Masaryk University

Faculty of Informatics

Virtual File System in DIVINE 4

Bachelor's Thesis

Katarína Kejstová

Brno, Fall 2017

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Katarína Kejstová

Advisor: Mgr. Vladimír Štill

Acknowledgement

I would like to thank all the people in the ParaDiSe laboratory for the support and for the pleasant work environment. Especially, I would like to thank Vladimír Štill for advising this thesis and his willingness to correct all my misunderstandings. Additionally, I would like to express my gratitude to Petr Ročkai for helpful consultations and sketches he draws when I seem to be confused.

Finally, I would like to thank my family for surviving all the 50 selfies I have taken and sent them while writing this thesis.

Abstract

This work focuses on the integration of Virtual Filesystem (VFS) into DIVINE 4 by exploiting the concept of system calls. The VFS, which was present in DIVINE 3 as a set of functions, is now available to programs running in DIVINE 4 through system calls and library wrappers.

VFS is now part of DIVINE's operating system (DiOS), which additionally maintains functionality that system calls expect from the operating system: scheduling, interrupts and switches between kernel and user mode.

Subsequently, we use the fact that DiOS provides a syscall interface and introduce two additional modes, which differ in the way of handling system calls. In the first, system calls are propagated to the host operating system. This mode is called the *passthrough* mode of DiOS. This mode additionally captures information about the system call into an output file, which can be later used as an additional input for the second mode. The second mode simulates system calls using the captured data. We call this mode the *replay* mode.

The implementation of these modes together with the integration of VFS enables DIVINE to process additional programs. In particular, programs that use C functions operating over POSIX library or functions from POSIX library directly, programs that rely on the POSIX-compatible filesystem, or programs that require input-output functionality.

Keywords

DIVINE, operating system, file system, system calls, POSIX, C, C++, implementation, verification

Contents

1 Introduction

With the constant addition of requirements, together with the demand on the growth of performance of modern applications, there is a pressure on both software and hardware to increase their speed.

Seeing that the current clock speed of transistors cannot be increased without overheating, manufacturers produce architectures with multiple processors communicating directly through shared memory and hardware caches. In order to benefit from multiprocessors and speed up the computing we often need to include parallelism in our application, i.e. build application comprising of multiple threads that can be executed on multiple processing units.

In such shared memory applications, we must ensure that shared resources are protected from concurrent access. By shared resources, we mean not only memory allocated on the heap and global variables, but also opened files, pipes, and sockets. If multiple threads manipulate the same data at the same time in a non-atomic style, these threads can overwrite each other's changes which can lead to an inconsistent state.

As multi-core CPUs are prevalent today, multithreaded applications are relatively wide-spread and widely used. These applications are quite hard to test for absence of not only common errors but also concurrency problems, such as race conditions or deadlocks.

Testing, used commonly nowadays, seems not to be satisfying enough, as a run of a test is affected by the scheduling of threads. That mostly means that the issue may not be found by simply running tests, even if the error is present. The issue often appears in the case of errors occurring only in very specific situations, e.g. exact threads interleaving.

The use of formal methods brings a possibility to not only avoid problems in testing parallel applications but also to prove the correctness of a program. One of the methods of formal verification is model checking. DIVINE is a modern model checker able to verify multithreaded applications written in real-world programming languages such as C and C++.

It is common that these applications communicate with the user or surrounding environment during their execution, for example by opening a file, or by writing to the standard output. As DIVINE verifies real-world applications, it is essential to support, among others, verification of operations over the filesystem, as these operations are the main part of the communication between a user and a system, and therefore often the key part of a program.

In DIVINE 3, the operations over filesystem were supported, however, the access to the filesystem was uncontrolled, as the program called operations over the filesystem directly. The introduction of integrated operating system DiOS in DIVINE 4 brings the possibility to control access to the filesystem in a way similar to the traditional UNIX-like operating system, e.g. through system calls. The goal of this thesis is to examine this possibility and produce a mechanism to work with the filesystem safely with the use of DiOS.

The structure of this thesis is the following: Chapter 2 provides necessary fundamentals of crucial parts of UNIX-like operating systems. Subsequently, Chapter 3 provides a detailed description of DIVINE's components essential for our implementation. Chapter 4 gives detailed overview of the implementation of system calls in UNIX-like system and is followed with the implementation of syscalls in DIVINE in Chapter 5. Chapters 6 and 7 introduce new modes of DiOS (namely *passthrough* and *replay*) by applying different approach to system calls handling. Finally, Chapter 8 summarizes our contribution and discusses possible future work.

2 Preliminaries

The structure of the filesystem used in DIVINE is inspired by the UNIX filesystem architecture, thereupon I will introduce this type of filesystem more precisely.

2.1 File System in UNIX

A filesystem is an aggregation of data structures and methods over them that an operating system employs to keep track of the state of files on a disk.

Moreover, a filesystem is a hierarchical storage of data defined by the specific operations provided over the data, such as creation, deletion or mounting. Filesystems contain files, directories and associated control information [10].

File types specific for the UNIX filesystem are regular files, directories and symbolic links, named pipes, and special files. To obtain properties of any file type, we use the system call *stat()*.

```
$ stat file.c
  File: 'file.c'
  Size: 199             Blocks: 8          IO Block: 4096
regular file
Device: 801h/2049d      Inode: 5116012     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/  kejsty)   Gid: ( 1000/
kejsty)
Access: 2017-02-03 14:53:41.597784860 +0100
Modify: 2017-01-26 23:18:55.873924732 +0100
Change: 2017-01-26 23:18:55.873924732 +0100
Birth: -
```

Figure 2.1: Regular file properties presented as output of invoking the stat utility on a regular file file.c. This utility internally uses the system call *stat()* and reproduces its output.

Every regular file has several properties, as shown in Fig. 2.1, among other permissions or size of the file. The most important detail seen in this figure is the *Inode*. The inode, often called file metadata, is an object, holding every single information needed by the kernel to operate with the file or directory. This number uniquely identifies each file within a filesystem, as there can be more links to one file in address hierarchy (e.g. hard links).

2.2 Virtual File System in UNIX

The Virtual File System, sometimes called *Virtual File Switch* or *VFS*, is the subsystem of the Linux kernel that implements the file and file system-related interface provided to user-space programs. It brings the ability for different filesystems to communicate via the standard Unix system calls, regardless of the structure of the filesystem or the underlying physical medium. Among other things, it means that we are able to use these system calls to copy or move data from one file system to another, which was not possible in older operating systems, such as DOS, without adding special tools.

