

# Reproducible Execution of POSIX Programs with DiOS\*

Petr Ročkai, Zuzana Baranová, Jan Mrázek,  
Katarína Kejstová, and Jiří Barnat

Faculty of Informatics, Masaryk University  
Brno, Czech Republic  
{xrockai,xbaranov,xmrazek7,xkejstov,barnat}@fi.muni.cz

**Abstract.** In this paper, we describe DiOS, a lightweight model operating system which can be used to execute programs that make use of POSIX APIs. Such executions are fully reproducible: running the same program with the same inputs twice will result in two exactly identical instruction traces, even if the program uses threads for parallelism.

DiOS is implemented almost entirely in portable C and C++: although its primary platform is DiVM, a verification-oriented virtual machine, it can be configured to also run in KLEE, a symbolic executor. Finally, it can be compiled into machine code to serve as a user-mode kernel.

Additionally, DiOS is modular and extensible. Its various components can be combined to match both the capabilities of the underlying platform and to provide services required by a particular program. New components can be added to cover additional system calls or APIs.

The experimental evaluation has two parts. DiOS is first evaluated as a component of a program verification platform based on DiVM. In the second part, we consider its portability and modularity by combining it with the symbolic executor KLEE.

## 1 Introduction

Real-world software has a strong tendency to interact with its execution environment in complex ways. To make matters worse, typical environments in which programs execute are often extremely unpredictable and hard to control. This is an important factor that contributes to high costs of software validation and verification. Even the most resilient verification methods (those based on testing) see substantial adverse effect.

In automated testing, one of the major criteria for a good test case is that it gives reliable and reproducible results, without intermittent failures. This is especially true in the process of debugging: isolating a fault is much harder when it cannot be consistently observed. For this reason, significant part of the effort involved in testing is spent on controlling the influence of the environment on the execution of test cases.

---

\* This work has been partially supported by the Czech Science Foundation grant No. 18-02177S and by Red Hat, Inc.

The situation is even worse with more rigorous verification methods – for instance, soundness of verification tools based on dynamic analysis strongly depends on the ability to fully control the execution of the system under test.

In this paper, we set out to design and implement a small and sufficiently self-contained model operating system that can provide a realistic environment for executing POSIX-based programs. Since this environment is fully virtualised and isolated from the host system, program execution is always fully reproducible. As outlined above, such reproducibility is valuable, sometimes even essential, in testing and program analysis scenarios. Especially dynamic techniques, like software model checking or symbolic execution, rely on the ability to replay interactions of the program and obtain identical outcomes every time.

## 1.1 Contribution

The paper describes our effort to implement a compact operating system on top of existing verification frameworks and virtual machines (see Section 3). Despite its minimalist design, the current implementation covers a wide range of POSIX APIs in satisfactory detail (see also Section 4.3). The complete source code is available online,<sup>1</sup> under a permissive open-source licence. Additionally, we have identified a set of low-level interfaces (see Section 2) with two important qualities:

1. the interfaces are lightweight and easy to implement in a VM,
2. they enable an efficient implementation of complex high-level constructs.

Minimal interfaces are a sound design principle and lead to improved modularity and re-usability of components. In our case, identification of the correct interfaces drives both *portability* and *compactness of implementation*.

Finally, the design that we propose improves robustness of verification tools. A common implementation strategy treats high-level constructs (e.g. the `pthread` API) as primitives built into the execution engine. This ad-hoc approach often leads to implementation bugs which then compromise the soundness of the entire tool. Our design, on the other hand, emphasises clean separation of concerns and successfully reduces the amount of code which forms the trusted execution and/or verification core.

## 1.2 Design Goals

We would like our system to have the following properties:

1. Modularity: minimise the interdependence of the individual OS components. It should be as easy as possible to use individual components (for instance `libc`) without the others. The kernel should likewise be modular.

---

<sup>1</sup> <https://divine.fi.muni.cz/2019/dios/>

2. Portability: reduce the coupling to the underlying platform (verification engine), making the OS useful as a pre-made component in building verification and testing tools.
3. Veracity: the system should precisely follow POSIX and other applicable standardised semantics. It should be possible to port realistic programs to run on the operating system with minimal effort.

Since the desired properties are hard to quantify, we provide a qualitative evaluation in Section 5. To demonstrate the viability of our approach, we show that many UNIX programs, e.g. `gzip` or a number of programs from the GNU `coreutils` suite can be compiled for DiOS with no changes and subsequently analysed using an explicit-state model checker.

