DiOS: A Lightweight Approach to Verifying POSIX-Based Programs[⋆]

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

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

Abstract. In this paper, we describe a novel approach to verification of programs which rely on POSIX APIs for their functioning. We leverage DiVM, an existing verification framework, which provides low-level (machine-like) interfaces. In our approach, we do not extend the verifier with special support for high-level features: instead, we have implemented a small operating system with support for POSIX-compatible threads, processes and filesystem. This operating system then becomes part of the run-time environment of the user program (and from the point of view of the verifier, is part of the system under test).
Additionally, we show that a small set of modular components can be combined to obtain various levels of POSIX API support. This means that DiOS can be configured to only provide features relevant to a particular program, yielding minimal overhead, without losing flexibility and while providing comprehensive API coverage.

1 Introduction

Automated verification of real-world software is a complex task involving a large number of components. The programming language in which the program of interest is written is one source of complexity: real-world languages rarely fit into neat formal semantic models. It is, in fact, more typical that they form a maze of corner cases and ad-hoc fixes. It has been a notable trend in the software verification community to move away from trying to capture this complexity in semantic models of the input language, and instead rely on existing compilers to produce a much simpler and semantically cleaner intermediate form. This allows the verification tool to focus on the complexity inherent in software analysis, instead of dealing with programming language idiosyncrasies.

While the input programming language is an immediate and unavoidable concern for a verification tool, it is far from the only problem. Programs do not exist in a vacuum – they execute within complex environments made of the hardware platform, the host operating system, a number of other software

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

components and even network resources, the human user and other factors. The issues that exist in formal treatment of programming languages are only more acute in the case of the execution environment. First of all, the hardware platform poses some unique challenges, especially concurrency and speculative execution of modern CPUs. Those two features have significant impact on execution of a large class of software. The upside is, that for the vast majority of programs, the remaining complexity of the hardware platform is completely abstracted away by the operating system.

On the other side of the environment, the behaviours of external factors like network resources and human users is barely amenable to detailed formal treatment. In engineering circles, those factors are typically modelled as maximally adversarial: the system has to behave correctly regardless of how malformed or even malicious its inputs are. This leaves us with the operating system and the accompanying software as crucial factors of the execution environment.

The paper is organised as follows: Section 2 details the underlying architecture of the verification system the OS was originally devised for. This covers the DIVINE model checker and the corresponding virtual machine, DiVM. Section 3 follows, giving a description of the implemented operating system itself, focusing on its most essential/distinctive features, including the file system, library support and internal kernel structure. Implementation details, as well as portability of the resulting OS, is covered in Section 4 and Section 5 gives an evaluation of the approach. Finally, Section 6 concludes the paper and outlines future work.

1.1 Goals

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 verification of POSIX-based programs. We would like the resulting system to have the following properties:

1. Modularity: minimise the interdependence of the individual OS components. It should be as easy as possible to use individual components (for instance `libc`) without the others. The kernel should likewise be modular.
2. Portability: restrict the coupling to the underlying verification engine, making the OS useful as a pre-made component in creating comprehensive verification tools.
3. Veracity: the system should precisely follow POSIX or other relevant standardised semantics. It should be possible to port realistic programs to run on the operating system with minimal effort.

Since the stated properties are hard to quantify, we provide a qualitative evaluation in Section 5. To demonstrate the usefulness of the approach, we show that a standard UNIX utility, `gzip`, can be easily ported to DiOS and loaded into an explicit-state model checker.

1.2 Contribution

The paper describes our effort to implement a compact operating system on top of an existing verification framework / virtual machine. Our choice of platform is DiVM [9], and its accompanying model checker DIVINE [1]. We have succeeded in implementing the operating system as designed. Even though the design is minimalistic, it is also complete, with fully-functional process manager and filesystem modules, written entirely in C++. The resulting system is not a standalone OS that could be booted on standard hardware, though: instead, it is part of a verification framework and serves as a stand-in for a real operating system during program verification. Besides the implementation itself, there are two main contributions stemming from this effort:

1. We have identified minimal interfaces required for implementing standard programming constructs like threads or exceptions. This allows verification tool implementers to provide those minimal interfaces, instead of directly supporting the complex high-level features.
2. DiOS is designed to be portable. Even though it was created on a particular verification platform, large and useful parts are implemented in plain C or in a subset of C++ that does not need extensive runtime support.

