A Truffle-based Interpreter for x86 Binary Code

Master Thesis
to obtain the academic degree of Diplom-Ingenieur
in the Master’s Program Computer Science
Abstract

Currently, C/C++ programs can be executed in a GraalVM using the LLVM interpreter Sulong. However, for Sulong source code is required, but sometimes only binary executables are available. In order to close this gap, we developed the vmx86 interpreter.

vmx86 is an interpreter for x86_64 binary programs for Linux based on the JIT compiler Graal and the language implementation framework Truffle. The interpreter parses machine code directly from an ELF file, generates Truffle call targets with the help of a heuristic and executes the Graal optimized code. Truffle call targets are required by Graal, and JIT compiled to machine code. The performance has been evaluated on some of the SPEC CPU2006 benchmarks. The benchmarks revealed that vmx86 increases the program execution time by approximately $3 \times$ to $25 \times$ compared to native execution. The completeness is sufficient to execute the GCC backend, and simple X11 programs, like xpdf. By implementing the Truffle Native Function Interface (NFI), other Truffle languages can transparently and safely execute native code by using vmx86. The vmx86 interpreter implements a sandbox, which guarantees that guest programs cannot crash the interpreter, or corrupt the interpreter’s data.

Abstract (Deutsch)


# Contents

1 Introduction ........................................ 6

2 Background Information ............................ 8
  2.1 x86_64 ISA ........................................ 8
  2.1.1 Registers ....................................... 8
  2.1.2 Address Calculation ............................ 10
  2.1.3 SIMD/Vector Extensions: SSE .................. 11
  2.2 Instruction Encoding ............................. 11
  2.2.1 Prefixes ....................................... 11
  2.2.2 ModRM ......................................... 12
  2.3 Linux Syscalls ..................................... 12
  2.4 Virtual Memory ................................... 13
  2.5 ELF .................................................. 13
  2.6 Static vs Dynamic Linking ......................... 14

3 System Overview .................................... 15
  3.1 Graal/Truffle ...................................... 15
  3.2 vmx86 ............................................. 16
  3.3 Truffle Interoperability ........................... 16

4 Implementation ..................................... 17
  4.1 x86_64 Instructions .............................. 17
  4.2 Truffle Frame ..................................... 17
    4.2.1 Registers ..................................... 17
    4.2.2 Flags .......................................... 18
    4.2.3 SSE Registers .................................. 18
  4.3 ELF Loader ......................................... 18
  4.4 Stack Memory Initialization ....................... 19
  4.5 Virtual Memory .................................... 20
    4.5.1 Pure Java Implementation ...................... 21
    4.5.2 Native Memory with JNI ....................... 22
    4.5.3 Native Memory with SystemJava ................ 23
  4.6 Syscalls ........................................... 23
    4.6.1 Virtual Filesystem ............................ 24
    4.6.2 poll ........................................... 25
8 Conclusion / Future Work 46

Acronyms 50


I Introduction

Graal is a new state of the art just in time (JIT) compiler for the Java Virtual Machine (JVM) written in Java [1]. Truffle [2] is a framework for the development of language interpreters. Graal optimizes Truffle language implementations by partially evaluating them during execution, which allows for their efficient execution.

Many language implementations, based on Truffle, exist, such as implementations of Ruby and JavaScript. In order to be able to execute programs written in C/C++ , the interpreter Sulong, which interprets Low Level Virtual Machine (LLVM) bitcode, is available [3]. LLVM bitcode is generated by compilers such as clang, which compile C/C++ to this bitcode. However, source code is not always available, for example, commercial third-party libraries are often available only in a binary format. To execute a program, for which no source code is available, the machine code must be decompiled per static program analysis in LLVM bitcode. The static analysis is not always possible, like for indirect jumps, or dynamically generated code. Dynamically generated code cannot be supported by ahead of time (AOT) compilation.

To give an understanding of the basic concept of x86_64 machine code programs, the following demo program for Linux on x86_64 is discussed in the next chapters:

```
_start: ; open("test.s", O_RDONLY)
   mov eax, 2
   lea rdi, [rip + file]
   xor esi, esi
   syscall
   cmp rax, 0
   jl L0
   ; exit(0)
   xor edi, edi
   jmp L1
L0: ; exit(1)
   mov edi, 1
L1: mov eax, 60
    syscall
    hlt
file: db "test.s", 0
```

Listing 1: Demo program for Linux on x86_64

The program in Listing 1 opens the file test.s and returns a status code 0, in case the file could be opened for reading, otherwise it returns a 1. The interaction with the operating system is managed per syscalls. The first syscall opens the file and is equivalent to the C code open("test.s", O_RDONLY). The second syscall terminates the program using exit. The hlt instruction after the exit(0) syscall is normally not executed, however it is available in most of the programs, in order to force a crash in case exit(0) is not executed correctly for whatever reason.
The goal of this work was to develop an interpreter, which directly executes x86 64 machine code, without translating the binary into an intermediate language, like LLVM bitcode beforehand. The program code is executed in a sandbox, to ensure that the executed program cannot crash the interpreter, or corrupt its data. Since the interpreter is written in Java, it is independent of the operating system and processor of the underlying machine. Due to the fact that no AOT translation into an intermediate language is required, it is also possible to execute programs, which can generate code at runtime.
2 Background Information

Since this interpreter executes Linux/x86_64 programs, it is essential to have basic knowledge about the x86_64 architecture, virtual memory, and Linux syscalls. A general overview is given in this section.

2.1 x86_64 ISA

The x86 architecture was originally developed as a 16-bit architecture by Intel [4], and is a complex instruction set computer (CISC) architecture [5]. Thus, a variety of complex instructions and addressing modes is available, which support high-level language constructs, like loops and array accesses directly in machine code. The architecture has been enhanced with a 32-bit mode and a 64-bit mode, additional registers, and instruction set extensions [6]. Most of the instructions can operate on one or two operands, either of which can reference memory, data that is part of the instruction (immediate operand), or registers. Operands can have different sizes, ranging from 8 to 64 bits.

2.1.1 Registers

Four general purpose registers are available: accumulator (AX), base (BX), counter (CX) and data (DX). Furthermore, special registers, like the base pointer (BP), stack pointer (SP), source index (SI), and destination index (DI) exist. With the 64-bit instruction set extension, eight additional registers (r8 to r15) have been implemented.

Some instructions implicitly use certain registers: string operations typically use the source and destination index to read characters from memory or write into memory whilst arithmetic operations like multiplication and division utilize the AX:DX register pair to handle large numbers or results and remainders.

![Registers on x86_64](image.png)

Figure 1: Registers on x86_64 [7]
All registers can be addressed as 8-bit, 16-bit, 32-bit and 64-bit registers. The following naming conventions apply for all general purpose registers, for example the naming convention for the accumulator is: AL = 8-bit, AX = 16-bit, EAX = 32-bit, RAX = 64-bit. The upper 8 bits of the lower 16 bits of the four general purpose registers can be addressed, e.g. AH, refer to Figure 1. The prefix E represents a 32-bit register, R represents a 64-bit register. Without a prefix, the register name refers to a 16-bit register. The L suffix represents the lower 8 bits, the H suffix represents the upper 8 bits of the lower 16 bits. These suffixes only exist for the four general purpose registers.

Addressing only 8 bits or 16 bits in a 32-bit register results in a drastic performance loss. In a superscalar processor a number of instructions can be executed simultaneously. In the case of a 8bit or 16bit register access, the processor must wait for all previous instructions that write into the same register. This relates to a performance loss. In the 64-bit mode, additional 32-bit are added to the existing 32-bit registers. During a write access to the lower 32-bit, the upper 32-bit are cleared, which eliminates the data dependencies to the previous instructions, and results in a performance increase, compared to the 8/16-bit case. The amount of parallel executions depends on the processor type.

An additional flag register exists (rflags) which contains the following information regarding the last operation and system level information:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry flag</td>
</tr>
<tr>
<td>PF</td>
<td>Parity flag</td>
</tr>
<tr>
<td>AF</td>
<td>Adjust flag</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign flag</td>
</tr>
<tr>
<td>TF</td>
<td>Trap flag</td>
</tr>
<tr>
<td>IF</td>
<td>Interrupt enable flag</td>
</tr>
<tr>
<td>DF</td>
<td>Direction flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>IOPL</td>
<td>I/O privilege level</td>
</tr>
<tr>
<td>NT</td>
<td>Nested task flag</td>
</tr>
<tr>
<td>RF</td>
<td>Resume flag</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual 8086 mode flag</td>
</tr>
<tr>
<td>AC</td>
<td>Alignment check</td>
</tr>
<tr>
<td>VIF</td>
<td>Virtual interrupt flag</td>
</tr>
<tr>
<td>VIP</td>
<td>Virtual interrupt pending</td>
</tr>
<tr>
<td>ID</td>
<td>CPUID instruction available</td>
</tr>
</tbody>
</table>

Out of the above mentioned flags, only the condition flags (CF, PF, AF, ZF, SF, OF), the direction flag (DF) and ID flags are relevant for normal programs, since the other flags are system level flags which are under control of the operating system. The important flags are described as follows:

- **carry flag:** the carry flag is mainly used by additions, subtractions, and rotations in order to indicate a carry or borrow. This is typically used to check, if a sum fits within the destination register. Another use case is an addition with operands longer than 64 bits. Using the carry flag and the *add with carry* operation, longer operands can be processed in multiple steps efficiently.
vmx86: A Truffle-based Interpreter for x86 Binary Code

- **parity flag**: indicates the parity of the lower 8 bits of the result of the last operation. The parity is an *exclusive or* of the lower 8 bits. It is rarely used in practice.
- **adjust flag**: is used for binary-coded decimal (BCD) arithmetic, it indicates a carry or borrow for the least significant nibble. Since BCD arithmetic is usually not in use anymore, this flag is also unused most of the time.
- **zero flag**: is set if the last result is zero.
- **sign flag**: indicates if the previous result is negative.
- **direction flag**: indicates the direction of string operations. If the flag is cleared, string operations are performed from the lowest to highest address, if it is set, string operations are performed from the highest address to lowest address.
- **overflow flag**: is set if the last result did not fit into the register, which indicates an overflow.
- **ID flag**: if software can write into this flag and read the written value, the *CPUID* instruction is supported, otherwise not.

All condition flags can be used by conditional jumps and conditional moves. Flags are always computed, regardless whether they are subsequently used.

x86 _64_ offers a variety of further registers that can control a number of attributes of the program execution and processor control, which are used by the operating system. The instruction pointer (RIP) can be manipulated via control flow instructions. Model-specific registers (MSRs) and control registers (CRs) are also part of this family. For the execution of user space programs, most of these registers are not of interest, since they are mainly controlled by the operating system and a standard process has no access to those registers. Exceptions are registers for performance measurement, provided the process has the necessary privileges.

