Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Reproducible Execution of POSIX Programs with DiOS

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

Faculty of Informatics, Masaryk University
Brno, Czech Republic

Received: date / Revised version: date

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. Components can be added to cover additional system calls or APIs or removed to reduce overhead.

The experimental evaluation has three 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. Finally, we consider its use as a standalone user-mode kernel.

1 Introduction

Software verification and validation is a field with a long and varied tradition and includes many methods, processes and techniques. While a certain amount of human involvement in the effort is quite unavoidable, the trend

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

is clearly towards maximal automation. For automation, consistency and reproducibility are clearly essential, but those properties are also of high value in manual efforts.

In this paper, we focus on the lower-level aspects of software quality assurance, with emphasis on:

- implementation-level pre- and post-conditions, invariants, and assertions,
- correctness of memory management (memory safety),
- low-level functional requirements (expressed as e.g. unit tests),
- programming errors in the interaction of programs with the operating system,
- medium-level functional requirements: the input-output behaviour of complete programs of moderate complexity.

In particular, validation and high-level verification (system testing, integration testing and non-testing-based approaches at a similar level of abstraction) are out of scope.

We have designed and implemented a compact model operating system, DiOS, with the aim to improve execution reproducibility in the context of the quality assurance goals outlined above. More specifically, we target the program-facing aspects of the POSIX standard, which are mainly defined in terms of a C API (Application Programming Interface). For this reason, our work is primarily applicable to C and C++ programs, which often use those operating system APIs directly. Additionally, it is indirectly applicable to programs written in higher-level languages, since even in those cases, interaction with the operating system is usually conducted through the POSIX C API.[1]

The model operating system that we describe in this paper is useful in its own right, for instance as a highly reproducible execution environment for traditional unit and functional tests. Its main strength, however, lies in its ability to complement existing rigorous software analysis tools, such as symbolic executors or full-blown software model checkers. In this role, our proposed operating system can be used as a stand-in for the native execution platform of the program under analysis. Moreover, thanks to its simplicity and modular design, DiOS can be adapted to various verification platforms, which will be demonstrated further in the paper.

In the rest of this section, we lay down the motivation, highlight the contributions of the present paper and set out the design goals of our effort. In Section 2, we discuss the prior art and related work. Section 3 describes the interface between DiOS and the underlying execution platform in general terms, while Section 4 describes the specific platforms to which DiOS has been ported and the state and capabilities of those ports. In Section 5, we

[1] One exception to this rule is the Go programming language, which at least in some configurations bypasses the C interface and interacts directly with the operating system kernel using system-specific conventions.

give a high-level view of the design and architecture of DiOS, abstracting technical and implementation details, which are then filled in by Section 6. We conclude the paper by presenting an empirical validation of the obtained results in Section 7 and give a final summary in Section 8.

1.1 Motivation

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.

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.

Consider `gzip`, a standard UNIX utility, which consists of an algorithmic core (the deflate compression algorithm) and a user interface, the latter of which interacts heavily with the operating system. An important part of this OS interaction deals with reading the input and writing the output. For the sake of simplicity, let us formulate the following property: decompressing a file, under conditions where memory allocation (via `malloc`) may fail, does not lead to any memory errors. In addition to the usual execution, this entails exploring all the error paths related to allocation failures. It is quite possible that these error paths have side effects: for instance, the input file might be closed, and the incomplete output file might be unlinked. The expected path (decompression is successful) also has a side effect: the input file is unlinked after decompression.

These side effects are what makes, in this case, the execution non-reproducible: if `gzip` succeeds, running the same command again will fail (the input file no longer exists), which is a behaviour clearly different from the previous execution. Moreover, model checkers usually do not restart execution from scratch: instead, a model checker will return (backtrack) to the last `malloc` call where it did not yet explore both outcomes, and restore the program state. In this case, an inconsistency appears between the internal state of the program, and external state of the environment. After an error path has closed the input file, and the model checker rolls back to a previous state, the program believes it has a valid file descriptor. The operating system (the environment), however, has already invalidated that file

descriptor and when the program attempts to use it, the operating system will indicate an error.

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 with its environment and obtain identical outcomes every time.

1.2 Contribution

The paper describes our effort to implement a compact operating system on top of existing verification frameworks and virtual machines (see Section 4) and its extension to execute natively, as a small user-mode kernel. In the initial phase, which was focused on execution in a virtual machine, we have identified a small set of interfaces between the VM and the operating system (see also Section 3) 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.

Such minimal interfaces represent 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*.

Additionally, the most important parts of this interface are sufficiently easy to implement in terms of native (host) operating system services, which in turn made the extension to native execution possible.

Despite its minimalist design, the current implementation covers a wide range of POSIX APIs in satisfactory detail (see also Section 5.3). The complete source code is available online,[2] under a permissive open-source licence.

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.

1.3 Design Goals

We would like our system to have the following properties:

[2] `https://divine.fi.muni.cz/2020/dios/`

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.
2. Portability:[3] 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 model operating system with minimal effort.
4. ABI compatibility: it should be possible to generate a working native executable from bitcode that has been built for DiOS.

Since the desired properties are hard to quantify, we provide a qualitative evaluation of the outcomes in Section 7.

2 Related Work

Programs which interact with the surrounding operating system have been studied from different perspectives. Our main focus is on the reproducibility of the program execution, and on the ease of use of the given method, particularly with application programs and in conjunction with testing, as the principal method of assurance of correctness.

2.1 Execution Reproducibility

Some of the simpler approaches either offer a substitute of the necessary system facility, or let the program interact with the actual operating system and attempt to capture the relevant data for later use/analysis. As for the latter case, 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 [5] bundles the environment variables, files and library dependencies it observes so that the executable can be run on a different system. While the tool was not intended for analysis of programs, but rather to make their execution reproducible in different environments,

[3] Ideally, since reproducibility is the main motivation for DiOS, a given program on given inputs would lead to an identical state space across all supported platforms. Considering the difficulty of the problem, this was not a major priority. Instead, we aim for a less ambitious variant of this goal, that is, the program exhibits the same higher-level semantics. In particular, considering any pair of platforms, the program contains the same set of errors (as long as all errors in the program fall into the intersection of error classes detected on both). Alas, it is much harder to demonstrate this modified thesis rigorously, though our evaluation does support it.

it goes to show how interaction of a program with the system binds it to the surrounding environment and how the potentially different systems (in terms of missing dependencies) might obstruct reproducibility.