The implementation is available online[1], under a permissive open-source licence, and is also bundled with the model checker DIVINE.

1.3 Rationale

With the hereby presented approach, we aim to improve usability and practicality of verification of real-world software. Particularly, the verification tool cannot rely on the surrounding environment to remain uniform, rather, it is up to the verifier to maximize its own portability. With the concept of an OS operating within the verification tool, the ambition of smoothing out the differences of the host OS is taken one step further. The minimal implementation of an operating system controls/directs the program execution and, at the same time, discards any unnecessary platform-dependend features.

The fact that the OS is shipped with the tool allows for flexibility of use and easy substitution in terms of system components. Additionally, modularity of components reduces the bulkiness and complexity of the verification engine proper. This tends to reduce the number of mistakes and leaves space for the verifier to be partially tailored for the analysed program.

In line with those considerations, we have chosen POSIX, a widespread, largely standardized and well-defined interface, as the basis of our proposed operating system. In a certain sense, it is an obvious choice for a system aimed at real-world program verification.

[1] `https://divine.fi.muni.cz/2018/dios/`

1.4 Related Work

Automated software verification is a field with bad reputation: the promise of powerful, computerised analysis of software correctness is just barely out of grasp, and has been for nearly two decades. Labour-intensive methods, like computer-assisted theorem proving, are known to give good results, but with a very steep price tag. One of the most successful projects in this area is seL4, a microkernel with an end-to-end correctness proof [7].

While automated tools do exist, they are fraught with difficulties. Either the tools use very rough approximations, or else, they cannot be used on realistic software as a whole. The main focus of verification (as opposed to bug hunting) tools are synthetic benchmarks and small, isolated code fragments extracted from existing software, as epitomised by SV-COMP [3], the software verification competition.

Unfortunately, work on verification of entire programs that make extensive use of operating system services is scarce. One tool which allows a degree of such interaction is KLEE [4], which provides a small subset of the standard C library in a fashion similar to our present work: compiled and loaded alongside the program under test. The scope of this support is, however, much narrower than what is provided by DiOS. Additionally, KLEE can allow the program to talk to the host operating system, in a feature similar to what we describe in Section 3.6.

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 [11] and CHESS [8]. Those approaches, though, 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.

On the other hand, standard (offline) model checkers, in turn, rarely support more than a handful of interfaces. The most widely supported is the POSIX threading API, which is supported by tools such as Lazy-CSeq [5] and its variants, by Impara [10] and a few other tools.

2 Preliminaries

For the purpose of this paper, we will regard verification tools as virtual machines. This is more convenient in operating system design than to think of verification tools as implementing programming language semantics. After all, a normal operating system targets a particular hardware platform (machine), rather than abstract computational semantics. The important issue is that the verification tool is capable of executing programs. Clearly, the machine has to be aware of the semantics of the program so it can, besides just executing it, also decide on its correctness.

As stated, our primary target platform is DIVINE, a virtual machine and an explicit-state model checker, and we expect the verified programs to be written in C and C++ programming languages. When we talk about a program, we mean a source file, either in C or C++, or the LLVM bitcode form produced by a compiler from this source.

2.1 Virtual Machine

We have certain expectations from the virtual machine in question. Formally, the operating system uses a set of *hypercalls* to communicate with the underlying machine. However, most of these hypercalls can be either sufficiently emulated using other constructs available in C or else their use is limited to a specific subset of DiOS functionality. In those cases, the functionality can be disabled to allow execution in an environment without equivalent facilities. A more detailed account of these considerations will be presented in Section 4.