### 2.1.2 Address Calculation

The address calculation for memory accesses is very powerful: from an offset, a base, an index and a scale, the effective virtual address is calculated. Offset and scale are immediate operands, base and index are registers. The effective address is calculated as shown in equation (1) below.

\[
EA = offset + base + index \cdot scale
\]

*EA* is the effective address and *scale* can only have a value of 1, 2, 4 or 8. Base and offset are suited to access structures; index and scale are mainly used for array accesses. If an address-calculation element is not necessary, it can be omitted in the instruction in order to save memory.

The address calculation is so flexible it can be executed via one explicit instruction (*LEA* – Load Effective Address). The arithmetic can be used for other calculations as well, not necessarily an address. For example, *LEA* can be used also as an addition with three operands, which does not update the flags. Certain multiplications can be encoded with *LEA* as well.
2.1.3 SIMD/Vector Extensions: SSE

x86-64 processors support an instruction set extension, which is named Streaming SIMD Extensions (SSE), and later processor models also support Advanced Vector Extensions (AVX). SSE implements 16 vector registers with 128-bit width, which can be used for floating point numbers and integers. Typical SSE operations execute an operation on multiple packed values in parallel. AVX introduces additional 16 vector registers, and extends the width of all vector registers to 256 bits (or 512 bits for AVX-512). Additionally, AVX introduces a new instruction encoding as well as new instructions.

Handwritten SSE code is typically used by implementations of multimedia codecs, or for acceleration of matrix operations in machine learning, and simulation software. Compilers, however, generate SSE instructions to implement floating point computations, since SSE supports the standard IEEE754 floating point format with 32-bit and 64-bit width, and SSE is faster than the traditional x87 floating-point unit (FPU).

The vmx86 interpreter partially supports SSE, thus can execute float operations, and certain multimedia codecs, like JPEG and PNG encoders/decoders, as well as other multimedia related libraries like libavcodec.

2.2 Instruction Encoding

Since x86 is a CISC architecture, the encoding of instructions is non-trivial. An instruction consists of 1 to 15 bytes, optional prefixes, an opcode, mostly a ModRM byte, which describes the operands and addressing modes, optionally of a scale/index/base (SIB) byte for indexed memory access, and sometimes of one or more immediate values. There is a variety of instruction prefixes, which can have different meanings, depending upon the opcode. Some instructions (e.g. PUSH) have the operand directly coded in the opcode.

2.2.1 Prefixes

Prefixes change the behavior of an instruction. With the help of the operand size override prefix (value 0x66), the operand size can be modified to an alternative value. If the operand size is 32bit by default, with the help of the operand size override, a 16-bit operand can be encoded.

Address size override (value 0x67) has a similar functionality, although dedicated to address sizes. The lock prefix (value 0xF0) is used to execute a command in an atomic mode, which is necessary for synchronization in parallel execution. Since the interpreter supports a single thread only, lock prefixes are ignored.

String operations can use a large variety of further prefixes, such as:

- REPNZ/REPNE (value 0xF2) in order to repeat the instruction until the accumulator is not equal to zero.
- REP (value 0xF3) in order to execute the instruction RCX times.
- REPZ/REPE (value 0xF3) to repeat the instruction, until the accumulator is zero.

The availability of the prefixes depends on the instruction. It is important to notice that these REP prefixes are only valid for string operations.

Daniel Pekarek February 28, 2019 11/52
SSE operations repurpose prefixes by using a prefix together with an opcode byte, which represent an actual opcode.

Every memory access has an address operand, and a segment register, which was implemented in the original 8086 processor, to access more than 64kB of memory. With the help of segment override prefixes (0x64 for the FS segment and 0x65 for the GS segment), the segment for memory access can be determined. By default, the DS segment is used. Since a linear address space is always used on x86_64, the segment represents an offset. DS is always zero, therefore a memory access to DS:42 points to address 42. With segment override for FS and FS base set to 100, the pointer FS:42 points to 142 instead.

In the 64bit mode also a REX prefix is available, which consists of multiple fields which can select registers r8 to r15 and/or promote operands to 64bit. The REX prefix consists of the following bits:

- w: promote operands to 64bit.
- r, x, b which are used together with the registers from ModRM and SIB bytes to encode registers r8 to r15.

### 2.2.2 ModRM

The ModRM byte describes one to two operands in one instruction, and can also be used for the extended opcode. ModRM consists of three fields: mod, reg and r/m. The field reg is a register operand (r0 - r7) or an extended opcode, depending on the instruction. The fields mod and r/m can be decoded into an operand with the help of a table. The operand is either a register, or a memory operand. Certain combinations of mod and r/m require an immediate value for offsets or a SIB byte, in which complex memory accesses are encoded.

### 2.3 Linux Syscalls

Syscalls can be used to call functions of the operating system [8], like for example opening a file. In the intro example (Listing 1), syscalls are used to open a file and terminate the program. A syscall comprises of a syscall number in the register RAX, as well as arguments in the registers RDI, RSI, RDX, R10, R8 and R9. The result is passed from the operating system to the program via the register RAX. The processor sets RCX to the address of the next instruction after the syscall, and the flags are stored in R11 [9].

Linux supports more than 300 syscalls, which support the important functions of the kernel. Syscall numbers vary extensively between different architectures and are defined in the header /usr/include/asm/unistd.h. On x86, there are two headers, for x86_64, the numbers are located in /usr/include/asm/unistd_64.h.

Some syscalls like `ioctl`, `fcntl`, or similar, accept a subcommand number as an additional argument. Consequently, these syscalls support different functionalities in reality. For most of the programs it is not necessary to implement all syscalls or subcommands, because in reality a single program rarely uses all syscalls. Tests show that approximately 65 syscalls are sufficient to execute the SPEC CPU2006 benchmarks, all shootout benchmarks, and some command-line interface (CLI), as well as X11 programs. In case a syscall is not implemented,
this can be indicated to the program as an error code ENOSYS [10]. Some programs support fallback routines, in case certain syscalls are not available.

2.4 Virtual Memory

Every process is allocated to a virtual linear address space, which gives the process the impression to have full access to the complete memory. Virtual addresses are translated to physical addresses via the processor’s memory management unit (MMU). A physical address points to a location in the installed physical memory. The MMU is configured by the operating system. In case the program tries to access a virtual address, for which no mapping to a physical address exists, an interrupt, called page fault, of the operating system is triggered, and the operating system can react accordingly. If the memory is simply swapped to the hard disk, it can be fetched into the main memory, and an entry into the page table is created, so the processor can translate the address. If the address references unallocated memory, the operating system can trigger a segmentation violation error, and the program is terminated by default.

Virtual memory is organized in pages, which have a default size of 4kB on x86. In order to execute a program in the interpreter, virtual memory must be modeled with the help of pages, since the memory management with the mmap syscall always requests pages in the address space of the process. It is not necessary to model the low level details of MMU configuration, which is only relevant to the operating system itself, since the interpreter comprises the processor and the operating system, hence it is not necessary to model the operating system internal functionality, like the swapping of memory space, or configuration of the processor’s MMU registers.

2.5 ELF

The executable and linkable format (ELF) file format is a standard format supported by Linux to store executable programs and shared libraries. An ELF file consists of a header, segments, and sections [11]. Segments are used by the kernel, in order to load program code and data into the virtual memory [12]. Segments consist solely of a type (e.g. PT_LOAD), the virtual address to which the segment shall be loaded, the offset in the ELF file, in which the data to be loaded can be located, a size, and access permissions (READ, WRITE and EXEC). Sections however, are of more complex nature, and are used by the linker, or dynamic linker, in order to combine multiple ELF files to one process image. Code, data, symbols, strings, debug information, and others can be stored in sections, for example. Sections have a name like “.text”. A segment can contain multiple sections, where else a section can be spread across multiple segments.

In case an ELF file is fully relocatable, it can be loaded to an arbitrary page aligned address. All segments have to keep their relative offset as described in the ELF header, consequently, the offsets between segments may never be changed.
2.6 Static vs Dynamic Linking

For a statically linked program only the loadable segments are relevant. The segments contain the executable code and all data of the program. Since the segments are loaded and there is no necessity to modify the code, sections are not necessary, except for debugging.

With a dynamically linked program, a special segment is available (PT_INTERP), which contains the file name of the program interpreter, typically /lib/ld-linux.so. The program interpreter is an executable program in ELF format, which must be statically linked. It is additionally loaded to the program in the address space, and the program execution is started at the program interpreter, not at the actual program. The program interpreter reads information form the sections, to load dependent libraries and resolves symbols at runtime. After all dependent libraries are loaded by the program interpreter, the program execution is resumed at the entry point of the actual program.
3 System Overview

This section gives a general overview of the system architecture of vmx86, as well as the GraalVM, and the interactions via Truffle interoperability.

The x86_64 interpreter – vmx86 – is based on the Truffle [2] framework. Truffle is a language implementation framework, which is used with the Graal [1] compiler, in order to develop a high performance interpreter, with limited development resources. The interpreter utilizes the JIT compiler Graal.

![vmx86 system overview](image)

As you can vide in Figure 2, a program, which is available as an ELF file, is loaded from disk at runtime, parsed by the ELF parser, which was developed for the interpreter, and the loadable segments are written into the virtual RAM. Consequently, the code is parsed by the x86_64 machine code parser into executable Truffle nodes, which are executed in the main loop of the dispatch logic of the interpreter. Interactions with the environment are supported through syscalls, so an interpreted program can access the network, filesystems, and various security functions like getuid.

Truffle nodes represent executable statements in a program. An x86_64 instruction can consist of multiple Truffle nodes, for example a node for the instruction, and additional nodes for retrieving the operands.

Syscalls are implemented using a syscall node, which calls into a POSIX library. The POSIX library provides the base for the 65 most important Linux syscalls.

3.1 Graal/Truffle

Graal is a JIT compiler for the JVM, written in Java [1]. Graal utilizes the JVM Compiler Interface (JVMCI) of the JVM, in order to install compiled code into the virtual machine (VM). Truffle is a language implementation framework, with which language interpreters can be implemented in Java. The Graal compiler has special knowledge of Truffle, and therefore interpreters implemented in Truffle can be optimized very efficiently [2]. Consequently, it is possible to execute a program, interpreted by a Truffle based interpreter at a performance level comparable to a Java program. vmx86 generates Truffle nodes from x86_64 machine code, which can be optimized and compiled by Graal.
With Truffle, abstract syntax tree (AST) interpreters are developed. The AST of the interpreted programs is consequently directly executed with Truffle. Every node in the AST is mapped to a Truffle node, which can also have, if necessary, child nodes. A Truffle node contains an `execute` method, which executes the node and all respective child nodes and returns the result. In some cases, no result is returned, but e.g. written into registers, or memory. Truffle also supports that the AST can be rewritten at runtime, in order to optimize performance based on the actual data types and value ranges [2].