### 1.3 Related Work

Execution reproducibility is a widely studied problem. A number of tools capture *provenance*, or history of the execution, by following and recording program’s interactions with the environment, later using this information to reproduce the recorded execution. For instance, ReproZip [4] bundles the environment variables, files and library dependencies it observes so that the executable can be run on a different system. Other programs exist, that instead capture the provenance in form of logs [7], or sometimes more complex structures – provenance graphs in case of ES3 [5].

SCARPE [7] was developed for Java programs and captures I/O, user inputs and interactions with the database and the filesystem into a simple event log. The user has to state which interactions to observe by annotating the individual classes that make up the program, since the instrumentation introduces substantial overhead, and recording all interactions may generate a considerable amount of data (for example, capturing a large portion of the database).

Another common approach to dealing with the complexity of interactions with the execution environment is *mocking* [14, 15]: essentially, building small models of the parts of the environment that are relevant in the given test scenario. A *mock object* is one step above a stub, which simply accepts and discards all requests. A major downside of using mock objects in testing is that sufficiently modelling the environment requires a lot of effort: either the library only provides simple objects and users have to model the system themselves, or the mock system is sophisticated, but the user has to learn a complex API.

Most testing frameworks for mainstream programming languages offer a degree of support for building mock objects, including mock objects which model interaction with the operating system. For instance the `pytest` tool [11] for Python allows the user to comfortably mock a database connection. A more complex example of mocking would be the filesystem support in Pex [10], a symbolic executor for programs targeting the .NET platform. KLEE is a symbolic executor based on LLVM and targets C (and to some degree C++) programs with a different approach to environment interaction. Instead of modelling the file system or other operating system services, it allows the program to directly

interact with the host operating system, optionally via a simple adaptation layer which provides a degree of isolation based on symbolic file models.

This latter approach, where system calls and even library calls are forwarded to the host operating system is also used in some runtime model checkers, most notably Inspect [19] and CHES [16]. However, those approaches only work when the program interacts with the operating system in a way free from side effects, and when external changes in the environment do not disturb verification.

One approach to lifting the non-interference requirement is *cache-based* model checking [13], where initially, the interactions with the environment are directly performed and recorded in a cache. If the model checker then needs to revisit one of the previous states, the cache component takes over and prevents inconsistencies from arising along different execution paths. This approach is closely related to our *proxy* and *replay* modes (Section 4.1), though in the case of cache-based model checking, both activities are combined into a single run of the model checker. Since this approach is focused on piece-wise verification of distributed systems, the environment mainly consists of additional components of the same program. For this reason, the cache can be realistically augmented with process checkpointing to also allow backtracking the environment to a certain extent.

Finally, standard (offline) model checkers rarely support more than a handful of interfaces. The most widely supported is the POSIX threading API, which is modelled by tools such as Lazy-CSeq [6] and its variants, by Impara [18] and by a few other tools.

## 2 Platform Interface

In this section, we will describe our expectations of the execution or verification platform and the low-level interface between this platform and our model operating system. We then break down the interface into a small number of areas, each covering particular functionality.

### 2.1 Preliminaries

The underlying platform can have different characteristics. We are mainly interested in platforms or tools based on dynamic analysis, where the program is at least partially interpreted or executed, often in isolation from the environment. If the platform itself isolates the system under test, many standard facilities like file system access become unavailable. In this case, the role of DiOS is to provide a substitute for the inaccessible host system.

If, on the other hand, the platform allows the program to access the host system, this easily leads to inconsistencies, where executions explored first can interfere with the state of the system observed by executions explored later. For instance, files or directories might be left around, causing unexpected changes in the behaviour<sup>2</sup> of the system under test. In cases like those, DiOS can serve to

---

<sup>2</sup> If execution A creates a file and leaves it around, execution B might get derailed when it tries to create the same file, or might detect its presence and behave differently.

insulate such executions from each other. Under DiOS, the program can observe the effects of its actions along a single execution path – for instance, if the program creates a file, it will be able to open it later. However, this file never becomes visible to another execution of the same program, regardless of the exploration order.

Unfortunately, not all facilities that operating systems provide to programs can be modelled entirely in terms of standard C. To the contrary, certain areas of high-level functionality that the operating system is expected to implement strongly depend on low-level aspects of the underlying platform. Some of those are support for thread scheduling, process isolation, control flow constructs such as `setjmp` and C++ exceptions, among others. We will discuss those in more detail in the following sections.

