to load the modified IR code into our execution engine. The execution engine calculates and inserts the offset into the IR code. Additionally the execution engine is responsible for creating and initializing the process-related structures such as memory, signals and file descriptor. Finally the IR code is translated into native code and executed.

We used our prototype to evaluate the isolation and performance of our approach. Our prototype was able to withstand illegal memory access and attempts to modify the signal handler and file descriptors by the subprocess. The results of the performance tests were better than anticipated. The *bzip2* executed by our prototype introduced only an overhead of 50% compared to the native compiled program. An optimization which trades isolation for performance by only modifying the `store` instructions (`load`s are ignored), thereby allowing arbitrary memory read access, reduced the overhead with bzip2 to 14% overhead. Thus, we believe that our solution is practical in certain scenarios where a lot of context switches and interprocess communication is done.

6.1 Future Work

Since our solution works on LLVM IR we are not able to prevent return-oriented programming [4,15] attacks within the subprocess. This protection can be achieved by a compiler backend which sets the most significant bits of each return address.

Another topic that has to be considered is performance. Future work should focus on improving the execution time of modified programs. The solution we propose is very similar to using position independent code. The program gets compiled with the 32-bit instruction set, except for memory and control flow instructions. These are compiled using 64-bit instructions. A fixed 64-bit offset register is added to the address of these instructions. By limiting the program to 32-bit instructions and 32-bit register, memory access can only occur within 4 GiB. The 64-bit offset register is set when the program is loaded into the address space and determines the position within the execution environment.

Bibliography

[1] The llvm compiler infrastructure project. `https://llvm.org/`. Accessed: 2012-12-12.

[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 295–308. ACM, 2013.

[4] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[5] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[6] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: a single address space system for 64-bit architecture address space. In *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*, pages 80–85. IEEE, 1992.

[7] Yasuhiro Endo, Margo Seltzer, James Gwertzman, Christopher Small, Keith A Smith, and Diane Tang. Vino: The 1994 fall harvest. 1994.

[8] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, 1998.

[9] Galen Hunt, James Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, et al. An overview of the singularity project. 2005.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. March 2013. `http://download.intel.com/products/processor/manual/253665.pdf`.

[11] John R Levine. Linkers and loaders, morgan-kauffman. Technical report, ISBN 1-55860-496-0.

[12] Jochen Liedtke. Improved address-space switching on pentium processors by transparently multiplexing user address spaces. Technical Report 933, November 1995. `http://l4ka.org/publications/`.

[13] George C Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[14] Julian Seward. bzip2, 1998.

[15] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[16] Christopher Small. A tool for constructing safe extensible c++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, pages 175–184, 1997.

[17] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[18] Volkmar Uhlig, Uwe Dannowski, Espen Skoglund, Andreas Haeberlen, and Gernot Heiser. Performance of address-space multiplexing on the pentium. Interner Bericht 2002-01, Fakultät für Informatik, Universität Karlsruhe, 2002. `http://l4ka.org/publications/`.

[19] AMD64 Virtualization. Secure virtual machine architecture reference manual. *AMD Publication*, (33047), 2005.

Appendix A

Listing 1: load.c

```
#include <stdio.h>
#include <stdlib.h>
const int MB = 1024 * 1024;
int main(int const argc, char const * const argv[]) {
   size_t size = 500;
   int   rep = 1;
   int   sum = 0;
   if(1 < argc) size = atoi(argv[1]);
   if(3 == argc) rep = atoi(argv[2]);
   char *ary = (char *) malloc(size * MB);
   if(NULL == ary) return 1;
   for(int j = 0; j < rep; j++)
      for(unsigned long long i = 0; i < size * MB; i++)
         sum += ary[i];
   printf("output: %d\n", sum);
   free(ary);
   return 0;
}
```

Listing 2: store.c

```
#include <stdio.h>
#include <stdlib.h>
const int MB = 1024 * 1024;
int main(int const argc, char const * const argv[]) {
   size_t size = 500;
   int   rep = 1;
   if(1 < argc) size = atoi(argv[1]);
   if(3 == argc) rep = atoi(argv[2]);
   char *ary = (char *) malloc(size * MB);
   if(NULL == ary) return 1;
```

```
    for(int j = 0; j < rep; j++)
        for(unsigned long long i = 0; i < size * MB; i++)
            ary[i] = i;
    printf("output: %d\n", ary[size * MB - 1]);
    free(ary);
    return 0;
}
```

Listing 3: sig.c

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

const char *prog;
void sighandler(int signo){
    printf("%s: Signal Handler\n", prog);
    exit(0);
}
int main(int const argc, char const * const argv[]) {
    prog = argv[0];
    signal(SIGUSR1, sighandler);
    sleep(5);
    printf("Don't print me\n");
    return 0;
}
```

Listing 4: stio.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int const argc, char const * const argv[]) {
    close(1);
    printf("%s: Hello\n", argv[0]);
    return 0;
}
```

Listing 5: mem.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <signal.h>

const char *prog;
const int FLAGS = MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED;
void sighandler(int signo){
   fprintf(stderr, "%s: Segmentation fault\n", prog);
   exit(EXIT_FAILURE);
}
int main(int const argc, char const * const argv[]) {
   prog = argv[0]; signal(SIGSEGV, sighandler);
   void *l = (void *) 0x110000000ull;
   char *p = (char *) mmap(l, 4096, PROT_READ |
      PROT_WRITE, FLAGS, -1, 0);
   if(l == p) {
      p[0] = 'O'; p[1] = 'K'; p[2] = '\0';
      printf("%s: %p: %s\n", argv[0], p, p);
      munmap(p, 4096);
   } else {
      fprintf(stderr, "%s: Failed to allocate memory at
         %p\n", argv[0], l);
   }
   return 0;
}
```