Since DIVINE is our primary platform, we will describe it in a little more detail. It has multiple components, the one which is most pertinent to our topic is DiVM, the virtual machine that DIVINE uses to execute programs. DiVM is responsible for generating the state space of the program, for memory manipulation and for ensuring the validity of the verification outcome. Internally, it works with LLVM instructions – that is, within the virtual machine and during the verification process, the to-be analysed programs are represented in the LLVM IR form.

3 Architecture of DiOS

This chapter aims to outline the structure of the proposed system, the respective components that form it and the means of communication of the components within the larger verification system, with special focus on DiOS.

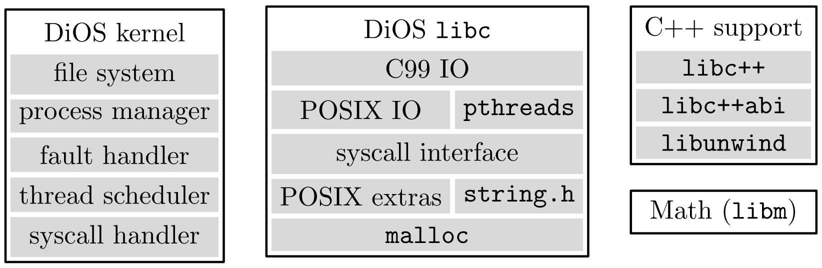

Fig. 1. The architecture of DiOS.

The system is formally split into a kernel and a userspace component, although the enforcement of this separation is optional, depending on the capabilities of the platform. The kernel can be linked into the program under test as a library, along with `libc` and other userspace components.

3.1 The Kernel

Since our goal is to mimic standard POSIX-compliant operating systems, we have followed the traditional architecture associated with those systems. There is a privileged *kernel*, which enforces process separation and abstracts hardware. Both those roles are diminished in the case of DiOS, because we target verification tools. Those tools are expected to enforce memory safety (partially obviating need for isolation features in the kernel) and to provide a fairly small interface (reducing the need for abstraction).

With one important exception, there is no platform interface to speak of: there is no hardware to be managed, all traditional facilities being replaced with virtual, memory-backed simulations. For instance, since there is no hardware block storage facility, the only file system implementation available in DiOS is a *memory filesystem*, similar to OpenBSD `mfs` or Linux `tmpfs`.

The one exception to this rule is the existence of a *system call passthrough* mode, where the DiOS kernel forwards system calls from the program under test, through the virtual machine all the way to the real host operating system. This mode of operation is described in more detail in Section 3.6.

3.2 System Calls

In previous iterations of DiOS, the system call mechanism was closely modelled on the traditional approach used in real processors. This was because the native platform that DiOS was initially written for offered an *interrupt* facility. It has been, however, realised, that this is rather uncommon in verification tools and that the approach hampers portability.

For this reason, the current system call interface of DiOS is built around concepts similar to the more modern approach used in *fast system call* implementations, embodied by the `syscall` and `sysret` instructions in current `x86-64` processors. The major advantage of this approach is that, unlike interrupts, the `syscall`-style traps can be easily emulated with plain function calls in C. More details about the emulation and the compromises it entails are provided in Section 4.

DiOS provides the standard subset of system calls that would be expected in a POSIX-compatible operating system. Those roughly fall into the following categories:

1. Thread and process management. This includes system calls to establish new threads, sufficient to implement the `pthreads` interface (the system calls themselves are not standardised by POSIX), the `fork` system call, `kill`, miscellaneous process and process group management (`getpid`, `getsid`, `setsid`, `wait` and so on). Notably, `exec` is currently not implemented and it is not clear if this can be done in a portable fashion. The thread- and process-related functionality is described in more detail in Section 3.4.
2. File system calls. 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 for thread-safe use of relative paths.
3. Network-related system calls. The standard BSD socket API is implemented, allowing threads or processes of the program under test to use sockets for communication.

3.3 Kernel Components

The decomposition of the kernel to a number of components serves two goals: first is resource conservation – some components have non-negligible memory overhead even when they are not actually used. This may be because they need to store auxiliary data along with each thread or process, and the underlying verification tool will need to track this data throughout the execution or throughout the entire state space, adding to a sizeable contribution.