The key idea behind the VFS consists of the introduction of a common file model capable of representing all filesystems. This model strictly mirrors the file model provided by the traditional Unix filesystem [4], so each filesystem implementation must provide a set of functions that converts its physical structure into the VFS's common file model in order to respect the model and become interoperable with other filesystems.

2.3 Interrupts

As the behaviour of interrupts is similar to the behaviour of system calls, we will mention few characteristics of them and subsequently compare system calls to interrupts in Sec. 4.1.

An interrupt is a mechanism of the operating system that provides an opportunity for the hardware to send a request for an attention from the operating system.

Operating systems need to distinguish between interrupts in order to react adequately to them. This is done by associating each interrupt with a unique value, often called the interrupt request line. Some of these are standardised for Linux, for example, value 1 stands for keyboard interrupt [3], others are dynamically selected.

Knowing which interrupt was raised, allows us to be able to handle it. For this purpose, kernel has a routine called the interrupt handler. In Linux, these handlers are typically C functions.

These functions are always invoked by the kernel as a response to an interrupt, and in order to be quick and uninterrupted, they are executed in special (atomic) context.

3 DIVINE and its File System

3.1 DIVINE

" DIVINE is a modern, explicit-state, LLVM-based LTL model checker that follows the standard automata-based approach to explicit-state model checking" [12]. Based on the LLVM toolchain, it can verify programs written in multiple real-world programming languages, including C and C++ [2]. The current version is DIVINE 4, released in January 2017.

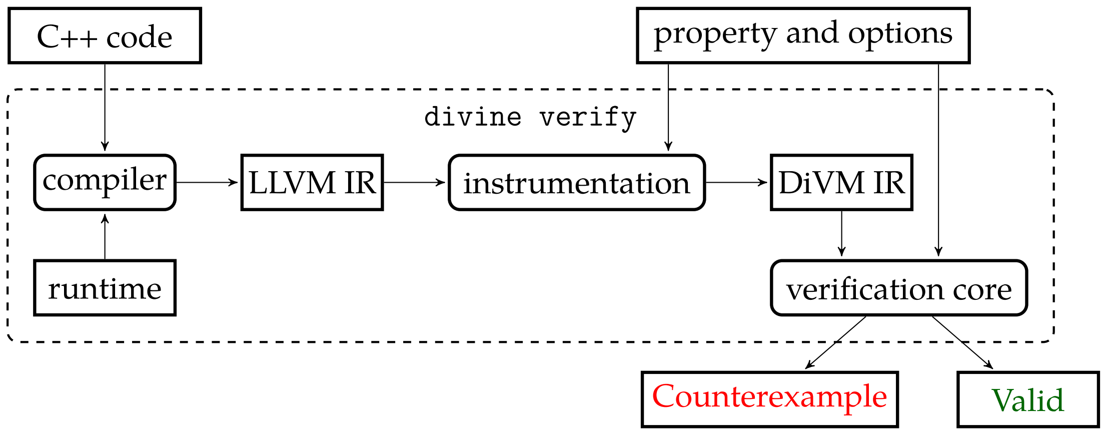

Figure 3.1: The workflow of DIVINE's verification of a given C++ program. [1]

As shown in Fig. 3.1, DIVINE takes a C or C++ program and translates it with Clang compiler, a C language family frontend for LLVM [13], into LLVM bitcode. This bitcode is consequently instrumented. This generated LLVM bitcode is then linked with the DIVINE-provided runtime libraries and verified against a given property.

When an error is discovered, DIVINE produces a readable counterexample, providing information about the erroneous run and a backtrace of the program from the state the error occurred in.

DIVINE can operate in two main modes: *run* and *verify*. The *run* mode investigate only single execution of a given program in some (random) execution order, i.e. random interleaving of threads together with the random selection of nondeterministic choices (such as the success of a memory allocation). Nevertheless, all behaviours such as memory safety or assertions are still checked. The *verify* mode, on the other hand, considers additionally all thread interleavings and all nondeterministic choices which are essential in case of parallel programs.

Additionally, DIVINE possesses another mode of operation: *sim*. This mode behaves similarly to a debugger, however, in addition to common debuggers, it allows the user to go back in the execution.

DIVINE operates over a state space of a program - an oriented graph where vertices represent states of the program (e.g. values in memory locations and registers). To visualise the state space, we can use the `divine draw` command. The output of such a command can be seen in the attached archive (`parallel.pdf`).

With the release of DIVINE 4, this model checker contains its own operating system DiOS, running in a DIVINE's virtual machine.

3.1.1 Virtual Machine

Programs in DIVINE are executed by a virtual machine (DIVINE VM or DiVM for short). DiVM provides the functionality needed to execute an entire operating system, DiOS.

The state of DiVM consists of two parts, a set of control registers and memory held in the heap.

Control registers, holding pointers or integers, are described by enum *ControlRegister*, and can be manipulated via the hypercall *control*, providing functionality such as setting or getting a value from a register. Each register has its own purpose.

Register *Flags*, used as a bitfield, represents a register holding several flags. As VM has many types of flags providing various functionalities and not all of them are important for the purpose of this thesis, I will only introduce some of them.

For example, setting flags *Accepting*, *Error* and *Cancel* indicates whether an edge in a state space should be considered as accepting, erroneous or should be abandoned, and thus not became part of the state space.

Setting the *Mask* flag will result in disallowing interrupts, hence the program will execute atomically until resetting the register back to 0. Flag *Interrupt*, on the other hand, gives rise to an instantaneous interrupt, in case the flag for masking is not set. Flag *KernelMode* indicates whether DiVM is running in kernel or user mode.

Moreover, there are two user registers, *User1* and *User2*. The first of them, i.e. *User1* is notable mainly because the instance of a system call is stored here before an interrupt is invoked, as will be shown in Chap. 5.

The memory in DiVM is represented as a directed graph, where nodes stand for objects, and edges represent pointers in these objects. DiVM holds for every object not only its data but also a shadow in the form of an image of the entire address space, e.g. additional memory associated with this object. This way the VM has complete knowledge about an object in the program, including information about uninitialised bytes. This feature is important in the implementation of syscall passthrough as will be shown in Chap. 6.