Another way to capture provenance of the running program is in form of logs, which are possibly harder to re-execute but easier to analyse. Sometimes more complex structures are used – provenance graphs in case of ES3 [7]. ES3 (The Earth System Science Server) records read and write accesses to files along with information about the relevant process, and then models the relationships between the affected files and the processes which accessed them. The result is a directed graph of files and processes, created during execution of a program. In particular, system calls can be traced to record the interactions.

SCARPE [10] was developed for Java programs and captures I/O, user inputs and interactions with the database and the file system 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).

2.2 Testing and Symbolic Execution

Another common approach to dealing with the complexity of interactions with the execution environment is *mocking* [17, 18]: 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 [14] for Python allows the user to comfortably mock a database connection. A more complex example of mocking would be the file system support in Pex [13], 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 [22] and CHESS [19].

2.3 Model Checking

Delegating the interactions to the real operating system, instead of providing equivalent substitutions, makes both reproducibility and verification more problematic. 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 (see also the example in Section 1.1).

A specific example of an approach based on a virtual machine which optionally passes through calls to the outside environment is the Model Java Interface (MJI). MJI is used within Java PathFinder (JPF), a verifier targeting Java bytecode programs, since JPF provides, in essence, its own Java Virtual Machine. Specifically, MJI allows the program running in the JPF-specific virtual machine to access the outer JVM.

One approach to lifting the non-interference requirement is *cache-based* model checking [16], 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 5.1), 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.

Perhaps the most similar to our approach is S2E [4], which serves as a platform for other analysis tools, and aims at verification of properties and behaviour of programs and other software. S2E uses a real software stack (programs, libraries, the operating system kernel and other parts), on the observation that the environment plays an important role in software analysis. Rather than relying on models, S2E employs virtualization for realistic analysis of systems that interact with their environment. The analysers built on top of S2E are, in a fashion similar to our approach, modular systems. An important difference is that in the case of S2E, tools analyse individual paths (executions) and group path families with a common property.

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 [9] and its variants, by Impara [21] and by a few other tools.

2.4 High-Assurance Systems

At the other end of the complexity scale of program interactions are efforts to model and reason at the level of processor instructions, and interactions

with small operating systems. One such project, focusing on verification of programs that interact with the underlying OS via system calls, was coordinated by Goel et al. [8]. The approach aims at verification of programs at the level of machine code instructions, notably the x86 instruction-set architecture (ISA), and thus requires that the program be first compiled into this representation. The x86 ISA is modelled in the ACL2 programming language, and the programs are verified against a user-supplied specification. The verifier provides a *logical mode* – which serves for reasoning about the program, and an *execution mode*, in which the program can be executed directly on the host system. The system call interface is not part of the architecture; it is instead provided by the underlying OS, making it hard to reason about reproducible behaviour of programs. In this sense, the project is a pilot study of user-level programs analysed in their machine-code form, which use system calls. However, this approach has its specific limitations: firstly, since the authors do not support SIMD instructions, they have to provide substitutions for the functions that produce these – such as `getchar`. Secondly, and more importantly, a lot of work has to be done manually, which unfortunately requires understanding of the employed proof techniques and the machine code instructions involved. In this paper, we instead focus on verification of programs on a higher level of abstraction and model the system call interface using its API-level description.

Further rigorous verification efforts concentrate on thorough formalization of the system: ideally, a proof of correctness is provided for each subpart and for their interaction. One of the examples is Olos [6], which is a very specific effort aiming at a fully verified system. The motivation here is that since correctness of an application that interacts with the system depends on the correctness of this system. The proposed system runs on a verified processor, uses a special programming language (known as C0), and a proven-correct compiler, among other things.

3 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. The platform requirements of various DiOS features are summarised in Table 1.[4]

[4] Non-determinism is required to implement two important classes of features: scheduling (threads, processes) and fault injection (which is, however, always optional). Additionally, a number of modules can transparently pass through indeterminate values (e.g. in the context of symbolic model checking), but do not directly make any use of non-determinism themselves.

Table 1. Summary of available features and what they require from the underlying verification platform: 'stack' means direct manipulation of the execution stack, 'nondet' means non-deterministic choice, 'memsafe' means that the feature relies on enforcement of memory safety. Further, 'sync systems' refers to synchronous systems, or rather, in our case, the synchronous mode of the scheduler, described in more detail in Section 5.1. Optional items are marked with $^+$.

feature	stack	nondet	memsafe	other
`malloc`		✓$^+$		memory management
threads	✓	✓		
sync systems	✓			
processes	✓	✓	✓	heap cloning
signals	✓			
system calls				supervisor mode$^+$
file system				
longjmp	✓			
exceptions	✓			
proxy mode				syscall execution
replay mode		✓		

3.1 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 of the system under test.[5] In cases like those, DiOS can serve to 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.

[5] If execution A creates a file and leaves it around, execution B might deviate from its expected course when it tries to create the same file, or might detect its presence and behave differently.

Unfortunately, not all facilities that operating systems provide to programs can be modelled entirely in terms of standard C.[6] 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.

3.2 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 bytes is sufficient to implement all the essential functionality: the result of `malloc` is likewise an array of bytes. In a flat model, though, the memory allocator must manage additional data structures, which are susceptible to corruption by misbehaving programs.

In normal operation, the size of an object allocated by `malloc` must be known a priori, and is supplied by the caller. Resizing objects (the equivalent of `realloc`), however, poses an issue: in a flat memory model, it generally requires allocating a new block of memory and moving the data, as the memory which belongs to a single object must be contiguous. For internal use by DiOS (both the kernel and the `libc`), it is both more convenient and more efficient if object resizing could happen in place.

Ultimately, a flat array 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 bounds checks and an inability to overflow pointers from one memory object into another.

[6] Here, we primarily refer to the C language, as opposed to the standard C library. In most use-cases, the C library is, in fact, provided by DiOS and for this reason, DiOS itself cannot rely on library facilities like `setjmp`. An exception to this rule is the 'native' port of DiOS, which uses a small set of functions from the host C library (see also Section 4.3).

3.3 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, it is part of standard program memory and can be directly accessed using standard memory operations.[7] If both reading and modification of the stack (or stacks) is possible, the operations that DiOS needs to perform can be implemented without special assistance from the platform itself:

- 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 to be able to transfer control to a particular stack frame, whether to a different frame within a single execution stack (to implement non-local control flow) or to a different stack entirely (to implement task switching). Neither of these two variants can be implemented in standard, portable C, even if the stack is directly accessible (though they can be both implemented in terms of a single non-local jump primitive). More details about the implementation of stack management on the two platforms which support stack switching are given in Section 6.3.