## 2.2 Program Memory

An important consideration when designing an operating system is the semantics of the memory subsystem of its execution platform. DiOS is no exception: it needs to provide a high-level memory management API to the application (both the C `malloc` interface and the C++ `new/delete` interface). In principle, a single flat array of memory is sufficient to implement all the essential functionality. However, it lacks both in efficiency and in robustness. Ideally, the platform would provide a memory management API that manages individual memory objects which in turn support an in-place resize operation. This makes operations more efficient by avoiding the need to make copies when extra memory is required, and the operating system logic simpler by avoiding a level of indirection.

If the underlying platform is memory-safe and if it provides a supervisor mode to protect access to certain registers or to a special memory location, the remainder of kernel isolation is implemented by DiOS itself, by withholding addresses of kernel objects from the user program. In this context, memory safety entails bound checks and an inability to overflow pointers from one memory object into another.

## 2.3 Execution Stack

Information about active procedure calls and about the local data of each procedure are, on most platforms, stored in a special *execution stack*. While the presence of such a stack is almost universal, the actual representation of this stack is very platform-specific. On most platforms that we consider,<sup>3</sup> it is part of standard program memory and can be directly accessed using standard memory operations. If both reading and modifications of the stack (or stacks) is possible, most of the operations that DiOS needs to perform can be implemented without special assistance from the platform itself. Those operations are:

---

<sup>3</sup> The main exception is KLEE, where the execution stack is completely inaccessible to the program under test and only the virtual machine can access the information stored in it. See also Section 3.2.

- creation of a new execution stack, which is needed in two scenarios: isolation of the kernel stack from the user-space stack and creation of new tasks (threads, co-routines or other similar high-level constructs),
- stack unwinding, where stack frames are traversed and removed from the stack during exception propagation or due to `setjmp/longjmp`.

Additionally, DiOS needs a single operation that must be always provided by the underlying platform: it needs to be able to transfer control to a particular stack frame, whether within a single execution stack (to implement non-local control flow) or to a different stack entirely (to implement task switching).

In some sense, this part of the platform support is the most complex and the hardest to implement. Fortunately, the features that rely on the above operations, or rather the modules which implement those features, are all optional in DiOS.

## 2.4 Auxiliary Interfaces

There are three other points of contact between DiOS and the underlying platform. They are all optional or can be emulated using standard C features, but if available, DiOS can use them to offer additional facilities mainly aimed at software verification and testing with fault injection.

*Indeterminate values.* A few components in DiOS use, or can be configured to use, values which are not a priori determined. The values are usually subject to constraints, but within those constraints, each possible value will correspond to a particular interaction outcome. This facility is used for simulating interactions that depend on random chance (e.g. thread scheduling, incidence of clock ticks relative to the instruction stream), or where the user would prefer to not provide specific input values and instead rely on the verification or testing platform to explore the possibilities for them (e.g. the content of a particular file).

*Nondeterministic choice.* A special case of the above, where the selection is among a small number of discrete options. In those cases, a specific interface can give better user experience or better tool performance. If the choice operator is not available but indeterminate values are, they can be used instead. Otherwise, the sequence of choices can be provided as an input by the user, or they can be selected randomly. The choice operation is used for scheduling choices and for fault injection (e.g. simulation of `malloc` failures).

*Host system call execution.* Most POSIX operating systems provide an indirect system call facility, usually as the C function `syscall()`. If the platform makes this function accessible from within the system under test, DiOS can use it to allow real interactions between the user program and the host operating system to take place and to record and then replay such interactions in a reproducible manner.

### 3 Supported Platforms

In the previous section, we have described the target platform in generic, abstract terms. In this section, we describe 3 specific platforms which can execute DiOS and how they fit with the above abstract requirements.

#### 3.1 DiVM

DiVM [17] is a verification-oriented virtual machine based on LLVM. A suite of tools based on DiVM implement a number of software verification techniques, including explicit-state, symbolic and abstraction-based model checking. DiVM is the best supported of all the platforms, since it has been specifically designed to delegate responsibility for features to a model operating system. All features available in DiOS are fully supported on this platform.

In DiVM, the functionality that is not accessible through standard C (or LLVM) constructs is provided via a set of *hypercalls*. These hypercalls form the core of the platform interface in DiOS and whenever possible, ports to other platforms are encouraged to emulate the DiVM hypercall interface using the available platform-native facilities.

#### 3.2 KLEE

KLEE [3] is a symbolic executor based on LLVM, suitable both for automated test generation and for exhaustive exploration of bounded executions. Unlike DiVM, KLEE by default allows the program under test to perform external calls (including calls to the host operating system), with no isolation between different execution branches. Additionally, such calls must be given concrete arguments, since they are executed as native machine code (i.e. not symbolically). However, if the program is linked to DiOS, both these limitations are lifted: DiOS code can be executed symbolically like the rest of the program, and different execution branches are isolated from each other.