As DIVINE cannot make use of a real operating system, an own POSIX-like operating system was implemented DIVINE 's operating system, DiOS in short. This way, the responsibility of scheduling threads is moved from the VM into the OS layer of DiOS, which simplifies the VM.

3.1.2 Operating System

DiOS provides to the verified program a subset of POSIX interfaces that should cover the requirements of a typical user-level program [2], as programs usually require a presence of services such as memory management or thread handling in the environment they run on.

Mainly, DiOS serves to cover requirements of a typical program, and to provide scheduler routine, determining the order of threads execution, i.e. deciding which of its states will be executed.

DiOS further supports fault handling and error tracing. For the purposes of this thesis, the most important component of DiOS is its own file system that supplies tools for loading a snapshot of another filesystem, and subsequently, uses it for simulation of operations over it.

DiOS runs in a *virtual* mode in the meaning that all interactions with the outside world are simulated. In the other words, no real I/O operation is allowed, due to the fact that verification of parallel applications in DIVINE consists of investigating all possible thread interleavings.

The root cause for this requirement is that these I/O actions have consequences in the outside world that cannot be replayed or undone. This restriction applies also to the host's file system. DiOS can not operate over the host's file system, as one interleaving could affect the result of another.

3.2 File System

DIVINE's Virtual file system is a POSIX-compatible filesystem implementation provided by DiOS. The effect of any operations performed by the VFS is not propagated to the host [2].

The file system in DIVINE is to a high degree inspired by UNIX filesystem idea. The main concept is that, on the abstract level, everything is a file.

The concrete objects representing file types are a regular file, write-only file, standard input, pipe, stream and datagram socket, and directory. These objects are held by an inode object, so every file type object is always strictly determined by its inode. By contrast, two different paths can be bound to the same inode object. This approach very closely corresponds to that used in UNIX.

After opening a regular file or pipe, the filesystem creates a descriptor for the file or pipe, i.e. over its inode. DIVINE implements special descriptors for a regular file, pipe, socket and directory, as descriptors provide operations such as read, write, close or offset, which behave differently for various descriptors.

Above this filesystem implementation stands a virtual filesystem (VFS), taking a responsibility for the filesystem and providing basic operations over the parts of the filesystem (i.e. files or sockets) needed by system calls.

Supported are not only basic functions like *open, read, write, close*, which are needed for the most basic communication and for basic C printing functions such as *printf*, and C++ operators on streams, but also more complex functions as *stat, fstat* or *lseek* and functions for sockets such as *socketpair, getsockname, bind or connect*.

DIVINE's VFS currently supports a significant part of functions defined in header files *unistd.h*, *sys/socket.h* and *sys/stat.h*.

The initial state of the filesystem is empty directory root, e.g. with path /, holding only . and .. directories. Every file that the program operates with must be created during the program execution. Herewith, DIVINE can only verify programs operating with files created during the execution of the program.

We dealt with this inconvenience by bringing the ability to create a snapshot of a directory selected by the user, and therefore use already constructed files instead of the necessity to create them first. DIVINE currently supports capturing regular files, directories and symlinks. The process of detecting hard links is done by inspecting the inode number of files.

4 System Calls in Linux

In this chapter, we will explore how traditional operating system from the POSIX family implements system calls and explain some parts important for syscalls, such as kernel mode, user space libraries or errno.

4.1 Syscalls

A syscall stands for a system call. The system call is the fundamental interface between an application and the Linux kernel [9]. This interface provides access to the hardware, that is fully controlled and also handles messages between the application and the kernel.

System calls can be apprehended as the layer between the hardware and user-space processes. Placing an additional layer between user-space, that is the application, and the hardware has several benefits. This way applications do not have to be concerned with the hardware specifications such as the type of disk or the filesystem. This layer also watches our permissions to requested actions, so an application is not permitted to directly perform any instruction it wants without permissions, which preserves stability and secure access to system resources. Last but not least, this approach makes programs more portable.

The concept of system calls in Linux has a lot in common to that of the interrupts as a syscall also represents an (user-space) request of a kernel service.

4.2 Wrapper Routines

As system call is an interface directly communicating with the kernel, it shall not be called directly from an application. Instead, it is used through a wrapper function (or wrapper routine), whose only purpose is to issue a system call.

These wrapper routines can be found on typical Unix system in `libc` standard C library. In Linux, this library is provided by the `glibc` the GNU C Library [6]. Usually, each system call has its own corresponding wrapper routine [4]. Hence, these routines can be seen as the interface of an operating system.

Moreover, each system call on POSIX-like systems has internally assigned a specific number, the system call number. The number together with `libc` generic function `syscall` allows us to invoke the system call directly, without possessing the knowledge of the exact syscall function name.

Let us inspect the behaviour of the wrapper routines and system calls in a simple example.

Having a simple C program using the `libc` function *write* (see Fig. 4.1), we will use *ltrace* tool to examine the internal interpretation of it. The *ltrace* utility displays a set of user-space calls of a program [5].

```c
#include <unistd.h>

int main() {
    write(1, "Hello World", 11);
    return 0;
}
```

Figure 4.1: A simple program in C programming language that writes "Hello World" on the standard output (the standard output has the file descriptor 1).

```
ltrace -S ./example
...initialization
__libc_start_main(0x400526, 1, 0x7ffd80c3d778, 0x400550 <unfinished ...
write(1, "Hello␣World", 11 <unfinished ...>
SYS_write(1, "Hello␣World", 11Hello World)
<... write resumed> )
SYS_exit_group(0 <no return ...>
```

Figure 4.2: The output of the ltrace utility investigating the program from Fig. 4.1.

The output of the trace utility (from Fig. 4.2) demonstrate that the wrapper routine for *write* (form the `libc` library) calls the syscall implementation *SYS_write*. The implementation is provided by the operating system and thus is executed in the kernel mode.

4.3 Kernel Mode

In the majority of operating systems, the CPU spends its time in two different modes, kernel mode and user mode. In the user mode, the executed program has no ability to directly access the hardware or reference memory not owned by the program without asking for the permission from the operating system first. This mode is present during basic interaction with the system or during the run of an application.

On the other hand, in kernel mode, the code has complete and unrestricted access to the hardware, so every CPU instruction can be executed and any memory address referenced in this mode. The kernel mode is generally reserved for the lowest-level and trusted functions of the operating system. System calls are executed in the kernel mode. They are expected to be quick.