By using SubstrateVM, Java programs can be compiled to executable programs, which have no dependency on the JVM anymore. This is of great interest for programmers of language interpreters, since they can generate binaries, which contain the language interpreter, and the Graal compiler. Due to the translation with SubstrateVM, the startup overhead of the JVM is eliminated, and an interpreter starts almost as fast as an interpreter developed in C, e.g. CPython (Python) or MRI (Ruby). However, the interpreter code being executed before Graal compilation is triggered, is statically compiled and does not utilize a JIT compiler, which can result in a performance loss, compared to the JVM during the startup phase.

3.2 vmx86

vmx86 is an interpreter based on Truffle, which executes x86_64 programs for Linux in ELF format on a JVM. In conjunction with Graal, and after a sufficiently long warmup phase, which depends on the program, vmx86 can reach a performance of approximately 3× - 25× slower than native execution. During the warmup phase, the performance loss can be much higher (250× slower or more).

In a GraalVM it is possible to mix multiple languages in a single program. This is achieved by implementing an interpreter with Truffle for each language. With vmx86, interoperability to x86_64 libraries can be provided for a GraalVM. For example, the LLVM bitcode interpreter can use existing code from shared libraries without having to decompile them to LLVM bitcode.

Since memory and syscalls are completely virtualized, it is possible to execute Linux/x86_64 programs on any system, on which Graal/Truffle is available. Consequently, it is possible to execute Linux/x86_64 programs using vmx86 on MacOS, Windows, and even Linux/AArch64. The functionality of certain syscalls is limited, e.g. the support of some filesystem features, like extended attributes, or symlinks on Windows.

3.3 Truffle Interoperability

With the help of Truffle interoperability, Truffle languages can exchange data with minimum overhead, and call functions in foreign languages [13]. Since there are no function signatures in machine code, vmx86 implements the Truffle Native Function Interface (NFI), which can call native functions. The Truffle language, which uses the Truffle NFI, has to supply the function signature of the called function before the function is called.

Truffle NFI offers function calls with integer/float arguments, and the possibility to exchange strings, object handles, and callback functions, with native code. A large number of the above mentioned functions is already implemented in vmx86, at least in a rudimentary way.
4 Implementation

In this section, the implementation details of vmx86 are described. This includes the virtual memory implementation, ELF loader, machine code parser, and Truffle interoperability related features.

As already described, vmx86 uses the Truffle framework to utilize Graal as an efficient JIT compiler, and in order to offer functionalities for other Truffle languages. Since Truffle language implementations are AST interpreters, machine instructions are mapped to Truffle nodes. Multiple machine instructions represent a basic block which in return is a Truffle node. Between the basic blocks, the control flow is handled by the dispatch node. Depending on the instruction pointer (RIP), the basic block is selected.

Since Graal is a method based compiler, guest language implementations must contain Truffle methods, the so called Truffle call targets. A method based compiler processes methods only, therefore, Truffle call targets are the compilation units. The machine code however, does not contain sufficient information, in order to determine in a fast and efficient way what a method is, hence Truffle call targets are created with the help of a heuristic. A Truffle call target contains a dispatch for a fixed amount of basic blocks. The Truffle call targets are selected from a dispatch node, or if no Truffle call target exists for an address, it will be generated.

4.1 x86_64 Instructions

A parsed instruction is stored as a Truffle node. There are Truffle node implementations for every x86_64 instruction. Instructions (with the exception of \texttt{NOP}) have operands, which can be read, or written to, or immediate operands, which are always of constant value. If an operand is a register, or memory operand, a respective read or write node is created, with which during runtime, access to operands is granted. A node can give information about which registers are read or written to. This leads to optimization in the dispatch node. For details vide section 4.8.

4.2 Truffle Frame

A Truffle frame allows storing local variables of a function. Graal takes care that JIT compiled code does not allocate objects on the heap, and stores the data directly in the stack / registers. In case such an object is passed to foreign code, the object is actually allocated on the heap. This must be avoided due to performance reasons.

All registers are stored in the frame, and therefore handled like local variables of a function by Truffle. This is due to performance reasons because Graal can allocate such variables to registers.

4.2.1 Registers

All 64-bit registers are stored as \texttt{long} words, since no further information for 64-bit registers is required. Read nodes mask the part they need, unless they read the whole 64bit register.
Write nodes combine the already existing value with the new value, unless the complete 64-bit register is written.

### 4.2.2 Flags

All flags are stored as `boolean` variables. This allows Graal to determine if a flag is used at a later stage. When a guest program wants to read out the `rflags` register, the individual flag bits are translated into the numeric values for the `rflags` register. Otherwise when a guest program wants to write to the `rflags` register, the individual flag bits are extracted from the numeric value and stored in the `boolean` variables.

### 4.2.3 SSE Registers

SSE registers have a width of 128 bit and are implemented as a `Vector128`. A `Vector128` is a value type and consists of two `long` words. In order to avoid allocations with almost every operation, a new `Vector128` is created, and stored in the frame, so it can be avoided that a `Vector128` from the frame can be passed to foreign code, which would result in an allocation of the `Vector128` on the heap.

During initial experiments, we used an array of two `long` words in the `Vector128` class instead of two `long` variables. This lead in some cases to unwanted allocations of the array on the heap. By using two single `long` variables, the allocation is always avoided.

### 4.3 ELF Loader

The ELF loader reflects the behavior according to the implementation of the ELF loader in the Linux kernel. The ELF loader loads the ELF file at the beginning of the execution into memory. The `PT_LOAD` segments, and the `PT_PHDR` segment (if available), are loaded to the defined virtual addresses. In case a program interpreter is defined (a `PT_INTERP` segment exists), it will be additionally loaded to the program, and the entry point is set to the program interpreter. For debugging purposes, symbols from the ELF file of the program, and the program interpreter are loaded, if existing.

In case the program code is relocatable, a fixed load bias of `0x40000000` is used. Basically an arbitrary load bias can be selected to achieve address space layout randomization (ASLR). Practically debugging is simplified if no ASLR is used and a fixed load bias is used instead.

As soon as the segments are loaded, the stack is filled. Since no assumptions for specific C libraries, or even programming languages are made, it is irrelevant in which programming language the guest program was written. The program must be executable on Linux in order to be loaded correctly by the interpreter.
4.4 Stack Memory Initialization

Environment information like program arguments, environment variables, and further information (like the filename of the program binary), are transferred to the program via the stack, therefore, vmx86 has to set up the stack accordingly. The stack layout is part of the Linux ABI, and has the following structure:

```
--------------------------------------------- 0x7fff6c845000
0x7fff6c844ff8: 0x0000000000000000
  4fec: './stackdump'              <-----+
env / 4fe2: 'ENVVAR2=2'            |  <-----+
  4fd8: 'ENVVAR1=1'                |  <-----+
/ 4fd4: 'two\0'                    |  <-----+
args | 4fd0: 'one\0'                    |  <-----+
  4fc8: 'zero\0'                   |  <-----+
3020: random gap padded to 16B boundary
3019: '\x86_64\0' <-+           |       |
auxv 3009: random data: ed99b6...2adc07 |  <-+       |
data 3000: zero padding to align stack |  <-+       |
          . . . . . . . . . . . . . . . . . . . . . |
2ff0: AT_NULL(0)=0                     --+       |
2fe0: AT_PLATFORM(15)=0x7fff6c843019     ---+       |
2fd0: AT_EXECFN(31)=0x7fff6c844fec       ------|-----+ |
2fc0: AT_RANDOM(25)=0x7fff6c843009       ------+-----+ |
ELF 2fb0: AT_SECURE(23)=0
auxiliary 2fa0: AT_EGID(14)=1000
vector: 2f90: AT_GID(13)=1000
(id,val) 2f80: AT_UID(12)=1000
pairs 2f70: AT_UID(11)=1000
  2f60: AT_ENTRY(9)=0x4010c0
  2f50: AT_FLAGS(8)=0
  2f40: AT_BASE(7)=0x7fffe6c120200
  2f30: AT_PHNUM(5)=9
  2f20: AT_PHENT(4)=56
  2f10: AT_PHDR(3)=0x400040
  2f00: AT_CLKTCK(17)=100
2e00: AT_PAGESZ(6)=4096
  2e90: AT_HWCAP(16)=0xbfebfbff
  2eda: AT_SYSTIME(13)=0x7fff6c86b000
         . . . . . . . . . . . . . . . . . . . . . . . . . . . . . |
  2ec0: environ[2]=nil
  2ec0: environ[1]=0x7fff6c844fe2
  2eb8: environ[0]=0x7fff6c844fd8
  2eb0: argv[3]=nil
  2eaa: argv[2]=0x7fff6c844fd4
  2ea0: argv[1]=0x7fff6c844fd0
  2e98: argv[0]=0x7fff6c844fc0
0x7fff6c842e90: argc=3
```

Listing 2: Stack layout on Linux/x86_64 [12]

The focus of the stack layout is on the auxiliary vector (auxv), which is described in the next paragraphs.

With the auxv, information for the dynamic linking is transferred to the dynamic linker [14]. Part of the information is the original entry point, and the segment table.

The auxv entry AT_SYSINFO_EHDR is used to transfer the address of the virtual dynamic shared object (vDSO). Since the vDSO is not necessarily required, the respective functionality, and the auxv entry, have not been implemented in vmx86.

The auxv entry AT_HWCAP contains feature flags of the CPU and could be queried with CPUID. For every program start CPUID must be executed, therefore, it is more efficient to use the AT_HWCAP, because the CPUID instruction can be saved.
AT_UID, AT_GID, AT_EUID, and AT_EGID are used to pass the user ID (UID)/group ID (GID), respectively effective UID/effective GID. Alternatively, a combination of four syscalls could be called, however, this is not very performant. Since all four values are required for every program start, it is more efficient to pass them via auxv.

AT_RANDOM points to a 16 bytes long sequence of random numbers. These values are partly used by the dynamic linker and the C library of the system. Again, these values can also be queried by syscalls. Since they have to be retrieved for every program start, some syscalls can be eliminated, which results in a performance gain.

The vDSO is a virtual library that is linked into the address space of a process, and consequently can deliver functions for some syscalls, e.g., for querying the time, which are implemented in the userspace [15] [16]. The benefit of this methodology is, to avoid time consuming context switching to kernel mode. This results in a performance gain. Since the vDSO is optional, and all functions provided by the vDSO are also available as syscalls, the vDSO can be eliminated in the interpreter implementation.