In a 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. This means that a successful DiOS port is possible even if a suitable stack manipulation interface is not available.

3.4 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. incidence of

[7] 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 4.2.

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).

Non-deterministic 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.

4 Supported Platforms

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

4.1 DiVM

DiVM [20] 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*. They serve as an extension to the LLVM language, providing an interface between the operating system and the virtual machine. Hypercalls are used in a number of contexts: most importantly, they are used in memory management, but also for non-deterministic choice, for annotating the state space graph, and for accessing machine control registers. 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.

4.2 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 somewhat 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:[8]

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.

4.3 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

[8] A version of KLEE with fixes for those problems is available online, along with other supplementary material, from `https://divine.fi.muni.cz/2020/dios/`.

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.

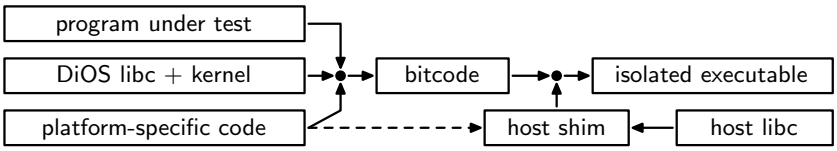

Fig. 1. Architecture of the native execution platform.

Like in KLEE, the native port of DiOS does not have access to in-place resizing 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 (more details are given in Section 6.3). The host stack unwinding code is directly used (the DiVM platform code implements the same `libunwind` API that most POSIX systems also use).

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, picking randomly or using an externally supplied list of outcomes are the only options.

5 Design and Architecture

This section outlines the structure of the DiOS kernel and user space, 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. If both are available, however, kernel memory protection is available even if the kernel and the user code share a single address space.

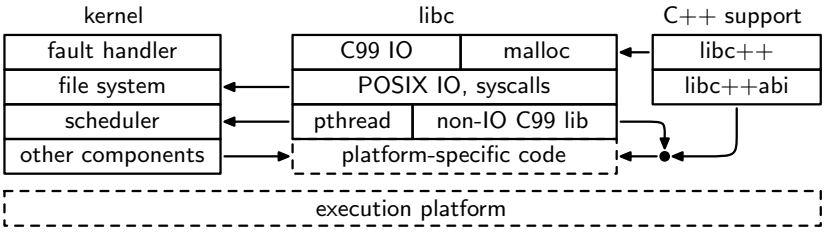

Fig. 2. The architecture of DiOS.

5.1 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 four 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 (cf. Section 6.4). It implements the `fork` system call and requires one of the two asynchronous schedulers.

POSIX System Calls. While a few system calls (mainly related to process management) 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 file system module, described in Section 6.5, 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 by every time it is observed, or a simpler variant, where ticks of fixed length are performed based on the outcome of a non-deterministic choice.

The system calls covered by the file system 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. `libc` uses it to flag assertion failures. A more detailed account of the DiVM fault mechanism and how it is used in DiOS is given in Section 6.6.

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.

5.2 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 3.3).

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.[9] 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[10] 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.

5.3 System Calls

The system call interface of DiOS is based on the ideas used in *fast system call* implementations on modern processors.[11] 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:[12] in addition to the kernel-side implementation, which may or may not be available depending on the active 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

[9] 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.

[10] The basis of this claim is largely empirical, based on years of experience in writing, reading and verifying multi-threaded code. The stubbed-out functions are designed for highly specialized scenarios – all the function intended for general use are implemented. An example of a lesser-used function would be `pthread_barrierattr_getpshared`, used to obtain the value of the process-shared attribute of its argument.

[11] For instance, on contemporary `x86-64` processors, this interface is available via the `syscall` and `sysret` instructions.

[12] 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.

thread- and process- related functionality was described in more detail in Section 5.2 and in Section 5.1.

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`).

5.4 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` & `regexec` 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.

5.5 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.[13]

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 [23].

5.6 Binary Compatibility

Programs are usually programmed against a particular API, or Application Programming Interface, which is a high-level description (in our case in terms of C function calls and C data types) of the interface they use to communicate with some underlying services (in our case, with the operating system). When a C compiler generates the intermediate code (and later machine code), it needs to emit instructions which realize those API calls, and which extract information from API-relevant data types. The specific instruction sequences, however, depend on a number of factors which are not part of the API itself, but are instead considered implementation details at this level of abstraction. Those factors are collectively known as the ABI, or Application Binary Interface.

When dealing with verification of real-world software, the specific ABI of the target platform becomes important, mainly because we would like to generate native code from verified bitcode files (when using either KLEE or DiVM). This means that at the bitcode level, the instruction sequence

[13] Of the 16 thousand lines of code in `libc++`, the version in DiOS differs from upstream in 29 lines (19 lines added, 10 removed). In case of `libc++abi`, 7 preprocessor directives have been added to make the library build in C++17 mode.

emitted by the compiler to call a function from the DiOS `libc` and the sequence to call that same function from the `libc` of the host operating system must be identical.

To this end, the layouts of relevant data structures and values of relevant constants are automatically extracted from the host operating system 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 7.4.

The extraction of the host ABI is performed at DiOS build time, using a tool (`hostabi.pl`) which is part of the DiOS source distribution. The technical details are discussed in Section 6.7.

6 Implementation

In this section, we discuss some of the implementation considerations, the challenges that we have encountered and the solutions that we chose. We will start by describing the boot sequence of DiOS in Section 6.1: how different platforms set up the environment and pass parameters and other data into DiOS. In Section 6.2, we will look at the details of the coroutine-based thread support, while Section 6.3 elaborates on the low-level aspects of stack management on the two platforms where it is supported. Afterwards, Section 6.4 deals with multi-process programs and the `fork` system call. In Section 6.5, we discuss the file system: both the implementation of the virtual file system itself and the protocol for seeding this virtual file system with the data from the host system. Finally, Section 6.6 is dedicated to the fault reporting mechanism employed by DiVM.