In these wrapper functions, everything is set up, registers are assigned the right values and only then the syscall is invoked in order to be fast and effective. Crashes in the kernel mode are mostly catastrophic, hence the permissions checking in the kernel is not being omitted.

4.4 Errno

The return value of a system call symbolises the result of the executed call. Although the meaning of this value depends mostly on the called system call, the return value -1 usually indicates that kernel was unsuccessful in accomplishing the given request. In this case, not only -1 is returned but also the *errno* variable is set to a specific *error code*.

The meaning of the error code corresponds to a symbolic error name. The list of them is specified by POSIX.1-2001 and C99 standard [8]. On that account, the errno provides a valuable feedback to the user program about the origin of the problem, such as file not found or insufficient permissions.

5 System Calls in DIVINE

The fact that system calls should be executed in the kernel mode and thus have different permissions together with an assurance that the execution of a system call will not be interleaved with the execution of another thread need to be reflected in our implementation.

To achieve the previously mentioned, the process of handling a syscall consists of two main constituents: the wrapper routines (Sec. 5.1) and the actual implementation of system call (Sec. 5.2) which runs in DiOS kernel. To do this, DiOS provides helpers which facilitate entering the kernel mode.

5.1 Wrapper Routines

```
SYSCALL(write, ssize_t, (int _1, const void * _2, size_t
    _3 ))
```

Figure 5.1: One line from file *syscall.def* providing declaration of write syscall for subsequent generation of components needed for the system call

```
#define SYSCALL(name, schedule, returnType, arguments)
returnType name arguments {
   returnType retval;
    __dios_syscall(_DiOS_SC::name, rv, _1, _2, _3, _4, _5
       , _6);
    return retval;
}
```

Figure 5.2: Simplified macro for generating wrapper routines implementations for declarations from *syscall.def*. Their form is shown in Fig. 5.1.

The wrapper routines in DIVINE are in fact implementations of functions from headers such as `unistd.h`, `fcntl.h`, `unistd.h`, etc.

Their main mission is to alert the operating system (DiOS) that a request by a verified program for a system call was raised.

Our approach identifies system calls by numbers. To cause the number to be compatible during the whole process we declared an enum (*_DiOS_SC*). This enum provides symbolic value to identify given system call inside DiOS according to its name. The compatibility is achieved by both DiOS and routines using the same enum.

As shown in Fig. 5.2, the implementation of wrapper routines is generic. Every wrapper routine call invokes the DiOS function *__dios_syscall* (Sec. 5.1.1) with particular system call identifier from *_DiOS_SC* and forwards input arguments.

This call to *__dios_syscall* exploits the fact that input arguments are called _1, _2, etc., as shown in Fig 5.1. Additionally, a class `Pad` is defined and seven objects, _1, _2,.. _7, are globally instantiated. This way, the first arguments are taken from the input arguments of the routine and the rest is filled with the instances of Pad. This solution is possible as the `Pad` instances are global.

Let us assume that verified program calls function *write* with some input arguments. The effect will be following:

Firstly, the wrapper routine generated from a prototype (Fig. 5.1) for *write* is executed. This wrapper routine calls *__dios_syscall* function (Fig. 5.1.1) and expects it to call the system call implementation with number defined by enum *_DiOS_SC* as write. The input parameters provided to this function together with mentioned enum suffice to identify the syscall inside DiOS.

After performing this function, the value received in the variable *retval* is returned. This way, the routine pushes the responsibility of invoking the system call to the DiOS.

5.1.1 Function `__dios_syscall`

The *__dios_syscall* is a DiOS routine of preparation of environment before invoking an interrupt to jump into DiOS (by calling the function *trap*).

The Fig. 5.3 shows simplified implementation of this function. The first two parameters, system call number and return value, were already mentioned. The remaining argument *rest* holds parameters for the system call.

```
void __dios_syscall( int syscode, void* ret, Args rest )
   {
   uintptr_t flag =  __vm_control( Get, Flags);

    __dios_trap( syscode, ret,  rest );

    __vm_control( Set, Flags, flag | Interrupted  );
}
```

Figure 5.3: Simplified implementation of the *__dios_syscall* function, taken from DIVINE source codes.

The run of the function is the following: At first, the current flags are stored into variable *flag* to recover them later. Subsequently, the function *trap* is called. This method creates a new system call instance and invokes it, as shown in Sec. 5.1.2. In the end, the call of *__vm_control* on the last line in Fig. 5.3 restores the DiOS with original flags stored in the variable *flag* together with an additional flag for an interrupt.

The *flags* are restored in order to save a program executed under mask (e.g. during executing an atomic section) even after executing a system call. The flag *Interrupted* is added to assure that the program will be interrupted so the scheduler can be called. This approach is necessary since the system call could have globally observable effect and thus thread rescheduling is necessary.

5.1.2 Function `trap`

Generally, the `trap` function creates an instance of system call holding data of syscall demanded by a program and triggers an interrupt. This is done as follows (Fig. 5.4):

First, an instance internally representing a syscall is created and afterwards filled with the necessary data, namely *syscode*, representing the DiOS internal system call number given by the enum *_DiOS_SC*; *ret*, the return value of the syscall; and *args*, input arguments for the given syscall (can be apprehended as a tuple).

Afterwards, the created instance is stored into *User1* register, a DiOS internal register, and an interrupt is initiated. The interrupt invocation means that DiOS musts run the scheduler. One important

```
static void trap(int syscode, returnType retval, Args
    rest) noexcept {
  _DiOS_Syscall instance;
    instance._syscode = ( _DiOS_SC ) syscode;
    instance._ret = retval;
    instance._args = rest;

    __vm_control( Set, User1, &instance );
    __vm_control( Set, Flags, Interrupted );
    __vm_control( Set, Flags, Mask );
}
```

Figure 5.4: Simplified implementation of function *trap*. Taken from DIVINE source code.

part of the reschedule process is entering the kernel mode by setting the flag *KernelMode*, second is recognising the instance of system call in the *User1* register and consequent handling of it.

5.2 Implementation of System Calls

The procedure of handling a syscall operating over a filesystem consists of calling the implementation using the interface provided by VFS while still entering the kernel mode.

For DiOS to have a simple access to system calls, an additional table of pointers to their implementations is held. In this table, the position of an implementation is determined by the value from the *_DiOS_SC* enum.