The stack pointer (RSP) points at program start to the number of the program arguments.

4.5 Virtual Memory

The virtual memory is essential for the execution of programs. It enables allocation of memory at almost arbitrary addresses, which is used by programs in ELF format, and the memory management with the help of mmap syscalls. The actual allocated memory does not necessarily have to be a continuous address space. Memory is segregated into pages, which have a standard size of 4kB in x86_64. The allocation of pages to the physical memory is normally achieved by a page table in the operating system, which is made available to the processor. The MMU of the processor translates the virtual address into a physical address at each memory access, checks the access permissions, and results either in a memory access with the translated address, or creates a page fault, which must be handled by the operating system.

In the vmx86 interpreter, the virtual memory is modeled with pages, and a number of sequential pages are managed in a single block. This improves the performance, since all pages do not have to be stored individually, which also results in a further performance increase, since the allocation and release of big memory spaces, also the caching of lookup results, is more efficient.

Big memory blocks increase the cache performance of lookup results, since more addresses can be handled by a cache entry. The allocation and release of memory is also simplified with the help of big blocks, since a small number of blocks in the memory allocator must be managed instead of single pages. Consequently, a big memory block can be allocated or released in one single step.

The virtual memory is implemented in the class VirtualMemory and their respective subclasses. Three implementations of the class exist: one is a pure Java implementation, the second implementation uses native memory due to performance reasons, and the third implementation uses a combination of both.
4.5.1 Pure Java Implementation

For any addresses, a Java implementation of a MMU can be utilized. This is achieved with a TreeMap, which maps addresses to memory blocks. One memory block can contain one or more pages. In case the access permissions of a part of the memory block are modified, the memory block will be split, in order to maintain the access permissions on page granularity. If memory is released or allocated, it is possible that memory blocks are split.

A TreeMap is utilized for the following reason: it is necessary to identify the memory block allocated to an address. In the TreeMap, the starting address of the memory block is stored as a key, and the value contains the memory block. In a standard map (e.g. HashMap) it is only possible to read out the memory block, if the starting address is known. By using a TreeMap however, it is efficiently possible to search for the next smaller key, which relates to the beginning of the memory block to which the target address points.

Since arbitrary pages can be released, it is also possible to release blocks within a memory block. In this case, the memory block must be split into two blocks, and the one in the center, which should be released, is removed from the mapping (refer to Figure 3). It is also possible to overwrite existing mappings when performing an allocation of memory. In this case, the existing block is also split in order to store the new access permissions, and in case of a memory mapped file, the actual file.

```
free  used  free
```

Figure 3: Freeing a memory area might split a memory block

In order to allocate addresses with mmap, if the executed program does not request an explicit address, the allocation of memory spaces is managed by a memory allocator with a double linked free list. Whenever memory is requested or released, the information in the memory allocator is updated.

For every allocated page, information like the access permissions, and in case a file is mapped, the file name, and the offset are stored. In the debugging mode of the interpreter, it is also possible to analyze mapped files, in order to obtain symbols of dynamically linked libraries.

A function is implemented that supports arbitrary implementations of pages, so a memory mapped I/O is supported. A custom implementation of a page can be created by implementing the page interface, which defines methods for read and write accesses. This can be used for example, for interoperability with other Truffle languages, in order to avoid the copying of data. Without memory mapped I/O, data must be copied to the memory, which leads to a performance loss in this particular case.

A weakness of the pure Java implementation is that the lookup in the TreeMap is very slow, and each memory access requires a lookup. This is why a least recently used (LRU) cache with three entries is used, which increases the hit rate to > 95% in most of the cases, according to benchmarks. However, in comparison to the MMU of the hardware, this performance is still slow.
4.5.2 Native Memory with JNI

In order to increase the performance of the translation of virtual addresses, the MMU of the hardware should be used as often as possible. This is achieved by allocating a big memory block as soon as the interpreter is started. The memory block is marked as non-readable and non-executable. Its main function is to reserve the memory block. When a program performs an allocation, the memory is mapped with the mmap syscall and the access permissions are set accordingly. To perform a memory access, the Java class Unsafe is used to perform a native memory access, which consequently causes the MMU of the hardware to translate the memory addresses, and to check the access permissions. The selection if native memory, or the Java implementation of the MMU is used, is achieved via a comparison. If the address is smaller than the native memory block size, the native memory is selected, otherwise the Java implementation is used. The memory layout is depicted in Figure 4.

<table>
<thead>
<tr>
<th>native</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast (code, stack, heap)</td>
<td>slow (mmap calls with fixed address &gt; split address)</td>
</tr>
<tr>
<td>JVM, ...</td>
<td>reserved space</td>
</tr>
</tbody>
</table>

Figure 4: JVM memory layout with the native memory block of vmx86

In order to handle segfaults in the interpreted program, if the native memory implementation is used, a custom segfault handler had to be developed. Without the custom segfault handler, the JVM would terminate with a core dump, if a segfault occurs within the interpreted program. In order to address this, a segfault handler is installed via Java Native Interface (JNI), which checks if the segfault is located within the allocated native memory block. In case the segfault is within the allocated native memory block, it has been triggered by the interpreted program, and the interpreter receives a notification. In a variable in the native memory, a flag is set, which is polled from the Java code directly after the memory access, with the help of the Unsafe class. If the flag is set, an exception is triggered. In case the segfault is not located in the allocated native memory block, the segfault handler of the JVM is invoked. In case of the execution of a segfault within the allocated native memory block, it is necessary to resume the execution of the Java code.

To continue execution in the case of a segfault, the interpreter has to skip the memory access instruction, and resume execution of the Java code. The implemented methodology is the following: with the help of XED (a library of Intel with which x86 instructions can be decoded), the length of the actual instruction is determined, the RIP of the JVM process is adapted, so that the next instruction is pointed at, and the execution can resume. This implementation leads to the scenario that the Java code has the impression, as if memory access returned random data, or did not perform a write operation. In specific, the original register content during a read access is returned to the Java code.

Unfortunately, it is not easily possible to directly throw an exception from the segfault handler, since Hotspot, Graal, and SubstrateVM must be adapted. The enhancement of Hotspot, Graal, and SubstrateVM is not scope of this work.

The polling logic degrades performance, since each memory access in the program being interpreted requires two memory accesses in the x86_64 interpreter: the first is the actual access, and the second is the access to the segfault flag variable. In case segfaults are not pos-
sible in the program, the polling of the segfault flag variable can be eliminated with the help of a configuration flag by the user. This leads, according to benchmarks, in particular cases to significant higher performance. If a segfault is triggered, the program gets the impression that random data was read from memory; write operations are ignored. The performance evaluation in Section 6 provides further details.

Prior to every memory access, a check is performed to determine whether the native memory or the Java memory should be accessed. Since with the native memory the reserved native memory block is involved, the sandbox property is valid. In particular it is not possible to crash the interpreter with the help of invalid memory access. Invalid memory accesses are thrown as an exception instead.

4.5.3 Native Memory with SystemJava

For SubstrateVM, the JNI code has been substituted with SystemJava. The code of the segfault handler is written in C and is statically linked into the SubstrateVM image. The rest is implemented with SystemJava. The consequence is that the JNI library is not required for the SubstrateVM image. Hence, no external dependencies pertain.

4.6 Syscalls

In order to ensure platform independent functionality, it was necessary to implement syscalls in Java. Consequently, vmx86 can be executed on other operating systems, e.g. Windows, MacOS, Solaris, and others. At the same time, the pure Java implementation makes it impossible to escape from the sandbox. Therefore, vmx86 can be embedded in other software without being a security risk.

65 Linux syscalls are implemented. The selection of the implemented syscalls is based upon tests of various installed programs on the developer’s computer. The syscalls required by the tested programs are the ones that are implemented. The tested programs comprise:

- SPEC CPU2006 benchmarks
- Language Shootout Game benchmarks
- some programs of the coreutils package
- the GCC backend
- GNU as and GNU ld
- xpdf
- xv
- boolector
- other CLI programs like bzip2, lame, and oggenc

Syscalls are executed in a syscall node, which selects the function of a POSIX library, based on the syscall number. The POSIX library implements the important functions, like file descriptor management, a virtual file system, and POSIX-/Linux specific functions like access of the clock, requests of resource limits, or similar. The library does not need any JNI code and is implemented in pure Java.
4.6.1 Virtual Filesystem

The virtual filesystem used in the interpreted program comprises a file descriptor manager, and a virtual filesystem implementation. When a file (or socket) is opened, a new Stream object is generated, and a file descriptor attached. The abstract Stream class has methods for reading and writing of data, as well as querying of information with the stat command. Every filesystem implementation creates a subclass of the Stream class in which the operation for the specific filesystem is implemented.

A special Stream is the DirectoryStream, which is used for readdir, respectively getdents to list the content of directories.

Native filesystem  A NativeFileSystem has been implemented, which directly exposes the filesystem of the host computer to the interpreted program. It is used for normal programs, which can interact with the files of the host computer. Note that one challenge with the NativeFileSystem is that the NIO API is used to interact with the host filesystem, which, depending on the host operating system, not necessarily supports the required features for Linux programs. When dynamically linked programs are executed, the dynamic linker of the glibc requires symlinks on shared object files, which are read via readlink, and this can lead to problems with Windows. Admin rights are required to create symlinks on Windows.

Tmpfs  The Tmpfs is implemented and exclusively located in the memory. It is very useful for unit tests, since the filesystem can be manipulated as desired, as it is required for the specific test case.

At program start three file descriptors are opened:

- 0 for the standard input (stdin)
- 1 for standard output (stdout)
- 2 for standard error (stderr)

We found that Linux links the standard input and standard output by default, so writing access on stdin is supported, as well as reading access on stdout. This function is working, as long as the streams are not redirected with e.g. a redirect in a shell. Unfortunately, it is not possible to implement this behavior completely in Java, so for the interpreter stdin and stdout are always linked, independent of redirections.

We also found that by using pure Java, it cannot be determined, cannot report if System.in, or System.out are a real file, or an attached terminal. In case both are attached to a terminal, this can be determined via System.console(). CLI programs usually request this information, in order to optimize their output for the terminal, or alternatively write a differently formatted text version into a file. Some programs like gzip refuse writing data to a terminal. Since it is not possible to retrieve the necessary information from Java, stdin and stdout are always reported as a terminal to the interpreted program. The reason why stdin and stdout are reported as a terminal is that most tested programs are console applications. In this case, the CLI programs have optimized their outputs for a terminal.
The reported terminal size has been defined as 80x24, which is the standard terminal size. The reason why the value is set to 80x24 is historical.