Table 2. Breakdown of the DiOS source code. The items marked with '→' further break down the total of the component given above them. The items marked $^+$ are original source code written for DiOS, the remainder comes from third parties with at most minor adaptations. Components typeset in monospace correspond to directory names in the DiOS source tree.

component	loc	component	loc	component	loc
kernel$^+$	5532	`libc`	23770	libc headers	11928
→ `sys`	2305	→ pthread$^+$	1024	→ `sys`$^+$	2092
→ `vfs`	2282	→ `sys`$^+$	663	→ pdclib	4084
→ `proxy`	945	→ `regex`	6755	→ standard	5752
		→ `stdio`	4013	`libm`	18591
		→ `time`	2252	`libc++`	16051
		→ `string`	1327	`libc++abi`	8755
		→ `stdlib`	1149		

The implementation is a mix of C and C++. The kernel, some of the DiOS-specific `libc` components and the C++ support libraries are implemented in C++17, while the remainder is implemented in C99. The entire code base is around 85000 lines of code, but a large majority of this comes from unmodified third-party libraries. DiOS-specific code is less than 10000 lines, or about 11 % of the total.

6.1 Boot Protocol

Like any other operating system, DiOS has a boot sequence: a list of steps, specific to a given platform, that puts the system into a state from which the DiOS kernel can take over. Once the kernel is ready, it sets up a single user-space process for the program under test and transfers control to that process – there is no extended user-space initialization sequence and no dedicated `init` program. In other words, DiOS executes a single user-space program which can perhaps use the `fork` system call to create new processes, but in the most common use-case, only a single process exists, and once this process exits (or is killed), the operating system shuts down again.

The boot process is split into two parts: the first part takes place outside of DiOS proper and involves loading the bitcode or machine code of both DiOS itself and of the program under test, preparation of configuration parameters and any additional data (e.g. a file system snapshot, see also Section 6.5) in such a way that the DiOS kernel can read and process them later. In order:

1. A DiOS *kernel configuration* is selected and loaded: in Section 5.1, we have outlined that a particular DiOS kernel can be pieced together from a selection of pre-made components. When DiOS is built, those components are compiled and linked to form a number of *kernel bitcode files*, one for each kernel configuration.
2. The user program is either loaded separately, or linked with the selected kernel bitcode file to form a single executable image. In either case, at this point, an execution environment is created and an address space is set up for the combined system.
3. Static inputs are prepared and copied into the address space of DiOS. The form of this data is an array of key-value pairs, where the key is a null-terminated ASCII string and the value is an arbitrary byte array.
4. Execution is transferred to the `__boot` function, which is part of DiOS proper. This function sets up the rest of the system – it takes care of processing the input data, initializing any internal kernel data structures and starting the program to be tested.

The data that is passed to DiOS contains the following items:

– command line arguments to be passed down to the program under test (not interpreted by DiOS in any way),

- the initial values of environment variables (likewise passed verbatim to the program),
- configuration parameters of DiOS itself (fault injection, how to react to faults signalled by the underlying platform, clock configuration, and so on)
- what to do with data that appears on `stdout` and `stderr` of the program,
- the content to be presented to the program on the `stdin` handle, and finally,
- the initial content of the file system (including metadata).

Since the entirety of the data is encoded in a simple, flat key-value map, it is easy to store this data, verbatim, in common data interchange formats like JSON or YAML. This greatly simplifies the task of reproducing a given execution – only the bitcode images and this block of data influence the behaviour of the system.[14]

While DiVM implements the above protocol natively, all other ports provide a small amount of *glue code* which transforms inputs from a platform-specific form into the form expected by DiOS. In those cases, it is also the responsibility of the platform glue code to invoke the stardard `__boot` function, as described above.

6.2 Thread Support

In Section 5.2, we have mentioned that the DiOS support for threading is based on coroutines. In this section, we will make that notion more precise. A coroutine is a computation that can be voluntarily *suspended* at an arbitrary point in its execution, and later resumed to continue execution from that point onwards. The suspension is also known as a *yield*. At the point of suspension, sufficient state must be stored in memory to allow the computation to continue even if the state of the processor will change in the meantime.

Commonly, two types of coroutines are considered. Those of the first type are known as *stackless* coroutines and usually encapsulate the entire state of the computation in a fixed-size structure. Such coroutines can call into standard subroutines, but cannot yield while a subroutine is executing.

In DiOS, we need to consider *stackful coroutines*, that is, each of the coroutines has a private execution stack associated with it. This is required because DiOS needs to be able to reschedule threads at arbitrary points in execution, including within arbitrarily nested stack frames (subroutine calls). DiOS, however, does not require a specific stack organisation (more

[14] Unless DiOS is configured for system call proxying, in which case it has no control over the outcomes of interactions with the host operating system. However, in this case, those interactions are recorded, and the recorded execution can then be replayed at will.

on this in Section 6.3) – it only needs to be able to perform a small number of operations on the coroutine stacks:

1. create a new stack and place a single stack frame corresponding to a function selected at runtime on this stack,
2. switch execution from one stack to another, in such a way that the suspended stack can be later resumed using the same primitive,
3. destroy an inactive stack and free up its associated resources.

The newly created stack must be in a state which makes it possible to start executing on it using the stack switching primitive from point 2. Those three operations are implemented in a platform-specific way, but then allow the remainder of thread support to be mostly platform-neutral.

Of course, the above scheme does not support *preemptive* scheduling,[15] which is, however, the norm for POSIX systems. To convert a cooperative system into a preemptive one without programmer intervention, we need to instrument the program under test to yield into the operating system at every 'interesting' (observable) point of its execution. This is achieved by using an external tool called LART,[16] which can, among other things, insert the requisite *yield* instructions into LLVM bitcode automatically. By default, only memory access instructions which access memory that cannot be statically shown to be thread-local are instrumented to yield into DiOS. This is sufficient to observe all possible behaviours of a multi-threaded program.

Once execution returns into the kernel (due to a yield), it will use non-deterministic choice (see also Section 3.4) to select which thread should continue execution, and use the stack switching primitive described above to resume this thread from the point at which it was suspended earlier.

6.3 Stack Management

In this section, we will describe how coroutine stacks are implemented on the two platforms which support task switching: DiVM and native execution.

DiVM On DiVM, the stack is represented as a linked list of frames, where each frame contains a snapshot of the state of the virtual processor. Each frame has a header, which contains a pointer to the caller's frame and the value of the program counter (pointing at the active instruction in the function to which the stack frame is associated). The remainder of the stack

[15] In a preemptive system, the executing thread does not need to perform any special action to be interrupted and removed from the processor (i.e. *preempted*). Systems based on this approach are more robust against misbehaving threads, at the expense of reduced efficiency and less intuitive behaviour.