These implementations are mostly unmodified, i.e. from DIVINE 3, with small changes such as elimination of interrupts before accessing the filesystem (as this is now handled by DiOS), or a propagation of the information about a demand to reschedule in case of blocking system calls.

Additionally, several functions working directly with the filesystem were not system calls. These functions have been completely rewritten as a combination of existing system calls.

For example, the function *send* in not a system call directly but can be implemented as a special case of system call *sendto*.

5.3 Further changes to VFS

After integrating the VFS some additional changes in this implementation has been made. The changes came as an output of an attempt to modularize DiOS together with the additional functionality of DiOS mentioned in next chapters.

This reimplementation includes some cleaning and changes in the structure of DiOS together with the system calls implementation. Withal, it also showed several disadvantages of the old implementation. For example, the table of system calls has been global and thus other parts of DiOS could simply access it. The access to the internal information of current DiOS's state by executed system calls shows up to be undesirable likewise. The main change is, however, introduction of configurations in DiOS (Sec. 5.3.1).

Furthermore, there were changes in the form of declarations (Fig. 5.1) to capture more information about the work with memory by syscalls, as this information becomes important.

Also, a `Syscall` class was introduced that provides handling of system calls and encapsulates the table of the implementations. Additionally, this class handles arguments unpacking process, needed due to the new form of declarations.

There were more participants working on this reimplementation and modularization. All further chapters build on this new implementation.

5.3.1 Configuration

The configuration in DiOS is a hierarchic structure of classes providing several functions for the operating system.

They build on the inheritance hierarchy, i.e. if a function not implemented by a subclass is called, the call is propagated to its superclasses. The hierarchy is build as follows: on the bottom is always a `BaseContext` (see Sec. 5.3.2). Above this class stands some additional classes. Finally, there is a `VFS` (see Sec. 5.3.3) implementing many system calls over the DiOS's filesystem.

This configuration including VFS is the default one. Other configurations will be demonstrated in next chapters.

5.3.2 Class `BaseContext`

To build configuration hierarchy, we need to start from the bottom. Therefore, we created a class `BaseContext`, that simulates this "bottom". This class implements every system call from the declaration file. However, the design is tricky: every such implementation informs the user that this system call has not been actually implemented and causes a fault in DiOS.

The result of this is that if a subclass (which will apply to every class mentioned in this thesis) implements such a system call, thanks to the inheritance hierarchy, the implementation of the subclass hides the implementation of the `BaseContext`.

Additionally, the VFS does not have to implement all system calls from declaration file, as some of them are provided by other parts of the configuration.

5.3.3 Class `VFS`

Since there is intention to support more types of syscall executions (as will be shown in next chapters) we decided to implement them as separate configurations.

For this purpose, we implemented a class VFS (standing for Virtual File System), that will fulfil the system calls implementations over DIVINE's filesystem (Sec, 5.3.3). This way, the implementation of system calls are not functions but methods of the VFS class.

The access to these methods is controlled, as a system call can be executed only by parts possessing the current configuration. Furthermore, the VFS fully represents the interface of the DIVINE's filesystem.

5.4 Example

In this section, an example of a program using system call will be shown.

We are able to verify this program after the integration of VFS into DIVINE. To reproduce it, please follow the instructions for installation and usage of DIVINE provided in Sec. A.2.2.

Let us consider the simple program shown below. This program creates a file (line 5), writes into it (line 7) and then closes it (line

8). Subsequently, this file is opened again (line 10), a read from it is invoked (line 12), the loaded string is compared with the inserted (line 13) and the file is closed again (line 14).

```
 1  int main() {
 2      char buf[8] = {};
 3      int fd = open( "test", O_WRONLY | O_CREAT, 0644 );
 4      assert( fd >= 0 );
 5      assert( write( fd, "tralala", 7 ) == 7 );
 6      assert( close( fd ) == 0 );
 7  
 8      fd = open( "test", O_RDONLY );
 9      assert( fd >= 0 );
10      assert( read( fd, buf, 7 ) == 7 );
11      assert( strcmp( "tralala", buf ) == 0 );
12      assert( close( fd ) == 0 );
13      return 0;
14  }
```

As the VFS is empty at the beginning (thus there will be no problem with creating the file on line 5), the program is valid. The result of DIVINE's verify is: **error found: no**.

More examples are available in the archive of this thesis.

6 Syscall Passthrough

Presently, DIVINE supports many syscalls and there is an intention to implement more of them in the future. However, DIVINE is actually incapable of having every one of them fully implemented since some syscalls build on operations which are unsupported in DIVINE (such as communication with outside world). This applies for example to sockets which communicate with the Internet.

We implemented a mechanism called *syscall passthrough* through which DiOS gains the ability to execute system calls in the host operating system (i.e. the operating system DIVINE is running on).

The motivation behind the *syscall passthrough* from DiOS to the host's kernel is to provide the ability to use almost all system calls and by that enlarge the number of programs that DIVINE can verify.

The idea of passing the calls through the DIVINE (shown in Fig. 6.1) is that a syscall invoked by a program is noticed by DiOS and then forwarded directly to the host's kernel via `libc` function `syscall`. Subsequently, the result of the syscall is returned back to DiOS to process it.

This approach brings the ability to run a program operating with almost any syscall, however, it is impossible to use this technique for the *verify* mode of DIVINE, as there would be a problem with thread interleaving already mentioned in Sec. 3.2. Nevertheless, this problem can be partially mitigated by recording and replaying syscalls, as will be shown in Sec. 7.

Of course, this approach is not *safe* for every syscall (as propagation of several system calls can have unexpected results, for example, system call *exit* kills the whole run of DIVINE, not only the program) and makes use of the fact that DIVINE can explore just a single run of the given program using the *run* mode.