When data is written to a file with the syscall `write`, and the length argument is too long, data is written as long as the memory is readable. The `write` syscall returns a value that is equal to the amount of bytes written. For example, `write(1, ptr, -1)` writes from `ptr` the amount of bytes that are mapped and readable. Only in case `ptr` points to a not mapped, respectively a not readable memory space, the `write` call is terminated with an `EFAULT` error.

### 4.6.2 poll

Some programs wait for I/O events by using `poll`, or `select`, for events of a file descriptor. Some programs utilize them to wait for events of multiple file descriptors. Typically, such a program waits in a loop with `poll`, or `select`, for events, and handles the event as soon as it occurs.

With this methodology, multiple connections in a server can be managed by a single thread. Shells also use `select`, respectively `poll`, to wait for user input. Programs like `telnet`, `nc`, and similar use `select`, respectively `poll`, to wait for user inputs, or incoming data on a socket. This leads to immediate data processing, as soon as data is available, without the involvement of multithreading. This subsequently leads to a performance gain, because no threads have to be scheduled.

In Java, the possibility exist to reconstruct the functions of `poll/ select` via a `Selector`. However, in Java it is only possible to use a `Selector` in connection with a network socket, and not in connection with a file. The consequence is that it is not possible in Java to implement `select` or `poll` on stdin. Consequently, interactive CLI programs are not supported without the necessary functions made available by JNI code. The implementation described above, using JNI, was not scope of this thesis.

When `poll` is executed, the interpreter code checks if all involved file descriptors are sockets. If not, an error is returned. In case all file descriptors are sockets, a `Selector` is created with the help of NIO, the sockets are registered on the `Selector`, and the `select` operation is executed. An implementation of `poll` exclusively for network sockets is sufficient to execute some X11 programs, like `xpdf` and `xv`.

The main difference between `poll` and `select` is that `select` can only handle file descriptors up to 1024, while there is no such limitation for `poll`. It is possible to implement the function of `select with poll`.

### 4.7 x86_64 Machine Code Parser

Machine code is parsed, as soon as it is executed for the first time (see section 4.8), and translated to the respective Truffle node. At this stage, the prefixes are parsed first, and stored in flag variables. As soon as the read byte is no valid prefix, it is interpreted as an opcode. There is a special case where the opcode is 0x0F. This means an additional opcode byte must be read. Depending on the opcode (if necessary), the ModRM, and the SIB byte are read,

---

1This size has been defined in history by typical video terminals, like VT100, IBM 3270, and others. Additionally, the Linux kernel reports 80x24 for unconfigured serial lines.
and/or the immediate operands. In case the prefix has no meaning for the instruction, the
prefix is ignored. With SSE instructions, explicitly no prefix instructions exist. With these
instructions, no prefix must be available. This is due to the x86 specification.

In case a REX prefix exists, it is used to decode the ModRM, in order to support all registers
and addressing modes. Some instructions use the REX.w bit to promote operands to 64bit.

In case the opcode is unknown, which means it does not relate to an implemented instruc-
tion, an IllegalInstruction is generated. Execution of an IllegalInstruction results in an error (i.e., a Java exception). This is important, since in some cases instructions are available in the code, which are not implemented in vmx86. However, they are never executed at runtime due to feature guards.

For example, a program can have functions for SSE and AVX2 implemented, and determine with CPUID at runtime, which function shall be used. It is possible that instructions are parsed at runtime, but never executed. Consequently, no error is caused, if a invalid, or not implemented instruction is parsed, but never executed. An IllegalInstruction can be generated if the instruction is guarded by a branch, which is never taken at runtime. The algorithm which is used to parse instructions of the program, is described in Section 4.8.

4.8 Dispatch

The control flow between Truffle call targets and basic blocks is implemented via a dispatch
node. A dispatch node decides, based upon the RIP register, which Truffle call target, respec-
tively which basic block, shall be executed next. With the dispatch node, special care must
be taken in regards of the structure of the execute method. The dispatch node consists of
a loop, which, depending on the RIP register, executes a node of the array of child nodes.
The executed node returns which node should be executed next. We spent much effort, until
Graal finally could handle the loop of the dispatch node, unroll it, and inline all basic blocks.

Figure 5: Truffle AST structure with dispatch nodes, basic blocks and instructions

4.8.1 Basic Blocks

A basic block consists of one or more x86 instructions, which are executed sequentially. A
basic block ends with a control flow instruction, like a jump. The instructions CALL, RET,
and SYSCALL also terminate a basic block. Calls terminate a basic block so that the size of
the basic block is kept small. The instructions **CALL** and **RET** are internally implemented as a jump with a **PUSH** (or **POP**) of the next address, so that the dispatch node does not require special knowledge about calls.

Syscalls terminate a basic block, because some syscalls like **exit** influence the program execution significantly, and therefore it is not useful to *always* parse code after a syscall, until it is executed for the first time.

### 4.8.2 Generating Call Targets

As already mentioned, Graal is a method based compiler, so all methods in a guest program must be detected and communicated to Graal via Truffle call targets. Since machine code does not have sufficient information about methods, it is not easy and efficient to accurately detect the methods of a program. For this reason, a simple heuristic has been developed, to create Truffle call targets in order to enable JIT compilation.

The methodology works as follows: starting with RIP a basic block is parsed. If the basic block ends with a direct jump, the jump is followed, and a new basic block is parsed at the target address of the jump. In case a basic block ends with a conditional jump, an additional basic block is parsed at the jump target, as well as directly after the jump. Only in case a basic block ends with an indirect jump, a call/return or a syscall, no new basic block is parsed. If a jump target is located in an already parsed basic block, the basic block is split at that location.

The details are described in detail in the following algorithm.

#### Algorithm

**A1**  
instructions ← new empty list, blocks ← new empty list, todo ← new empty list. Add the current RIP to todo.

**A2**  
If todo is empty, terminate. Otherwise set pc ← todo[0] and remove the first element from todo. If pc points into a block in blocks, split the block at pc and repeat step A2. If pc points to the beginning of a block in blocks, repeat step A2. Otherwise go to step A3.

**A3**  
Parse instruction at pc. If the instruction is a direct **JMP**, add the target address to todo, add the instruction to instructions and go to step A4. If the instruction is a conditional branch, add the target address to todo, add the instruction to instructions, increase pc by the instruction size and repeat step A3. If the instruction is a **CALL**, **RET**, **SYSCALL** or indirect **JMP**, go to step A4. Otherwise add the instruction to instructions, increase pc by the instruction size and repeat step A3.

**A4**  
Create a new basic block with the instructions in instructions, add it to blocks. Set instructions ← new empty list and go to step A2.

Figure 6 shows the result of parsing the demo program from Listing 1 into Truffle call targets (red borders) and basic blocks (black borders).

The algorithm ensures that code of branches and small loops are kept within a single Truffle call target, whilst at the same time it is ensured that self modifying code is parsed correctly,
as long as no modification occurs, after the code was parsed for the first time. Code that is never executed, e.g. after an `exit` syscall, is never parsed.

In a further test the impact of the strategy for calls has been investigated. Calls did not terminate the basic block, but executed a Truffle call instead. The challenge was that the code within a Truffle call target became too big, so the code cache of Hotspot had an overrun, and the warmup phase duration increased drastically. In addition, Graal failed to compile some Truffle call targets due to the size. It can be assumed that the warmup problem is due to loops being located within a Truffle call target, and hence, compiled at a later stage, because on stack replacement [17] is not implemented.

The strategy to terminate basic blocks at calls, and to not further parse after calls, leads to the scenario that loops in which functions are called are split across multiple Truffle call targets. This decreases the peak performance, but allows that Truffle call targets are called much more frequently and compiled earlier. This is a performance trade off.

As soon as a Truffle call target is parsed, it will not be changed anymore. If `RIP` points into an already parsed Truffle call target, a new Truffle call target is created at this location. A new Truffle call target is always created, when the outer dispatch node has to call a Truffle call target, but for the actual value of `RIP`, no Truffle call target exists.

The reason why code is parsed once it should be executed for the first time is that it is generally not possible to statically identify all jump and call targets, since targets of indirect jumps and calls are computed at runtime and therefore not known at this stage. Furthermore, with large programs, for which only a small part of the code is executed, only the actual executed code is parsed, which relates to an improvement of startup performance and memory utilization.

**Frame** Since a Truffle frame is only valid within a Truffle call target, and all registers are kept in the frame, it is necessary to transfer at execution of the Truffle call targets the register values, and return from the Truffle call targets the modified register values to the caller.
Our experiments indicated that with some small Truffle call targets, the code to copy the registers can be longer than the code of the created Truffle call target of the guest program to be interpreted. For this reason, a calculation is performed that tells which registers in the Truffle call target are read, and which registers are written. Only registers which are actually read are copied to the frame of the Truffle call target, and only registers that have been modified, are returned to the caller. This leads for instance to the situation that most Truffle call targets do not have to copy the SSE registers, since they are not used by most of the created Truffle call targets. The consequence is a significant performance increase, if little time is spent in the Truffle call target.

### 4.9 Interoperability: NFI Provider

Truffle supports combining various languages in a polyglot environment [13], so that code in one language can use functions, objects, and data in another language. An interface must be implemented in a Truffle language to achieve this interoperability. The challenge in this context is that neither in the machine code, nor in the eventually existing symbols of the ELF file, information regarding the function signatures exist, so it is not possible to automatically convert function arguments, and return values into the correct data type.

The remedy for this problem is the Truffle NFI, which can provide access to native functions. The interpreter vmx86 implements the Truffle NFI interface, and serves as a Truffle NFI provider, in order to execute libraries in a sandbox, even if the host system is not Linux/x86_64.

#### 4.9.1 Truffle NFI

The Truffle NFI is implemented as an internal Truffle language. It is possible to have a custom implementation of Truffle NFI. The default implementation of Truffle NFI is implemented based on libffi, so native code in the JVM process can be directly executed. Truffle NFI requires that the caller loads a library first, then resolves a symbol, and binds type information in form of a function signature to the symbol. This is prerequisite to call a function.

Functions can accept integer arguments that can have a length between 8bit and 64bit, floating point numbers with 32bit or 64bit, strings, object handles, and callbacks. Furthermore, a JNIEnv related structure exists with which some Truffle functions can be executed, for example to create references to Truffle objects in native code.

Truffle objects are opaque handles that are valid as long as the native function is executed. They can only be passed to callbacks, or to the caller. In the native code, they can not be directly used.

#### 4.9.2 Main Program

In order a library can be loaded in a program, the existence of a main program is always necessary. Since no assumption in vmx86 regarding the programming language or the C library is taken, it is not directly possible to load a shared object file and execute the respective functions. Instead, a main program is loaded, which defines a sigaltstack, which is normally used to execute signal handlers on an alternate stack. Afterwards, with the help of
a special syscall, which only exists in the interpreter, the address of three functions is passed to the interpreter:

- a function to load a library
- a function to resolve a symbol of
- a library and a function to release a library.