[16] LART is a comprehensive tool for transforming and instrumenting LLVM bitcode and is described in more detail in [20]. Appropriate calls to LART are performed automatically by compilation scripts included with DiOS.

frame stores function arguments and a function-specific set of virtual registers.

New stack frames (and stacks) can be created simply by allocating memory for them: they are not distinguished from other memory objects stored by DiVM, and their only special property is that either the 'currently executing frame' register, or another stack frame, points at them. Before bitcode is loaded into DiVM, it is enriched with metadata that allow sufficient level of reflection for DiOS to be able to directly construct frames for arbitrary functions and pass arguments to them.

The final ingredient is stack frame switching: in DiVM, the register which holds the currently executing frame can be directly overwritten from C code. When this happens, the virtual machine arranges for execution to continue in the target frame. Using this as a primitive, DiOS can implement both the coroutine yield/resume operations required for thread support, and a stack unwinder (which examines the stack frames in more detail, to decide whether they are the target frame, and to clean up any additional memory associated with those frames). The stack unwinder is then used in both `setjmp` / `longjmp` and to implement C++ exceptions on this platform.

Native When executing as a native program of the host operating system, DiOS has to conform to the stack conventions of the host. In practice, this means a contiguous stack, on which a stack frame structure is implied (one that is different for each processor architecture and host operating system). While it would be possible to implement stack management using CPU- and OS-specific assembly, this would make DiOS much harder to use in this regime.

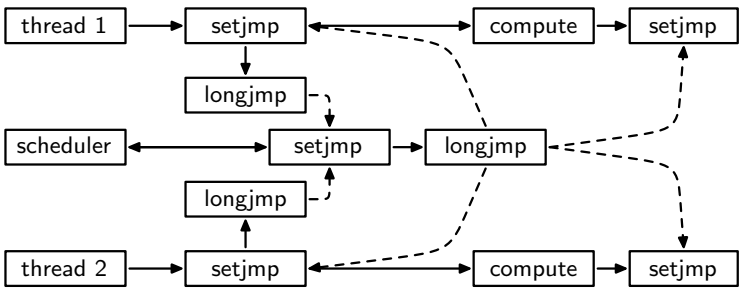

Fig. 3. The control flow across two coroutine-based threads that use the `setjmp` – `longjmp` mechanism to implement yields. The control starts at the scheduler, which calls the `setjmp` – `longjmp` sequence in a loop. The scheduler `longjmp` transfers control to each of the `setjmp` calls in each of the threads in a sequence chosen by the scheduler. The `setjmp` function returns twice: once immediately (solid line), second time (dashed) when control flow passes to it along a dashed line from `longjmp`. The destination of the `longjmp` in the middle depends on the decision of the scheduler and the state of individual threads at the time of the call.

Fortunately, with access to the underlying (host) C library, it is possible to use a combination of (standard) `setjmp` and `longjmp` functions together with the non-standard, but common, `alloca` primitive to implement a relatively portable system of stackful coroutines. Switching between existing coroutines is comparatively easy using a `setjmp` followed by a `longjmp` conditional on the `setjmp` result. The control flow of this construct is illustrated in Figure 3.

However, setting up a new stack is slightly more complicated, and we need to make a few assumptions about the host platform:

- `alloca` (or rather `__builtin_alloca`) does not check its argument, but instead simply performs arithmetic on the stack pointer,
- the platform calling convention can pass 3 pointer arguments in registers (can be narrowed down to one pointer at the cost of a minor increase in overhead, though modern architectures usually offer a sufficient number of register arguments),
- the stack grows towards lower addresses (this can be easily amended),
- it is possible to obtain memory that can be used as a stack using the host `mmap` system call.

To set up a new coroutine with a new stack, the following steps are performed in a `create_stack` function:

1. prepare a `jmp_buf` to store the context of the newly created coroutine,
2. use `mmap` to obtain memory for the new stack, and compute a pointer to its bottom end,
3. use `alloca` with the distance from the current stack frame (obtained using an address of a local variable) to the bottom of the new frame as an argument, with the result of adjusting the stack pointer to the bottom of the new stack,
4. perform a call on a trampoline function, which immediately calls `setjmp` to store the new context in the `jmp_buf` preallocated in step 1 and then returns,
5. the epilogue of `create_stack` undoes the adjustment of the stack pointer performed by the `alloca` builtin.

At this point, doing a `longjmp` into the context of the new coroutine causes the `setjmp` in the trampoline to return again (with a stack pointer pointing to the new stack), but this time, instead of returning to its caller, the trampoline simply calls the coroutine implementation itself.

6.4 Process Support

Support for processes in DiOS builds on thread support, since the threads of any given process run concurrently with threads in all other processes: from the point of view of thread scheduling, processes do not change the picture

significantly.[17] DiOS currently does not support the `exec` system call (it does not have an underlying mechanism to start executing a new program), but it does support `fork` when it runs on DiVM. In POSIX systems, `fork` causes the currently executing process to be duplicated into a new address space, and both processes continue execution by returning from the `fork` system call (the call returns different values in different processes, allowing them to differentiate themselves).

Real operating systems use a trick known as 'copy on write' to make the required address space duplication efficient. Additionally, the same mechanism (virtual memory, provided by the memory management unit of the CPU) is used to enforce process boundaries. However, DiOS does not have a virtual memory subsystem and instead relies on memory safety of the underlying platform to implement multiple address spaces (this is one of the reasons why process support is only available on DiVM). In this arrangement, all processes share the same address space, but they cannot access each other's memory. A major downside of this approach is that after a `fork`, one of the processes will see all memory objects shift to different locations and code which relies on specific numeric values of pointers could break.[18] Improvements in this area are a subject of future research.

6.5 Virtual File System

Support for file system operations is quite fundamental to any POSIX-compatible system. The POSIX semantics of files and directories are well established and well understood. Since the file system is a shared resource, the issue of concurrent access arises. Fortunately, the operations on directories (such as linking, unlinking and renaming of files) are straightforward, since they are guaranteed to be atomic. The situation regarding file content is slightly more complicated: POSIX does not specify the exact behaviour of concurrent write operations, and DiOS currently uses a very simple model, where the effect of each system call is atomic.[19] Finally, in the real world, operations on files are subject to a number of externally-caused error conditions (e.g. due to faulty hardware). In software, those effects can be emulated using fault injection, though this functionality is currently not offered by DiOS.