The second reason for splitting up the kernel is portability: some components have very simple requirements, but others need fairly sophisticated support from the underlying platform. This is mainly the case with thread support and C++ exception support, which both need an ability to manipulate the stack of the program, or, in the case of threads, even create new stacks and stack frames on demand. Naturally, it is not practical for all verification systems to provide such facilities. Allowing DiOS to be configured without thread support allows us to also target those verification platforms.

The components of the kernel are organised as a stack, where upper components can use services of the components below them. While this appears to be a severe limitation, in practice this has not posed substantial challenges, and the stack-organised design is much simpler than the alternatives.

The following components are available, some coming in several alternative implementations:

– A scheduler. There are 3 scheduler implementations, two of them asynchronous – that is, they implement thread-based parallelism. One is a standard scheduler which simply interleaves all runnable threads, useful for verification of safety properties in parallel programs. The other asynchronous scheduler includes a fairness provision, making it more suitable for verification of liveness properties. Finally, there is a *synchronous* scheduler, suitable for verification of C code synthesised from data-flow programs and hardware designs.
– File system. This component provides both a memory-backed filesystem as well as pipes and sockets. Most system calls are implemented in this component (i.e. both the second and third groups listed in Section 3.2).
– Fault handler. This component takes care of responding to error conditions indicated by the underlying verification platform. It is optional, since not all verifiers can report problems to the system under test. If present, the component allows the user to configure certain problems to be ignored during verification. The C library (cf. Section 3.7) also uses this component to report assertion failures and the like.

– Process manager. This component implements the `fork` system call, along with most other process-related services. It requires that one of the asynchronous schedulers is present below in the stack.
– The base component. This component provides fallback implementations of all system calls, that simply raise a runtime error. This means that the system call table is not affected by configuration changes, which is important because configurations can be selected at runtime. This component always forms the bottom of the configuration stack.

The first goal we set in Section 1.1, modularity, aims to separate the components and have them interact with one another using minimalist interfaces. One advantage that derives from the modular view is the option to swap a component for another with little extra effort – this is mainly illustrated in the filesystem and scheduler components in the above list, which each has multiple implementations. Certain components may be even left out entirely. Finally, minimal interfaces also make it easier to design and reason about the components and minimises the mistakes made if every component is responsible for a small well-defined part of the system.

On one hand, the respective components ought to have minimal dependencies between them. A different way as to how to regard modularity is based on the idea that not all programs use or need every feature of the operating system. A single-threaded program need not have process-specific functions, such as the `fork` system call or functions for manipulating attributes of (nonexistent) child processes at its disposal. Similarly, if a program does not require file system support, it suffices to supply it with a considerably reduced version of a file system.

3.4 Threads and Processes

One of the innovative features of DiOS is that it contains an implementation of the POSIX threading API, without requiring the verification tool to contain special thread support. This `pthread` implementation is in fact part of the `libc` shipped with DiOS and only relies on a few system calls provided by the DiOS kernel, and on the thread scheduler kernel component.

The scheduler is implemented in terms of a *nondeterministic choice* operator, which is usually available in software verification platforms. More details are provided in Section 4.1. The final ingredient is the ability to create and switch execution stacks. In this case, there is no choice but to implement this support in a platform-dependent way, since stack representation is platform-specific. On our primary platform, execution stacks have a particularly simple representation and DiOS only needs a hypercall for transferring control to a different stack – everything else is implemented in DiOS itself.

3.5 Synchronous Systems

As mentioned in Section 3.3, the flexible, component-based design allows components to be swapped in and out of the kernel. We have taken advantage of this flexibility when confronted with the task of designing a system for verification of C code synthesised from synchronous system specifications. In this case, we have replaced the standard asynchronous scheduler suitable for verification of software with a synchronous design.