As soon as the syscall is executed, the main program is interrupted at this location, and the state of the processor is stored. This enables that the execution can be resumed at a later stage.

NFI calls are executed by loading the library first, and resolving the desired symbol by using the functions that are registered by the syscall. The main program internally uses dlopen, dltsymp, and dlclose. The advantage of this method is that the dynamic linker of the C library can be used, which consequently means that no dedicated code for loading and linking of libraries must be developed. Also constructors, initializers, atexit handlers, and similar are executed correctly.

When the NFI context is closed, the main program resumes at the previously stored location. This means that all destructors and atexit handlers are executed. This is the reason why the signal stack is used for interop calls, otherwise the stack of the main program would be destroyed during NFI calls.

### 4.9.3 Native Pointers

Truffle NFI supports the `POINTER` data type which corresponds to a native pointer. Such a native pointer points to memory in the JVM process. The vmx86 interpreter uses virtual memory in a sandbox, so it is not possible to directly access native pointers.

In order to exchange native pointers per interop, native pointers are marked. The most significant bit of a native pointer on the location, where the native pointer is passed into vmx86 is set. This means that all native pointers within vmx86 are negative, which is usually invalid for Linux since negative addresses are reserved for the Kernel. However, according to tests, glibc can actually handle negative addresses. When accessing such a negative address in the interpreted code, the most significant bit is cleared, and the `Unsafe` class is used to access the memory. The sandbox functionality is violated, however the interoperability with native pointers is possible. In case the sandbox feature shall remain, the access to native memory can be deactivated via an option.

When a pointer to memory of vmx86 is passed per interop, the native implementation of the MMU must be utilized. In this case the address in the JVM process is calculated by vmx86 and passed by interop. The base of the native memory block is added to the virtual address to retrieve the address in the JVM memory, so it is possible to exchange pointers in both directions via interop.

Again, during research, the following problem surfaced: if a pointer points to a data structure, which contains a pointer, the pointer in the data structure is not translated, since neither Truffle, nor vmx86 can recognize that this is a pointer.
It is not possible to simply invert pointers instead of flipping the most significant bit. When calculating with an inverted pointer, the result is incorrect, because for example an addition with an inverted pointer is equivalent to a subtraction with a not inverted pointer.

### 4.9.4 Supported Messages

The resolving of symbols, assignment of a signature, and execution of an NFI function, is performed via interop messages. Currently the following messages are supported:

- **EXECUTE**: execute a resolved symbol that is assigned to signature
- **INVOKE**: execute the bind method, in order to bind a signature to a symbol
- **AS_POINTER**: interpret a symbol as a pointer
- **TO_NATIVE**: convert a pointer to a native pointer. This operation does not modify a pointer, since a pointer is always a native pointer.

### 4.10 Limitations

Since Linux and x86 are very complex, the interpreter is currently incomplete. It was not scope of this work to provide a complete implementation.

#### x87

Most of the tested programs do not use the traditional x87 FPU, which would calculate with 80bit floating point numbers, so the implementation of the x87 FPU has been omitted. As a consequence, only one SPECint benchmark did not complete, since it uses the x87 FPU for some mathematical operation.

#### AVX

None of the tested programs uses AVX, as long as the CPU does not signal AVX support via CPUID feature flags. Consequently, the complex encoding of AVX instructions and their 256bit respectively 512bit vector registers have not been implemented.

#### SSE4.2

SSE4.2 supports some instructions, which are used from certain programs, as long as the SSE4.2 feature flag is set in CPUID. Since all tested programs work without SSE4.2, if the feature flag is missing, SSE4.2 was not implemented.

#### Multithreading

Multithreading is used by many programs in order to perform certain functions simultaneously. The clone syscalls is used to spawn a new thread within a thread group, which uses the same memory as the parent thread. Multithreading requires synchronization mechanisms, like mutexes (in Linux: futex), atomic x86 instructions (using the LOCK prefix), memory fences, real time signals, and further more. A working implementation of multithreading was not scope of this work and cannot be easily implemented. This is the reason why multithreading has not been implemented.
**Multiprocessing**  Multiprocessing is usually required to execute further helper programs. For example, gcc internally calls the cc1 backend, the assembler, and other programs, and communicates with pipes. For this purpose, a new process is generated via fork, or vfork, and then the new program is loaded with execve. Some programs create worker processes with which communication is performed via pipe or shared memory.

Multiprocessing requires correct implemented real time signals, synchronization mechanisms, pipes, and shared memory. Furthermore, with fork, a copy on write memory image is created, which cannot efficiently be implemented with the JVM, without modifying it.
5 Execution Tracing

A very useful function for debugging is execution tracing, which allows to record and analyze the complete program run of memory access, register values, and operating system interactions. Therefore, it is not necessary to develop a debugger, since the program run can be analyzed in retrospect.

Since such a program run creates much data, a machine readable binary format has been developed to store the traces, as well as tools to view and analyze the data.

When for example the image viewer xv is stared, a 64x64 JPG image is loaded, and xv is terminated, the amount of trace data generated in binary format is approximately 14GB of size.

5.1 Efficient Binary Storage Format

The trace data format consist of a number of type/length/value (TLV) records, which comprise records for register values, program counter details, memory access, interpreter log events, and quite a few memory relevant syscalls.

The following records are defined as:

- **BrkRecord**: is generated when the `brk` syscall is executed. The record contains the new break, so it is traceable where the break was located.
- **CallArgsRecord**: is created with `call` instructions, and contains the first 6 integer arguments of the function call. In case the memory is readable at the address pointed at from the function arguments, the first 64 bytes are additionally deposited in CallArgsRecord.
- **CpuStateRecord**: is created with each executed instruction, and contains all register values of the CPU.
- **EofRecord**: as soon as the last instruction is executed, and the trace file is terminated, the EofRecord is written as the last record. This indicates if the trace file is complete.
- **LocationRecord**: contains the actual value of RIP and, if existing, additional information regarding this position, like the respective symbol, as well as the file that has been mapped to this address.
- **MemoryEventRecord**: with every memory access, this record is generated. It contains the address, as well as data, and the type of memory access.
- **MmapRecord**: This record is generated with each `mmap` syscall, so it can be traced which memory region has been mapped in the program. The record contains arguments, as well as the result of the `mmap` syscall.
- **MprotectRecord**: with every execution of the `mprotect` syscall, this record is created. It contains the arguments and the event of the syscall. This record gives information about the access permissions of a memory location at any time.
- **MunmapRecord**: is created with every `munmap` syscall, and contains the arguments, as well as the result of the syscall, consequently, it is possible to identify with the help of traces, when memory has been released.
• **StepRecord**: contains a LocationRecord and a CpuStateRecord, and is created with every executed instruction. A StepRecord contains all important information regarding the actual executed instruction.

• **SystemLogRecord**: contains the log message of the Java logging system. Since the POSIX library writes with every syscall a log entry when the `strace` flag is activated, the record contains the `strace` log.

### 5.2 Recording Execution Traces

Execution traces are created when certain events occur, and trace data is created at some locations in the interpreter.

In the trace memory subsystem, a memory event is generated for every executed memory access. It contains the address and the date. When executing a `mmap` syscall, the contents of the file, which shall be mapped into memory, is stored in a trace event, so that at a later stage the memory image of the traced process can be reconstructed. Before each instruction is executed, the content of all registers, as well as detailed information regarding the actual execution position in the program (e.g. function name, filename and offset within the file from which the code is retrieved), is stored as trace event.

An execution trace contains the complete program run with all related data, so the program run can be completely replayed at a later stage.

### 5.3 Reproducibility

Recording of traces is generally used for debugging purposes. For this work, a program that is executed in the same environment multiple times must result in the same traces. For this purpose, the behaviour of the instruction `RDTSC` is modified in such way that instead of the actual time, the amount of actual executed instructions is returned. Moreover, it must be ensured that only the absolute necessary environment variables are put on the stack, in order to minimize the changes between the program runs. As soon as a program interacts with the environment, by accessing the network for example, timing problems can occur, which can have a significant impact on the program execution.

For X11 programs, which communicate with the X11 Server per network sockets, it is practically impossible to obtain the same trace repeatedly. However, with simple CLI programs, which do not perform such interactions with the environment, traces can be easily reproduced.

### 5.4 emu86: ptrace Based Reference Trace Recorder

In order to be able to debug the interpreter itself, it is useful to execute a program on a real Linux/x86_64 system, and on the interpreter itself, in order to compare the traces. The first approach was to record a further trace on the host system, and compare the two traces. An interpreter was developed, based upon `ptrace`, which executes the machine instructions step by step via single stepping on the host CPU. The syscalls were implemented in Java with the same POSIX library like in vmx86. We initially decided for this approach since the system behavior is the same.
Before every executed instruction, the content of all the registers is written into a trace file. However, it is not possible to create events at memory access, since the memory access is directly executed by the processor.

The following challenges surface at this stage:

- Memory events cannot be recorded
- Complex programs (e.g. X11 programs), which would indicate a bug, cannot be equally recorded, since they would not show identical traces.

The consequence is that errors are identified at a later stage, where the context is not given, e.g. 100MB of trace file later. During the investigation, the following situation occurred: an erroneous value is written into memory, afterwards a function is called, which creates a load of trace data, and when the function returns, the value is read, and afterwards the error is populated. If no memory trace is available, the error can be detected at this stage. This is not very efficient for debugging, since it is possible only to detect the wrong value, but not the not the reason for that behavior.

For complex programs it is impossible to compare the traces from vmx86 and emu86, since if the program is executed twice with emu86, it shows different results. In order to tackle this issue, a verifier has been developed, which replays a trace on the host CPU, and consequently compares the memory and register content with the trace.

### 5.5 trchk: ptrace Based Trace Verifier

The verifier has a similar architecture as the emu86, however, no new trace is created, the existing trace is checked, and verified. At every memory event, the content of the memory of the ptraced process (this is the interpreted process) is checked against the memory event. After every executed machine instruction, the content of all registers is compared with the registers of the ptraced process. If deviations exist, they can be immediately detected. *mmap* events ensure that the respective memory region is loaded in the address space of the ptraced process. *mprotect* events are used to adjust the memory access permissions in the ptraced process.

Syscalls are not executed, the side effects for registers and memory are checked however. If the verifier passes the trace, and detects that no deviation of any machine instruction took place, it can be assumed that no error in the CPU implementation of the interpreter occurred in this program run.