However, there is also a number of limitations when KLEE is considered as a platform for DiOS. The two most important are as follows:

1. KLEE does not currently support in-place resizing of memory objects. This is a design limitation and lifting it requires considerable changes. A workaround exists, but it is rather inefficient.
2. There is only one execution stack in KLEE, and there is no support for non-local control flow. This prevents DiOS from offering threads, C++ exceptions and `setjmp` when executing in KLEE.

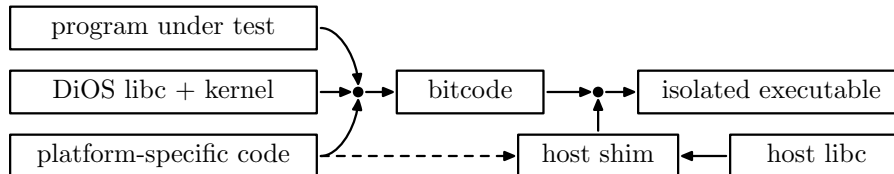
Additionally, there is no supervisor mode and hence no isolation between the kernel and the user program. However, in most cases, this is not a substantial problem. Non-deterministic choice is available via indeterminate symbolic values, and even though KLEE can in principle provide access to host syscalls, we have

not evaluated this functionality in conjunction with DiOS. Finally, there are a few minor issues that are, however, easily corrected:<sup>4</sup>

1. KLEE does not support the `va_arg` LLVM instruction and relies on emulating platform-specific mechanisms instead, which are absent from DiOS,
2. it handles certain C functions specially, including the `malloc` family, the C++ `new` operator, the `errno` location and functions related to assertions and program termination; this interferes with the equivalent functionality provided by DiOS `libc`, and finally
3. global constructors present in the program are unconditionally executed before the entry function; since DiOS invokes constructors itself, this KLEE behaviour also causes a conflict.

### 3.3 Native Execution

The third platform that we consider is native execution, i.e. the DiOS kernel is compiled into machine code, like a standard user-space program, to execute as a process of the host operating system. This setup is useful in testing or in stateless model checking, where it can provide superior execution speed at the expense of reduced runtime safety. The user program still uses DiOS `libc` and the program runs in isolation from the host system. The platform-specific code in DiOS uses a few hooks provided by a shim which calls through into the host operating system for certain services, like the creation and switching of stacks. The design is illustrated in Figure 1.



**Fig. 1.** Architecture of the native execution platform.

Like in KLEE, the native port of DiOS does not have access to in-place re-sizing of memory objects, but it can be emulated slightly more efficiently using the `mmap` host system call. The native port, however, does not suffer from the single-stack limitations that KLEE does: new stacks can be created using `mmap` calls, while stack switching can be implemented using host `setjmp` and `longjmp` functions.<sup>5</sup> The host stack unwinding code is directly used (the DiVM platform code implements the same `libunwind` API that most POSIX systems also use).

<sup>4</sup> A version of KLEE with fixes for those problems is available online, along with other supplementary material, from <https://divine.fi.muni.cz/2019/dios/>.

<sup>5</sup> The details of how this is done are discussed in the online supplementary material at <https://divine.fi.muni.cz/2019/dios/>.



On the other hand, non-deterministic choice is not directly available. It can be simulated by using the `fork` host system call to split execution, but this does not scale to frequent choices, such as those arising from scheduling decisions. In this case, a random or an externally supplied list of outcomes are the only options.

## 4 Design and Architecture

This section outlines the structure of the DiOS kernel and userspace, their components and the interfaces between them. We also discuss how the kernel interacts with the underlying platform and the user-space libraries stacked above it. A high-level overview of the system is shown in Figure 2. The kernel and the user-mode parts of the system under test can be combined using different methods; even though they can be linked into a single executable image, this is not a requirement, and the kernel can operate in a separate address space.

Like with traditional operating systems, kernel memory is inaccessible to the program and libraries executing in user-mode. In DiOS, this protection is optional, since not all platforms provide supervisor mode or sufficient memory safety; however, it does not depend on address space separation between the kernel and the user mode.

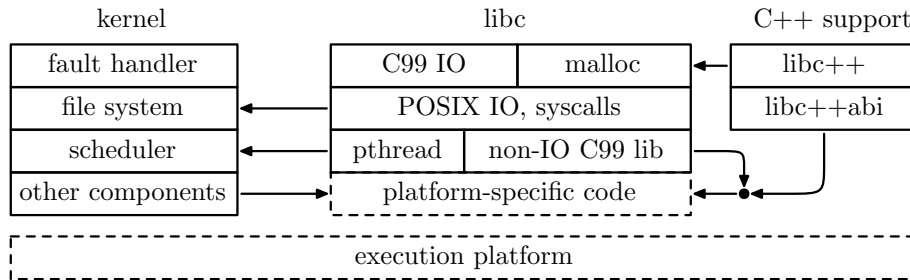


Fig. 2. The architecture of DiOS.