The principle of a synchronous system is that time passes in a consistent fashion and all of the units forming the system are subject to a global clock, which serves as a synchronisation mechanism. This means that, in synchronous systems, time works in so-called *ticks*. In a single step (that is, one tick), each component computes its outputs, taking into account its inputs and optionally some internal state.

We introduced *tasks* as an atomic synchronisation unit (with asynchronous tasks being simply the same as threads), and demonstrate the use of the synchronous scheduler on Simulink-synthesised programs. Simulink is a modelling environment for data-flow simulations. Our previous approach coupled to Simulink designs at a higher level [2], but the current approach is more faithful, since it uses the same C generator as the actual production synthesis [2]. With the synchronous scheduler in DiOS, it is possible to verify Simulink-synthesised programs without further changes in the verification core.

3.6 System Call Passthrough

As mentioned, traditional verifiers work with a simulated memory environment and do not interact with the surrounding operating system. This is because all behaviour of the program has to be recorded for the purpose of verification and the execution has to be reproducible, i.e. behave the same way (that is, every run has to be reproducible). Communication with the environment in terms of I/O and file manipulation violates both of these constraints; however, the verification system can be adapted to the new circumstances. For instance, we have extended the DIVINE verification framework with the option to forward system calls to the host operating system and manipulate real files and memory [6].

This capability is not unique to DIVINE, in fact: in principle, it is quite easy to extend other tools with a similar interface. For instance, the symbolic execution tool KLEE [4] allows system calls to be passed through to the host system, although it uses a different interface to this. Adapting KLEE to support the interface expected by DiOS is quite straightforward.

3.7 The C Library

DiOS comes with a complete ISO C99 standard library. This includes the `setjmp` and `longjmp` functions, which are often problematic in the context of verification. The implementation of those two functions is closely related to C++ exception support, and is described in more detail in Section 3.8.

The remainder of the C library can be broken down into a few categories:

[2] There are multiple tools which generate C code from Simulink designs, one such tool is Simulink Coder.

- 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 most recent error code due to an API call. On platforms with threads (like DiOS), it must be thread-local.
- Multibyte strings: conversions of unicode character sequences to and from UTF-8 is supported.
- Time-related functions: time and date formatting (`asctime`) is supported. Obtaining and manipulating wall time, and using and setting timers, is not, although the relevant functions are present as simple stubs.

In addition to ISO C99, there are a few extensions (not directly related to the POSIX OS 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.

3.8 C++ Exceptions and `longjmp`

DiOS also includes low-level support for non-local jumps and stack unwinding. This functionality is implemented by manipulating the execution stack in a platform-dependent way. The stack unwinder provided by DiOS is then sufficient to implement C++ exceptions using off-the-shelf components [12]. By default, DiOS ships with an essentially unmodified `libc++abi` runtime support library. Together, these components, along with metadata instrumented into the program, enable transparent and faithful support for C++ exceptions.

Additionally, the same low-level stack unwinder is used in the implementation of the C99 functions `setjmp` and `longjmp`, which can be used by user programs to directly jump from one function to one of its (even indirect) callers. The low-level unwinder is described in more detail in Section 4.2.

4 Implementation & Portability

As suggested in previous sections, DiOS requires (or in some cases, can take advantage of) certain features in the underlying verification platform. We first list those requirements and their meaning. The rest of this section will then explain how the most important features of DiOS are implemented and how they use those platform features. This information is also summarised in Table 1.

Table 1: Summary of available features and what they require from the underlying verification platform: 'stack' means direct manipulation with the execution stack, 'nondet' means nondeterministic choice, 'memsafe' means that the feature relies on enforcement of memory safety. 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	✓			
passthrough mode				syscall execution
replay mode		✓		

The relevant platform features are as follows:

- Nondeterministic choice. The only type of choice required by DiOS is picking a value from a small set of integers. The scheduler uses nondeterministic choice to pick the thread to execute, `malloc` uses it to decide whether it should fail, and so on.
- Memory management. The implementation of `malloc` needs to obtain memory from the system, and this operation must be directly supported by the underlying virtual machine.
- Stack manipulation. This is, arguably, the trickiest of the features that DiOS requires. It entails iteration over individual stack frames (activation records), access to return address and the ability to edit the stack. In particular, the current implementation expects that the execution stack is a linked list of frames.
- Memory safety. The process support in DiOS relies on the assumption that the verification platform enforces memory safety: it must be impossible to construct "accidentally valid" pointers to existing memory.