Note that no bug had occurred, and consequently has been detected at this stage, this does not imply that the CPU implementation is bug free. It can only be ensured that the respective program run behaved as expected.

For illustration: if a PNG image is viewed with xv, no error occurred; if a JPEG image is loaded instead with xv, different instructions are executed, which can result in a previously undetected bug.

A typical use case is shown in figure 7 where libjpeg incorrectly decoded JPEG images. It was not obvious to figure out the nature of the problem by hand. By using trchk, it occurred that *psrlw* was implemented incorrectly. The cause of the problem was that a logical shift was
implemented as an arithmetic shift. The tool trchk quickly detected the problem and located the deviation in the instruction.

![Bug](image1.png)  ![After bugfix](image2.png)

Figure 7: While the PNG decoder correctly decompresses the image, the JPEG decoder uses different instructions and exhibits a bug.

The tool trchk has been implemented to check the traces of X11 programs, like for instance a 14GB trace file of xv, in which a JPEG picture with a size of 128x128 pixel has been loaded. The manual analysis of such a big trace file is infeasible, whilst trchk indicated within minutes the location of the error in the trace. Although the bug originally occurred with the big image as seen in figure 7, a much smaller image was used to debug the problem, since the trace file was significantly smaller, and therefore easier to handle.

This approach saved a lot of time for debugging.

5.6 trcview: Execution Trace Viewer

Trace data is machine readable, but not for humans. In order to address this, a viewer for trace data was developed. The viewer loads all step events, and analyzes function calls, in order to calculate a call tree, so it is possible to make available a function for the user to step over call instructions, like in a debugger. Function calls not of interest for the user, can be easily skipped. This feature saves a lot of time in the debugging process, and helps to understand traces.

Since the complete trace is already recorded, arbitrary navigation is supported. Basically all necessary information would be available due to the existence of memory events to support memory inspection. Second functionality would be of great leverage in the analysis process, and the function would be of this nature: the user inspects an instruction that retrieved a value from memory, and it would be of interest to know which instruction wrote that value into memory. A concept for these features exists, however they are not implemented in trcview due to time constraints.

The trcview interface as depicted in figure 8 consists of three sections: the left section contains the stack trace, the center area contains the execution trace of the current function, and the right section contains the register content of the currently selected instruction. In the status line, the current program counter (PC), and the file, in which the program counter is located, if this information is available, is shown.
### 5.7 Computing the Call Tree

The call tree is created with a recursive algorithm, which parses step events, and collects the instructions in a list, until a `RET` instruction is reached. As soon as a `RET` instruction is reached, the previously parsed step events are aggregated to one node, which is returned from the algorithm. If a `CALL` instruction is encountered, the algorithm is recursively called, and the complete call is parsed into a call instruction node. The GUI shows all the instructions, which are directly referenced from the current node, which is equivalent to the current function. If a call is double clicked, or enter is hit, the function call is followed, and the call node becomes the new current node.

**Algorithm**

A1 \[ \text{block} \leftarrow \text{new empty list} \]

A2 Parse event stream and process all step events. If the instruction is a `CALL` instruction, call the algorithm recursively and add the result to `block`. If the instruction is a `RET` instruction, go to step A3. Otherwise, generate a new instruction node and add it to `block`. If no more event is available, go to step A3.

A3 Return a new function node which contains the instructions from `block`
6 Evaluation

The completeness, performance, and correctness of the vmx86 interpreter has been evaluated with various programs and benchmarks. During the investigations, it showed that the lack of multithreading/multiprocessing, and the weak performance in comparison to native execution are existent. The goal of these benchmarks is to prove that the system is able to execute real world programs within reasonable time.

All benchmarks have been performed on the developer’s computer with an Intel i7-4770 CPU, 16GB RAM, and an Intel 520 series SSD with 240GB capacity. The software used, was Archlinux with:

- Linux kernel 4.17.11
- glibc 2.28
- GCC 8.2.1
- LLVM 7.0.1

For Graal, Truffle, and Sulong, the commit 357d66b11f6e8813489e811766718e0d4d7e68d92 from December 12th, 2018 was used.

For all benchmarks, custom benchmark harnesses were used. The details are described in the following sections.

6.1 Language Shootout Benchmarks

The language shootout benchmarks is a collection of toy programs, which implement various algorithms in multiple programming languages. Consequently, they are well suited for preliminary experiments about performance of certain language implementations.

The benchmark harness consists of a python script, which executes the program using QEMU, and vmx86, and measures the execution time of the program. The measured time values are written into a CSV file.

In vmx86 all benchmarks written in C and C++ were successfully executed, as long as no multithreading was required. The performance varied extensively, depending on the benchmark, so a slowdown of up to $250 \times$ was measured, which means that the program code has not yet been compiled by the JIT compiler. The performance could be improved by modifying the benchmark in order to execute the benchmark multiple times within a loop. Since the focus of interest is on SPEC CPU2006 benchmarks, because they represent the approved benchmarks, the language shootout benchmarks were not modified.

6.2 SPEC CPU2006

The SPEC CPU2006 benchmarks represent a selection of real world programs, and standardized inputs for these programs. They can be used to compare the system performance of various systems by executing the standardized program with the standardized input and measuring the elapsed time.

The following SPEC CPU2006 benchmarks can be successfully performed on vmx86:

Daniel Pekarek February 28, 2019
• **401.bzip2**: Compression
• **429.mcf**: Combinatorial optimization
• **445.gobmk**: Artificial Intelligence: go
• **456.hmmer**: Search gene sequence
• **462.libquantum**: Physics: quantum computing
• **464.h264ref**: Video compression
• **470.lbm**: Fluid dynamics
• **473.astar**: Path-finding algorithms

The slowdown in comparison to the native execution without interpreter has been quantified 3× up to 25×. This indicates a significant slower peak performance. In order to measure the peak performance, the SPEC benchmarks had been adapted, so that the main program is executed in a loop multiple times. Since the main program is executed multiple times, the Truffle call targets are called often enough, so that the JIT compiler compiles and optimizes the benchmarks.

Benchmarks were executed with 100 iterations. The arithmetic mean of the last 10 iterations is shown in the plot of figure 9. The benchmarks have been compiled with `gcc -O3` and GCC version 8.2.1 has been utilized.

The benchmark harness consists of multiple shell scripts, which compile the benchmarks with GCC and clang, generate LLVM bytecode, and execute the benchmarks with QEMU, vmx86, and Sulong. The modified SPEC benchmarks directly write the execution times to stdout in CSV format, and the shell scripts redirect stdout into individual CSV files. An R script was used to parse the CSV files and generate the plot, which can be seen in figure 9.

In the plot, each benchmark is shown in comparison to QEMU, and normalized to native execution. For every benchmark, the virtual memory has been configured, so that segfaults are ignored once, and in a second configuration segfaults are correctly handled. The resulting difference shows that with the help of integration of the virtual memory into HotSpot, a higher performance can be achieved in benchmarks with a load of memory accesses.

A big drawback is however that during the warmup phase, the slowdown is in the range of 250×, which is not shown in the plot.

Since a run of all supported SPEC CPU2006 benchmarks with the training data sets requires more than 24 hours, the decision was taken to execute a single run only. Therefore, not enough information for more expressive plots, like box plots, exist.

### 6.3 SubstrateVM

With SubstrateVM, Java programs can be compiled to native programs, which can be executed on vmx86, as long as no multithreading is involved. It is also possible to compile vmx86 with SubstrateVM, and execute vmx86 in vmx86 afterwards. The execution however, terminates as soon as Graal starts a compiler thread. This means that the program in the inner vmx86 interpreter must be terminated quickly, otherwise it terminates before it has been successfully executed.
The performance of vmx86 in vmx86 is very slow, since both instances of vmx86 run in interpreted mode at this stage. The outer instance of vmx86 compiles first, since the code of vmx86 must be interpreted in addition to the guest program. At this stage, the performance increase can be easily observed with console programs. If the guest program runs for a too long time period, the outer interpreter terminates, since the inner interpreter tries to start a compiler thread for the JIT compiler.

6.4 lame, groff, as, ld, objdump, xv, xpdf

The following programs were successfully tested and executed on the vmx86 interpreter:

- **lame**: MP3 encoder
- **groff**: GNU version of the troff typesetting system
- **as**: GNU assembler
- **ld**: GNU linker
- **objdump**: tool to inspect object code files
- **oggenc**: OGG encoder
- **cc1**: backend of GCC for the C language
- **xv**: image viewer for the X11 window system
- **xpdf**: PDF viewer for the X11 window system
• various programs from the coreutils project (ls, echo, printf, env, cat, date, dirname, basename, base32, base64, head, tail, md5sum, sha1sum, pwd, printenv, readlink, seq, sort, tac, uname, wc, whoami)

• various programs from the util-linux package (cal, hexdump, uuidgen, fdisk -l)

If the backend of GCC (cc1) is directly executed, it is possible to compile C source code to assembler code, assembler code to object code, and link the object code to an executable program, which can be executed in vmx86. Every tool in these steps can be executed on vmx86.

Thanks to rudimentary network functions, it is also possible to execute simple X11 programs, like the image viewer xv, or the PDF viewer xpdf within vmx86. The DISPLAY environment variable must point to an IP address, otherwise the X11 client library tries to communicate per shared memory object with the X11 server, which is currently not implemented in vmx86. Since X11 programs are interactive programs, the startup overhead is clearly visible, for instance, with xpdf the pages must be flipped 15-20x before the flipping of a page has almost the same speed as usually expected.

The X11 programs have been tested against an X11 server of TurboVNC in version 2.1.2. Testing with a different X11 server revealed problems with establishing the connection. The nature of this behavior could not be investigated due to a time constraint, however, it can be expected that the glitch is due to a possibly incorrect implementation of a syscall (bug).

6.5 CTF Examples

Capture the flag (CTF) competitions are hacker competitions with the goal to steal flags from programs by exploiting bugs/security vulnerabilities. CTF competitions often provide interesting programs as part of their challenges. Some of them use edge cases of the operating system, or runtime environment, like the standard C library. Some of these programs have been tested on vmx86. During these tests some problems in vmx86 have been identified and fixed, for instance writing on stdin.

CTF competitions in particular:

• hitcon2018: a challenge called Abyss consisted of three parts:
  1. a hypervisor based on KVM developed for this challenge,
  2. a custom kernel running on the hypervisor
  3. a userspace program, which is executed by the custom kernel, and implements an interpreter for a programmable stack machine.

Because the userspace program was a Linux program, it was tested with vmx86. Both, the standard execution path, as well as the shellcode injected by an exploit, were correctly executed by vmx86.