Our implementation of the file system builds on the traditional approach, where all file-like objects (including directories) are backed by inodes, which

[17] There are typically more opportunities for state space reductions in inter-process concurrency (when compared to thread-based concurrency) due to less shared state, but this is currently outside the scope of DiOS.

[18] Specifically, hash tables and binary search trees that use pointers as keys are vulnerable to this problem.

[19] Of course, the effects of multiple concurrent calls to `write` may be ordered arbitrarily, and the scheduler will in fact ensure that all possible orderings of writes are explored.

are, in our case, represented using in-memory data structures. For reasons of implementation simplicity, inodes in DiOS form a simple object-oriented class hierarchy with late binding. The basic set of operations over an inode is (not all inode types support all of them):

- determine the access mode, size, ownership, inode number and other metadata, and adjust those where applicable,
- read and write data from the inode,
- open and close the inode (for keeping track of the number of live references),
- link and unlink the inode into the directory tree (likewise for keeping track of reference counts),
- perform socket operations (`listen`, `connect`, `bind`, `accept`, `receive` and so on).

Of notable interest is the reference counting mechanism, which is essentially mandated by POSIX semantics: an inode must remain valid as long as there are either directory links which refer to it, or while it is present in the open file table of a running process. Inodes which are no longer referenced can be recycled and their associated storage likewise released for reuse.

The inode types available in DiOS are the following:

- the standard input of the executing program,
- output inodes which copy any data they receive into the *execution trace* which usually becomes part of the verification report (cf. Section 6.6), in a line-buffered and an unbuffered variant, usually connected to the `stdout` and `stderr` of the program under test,
- a regular file inode, which simply stores an array of bytes,
- a symbolic (soft) link,
- directory inodes, which contain named links to other inodes (those links are made available to the user space using the standard `read` function),
- pipes, which push data in a single direction, and finally
- sockets, which are used for both inter-process and network communication (both stream and datagram sockets are supported).

DiOS notably lacks support for special files called (character and block) *devices*, which in normal UNIX represent peripherals and other special file types, for instance `/dev/null`.[20]

While inodes enshrine the content and semantics of individual file system objects, the interaction of programs with the file system is mediated by per-process *file descriptors*, which refer to entries in a global *open file table*. File descriptors are thin references – in user space, they appear as small integers, while in the kernel, they are used as indices into a table of references to open files. While a single inode may have multiple entries in the open files table,

[20] The existence and semantics of `/dev/null`, along with `/dev/zero`, are mandated by POSIX, but currently not available on DiOS. This is expected to be fixed in a future revision.

it is also possible that multiple file descriptors point to the same record in this table. The *offset* (the position within the file where the next `read` or `write` operation will take place) and certain flags are a property of the open file record and may be therefore shared by multiple file descriptors, even across process boundaries.

The remaining component of the file system API are *paths*, which describe the locations of files (and other objects) in the directory tree. Paths appear most prominently in the `open` call, which takes a path as input and returns a file descriptor as its output, thus allowing the process to interact with the designated file further. Two types of paths exist: *absolute*, which are anchored to a per-process *root directory* and *relative*, which refer to a (more dynamic and also per-process) *working directory*. Additionally, POSIX allows the programmer to specify paths relative to any directory, to which they have a valid file descriptor (using the `openat` family of functions; in DiOS, this is the primary interface).

Since DiOS does not have access to any kind of persistent storage (such access would violate the desired reproducibility guarantees), it is important that the file system can be conveniently seeded with data upon boot. This is done using the protocol which was described in Section 6.1: each inode is serialized using 3 blobs in the key-value data structure, using a common `vfs.N` prefix for the keys, where `N` is a unique sequence number, and `name`, `stat` and `content` are suffixes. The `name` attribute is the path under which the object is linked, `stat` contains the inode metadata and `content` contains the data (the content of regular files and the target path for symlinks). Like any other data passed to DiOS during boot, the file system snapshot is easily stored and re-used, allowing exact reproduction even if the original (host) file system from which the data was captured has changed in the meantime.

6.6 DiVM Fault Mechanism

An important function of DiOS is, in the context of verification, the reporting of errors that arise in the program execution back to the user. There are two components to this:

1. the interface between the underlying verifier, which reports errors to the DiOS kernel in a manner similar to how CPUs notify kernels of standard operating systems about error conditions such as access to unmapped memory, writes to read-only memory, division by zero and so on,
2. the *tracing* interface, which allows DiOS to report arbitrary text output back to the user, usually via the verification tool.

In addition to the errors detected by the virtual machine, the executing code itself may signal a problem, and those are routed through the same *fault handler* that the virtual machine invokes. A typical example would be an assertion violation, or a variation thereof, like detection of deadlocks on pthread mutexes, or detection of invalid (overlapping) `memcpy` or `strcpy` arguments.

The fault handler then decides how to handle each reported error:

- ignore the error completely, resuming execution,
- report the error but continue execution, or
- report the error and kill the program under test,

based on the configuration parameters passed to the fault handler using the system option mechanism described in Section 6.1. If execution is to be resumed, an additional challenge arises in case the error was caused by a branch (e.g. due to a conditional branch which depends on an uninitialized value) or a call instruction (e.g. a wrong number or type of arguments). Since the virtual machine invokes the handler as a normal function sitting on top of the stack which caused the error, it is not possible to return normally from the handler. For this reason, the virtual machine passes the frame and the program counter value which describe the location at which execution should resume in case the fault is ignored, and the fault handler uses the control transfer mechanism described in Section 6.3.

6.7 Host ABI Compatibility

As outlined in Section 1.3 and Section 5.6, one of the goals of DiOS is to be binary compatible with the host operating system. The main motivation is to be able to directly re-use the bitcode that was used for verification to generate native code to be used in production. Unfortunately, the `libc` ABI is not standardized across different operating systems (and sometimes not even across different versions of the same operating system).[21] Since DiOS aims to be a generic POSIX system (instead of emulating a particular implementation), it is not realistic to hard-wire a particular ABI into DiOS; instead, it should be able to automatically re-use the ABI provided by the host operating system.