– Heap cloning. Like memory safety, this feature is only required for process support – the `fork` system call needs to be able to clone all memory reachable through a given pointer.
– Supervisor mode. While this feature improves the credibility of the verification outcome, it is not strictly required by DiOS.
– Host syscall execution. This feature was already described in Section 3.6. It can be used along with a single-execution mode of the verifier to observe program behaviour in its native environment.

4.1 Threads and Processes

As outlined in Section 3.4, the threading support in DiOS has two main components. The first is a low-level implementation of a handful of primitives, in particular a simple thread scheduler based on nondeterministic choice, and support for creating new threads and their forcible destruction.

The second component is the POSIX threading API, which builds on top of this low-level thread support. The POSIX thread API is quite comprehensive and contains over 100 functions. The current DiOS implementation provides approximately half of them, including all that are commonly used. Among other things, the implementation is sufficiently complete to support the C++11 threading library. The implementation of the high-level POSIX API is, in terms of the low-level DiOS-specific API, quite straightforward. Therefore, in the remainder of this section, we focus on the low-level interface.

The most interesting part of the threading support is the scheduler. Clearly, the operating system alone cannot contain the entire machinery to allow verification of threaded programs. One external components is required: the input program needs to be instrumented with *interrupt points*. This is a simple procedure when performed using a low-level program representation: for LLVM IR, the instrumentation takes fewer than 200 lines of C++ code. The instrumentation simply needs to insert calls to a C function called `__dios_interrupt`, which first takes care of internal bookkeeping and then notifies the verification tool to generate and store a new program state.[3] Finally, it invokes the scheduler, which non-deterministically chooses the next thread to execute.

While a naive implementation of this approach would not be very efficient, it is still suitable for verification of small programs. With additional support from the verification platform, it is straightforward to include state space reductions based on partial orders and/or invisible actions. In this case, the instrumentation can make the calls to `__dios_interrupt` conditional: the scheduler will then be only called if a visible action has been performed by the currently executing thread.[4]

[3] This notificaton mechanism is optional, and exists mainly for the convenience of the verification tool.

[4] This is exactly how state space reductions are implemented in DIVINE: the virtual machine keeps track of memory accesses and provides a hypercall to test for their visibility. The instrumentation ensures that `__dios_interrupt` is only called when needed.

4.2 Stack Unwinding

The DiVM implementation of the low-level stack unwinder in DiOS relies on the linked-list stack representation used on this platform. Additionally, the layout of each stack frame is such, that the return address and return values can be directly manipulated by reading and writing into the stack frame.

The unwinder is implemented as a C function, `__dios_unwind`, which has its own stack frame, and operates by unlinking frames from the middle of the stack. The memory used for local variables associated with the unlinked frames is freed. Control then returns to the caller of `__dios_unwind` which may further manipulate the values in the target frame, e.g. to influence control flow (this is important in both `longjmp` and in C++ exceptions). For this further manipulation, DiOS provides a pair of functions, `__dios_set_register` and `__dios_get_register`, which are lower-level counterparts to the standardised `libunwind`[5] API for manipulating control flow.

4.3 Binary Compatibility

When dealing with verification of real-world software, the exact layout of data structures becomes relevant, mainly because we would like to generate native code from the verified bitcode file. To this end, the layouts of relevant data structures and values of relevant constants are automatically extracted from the host operating system and used in the DiOS `libc`. As a result, the native code generated from the verified bitcode can be linked to native (host) libraries and executed as usual.