• hxp2018: in the challenge yunospace, it was the possibility to execute shellcode with exactly 9 bytes. This shellcode should output data, which is located directly after those 9 bytes, in which the shellcode itself is stored. The 9 bytes are not sufficient to store...
shellcode, which invokes the `write` syscall with an exact address and length argument. Instead, this shellcode is depending on undocumented behavior of the Linux kernel, as well as easily overlooked features of the processor.

In order to execute the shellcode, support for some special cases had to be implemented in the interpreter, like e.g. writing to stdin, as well as the `write` syscall with a too large length argument, which would force a memory access violation during the writing operation of the device. The interpreter vmx86 can execute the shellcode. A test case based on this shellcode has been added in the source code repository of vmx86.

At CTF competitions, vmx86 was not of great help so far, since the traces have been simply too big, and could not be analyzed, or certain functionality was missing in the interpreter, so it could not execute the program. The missing functionalities have been implemented after the competitions. Sometimes programs called functions, which cannot be easily implemented, like e.g. `fork`.

### 6.6 Truffle NFI

In Sulong, the vmx86 interpreter has been configured as a Truffle NFI backend, so native functions can be executed with vmx86. It was necessary to reverse the search order of libraries, so that symbols are searched in libsulong, and libc++ first, and consecutively in the libraries given by command line arguments. For libsulong and libc++, the standard NFI implementation of Truffle, using libffi is utilized, all the other libraries are handled by vmx86. The reason for this approach is that Sulong uses native memory, and problems exist with pointers in structures, and pointers to pointers, if vmx86 is used. Furthermore, the implementation of the NFI interface in vmx86 is incomplete, which means that certain operations, involving the `ENV` argument of Truffle NFI, which is used by libsulong, are currently not supported.

To perform the benchmarks, a shell script was executed, which invokes the benchmarks using vmx86 and Sulong. The benchmarks write the execution times to stdout in CSV format, which is redirected by the shell script into files. The files were analyzed with an R script, and the plot for figure 10 was generated. The benchmarks perform the operation of interest multiple times, in order to avoid measuring the warmup phase. Since the warmup phase of Sulong is quite long (approximately 10,000 iterations before functions are compiled), 100,000 warmup iterations were performed, followed by 100 measured iterations. The arithmetic mean of the 100 measured iterations is shown in the plot mentioned above.

The benchmarks reveal that due to the poor implementation of Truffle NFI in vmx86, the NFI overhead leads to a performance loss of approximately 50%.

### 6.6.1 OpenSSL

OpenSSL [18] is a cryptography library which supports low-level cryptographic operations as well as the SSL and TLS protocols. An appropriately modified Sulong can calculate SHA1 hash values, where the OpenSSL library (`libcrypto`) was executed by vmx86. The performance overhead, caused by Truffle NFI, is measurable, and is approximately 3 times slower than the direct execution in vmx86.
6.6.2 cairo

Cairo [19] is a general purpose library to generate graphics. The following benchmark directly uses cairo to generate a “basket” and save it in a PostScript file. Since no additional wrapper library is involved, every cairo function call leads directly to a Truffle NFI call. As depicted in figure 10, the performance is reduced to approximately 50%.

6.6.3 libpng

With libpng [20], graphics in PNG format can be compressed and decompressed. The library libpng can be used with the help of vmx86, directly from Sulong. However, some functions in libpng utilize e.g. FILE objects of the libc, which contain pointers. In order to call libpng functions from Sulong, it is necessary to develop a library, which provides file operation
functionality, like wrappers around `stdio` functions. This library must be located in the same address space as `libpng`, so that the pointers in `FILE` objects can be used without translation in `libpng`.

If PNG data is read, or written line by line, instead of using the more efficient functions, which read/write all lines of a PNG file in an array, then the limitations of `vmx86` are avoided. Since every `libpng` function is directly called from `Sulong`, without a wrapper library in between, the overhead for `Truffle NFI` calls occurs with every call of a `libpng` function. As shown in figure 10, the performance is reduced to approximately 50%.

6.6.4 libavcodec

The `libavcodec` library [21] provides functions for encoding and decoding of audio and video formats, and is used by media players, like VLC, and format converters like ffmpeg.

For the `libav` benchmark, a native library was developed, which exports a function to decode an MP3 file. This function is consequently called from `Sulong`, in order to convert an MP3 file into a WAV file, and resample it to 44.1 kHz. With this approach, all limitations of the `Truffle NFI` implementation are mitigated. However, only one single `Truffle NFI` call is executed. This leads to the fact that the overhead of the `Truffle NFI` call is neglectable, compared to the execution time of the function. It appears that the `Truffle NFI` does not incur any measurable overhead in this particular case.

6.7 Discussion

The benchmarks show that the performance is sufficient to execute real world programs within reasonable time. The `Truffle NFI` implementation is good enough to access a wide range of native libraries from other `Truffle` languages. However, the restrictions are a slow warmup phase and a high overhead for `Truffle NFI` calls, as well as slow performance compared to native execution.
7 Related Work

This section covers related projects which implement similar use cases. The main focus here is on the Truffle based Sulong, as well as QEMU, which utilizes its own JIT compiler.

7.1 Sulong

Sulong [3] is an existing interpreter based on Truffle, which can efficiently execute C and C++ programs. The interpreter reads LLVM bytecode, and therefore has more information about the program to be executed than the x86_64 interpreter, like e.g. functions and data types. The substitution of functions with other functions is easily supported, since the linking is implemented in Sulong. On machine code level, however, a lot less information is available, and dynamic linking is a lot more complex and, depending on the utilized C library. This is the reason why vmx86 uses the dynamic linker of the system and heuristics to reconstruct other information like function boundaries.

7.2 QEMU

QEMU [22] is an interpreter which can execute machine code. QEMU can execute Linux userspace programs similar to vmx86, as well as the complete machine, like e.g. a PC including emulated peripheral devices, and an operating system kernel like Linux, or Windows. The machine code is translated into an intermediate language, which is consequently translated by the tiny code generator (TCG) into machine code of the host computer. One single basic block is always handled at a time and immediately compiled during the first execution. Hence, there is almost no startup overhead, optimization across multiple basic blocks however, is not possible.

Syscall arguments are translated by QEMU in such a way that they are executed directly by the syscalls of the host system. Only structure layouts, pointer sizes, and magic values are exchanged. Syscalls are not emulated in QEMU, consequently, the userspace interpreter of QEMU can only be executed on a Linux host. The vmx86 interpreter instead, emulates syscalls in such a way that vmx86 can be executed on any operating system supported by Graal/Truffle. Code is compiled after an initial warmup phase, so that optimizations across multiple basic blocks can be performed by the compiler. During the startup phase, a significant slowdown in performance can be observed.
8 Conclusion / Future Work

This project is a feasibility study and demonstrates that the implementation of a x86_64 interpreter based on Graal/Truffle can be implemented within a reasonable time and effort. Since it is a feasibility study, this master thesis is the prerequisite for further investigation, development, and prototyping of enhanced features, which could also lead to a commercial product.

Most of the instructions and syscalls used by simple programs, have been implemented. Major limitations include the lack of support for multithreading, multiprocessing, limited poll, and select syscalls. The missing x87 FPU appears to be only occasionally a problem. All these features can be implemented at a later stage, however, some of those features, like multiprocessing, require significant implementation effort.

Testing on further platforms, like Windows, as well as systems with big endian memory layout (e.g. ppc64), would be of great interest. The specific challenges in this context are:

- **Windows**: this operating system uses a completely different API compared to the POSIX API or Linux syscalls. Some Linux functions do not have a direct equivalent in the Windows API. At this stage, it must be noted that certain functionality is also missing in the Java standard library today, including poll on files.

- **Big Endian Systems**: since x86 is a little endian system, it is necessary to translate memory accesses. Performance implications of this translation would be a further point of interest. The NFI backend will require special attention to support endianess conversion.

**Peak Performance**  The peak performance increase should be of great focus for further developments of the vmx86 interpreter. The heuristic for generating call targets would require again special attention. It is quite possible that a new heuristic shall be developed based upon the experience gained form the current investigations.

**Native Pointers in Truffle NFI**  A systematic approach, to solve the challenge of exchanging native pointers using Truffle NFI, is a suggested field of interest, since if this functionality can be implemented, the sandboxed vmx86 could always be used instead of the default Truffle NFI implementation using libffi, which will lead to a security increase.

**Startup Overhead**  Optimizing the startup overhead would lead to a perceived performance increase for interactive programs and applications. This would be the prerequisite for the utilization of interactive programs in vmx86.

**Security Features**  To increase the security of interpreted programs, it is suggested to develop security features, and implement them directly in the interpreter. One methodology could be deploying a shadow stack, in order to avoid return-oriented programming (ROP) type exploits on on any program. Another feature could be a taint analysis, to detect injected code during program execution. This could be used to prevent code injections, even if buffer overflows are present.
Debugging  It is suggested to investigate optimizations in the trace viewer, as well as the underlying file format. Consequently, this can lead to a very powerful debugging tool, with features that are not available, and supported in most other debugging solutions. One of these features would be tracing of memory values, which would reveal, when a value was last modified in memory. Since traces are recorded, which means they are stored, they can be individually inspected/analyzed, independent of the program, which means without re-running the program.

One thought would be developing additional types of analyses for trace files. For example, an analysis could find irregularities in a program run, like a buffer overflow.
vmx86: A Truffle-based Interpreter for x86 Binary Code

References


Acronyms

AOT  ahead of time. 6, 7
ASLR  address space layout randomization. 18
AST  abstract syntax tree. 16, 17, 26
auxv auxiliary vector. 19, 20
AVX  Advanced Vector Extensions. 11, 26, 31
BICD  binary-coded decimal. 10
CISC  complex instruction set computer. 8, 11
CLI  command-line interface. 12, 23–25, 34
CTF  Capture the flag. 41
ELF  executable and linkable format. 13–16, 18, 20, 29
FPU  floating-point unit. 11, 31, 46
GID  group ID. 20
JIT  just in time. 6, 15–17, 27, 38–40, 45
JNI  Java Native Interface. 22, 23, 25
JVM  Java Virtual Machine. 6, 15, 16, 22, 29, 30, 32
JVMCI JVM Compiler Interface. 15
LLVM  Low Level Virtual Machine. 6, 7, 16, 38, 39, 45
LRU  least recently used. 21
MMU  memory management unit. 13, 20–22, 30
NFI  Native Function Interface. 16, 29–31, 42–44, 46
ROP  return-oriented programming. 46
SIB  scale/index/base. 11, 12, 25
SSE  Streaming SIMD Extensions. 11, 12, 18, 26, 29
TCG  tiny code generator. 45
TLV  type/length/value. 33
UID  user ID. 20
vDSO virtual dynamic shared object. 19, 20
VM   virtual machine. 15
Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.