The ABI of a typical POSIX `libc` has a number of components, mainly:

1. any non-standard (platform-specific) API calls or variable references that results from the expansion of a standard API call implemented using macros (e.g. `__errno_location` expanded from the `errno` macro-variable or `__assert_failed` expanded from the `assert` function-like macro),
2. the numeric values of any POSIX-specified symbolic constants (e.g. `O_RDONLY`, `MAP_FIXED` and so on),

[21] Not all aspects of the ABI are relevant at the bitcode level. For example, the *function calling convention* used by a given platform is specified in terms of low-level, architecture-specific notions, like the names of CPU registers or the minutiae of stack management. These do not affect DiOS directly, and we simply rely on the LLVM native code generator to deal with this part of the ABI correctly.

3. the size and internal layout of any transparent `struct` types[22] specified by POSIX,
4. the size of opaque (i.e. field access is not allowed by POSIX for those types) but user-allocated types (`jmp_buf`, `pthread_mutex_t`, ...),
5. the sizes and signedness of named integral types (`pid_t`, `mode_t`, ...),
6. system call numbers (i.e. the values of the `SYS_*` macros).

Most of the information on the list is encoded in the C header files provided by the host operating system and hence available to the system-provided C compiler. Fortunately, for most of the items on the list, it is not hard, even if perhaps tedious, to write C programs which simply print out the relevant bits of information in a form that can in turn be used to automatically generate the ABI-relevant sections of DiOS header files. We use a straightforward script to write, compile and execute the relevant C program. The output of this program is a single header file, which is made available as part of the DiOS `libc` under the name `sys/hostabi.h`. This file is then included in DiOS versions of standard headers and the information stored there is used to build up the DiOS ABI.

The only type of information which is not easily extracted using this automated approach falls under the first item of the above list. For such macros, DiOS must be manually modified to be made aware of their expansions on the targeted host system. While this is unfortunate, such macros are comparatively rare and we are not aware of an approach which could reliably automate the process.

7 Evaluation

We have tested DiOS in a number of scenarios, to ensure that it meets the goals that we describe in Section 1.3. 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 [15], for verification of liveness (LTL) properties of synchronous C code synthesized from Simulink diagrams, and for runtime verification of safety properties of software [12]. DiOS has also been used for recording, replaying and fuzzing system call traces [11].

[22] Transparent (or non-opaque) types in the sense that user programs are allowed to directly access their fields by name or via macro expansion. In those cases, the compiler computes field offsets into the `struct` at compile time and hard-codes the results into the generated bitcode or machine code.

Table 3. A summary of the test programs used in evaluation. Please note that there is overlap between the individual tags. Tags marked with → are a subset (or very nearly a subset) of the (unmarked) tag given above them.

tag	cases	included programs...
c	1258	are written in C
→ svcomp	504	were taken from SV-COMP
→ posix	59	specifically check POSIX APIs
→ libc	239	focus on testing the standard library
→ lang-c	56	test C language features
c++	1669	are written in C++ (up to C++17)
→ libcxx	910	focus on testing the standard library
→ bricks	376	are unit tests of a utility library
→ lang-cpp	31	test C++ language features
→ exception	97	throw an exception at runtime
threads	320	use POSIX, C11 or C++11 threads
→ pthread	174	use the POSIX threading API
→ weakmem	77	execute under a relaxed memory model
sym	678	require symbolic execution to verify
error	612	contain a safety violation

7.1 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 3000 test programs which cover various aspects of the entire verification platform.[23] 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 memory, assertion violations, deadlocks and arithmetic errors. Each of the programs is tagged (labelled) with a list of categories.[24] Both the summary in Table 3 and the following breakdown are based on those tags.

In the case of parallel programs (about 320 in total, tag `threads`), 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 were used to cover all feasible paths through the program; those programs carry the `sym` tag.

[23] All test programs are available online at `http://divine.fi.muni.cz/2020/dios/`, including scripts to reproduce the results reported in this and in the following sections.

[24] Each test program contains the list of its assigned tags near the top (first or second line) embedded in a comment in a machine-readable format. Names of all parent directories in which the test cases are stored are appended to this list. Please note that the tags are assigned and reviewed mostly manually, and hence it is possible that minor inaccuracies have crept in.

The majority (1600) of the programs are written in C++ (tag `c++`), the remainder in C (tag `c`), while a sixth of them (approximately 500) were taken from the SV-COMP [2] benchmark suite (tagged `svcomp`). Roughly a fifth of the programs contain a safety violation (tagged `error`), 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.

All tests were performed on two host operating systems: Linux 4.19 with `glibc` 2.28 and on OpenBSD 6.6, with no observed differences in behaviour.

7.2 The KLEE Port

The KLEE port was evaluated on a subset of the programs considered in Section 7.1. In particular, parallel programs (tag `threads`) and programs which use non-local control flow (`setjmp` and C++ exceptions, tagged `setjmp` and `exception`, respectively) were excluded. More than half of the test cases (approximately 1900) have made it into the KLEE evaluation set.

The selection notably included test cases focused on file system and socket support (50 programs, tagged `posix`) and those exercising the standard C and C++ libraries shipped with DiOS (234 cases tagged `libc` and 829 tagged `libcxx`, respectively). The set also included most of the SV-COMP benchmarks (434 programs), and in those cases, like with DiVM, all possible inputs were covered (using symbolic memory support built into KLEE). The KLEE port also supports fault injection, which was enabled in the test programs which call for its use.

Out of the selected subset, 5 test cases timed out or were too slow and one test case failed: in this case, an out-of-bounds memory access was not detected by KLEE, since it fell within the small metadata area that the DiOS memory allocator adds to all objects. This deficiency could be addressed by altering the metadata scheme used by the DiOS KLEE port. The remaining test cases have all completed successfully, and KLEE has identified all the annotated safety violations in these programs.

7.3 The Native Port

The other DiOS port targets native execution. We have used the same approach as we have for evaluating the DiVM and KLEE ports, compiling the applicable subset of the 3000 test programs and executing each of them. In this case, the selected subset contained slightly more than 1700 test programs. Unlike the KLEE port, most parallel programs were included in the set, as were programs which use C++ exceptions and the `setjmp` function. The biggest group that was excluded from this part of the evaluation were benchmarks from SV-COMP, since they very often rely on indeterminate input values.

Unlike the other two ports, however, this evaluation approach fails to cover an important port-specific use case: replaying counterexamples obtained from a DiVM-based model checker. The motivation for this would be to leverage standard debugging tools such as `gdb` for a more comfortable analysis of the underlying cause of the detected problem. The methodology does, however, establish the viability of the other important use case: isolation of the program under test from the host operating system, with the intention of improving reproducibility of any given trial execution. This particularly pertains to programs which interact with the file system and to parallel programs. Finally, we have considered the case of fault injection, which we have found viable, even though it is harder to use with the native port than it is with the DiVM and KLEE ports.

