Applying OS Fuzzing Techniques To Unikernels

Oliver Dunk

Supervisors:
Dr. Pierre Olivier, University of Manchester
Dr. Razvan Deaconescu, University Politehnica of Bucharest
Felipe Huici, NEC Laboratories Europe

April 2021

1 Abstract

Fuzzing is a testing technique which involves running programs with random inputs, often guided by instrumentation data such as code coverage. Unikernels are a new type of Operating System, where an application and kernel are compiled in to a single kernel image. Unikernels are currently the focus of active research and are growing in popularity due to the possibility of using them in cloud environments. In this project, I investigated the application of fuzzing to unikernels. While there is significant research applying fuzzing to traditional operating systems like Linux, the same research does not exist for unikernels. I propose a breakpoint based approach to collecting coverage information for input generation, build a new fuzzer that uses this technique, and evaluate its effectiveness.

Contents

2 Introduction

Fuzzing is a testing technique in which programs are repeatedly passed random data in an attempt to find exploitable bugs in their implementation. Code coverage is measured while running with a particular input, and promising inputs which generate new coverage are explored further by running a number of mutated versions of them. This technique has been applied extensively to OSes (Operating Systems) like Linux, and is responsible for finding a large number of security issues. However, there has been little work applying this same technique to non-mainstream OSes. Unikernels are a type of OS designed for use in cloud applications, where a bespoke operating system is built by compiling the kernel and application together in to a single binary. These are growing in popularity, so the application of fuzzing provides an important opportunity to identify security critical bugs in these new projects. On top of this, many unikernels claim to provide POSIX compatibility through a system call interface, providing an opportunity to investigate the use of fuzzing for determining compatibility.

Applying fuzzing tools which already exist to unikernels is a challenge, due to constraints such as the requirement of platform specific features (kcov on Linux) or support for higher level languages (such as Go). I started my project by researching these challenges, and then based on this research, built a new fuzzing tool in C++. The fuzzer input generation is guided using coverage collected by a modified version of QEMU which inserts breakpoints at the start of functions within the kernel. I demonstrated the effectiveness of this approach by identifying a number of bugs, including two in Unikraft and five in HermiTux (see Appendix). I also show that this fuzzer is generic by demonstrating that it can be ported to other OSes like Linux with minimal effort.

2.1 Aims and Objectives

The following list is taken from the original project description. For each task, I have added my own criteria which I will use to decide if the task was completed.

Aim/Objective	Criteria for success
Study of fuzzing concepts and existing tools	For this to be successful, I hope to be able to demonstrate in this report where I have found approaches in existing work that I can adapt, as well as gaps in current research that I have taken steps towards addressing.
Select the appropriate tool, define a fuzzing strategy, and select a target unikernel model such as OSv, Rumprun or HermiTux	I will consider this successful if I am able to choose and justify a fuzzing strategy, explaining how the strategy could be used to fuzz the unikernel I chose.
Apply the fuzzing strategy and fix or report the potentially uncovered bugs	I will consider this achieved if I am able to show evidence of applying fuzzing techniques to one or more unikernels. I also expect to be able to include a number of bugs in this report that I have found and reacted to.

3 Background and Challenges

3.1 Unikernels

Originally described in Madhavapeddy et al.'s Unikernels: library operating systems for the cloud [14], unikernels are bespoke OSes, which run a single program chosen at compile-time. Unlike traditional OSes, which have many address spaces and provide support for running multiple programs simultaneously, unikernels operate in a single address space and programs run directly with the kernel in a virtual machine, on a hypervisor.

One major benefit of this approach is the reduced size of the deployed binary. Since unused parts of the operating system can be removed, the attack surface is reduced, and the binary itself is smaller. If a particular program does not work with files, for example, large sections of library code to work with files can be excluded altogether.

As well as this, unikernels have the potential to perform faster and use less system resources. Unikraft claim boot times of 10ms or less and a memory footprint of just a few MBs of RAM [13]. This makes them particularly well suited to deployment in cloud environments, where it would be common for a machine to be dedicated to a single application.

Many existing unikernels such as HermiTux [17], Unikraft [13], OSv [12] and rumprun [10] offer a POSIX-like interface for communicating with the kernel. Unikraft provides a system call shim layer [19] which directly maps system call numbers to the corresponding handler, and OSv has compatibility with the Linux ABI [18]. This compatibility is present in other unikernels like HermiTux and Rump too. With this in mind, this could allow fuzzers built for POSIX-like OSes to be applied in the project.

3.2 Fuzzing

Fuzzing is a process which consists of a program being run repeatedly with random, although often guided, inputs. The goal is to explore as much of the code as possible, including paths which may be rarely used, as these often exhibit undefined behaviour when passed unexpected inputs. Fuzzing can take place in userspace (AFL [5]) or the kernel (syzkaller [7]). The strategy taken for input generation can often be split in to one of three broad categories [4]:

3.2.1 Blackbox Fuzzing

Blackbox fuzzers have no understanding of a program's internal structure, so these types of fuzzers usually generate inputs entirely randomly. This makes input generation fast, but statistically a much larger number of inputs are required to test the program thoroughly, as the vast majority of inputs will make repeated use of the same code paths or cause the program to terminate at a very early stage of execution. One approach taken to improve the effectiveness of blackbox fuzzing is providing a well-formed input known as a seed, which is then mutated to produce new inputs. These mutations are small changes such as flipping a bit or overwriting a small section of the input. The mutations usually only affect a small part of the input, which increases the chance the mutated input will be similar enough to the well-formed input to trigger interesting code paths in the program.