6.1 The `libc` Function `syscall`

When a program executes a syscall, the process running the program must provide the kernel among others with a parameter named *system call number*. This number identifies this call but is architecture specific. For example, the x86_64 architecture provides 322 system calls and

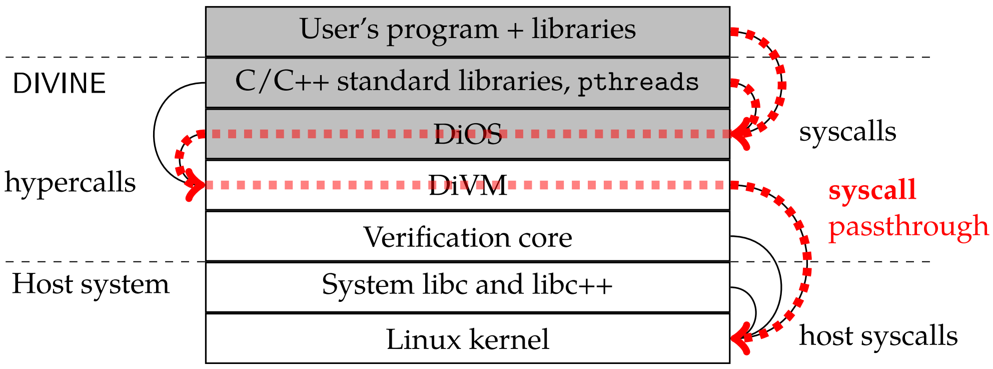

Figure 6.1: A scheme showing the propagation of a system call request through DIVINE into the host system [11].

```
long syscall(long number, ...);
```

Figure 6.2: Declaration of `libc` function `syscall` from header file *sys/syscall.h*. Taken from the Linux manual pages [9].

the x86 architecture provides 358 system calls [15] as it contains also additional x32-specific system calls.

In DIVINE, we implemented the support passthrough of x86_64 architecture system calls.

6.2 Passthrough Support in DiVM

```
__vm_syscall( __dios::_VM_SC_write,
      _VM_SC_Out | _VM_SC_Int64, &written,
      _VM_SC_In | _VM_SC_Int32, fd,
      _VM_SC_In | _VM_SC_Mem, length, message,
      _VM_SC_In | _VM_SC_Int64, length );
```

Figure 6.3: An example of invocation of `__vm_syscall` that performs a `read` syscall passthrough.

To implement *syscall passthrough* we need an additional functionality from DIVINE, which will allow the call of the `libc` function `syscall` (see Fig. 6.1) on the host system.

This cannot be done by DiOS directly as DiOS has no ability to call functions outside the verified program. Thus, every operation outside DIVINE must be realised through DiVM. For this reason, DiVM provides an hypercall `__vm_syscall` which serves as the wrapper of `libc` function `syscall`. An example of usage is shown in Fig. 6.3.

The function `__vm_syscall` takes at least three parameters. First is the system call number, second is the type of an output variable, and the third is the address of the output variable, i.e. an address for storing the result of the propagated system call. Additional arguments are optional but must follow several rules:

- If the passed argument contains a value to be read by a syscall, the additional flag _VM_SC_Out must be passed. Otherwise, if we pass an address that expects to be filled, we provide flag _VM_SC_In

- If the passed argument is a scalar of size 32 or 64 bits, the flag must contain _VM_SC_32 or _VM_SC_64. On the other hand, if it is not a scalar, i.e. array, structure or a pointer to a structure, we pass flag _VM_SC_Mem

- If we pass the flag _VM_SC_Mem, an additional argument providing the length of the valid memory must be involved, so the memory can be correctly copied between DIVINE and the host system.

Arguments are passed in the order flag; argument.

This way, the `__vm_syscall` does not have to possess any system-specific knowledge and thus can be more general and usable for any system call. However, DiOS has the responsibility of providing correct argument's metadata.

Recall the example from Fig. 6.3. The argument written is an *address* for the return value of the propagated syscall, so the flag _VM_SC_Out is added. The return value of `libc` function write is of type `size_t`, which has usually 64 bits, thus the argument also contains flag _VM_SC_64.

On the other hand, the argument message, which represent the second argument of the `libc` function write, is of a type *const char **. Hence, flags passed for this argument are _VM_SC_In, because it *contains* a value, together with the flag _VM_SC_Mem, as the type represents an array of characters, not a scalar. Since the _VM_SC_Mem flag is added, before passing the argument message, an additional argument is included, specifically the length of the message array.

The function `__vm_syscall` returns an integer representing the value of errno after executing the system call on the host system.

6.3 Storing System Call Numbers

For simpler representation of system calls in the kernel, every syscall has its own unique number. In Fig. 6.3 the first passed argument was introduced as system call number even though a variable __dios::_VM_SC_write occurs in the given example.

To support system calls in DIVINE, we need to hold their identifiers in easily accessible way. Due to the fact that every system call has its own wrapper which knows the name of the system call, we only need to map names to the identifiers. This is done by declaring an enum *_VM_SC* that fully supplies this functionality.

To create this enum, we use simple macro together with a declaration, that is a modification of a syscall table [14] from Linux. As an enum is a set of named values, we can interpret it as a mapping of syscall names to their identifiers.

For example, the value of _VM_SC_write is 1, as the file *systable.def* contains a line SYSCALL(1,write).

6.4 Implementation

For the purposes of integrating *syscall passthrough* into DIVINE, we created a class `Passthrough` that is part of new DiOS's configuration. The configuration has to differ from the default one, as concurrently supporting the VFS and *syscall passthrough* does not make sense.

For this reason, we added a new mode: *passthrough mode*. This mode is obtained by replacing `VFS` for `Passthrough` in configuration and does not support the *verify* mode of DIVINE.

As the number of system calls is large (typically a few hundred), we decided to generate these wrappers using a macro 6.4.1 and C++ templates.

6.4.1 Automatic Generation of Implementations

```
SYSCALL( write, ssize_t,
        (int _1, COUNT(const void *) _2, size_t _3 ))
```

Figure 6.4: One line from syscalls declaration.

```
#define SYSCALL( name, schedule, returnType, arguments )
returnType name arguments {
    returnType returnValue;
    int outType = _VM_SC_Int32 | _VM_SC_Out;
    tuple input = {_VM_SC_ ## name, outType, returnValue
       };

    errno = parse(input, _1, _2, _3, _4, _5, _6, _7);
    return returnValue;
}
#include <sys/syscall.def>
#undef SYSCALL
```

Figure 6.5: A simplified pseudo code of the macro for generating *syscall passthrough* functions inside `Passthrough` class.

As already mentioned, the declarations of system calls had slightly changed. The new form is shown in Fig. 6.4. Some input arguments contain additional metadata, namely *Count, Mem, Out, Struct*, which helps to parse the arguments for the `__vm_syscall` function.

The Fig. 6.5 shows generation of wrapper functions for system calls in *passthrough mode*, which is made automatically. Thanks to the structure of the declarations, functions can be created relatively simply.

The process of this generation is shown in Fig. 6.5.