Unlike with the other ports, exploring multiple different executions of a given test program is not entirely straightforward – for this reason, for the bulk evaluation, we have taken a single execution for each program, with fault injection disabled. Each non-deterministic choice (see also Section 3.4) was taken based on a pseudo-random sequence with a fixed (but configurable) initial seed. This means that in many cases, errors the presence of which depends on the outcomes of a race conditions were not detected. A further limitation stems from the fact that memory errors, especially out-of-bounds memory access, cannot be reliably detected. Some of the programs finished successfully, even though they contained a memory error.

After excluding undetected race conditions and memory errors, remaining test programs have all behaved as expected, either crashing (due to a failed assertion, a segmentation fault, a division by zero and so on) or completing without errors, in accordance with the annotations in the programs.

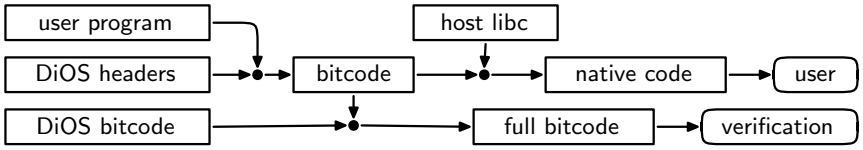

Fig. 4. Building verified executables with DiOS.

7.4 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 4) 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 (many of which 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 testsuite of the Eigen project, which provides efficient C++ implementations of linear algebra operations, was built and tested with success,
- SQLite 3.28.0, a widely used embedded SQL database engine, builds with minor limitations (support for `dlopen` is missing from DiOS), but binaries fail to work due to ABI incompatibility of the DiOS `libpthread` library,
- Berkeley DB 4.6.21, another database management library, more tightly coupled to the client application, builds okay but exhibits the same `libpthread` problem as SQLite,
- `zlib` 1.2.11, a simple compression library, builds and works as expected,
- `libpng` 1.6.37, a library for reading and writing Portable Network Graphics files, builds and works as expected, including example programs which demonstrate basic PNG file manipulation,
- 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 `mtools` failed due to missing `langinfo.h` support.

8 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. We have also demonstrated that bitcode of a program compiled for DiOS can be further processed into a native executable targeting the host system (with some limitations).

Implementation-wise, there are two important future directions: first, the coverage and compatibility of the DiOS-provided API with real operating systems can be further improved. Second, existing ports can be extended to cover more functionality, and it is possible and desirable to create new ports for additional verification platforms.

In terms of design challenges, we would like to improve support for multi-process and multi-image programs, including support for the `exec` syscall and for dynamically loaded shared libraries. We also believe that the mechanism for thread scheduling, most importantly for integration with sophisticated state space reductions, can be both simplified and significantly improved.

References

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. In *ATVA 2017*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017. URL `https://divine.fi.muni.cz/2017/divine4`.
2. Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses report on SV-COMP 2016. In *TACAS*, pages 887–904. Springer, 2016. doi: 10.1007/978-3-662-49674-9_55.
3. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.
4. Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1), February 2012. doi: 10.1145/2110356.2110358.
5. Fernando Chirigati, Dennis Shasha, and Juliana Freire. Reprozip: Using provenance to support computational reproducibility. In *Theory and Practice of Provenance*, pages 1:1–1:4, Berkeley, CA, USA, 2013. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2482949.2482951`.
6. Matthias Daum, Norbert Schirmer, and Mareike Schmidt. From operating-system correctness to pervasively veried applications. In *Integrated Formal Methods*, pages 105–120, 10 2010. doi: 10.1007/978-3-642-16265-7_9.
7. James Frew, Dominic Metzger, and Peter 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.
8. Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of x86 machine-code programs that

make system calls. *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 91–98, 2014.
9. 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 *Automated Software Engineering*, pages 807–812, 2015. doi: 10.1109/ASE.2015.108.
10. Shrinivas Joshi and Alessandro Orso. SCARPE: A technique and tool for selective capture and replay of program executions. In *International Conference on Software Maintenance*, pages 234 – 243, 2007. ISBN 978-1-4244-1256-3. doi: 10.1109/ICSM.2007.4362636.
11. Katarína Kejstová. Model checking with system call traces. Master's thesis, Masarykova univerzita, Fakulta informatiky, Brno, 2019. URL `http://is.muni.cz/th/tukvk/`.
12. Katarína Kejstová, Petr Ročkai, and Jiří Barnat. From model checking to runtime verification and back. In *Runtime Verification*, volume 10548 of *LNCS*, pages 225–240. Springer, 2017. doi: 10.1007/978-3-319-67531-2_14.
13. Soonho Kong, Nikolai Tillmann, and Jonathan de Halleux. Automated testing of environment-dependent programs - a case study of modeling the file system for pex. In *International Conference on Information Technology: New Generations*, pages 758–762, 2009. doi: 10.1109/ITNG.2009.80.
14. Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. pytest 4.5, 2004. URL `https://github.com/pytest-dev/pytest`.
15. Henrich Lauko, Vladimír Štill, Petr Ročkai, and Jiří Barnat. Extending DIVINE with symbolic verification using SMT. In *TACAS*, pages 204–208, Cham, 2019. Springer.
16. Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, Mitsuharu Yamamoto, and Koichi Takahashi. Modular software model checking for distributed systems. *IEEE Transactions on Software Engineering*, 40:483–501, 05 2014. doi: 10.1109/TSE.2013.49.
17. Tim Mackinnon, Steve Freeman, and Philip 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`.
18. S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *International Conference on Quality Software*, pages 127–132, 2014. doi: 10.1109/QSIC.2014.19.
19. Madan Musuvathi, Shaz Qadeer, Tom Ball, Gerard Basler, Piramanayakam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*. USENIX, 2008.
20. Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model checking with LLVM and graph memory. *Journal of Systems and Soft-*

ware, 143:1 – 13, 2018. doi: 10.1016/j.jss.2018.04.026.
21. Björn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design*, pages 210–217. IEEE, 2013. doi: 10.1109/FMCAD.2013.6679412.
22. Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. Technical report, 2008.
23. Vladimír Štill, Petr Ročkai, and Jiří Barnat. Using off-the-shelf exception support components in C++ verification. In *Software Quality, Reliability and Security*, pages 54–64. IEEE, 2017. doi: 10.1109/QRS.2017.15.