5 Evaluation

The goals set out in Section 1.1 have been, to a large degree, met. In particular, the modularity of the system is well established, as it is used in practical verification tasks in many different configurations, whether it is as a substrate for verification of parallel algorithms that use the POSIX threading API, or in verification of LTL properties of synchronous, Simulink-derived systems.

To evaluate both the veracity and general usability of DiOS, we have done a case study based on the `gzip` compression utility. While this is a simple utility (compared to typical programs), it serves very well to illustrate problems inherent in automated software verification. Even though the implementation of the compression algorithm used in `gzip` is entirely platform-independent, there is a range of platform-dependent issues that `gzip` needs to deal with. This includes input and output, but also resource limits, command line parsing (`getopt`) and

[5] The `libunwind` library is part of the application binary interface on many POSIX operating systems, and its interface is used, for instance, by C++ runtime libraries. DiOS provides a fairly complete `libunwind` implementation based on the three functions described here. More details about the relationships and inner workings of these components are given in [12].

so on. Since POSIX platforms (and pre-POSIX UNIX) have a number of subtle differences, `gzip` also bundles a portability library known as `gnulib` which comprises almost 18 thousand lines of code and often relies on minute details of the operating system (for comparison, `gzip` itself is less than 7000 lines).

The good news is that `gzip`, along with `gnulib`, can be compiled for DiOS with only 4 minor changes (each 1-2 lines) in the `gnulib` implementation, and no changes in `gzip` itself.[6] The resulting LLVM bitcode file can be loaded and verified using DIVINE. Additionally, since DiOS `libc` is binary compatible with the host version of the library (cf. Section 4.3), the verified bitcode can be further translated into native code and executed in the host operating system as a standard program.

Table 2: Selected benchmarks verified within DiOS.

category	programs	states	search	counterexample
posix	59	3124	2:29	6:24
pthread	170	4556 k	3:16:14	22:10
svcomp	113	4149 k	105:25:19	29:42
weakmem	82	5624 k	5:45:33	32:48
libcxx	309	1765 k	28:52	22:56
undef	35	119	0:27	2:27

Finally, in addition to these qualitative tests, we have successfully used DiOS in verification of over 900 benchmarks and test cases. The total verification time was over 112 hours and generated over 12 million states. The benchmark set included C and C++ programs of various complexity and reliance on OS facilities. Some of the categories are listed, along with program count, state count and total verification time, in Table 2.

6 Conclusion & Future Work

We have presented DiOS, a POSIX-compatible operating system designed for verification of programs. 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 preliminary case study shows, the proposed approach is a viable way forward.

There are two important future directions: further extending the coverage and compatibility of DiOS with real operating systems, and porting DiOS to other verification tools.

[6] The complete patch for the `gzip-1.8` source tarball is available online at `https://divine.fi.muni.cz/2018/dios/`

Bibliography

[1] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. 2017.

[2] Jiri Barnat, Jan Beran, Lubos Brim, Tomas Kratochvíla, and Petr Ročkai. Tool chain to support automated formal verification of avionics Simulink designs. In *FMICS*, number 7437 in LNCS, pages 78–92. Springer, 2012. URL `http://dx.doi.org/10.1007/978-3-642-32469-7_6`.

[3] Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses report on SV-COMP 2016. In *TACAS*, pages 887–904. Springer, 2016. ISBN 978-3-662-49673-2. doi: 10.1007/978-3-662-49674-9_55.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[5] O. Inverso, T. L. Nguyen, B. Fischer, S. L. Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 807–812, Nov 2015. doi: 10.1109/ASE.2015.108.

[6] 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.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM. doi: 10.1145/1629575.1629596.

[8] 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, December 2008.

[9] Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model checking with LLVM and graph memory. 2017. URL `https://arxiv.org/abs/1703.05341`. Preliminary version.

[10] Björn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design*, pages 210–217. IEEE, 10 2013. doi: 10.1109/FMCAD.2013.6679412.

[11] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, 2008.

[12] Vladimír Štill, Petr Ročkai, and Jiří Barnat. Using off-the-shelf exception support components in C++ verification. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 54–64, 2017.