The generated function creates an input tuple which will be propagated through the whole parsing process and will progressively build a full input tuple for the `__vm_syscall`. The variable *outType* contains metadata for the return value demanded by the `__vm_syscall`.

Finally, there is a call of a function *parse*. This call exploits the fact, that input arguments are called _1, _2, etc., as shown in Fig 6.4. Additionally, a class `Pad` is defined, and seven objects, _1, _2,.. _7 are globally instantiated. This way, the first arguments are taken from the input arguments of the function, and the rest is filled with the instances of Pad.

6.4.2 Parsing the Input Arguments

The mission of *parse* function is to process input arguments given to the wrapper of `Passthrough` into a form suitable for the `__vm_syscall`, shown in Fig. 6.3. In other words, the *parse* function builds suitable *input tuple*.

This process consists of two parts: resolving a type of the given argument together with assigning the right flags, which is done by the *process* function and gives us a tuple *addition* which is subsequently inserted into this tuple into the *input tuple*.

This way, the tuple is progressively built as the arguments are processed and the result is always appended to the end of the tuple. At the end, the *input tuple* contains all arguments needed by `__vm_syscall`.

As mentioned in Sec. 6.4.1, the parse function is called with input arguments followed by instances of the class Pad. This design allows us to stop the recursion after receiving an instance of the Pad as it corresponds to the end of inputs arguments belonging to the syscall. This behaviour is achievable thanks to the template specialisation in C++.

7 Syscall Replay

The implementation of *syscall passthrough* in Chap. 6 brings another ability. It creates the possibility to capture values of arguments, together with results of every invoked system call, while passing the system call to the host operating system. This is achievable as our implementation knows input argument types including their size (including buffers) and also the type of the output argument.

We take advantage of this by storing all this data into an output file and subsequently allowing to load this file back into DIVINE and use it for a new *replay mode* of DiOS. This mode allow us to use knowledge captured in *passthrough mode* (which supports only *run mode*) of DIVINE, in an new *replay mode* of DiOS supporting also the *verify* mode of DIVINE.

This mode brings two main advantages. First, it allows us to "step backward in time", i.e. derive a reversible debugger from DIVINE by recording a run of a program in *passthrough* mode and subsequently using the output for a *simulation* run in *replay* mode.

Second, as already mentioned, it allows us to (partially) verify programs that primarily depend on actions different than system calls, as their behaviour often leads to invoking the same syscall sequence. DIVINE is thus free to explore all interleavings of such a programs, which was not possible in the *passthrough* mode.

7.1 Implementation

We created a `Replay` class and by that introduced another configuration mode of DiOS: *replay mode*. Before the start of verification process, the input file (currently expected to be `passthrough.out`) is read and parsed (Sec. 7.1.1). This way, there is no communication with the outside world during verification, all actions are just simulated in accordance with the captured data. After the parse, system calls are stored as a set of `currently available` syscalls (Sec. 7.1.2).

Therefore, when a system call is invoked by a program, the Replay class looks at the currently available system calls. If there is a match (Sec. 7.1.3), the system call takes place. From to point of view of a program, the behaviour is indistinguishable from *syscall passthrough*.

Otherwise, the branch of execution is cancelled by setting the *Cancel* flag via the hypercall *control*. This causes the edge leading to the current state to be abandoned and thus it will not become a part of the state space.

7.1.1 Parsing

As DiOS knows during an execution of a syscall in the *passthrough mode* the metadata need by `__vm_syscall`, we simply use this knowledge to store these data using a backwards compatible design. We decided to store the output file in a binary form which brings discomfort in case of reading by a human but is comfortable for parsing.

The first notable detail is that the storing must occur *after* the system call execution, as the value of some arguments of output type (having flag *_VM_SC_Out*) are filled by the system call itself. This is no problem, as the parse method of the class `Passthrough` works recursively so we collect the data during backtracking from the recursion.

For primitive types we store only the value, for a structure or a buffer, we save the size followed by the content. The structure of the output for a system call is:

SYSCALL:{system call number}{the length of the input arguments plus the output}{the content}{the value of errno}

This way, the process of the parsing is straightforward. After reading the leading "SYSCALL" and the system call number, the length of content is read and thus we know the size of a segment of characters representing the input arguments. At the end, the errno is read. Recall, that as the size of system call number as well as the size of errno is known considering they are scalars.

If any error occurs during the parsing, the process is interrupted and an error is raised with a suitable message.

The second notable detail is that the write into the file cannot be done with the *write* syscall as this would cause an infinite recursion (this write would need to write metadata about itself, etc.). This is solved by calling the `__vm_syscall` directly.

7.1.2 Storing Recorded Sysycalls

As already mentioned, after parsing the input, system calls are stored as a set of *currently available* syscalls for the given state. The system call itself is held as a class encapsulating a number (the system call number), a content (the input data for a system call) and the errno. Additionally, this class holds a set of its successors. The successor of a system call is a syscall, which cannot be executed before executing its predecessor.

Therefore, after the simulation of a system call is finished, all its successors are added into the set of *currently available*.

Currently, every system call has only one successor which is the subsequently parsed system call. The form of implementation allows us to add heuristics for selection of successors in the future.

7.1.3 Matching

An invocation of a system call by a program verified or run in the *replay mode* is allowed only if it occurs in the *currently available* set. For this reason, we develop a matching algorithm that compares the system call to those currently available.

First, we consider only the system call number, as the process of deeper check is more complicated. If there is no match in these numbers, we refuse to continue which causes cancellation of the executed branch.

Afterwards, we perform a deeper check. The system call is suitable to some from the *currently available* set if it satisfies the following conditions: the system call number matches and for every argument it is true that if the argument is of input type (recall the flag *_VM_SC_In*), its value matches the captured. If the argument is of output type (i.e. flag *_VM_SC_Out*), it is filled with captured data.

If any of input types of arguments do not match, the system call is marked as not a match. If there is no deeper match found we refuse to continue and cancel the branch.

7.2 Example

We wrote small programs designed mainly to test the *passthrough* and the *replay* mode:

- **rw** which creates, writes to and reads from a file
- **rw-parallel** in which one thread reads from and second writes to a file
- **network** with simple HTTP client that opens a TCP IP connection to a fixed IP address, executes a request and writes out the result.

We tested these programs in both modes. They demonstrate our approach works. These mentioned programs can be found in the archive. To show the thread interleaving in the *replay* mode, we generated a graph (this is already built-in feature in DIVINE) that shows the state space of the program **network**. This graph can be found also in the archive of this thesis in *parallel.pdf*.