### 4.1 Kernel Components

The decomposition of the kernel to a number of components serves multiple goals: first is resource conservation – some components have non-negligible memory overhead even when they are not actively used. This may be because they need to store auxiliary data along with each thread or process, and the underlying verification tool then needs to track this data throughout the execution or throughout the entire state space. The second is improved portability to platforms which do not provide sufficient support for some of the components, for

instance thread scheduling. Finally, it allows DiOS to be reconfigured to serve in new contexts by adding a new module and combining it with existing code.

The components of the kernel are organised as a stack, where upper components can use services of the components below them. While this might appear to be a significant limitation, in practice this has not posed substantial challenges, and the stack-organised design is both efficient and simple. A number of pre-made components are available, some in multiple alternative implementations:

*Task scheduling and process management.* There are 4 scheduler implementations: the simplest is a *null* scheduler, which only allows a single task and does not support any form of task switching. This scheduler is used on KLEE. Second is a synchronous scheduler suitable for executing software models of hardware devices. The remaining two schedulers both implement asynchronous, thread-based parallelism. One is designed for verification of safety properties of parallel programs, while the other includes a fairness provision and is therefore more suitable for verification of liveness properties.

In addition to the scheduler, there is an optional process management component. It is currently only available on the DiVM platform, since it heavily relies on operations which are not available elsewhere. It implements the `fork` system call and requires one of the two asynchronous schedulers.

*POSIX System Calls.* While a few process-related system calls are implemented in the components already mentioned, the vast majority is not. By far the largest coherent group of system calls deals with files, directories, pipes and sockets, with file descriptors as the unifying concept. A memory-backed filesystem module implements those system calls by default.

A smaller group of system calls relate to time and clocks and those are implemented in a separate component which simulates a system clock. The specific simulation mode is configurable and can use either indeterminate values to shift the clock every time it is observed, or a simpler variant, where ticks of fixed length are performed based on the outcome of a nondeterministic choice.

The system calls covered by the filesystem and clock modules can be alternately provided by a *proxy* module, which forwards the calls to the host operating system, or by a *replay* module which replays traces captured by the *proxy* module.

*Auxiliary modules.* There is a small number of additional modules which do not directly expose functionality to the user program. Instead, they fill in support roles within the system. The two notable examples are the *fault handler* and the *system call stub* component.

The fault handler takes care of responding to error conditions indicated by the underlying platform. It is optional, since not all platforms can report problems to the system under test. If present, the component allows the user to configure which problems should be reported as counterexamples and which should be ignored. The rest of DiOS also uses this component to report problems detected by the operating system itself, e.g. the `libc` uses it to flag assertion failures.

The stub component supplies fallback implementations of all system calls known to DiOS. This component is always at the bottom of the kernel configuration stack – if any other component in the active configuration implements a particular system call, that implementation is used. Otherwise, the fallback is called and raises a runtime error, indicating that the system call is not supported.

## 4.2 Thread Support

One of the innovative features of DiOS is that it implements the POSIX threading API using a very simple platform interface. Essentially, the asynchronous schedulers in DiOS provide an illusion of thread-based parallelism to the program under test, but only use primitives associated with coroutines – creation and switching of execution stacks (cf. Section 2.3).

However, an additional external component is required: both user and library code needs to be instrumented with *interrupt points*, which allow thread preemption to take place. Where to insert them can be either decided statically (which is sufficient for small programs) or dynamically, allowing the state space to be reduced using more sophisticated techniques.<sup>6</sup> The implementation of the interrupt point is, however, supplied by DiOS: only the insertion of the function call is done externally.

The scheduler itself provides a very minimal internal interface – the remainder of thread support is implemented in user-space libraries (partly `libc` and partly `libpthread`, as is common on standard POSIX operating systems). Even though the implementation is not complete (some of the rarely-used functions are stubbed out), all major areas are well supported: thread creation and cancellation, mutual exclusion, condition variables, barriers, reader-writer locks, interaction with `fork`, and thread-local storage are all covered. Additionally, both C11 and C++11 thread APIs are implemented in terms of the `pthread` interface.