8 Conclusion

We integrated the Virtual File System (VFS) from DIVINE 3 into DIVINE 4. The access of VFS in DIVINE 3 was uncontrolled, as a verified program called the implementations of system calls directly.

One part of the integration was to change this approach. This was successfully done by inserting DiOS in-between the program and the implementation. This way, the program's access to the filesystem is fully controlled by DiOS. Additionally, the implementation of functions called directly by the program simulates the functionality of system call wrappers that commonly occur in UNIX-based operating systems.

The success of this integration is shown in the example provided in Sec. 5.4 together with several tests produced for purposes of testing this integration.

Furthermore, we implemented two new modes of DiOS configuration: the *passthrough* mode and the *replay* mode. The current modes of DiOS are the following:

- *virtual*, or the default mode, in which every interaction with the outside world is simulated. The filesystem is part of the state of the verified program. This mode may be used in all modes of DIVINE.

- *passthrough* mode that, with the help of the `__vm_syscall` function, executes system calls in the host operating system. This way, DIVINE interacts with the environment, thus this mode is usable only for the *run* mode.

- *replay* mode, which reproduces (simulates) syscalls using the trace recorded in the *passthrough* mode, but does not interact with the host OS. Thereupon, this mode can be again used in all modes of DIVINE.

The proposed integration of VFS is already included in the current version of DIVINE. The integration of new modes of DiOS are planned in the future, however, a modified distribution od DIVINE that contains all the changes described in this thesis is provided in the appendix.

The work described in this thesis is additionally described in the article *From Model Checking to Runtime Verification and Back* [7] which was sent to the *The 17th International Conference on Runtime Verification*.

8.1 Future work

The *replay* mode has several limitations in our current implementation.

As some syscalls are blocking (their execution blocks thread until they finish), the run of a program in *passthrough* mode may cause a deadlock of the model checker if the executed syscall blocks and waits for another thread (since there is only one thread executing everything in DIVINE).

The possible solution would be to convert system calls to non-blocking (e.g. add the O_NONBLOCK flag) or to create a separate thread for every system call propagated to the host operating system.

An additional limitation is that our current implementation covers only a subset of POSIX. For example, the `libc` function `gethostbyname` is not implemented, which means that many interesting programs could not be run or verified.

A Archive and Manual

A.1 Archive Structure

The archive submitted with this thesis contains the sources of the thesis itself and a file **thesis.zip** whose unzipping gives us two directories.

C programs for the *passthrough* mode and the pdf of generated graph from the *replay* mode of DiOS can be found in the directory **examples**. As not all changes mentioned in this thesis were a part of the current version of DIVINE at the time of the submission of this thesis, the comprising a modified distribution of DIVINE 4 is in directory **divine4-passthrough**.

The implementation of system calls is included primarily in the **runtime/dios/filesystem** subdirectory.

A.2 DIVINE

A.2.1 Instalation

In order to compile DIVINE, it is necessary to have an up-to-date Linux distribution with a C++14 capable compiler. The compilation was tested with GCC 5.3.0 and Clang 3.7.0, both using libstdc++ 5.3.0 as the C++ standard library and CMake [16]

After unzipping the provided **thesis.zip**, please follow these instructions:

```
cd divine4-passthrough
make
make install
```

A.2.2 Running provided examples

A program can be verified by DIVINE using the **divine verify** command, by simple typing

```
divine verify program.c
```

To run the program in the *passthrough* mode, use following command:

```
divine run -o configuration:passthrough -o nofail:malloc program.c
```

The program will be executed in *passthrough* under the assumption that malloc does not fail. Additionaly, a syscall trace of the execution will be stored in a file `passthrough.out`. This output file can be subsequently used to verify the program using *replay* mode by typing:

```
divine verify -o configuration:replay -o nofail:malloc program.c
```

To visualise the generated state space of the program in the *replay* mode (illustrating that for a parallel program new interleavings occur), the `draw` mode of DIVINE can be used:

```
divine draw -o configuration:replay -o nofail:malloc program.c
```

Please refer to **divine cc –help** for more details, or read the manual [2].

Bibliography

[1] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of c and c++ with divine 4. Submitted, `https://divine.fi.muni.cz/2017/divine4/paper.pdf`, 2017.

[2] Jiří Barnat, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DIVINE 4: Model checking for everyone. `http://divine.fi.muni.cz/`, 2016.

[3] Randolph Bentson. *Inside Linux: A Look at Operating System Development*. Specialized System Consultants, Inc., Seattle, WA, USA, 1997.

[4] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[5] Rodrigo Rubira Branco. Ltrace internals. In *Proceedings of the Linux Symposium*, page 41. Linux Symposium, Ottawa, Ontario, Canada, 2007. `https://www.kernel.org/doc/mirror/ols2007v1.pdf`.

[6] Free Software Foundation, Inc. *The GNU C Library Reference Manual*, 2.25 edition, 1993–2017. `http://www.gnu.org/software/libc/manual/pdf/libc.pdf`.

[7] Katarína Kejstová, Petr Ročkai, and Jiří Barnat. From model checking to runtime verification and back. Submitted, `https://divine.fi.muni.cz/2017/passthrough/paper.pdf`, 2017.

[8] Linux. *ERRNO(3) Linux Programmer's Manual*, 4.11 edition, 2016. `http://man7.org/linux/man-pages/man3/errno.3.html`.

[9] Linux. *SYSCALLS(2) Linux Programmer's Manual*, 4.08 edition, 3 2016. `http://man7.org/linux/man-pages/man2/syscalls.2.html`.

[10] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[11] Vladimír Štill. presentations. `https://github.com/vlstill/presentations/blob/master/rh_1705/pres.md`, 2017.

[12] Vladimír Štill, Petr Ročkai, and Jiří Barnat. *DIVINE: Explicit-State LTL Model Checker*, pages 920–922. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[13] The University of Illinois. Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[14] Linus Torvalds. linux. `https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl`, 2017.

[15] Filippo Valsord. Searchable linux syscall table for x86 and x86_64. `https://filippo.io/linux-syscall-table/`, 2016.

[16] Vladimír Štill. LLVM Transformations for Model Checking. Master's thesis, Masarykova univerzita, Fakulta informatiky, Brno, 2016. `http://is.muni.cz/th/373979/fi_m/`.