## 4.3 System Calls

The system call interface of DiOS is based on the ideas used in *fast system call* implementations on modern processors.<sup>7</sup> A major advantage of this approach is that system calls can be performed using standard procedure calls on platforms which do not implement supervisor mode.

The list of system calls available in DiOS is fixed:<sup>8</sup> in addition to the kernel-side implementation, which may or may not be available depending on the active

---

<sup>6</sup> In DIVINE [1], a model checker based on DiVM, interrupt points are dynamically enabled when the executing thread performs a visible action. Thread identification is supplied by the scheduler in DiOS using a platform-specific (hypercall) interface.

<sup>7</sup> For instance, on contemporary x86-64 processors, this interface is available via the `syscall` and `sysret` instructions.

<sup>8</sup> The list of system calls is only fixed relative to the host operating system. To allow the system call proxy component to function properly, the list needs to match what is available on the host. For instance, `creat`, `uname` or `fdatasync` are system calls on Linux but standard `libc` functions on OpenBSD.

configuration, each system call has an associated user-space C function, which is declared in one of the public header files and implemented in `libc`.

The available system calls cover thread management, sufficient to implement the `pthread` interface (the system calls themselves are not standardised by POSIX), the `fork` system call, `kill` and other signal-related calls, various process and process group management calls (`getpid`, `getsid`, `setsid`, `wait`, and so on). Notably, `exec` is currently not implemented, and it is not clear whether adding it is feasible on any of the platforms. The thread- and process- related functionality was described in more detail in Section 4.2.

Another large group of system calls cover files and networking, including the standard suite of POSIX calls for opening and closing files, reading and writing data, creating soft and hard links. This includes the `*at` family introduced in POSIX.1 which allows thread-safe use of relative paths. The standard BSD socket API is also implemented, allowing threads or processes of the program under test to use sockets for communication. Finally, there are system calls for reading (`clock_gettime`, `gettimeofday`) and setting clocks (`clock_settime`, `settimeofday`).

#### 4.4 The C Library

DiOS comes with a complete ISO C99 standard library and the C11 thread API. The functionality of the C library can be broken down into the following categories:

- Input and output. The functionality required by ISO C is implemented in terms of the POSIX file system API. Number conversion (for formatted input and output) is platform independent and comes from `pdclib`.
- The string manipulation and character classification routines are completely system-independent. The implementations were also taken from `pdclib`.
- Memory allocation: new memory needs to be obtained in a platform-dependent way. Optionally, memory allocation failures can be simulated using a non-deterministic choice operator. The library provides the standard assortment of functions: `malloc`, `calloc`, `realloc` and `free`.
- Support for `errno`: this variable holds the code of the most recent error encountered in an API call. On platforms with threads (like DiOS), `errno` is thread-local.
- Multibyte strings: conversion of Unicode character sequences to and from UTF-8 is supported.
- Time-related functions: time and date formatting (`asctime`) is supported, as is obtaining and manipulating wall time. Interval timers are currently not simulated, although the relevant functions are present as simple stubs.
- Non-local jumps. The `setjmp` and `longjmp` functions are supported on DiVM and native execution, but not in KLEE.

In addition to ISO C99, there are a few extensions (not directly related to the system call interface) mandated by POSIX for the C library:

- Regular expressions. The DiOS `libc` supports the standard `regcomp` & `regex` APIs, with implementation based on the TRE library.
- Locale support: A very minimal support for POSIX internationalisation and localisation APIs is present. The support is sufficient to run programs which initialise the subsystem.
- Parsing command line options: the `getopt` and `getopt_long` functions exist to make it easy for programs to parse standard UNIX-style command switches. DiOS contains an implementation derived from the OpenBSD code base.

Finally, C99 mandates a long list of functions for floating point math, including trigonometry, hyperbolic functions and so on. A complete set of those functions is provided by DiOS via its `libm` implementation, based on the OpenBSD version of this library.

#### 4.5 C++ Support Libraries

DiOS includes support for C++ programs, up to and including the C++17 standard. This support is based on the `libc++abi` and `libc++` open-source libraries maintained by the LLVM project. The versions bundled with DiOS contain only very minor modifications relative to upstream, mainly intended to reduce program size and memory use in verification scenarios.

Notably, the exception support code in `libc++abi` is unmodified and works both in DiVM and when DiOS is executing natively as a process of the host operating system. This is because `libc++abi` uses the `libunwind` library to implement exceptions. When DiOS runs natively, the host version of `libunwind` is used, the same as with `setjmp`. When executing in DiVM, DiOS supplies its own implementation of the `libunwind` API, as described in [20].

#### 4.6 Binary Compatibility

When dealing with verification of real-world software, the exact layout of data structures becomes important, mainly because we would like to generate native code from verified bitcode files (when using either KLEE or DiVM). To this end, the layouts of relevant data structures and values of relevant constants are automatically extracted from the host operating system<sup>9</sup> and then used in the DiOS `libc`. As a result, the native code generated from the verified bitcode can be linked to host libraries and executed as usual. The effectiveness of this approach is evaluated in Section 5.3.

---

<sup>9</sup> This extraction is performed at DiOS build time, using `hostabi.pl`, which is part of the DiOS source distribution. The technical details are discussed in the online supplementary material.

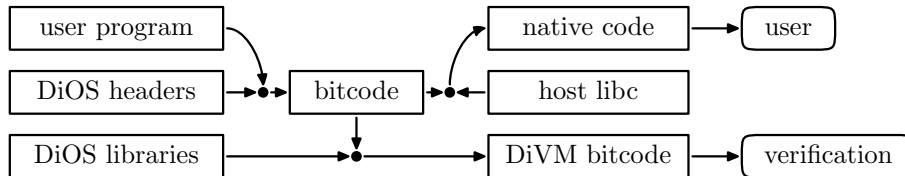


Fig. 3. Building verified executables with DiOS.

## 5 Evaluation

We have tested DiOS in a number of scenarios, to ensure that it meets the goals that we describe in Section 1.2. The first goal – modularity – is hard to quantify in isolation, but it was of considerable help in adapting DiOS for different use cases. We have used DiOS with success in explicit-state model checking of parallel programs [1], symbolic verification of both parallel and sequential programs [12], for verification of liveness (LTL) properties of synchronous C code synthesized from Simulink diagrams, and for runtime verification of safety properties of software [9]. DiOS has also been used for recording, replaying and fuzzing system call traces [8].

### 5.1 Verification with DiVM

In this paper, we report on 3 sets of tests that we performed particularly to evaluate DiOS. The first is a set of approximately 2200 test programs<sup>10</sup> which cover various aspects of the entire verification platform. Each of them was executed in DiOS running on top of DiVM and checked for a number of safety criteria: lack of memory errors, use of uninitialized variables, assertion violations, deadlocks and arithmetic errors. In the case of parallel programs (about 400 in total), all possible schedules were explored. Additionally, approximately 700 of the test programs depend on one or more input values (possibly subject to constraints), in which case symbolic methods or abstraction are used to cover all feasible paths through the program. The tests were performed on two host operating systems: Linux 4.19 with `glibc` 2.29 and on OpenBSD 6.5, with no observed differences in behaviour.

The majority (1300) of the programs are written in C, the remainder in C++, while a third of them (700) were taken from the SV-COMP [2] benchmark suite. Roughly half of the programs contain a safety violation, the location of which is annotated in the source code. The results of the automated analysis are in each case compared against the annotations. No mismatches were found in the set.

<sup>10</sup> All test programs are available online at <http://divine.fi.muni.cz/2019/dios/>, including scripts to reproduce the results reported in this and in the following sections.

## 5.2 Portability

To evaluate the remaining ports of DiOS, we have taken a small subset (370 programs, or 17%) of the entire test suite and executed the programs on the other two platforms currently supported by DiOS. The subset was selected to fall within the constraints imposed by the limitations of our KLEE port – in particular, lack of support for threads and for C++ exceptions. We have focused on filesystem and socket support (50 programs) and exercising the standard C and C++ libraries shipped with DiOS. The test cases have all completed successfully, and KLEE has identified all the annotated safety violations in these programs.

## 5.3 API and ABI Coverage and Compatibility

Finally to evaluate our third goal, we have compiled a number of real-world programs against DiOS headers and libraries and manually checked that they behave as expected when executed in DiOS running on DiVM, fully isolated from the host operating system. The compilation process itself exercises source-level (API) compatibility with the host operating system.

We have additionally generated native code from the bitcode that resulted from the compilation using DiOS header files (see Figure 3) and which we confirmed to work with DiOS libraries. We then linked the resulting machine code with the `libc` of the host operating system (`glibc 2.29` in this case). We have checked that the resulting executable program also behaves as expected, confirming a high degree of binary compatibility with the host operating system. The programs we have used in this test were the following (all come from the GNU software collection):

- `coreutils 8.30`, a collection of 107 basic UNIX utilities, out of which 100 compiled successfully (we have tested a random selection of those),
- `diffutils 3.7`, programs for computing differences between text files and applying the resulting patches – the diffing programs compiled and `diff3` was checked to work correctly, while the `patch` program failed to build due to lack of `exec` support on DiOS,
- `sed 4.7` builds and works as expected,
- `make 4.2` builds and can parse makefiles, but it cannot execute any rules due to lack of `exec` support,
- the `wget` download program failed to build due to lack of `gethostbyname` support, the cryptographic library `nettle` failed due to deficiencies in our compiler driver and `mttools` failed due to missing `langinfo.h` support.

## 6 Conclusions & Future Work

We have presented DiOS, a POSIX-compatible operating system designed to offer reproducible execution, with special focus on applications in program verification. The larger goal of verifying unmodified, real-world programs requires the

cooperation of many components, and a model of the operating system is an important piece of the puzzle. As the case studies show, the proposed approach is a viable way forward. Just as importantly, the design goals have been fulfilled: we have shown that DiOS can be successfully ported to rather dissimilar platforms, and that its various components can be disabled or replaced with ease.

Implementation-wise, there are two important future directions: further extending the coverage and compatibility of DiOS with real operating systems, and improving support for different execution and software verification platforms. In terms of design challenges, the current model of memory management for multi-process systems is suboptimal, and there are currently no platforms on which the `exec` family of system calls could be satisfactorily implemented. We would like to rectify both shortcomings in the future.

## Bibliography

- [1] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill. Model checking of C and C++ with DIVINE 4. 2017.
- [2] D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses report on SV-COMP 2016. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 887–904. Springer, 2016. ISBN 978-3-662-49673-2. doi: 10.1007/978-3-662-49674-9\\_55.
- [3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.
- [4] F. Chirigati, D. Shasha, and J. Freire. Reprozip: Using provenance to support computational reproducibility. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance, TaPP '13*, pages 1:1–1:4, Berkeley, CA, USA, 2013. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=2482949.2482951>.
- [5] J. Frew, D. Metzger, and P. Slaughter. Automatic capture and reconstruction of computational provenance. *Concurr. Comput. : Pract. Exper.*, 20(5):485–496, 2008. ISSN 1532-0626. doi: 10.1002/cpe.v20:5. URL <http://dx.doi.org/10.1002/cpe.v20:5>.
- [6] O. Inverso, T. L. Nguyen, B. Fischer, S. L. Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 807–812, 2015. doi: 10.1109/ASE.2015.108.
- [7] S. Joshi and A. Orso. Scarpe: A technique and tool for selective capture and replay of program executions. pages 234 – 243, 2007. ISBN 978-1-4244-1256-3. doi: 10.1109/ICSM.2007.4362636.
- [8] K. Kejstová. Model checking with system call traces. Master’s thesis, Masarykova univerzita, Fakulta informatiky, Brno, 2019. URL <http://is.muni.cz/th/tukvk/>.



- [9] K. Kejstová, P. Ročkai, and J. Barnat. From model checking to runtime verification and back. In *Runtime Verification*, volume 10548 of *LNC3*, pages 225–240. Springer, 2017. doi: 10.1007/978-3-319-67531-2\\_14.
- [10] S. Kong, N. Tillmann, and J. de Halleux. Automated testing of environment-dependent programs - a case study of modeling the file system for pex. pages 758–762, 2009. doi: 10.1109/ITNG.2009.80.
- [11] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laughner, and F. Bruhin. pytest 4.5, 2004. URL <https://github.com/pytest-dev/pytest>.
- [12] H. Lauko, V. Štill, P. Ročkai, and J. Barnat. Extending DIVINE with symbolic verification using SMT. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 204–208, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17502-3.
- [13] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi. Modular software model checking for distributed systems. *Software Engineering, IEEE Transactions on*, 40:483–501, 05 2014. doi: 10.1109/TSE.2013.49.
- [14] T. Mackinnon, S. Freeman, and P. Craig. Extreme programming examined. chapter Endo-testing: Unit Testing with Mock Objects, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-71040-4. URL <http://dl.acm.org/citation.cfm?id=377517.377534>.
- [15] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *2014 14th International Conference on Quality Software*, pages 127–132, 2014. doi: 10.1109/QSIC.2014.19.
- [16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*. USENIX, 2008.
- [17] P. Ročkai, V. Štill, I. Černá, and J. Barnat. DiVM: Model checking with LLVM and graph memory. *Journal of Systems and Software*, 143:1 – 13, 2018. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2018.04.026>.
- [18] B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design*, pages 210–217. IEEE, 2013. doi: 10.1109/FMCADE.2013.6679412.
- [19] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, 2008.
- [20] V. Štill, P. Ročkai, and J. Barnat. Using off-the-shelf exception support components in C++ verification. In *Software Quality, Reliability and Security (QRS)*, pages 54–64, 2017.