3.2.2 Grammar-Based Fuzzing

Grammar-Based fuzzing exploits structure in the expected input of programs. For example, a program may be expecting JSON, in which case rules could be defined to ensure curly braces and quotes are balanced, and that the data types of particular fields are respected. This input generation strategy works especially well for many types of programs, although one limitation is that the process of writing the grammar itself requires a lot of manual effort.

3.2.3 Whitebox Fuzzing

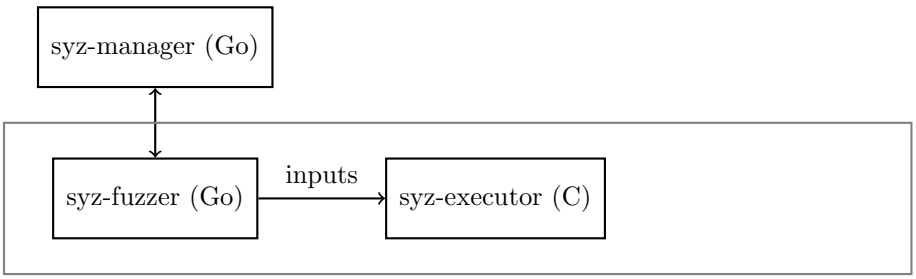
The final type of fuzzing, and the focus of my project, is whitebox fuzzing. In this case, the fuzzing strategy relies on the programmer having full access to the source code of the program. This has two big benefits. Firstly, the code can be instrumented such that feedback is provided after any given input is run. This is usually in the form of coverage, and allows mutated inputs, or mutations themselves, to be favoured based on if new code has been covered. Additionally, this coverage provides a useful metric as to how many execution paths have been tested by the fuzzer. Mutation strategies can be compared based on the coverage generated, and fuzzing can stop when the code coverage begins to converge.

3.2.4 OS Fuzzing

One potential target for fuzzing is the kernel of an operating system. Fuzzers usually depend on having a well defined entry point, and in the case of an operating system, this is usually either the system call interface or the wrappers

around it provided by libc. Fuzzing an operating system is as a whole fairly similar to fuzzing a program in userspace, however it does present additional challenges. For example, the stability of the entire operating system may be affected. As a result, fuzzing is usually run inside a virtual machine, with at least some code for persisting state such as the corpus of interesting inputs running on the host. The host is also responsible for starting new virtual machines when an existing one crashes or becomes unresponsive.

Google's syzkaller [7] is one of the most well known operating system fuzzers. In syzkaller, syz-manager starts virtual machines running a version of the kernel built with kcov coverage support. Inside, the syz-fuzzer process is started, an application written in Go which generates inputs and collects coverage data. To execute inputs, the short-lived syz-executor is started. syz-fuzzer and syz-manager synchronise data over time, and this allows syz-manager to persist the fuzzing state even if the virtual machine exits and a new one needs to be started.



3.3 Challenges applying existing techniques to unikernels

A large amount of work has been put in to fuzzing tools since at least the early 2000s [4]. Consequently, it was important early on in the project to understand if these existing tools were sufficient for doing similar testing of unikernels. Unfortunately, there were several challenges which meant existing tools could not be easily reused.

3.3.1 Language Support

Google's syzkaller [7] is an "unsupervised coverage-guided kernel fuzzer", which initially made it a promising project. However, syzkaller generates sequences of system calls inside a process called syz-fuzzer. This is written in Go, a language supported by only some of the unikernels I hoped to support. Additionally, even considering just the unikernels which claim Go support, this is tested less than languages like C and therefore significantly more unstable.

3.3.2 kcov

The Linux kernel can be built with the `CONFIG_KCOV` flag. This adds instrumentation which collects kernel code coverage during system calls as well as the operands used in comparisons. This is relied on by syzkaller to perform coverage-based fuzzing [8].

Currently, this is specific to the Linux kernel. Additionally, even if it was supported by other OSes, making it a requirement would limit fuzzing to OSes which either implemented it already or had time available to implement it. This was a requirement I wanted to avoid.

3.3.3 OS maturity

Linux is the target of the vast majority of kernel fuzzing research. Since it has been tested extensively, crashes are rare, and any overhead introduced by the need to start a new virtual machine when one running the fuzzing process faults is negligible. On the other hand, many unikernels have never been tested by fuzzers before. Consequently, a mechanism is needed for restricting the input that is generated such that known bugs are not hit repeatedly. Additionally, since unikernels only run a single program, the security risk associated with allowing a program which usually runs in user space to invoke undefined behaviour in the kernel is less obvious. This means input validation and error codes implemented by Linux may not be implemented by unikernels for performance reasons. This means even a well tested unikernel may appear more unstable during fuzzing.

Bugs may also be present in less mature OSes which makes porting complex fuzzers hard. As an example, during this project I encountered a page fault when statically initialising a large vector in a C++ program (see Appendix). This global vector contained a description of the available system calls and their arguments, which was required by my fuzzer to generate inputs. Due to the bug, when compiling for Unikraft, I was forced to limit the number of system calls that my fuzzer was aware of. I do expect that this limitation will be short-lived, as the bug has been reported to the Unikraft team and is already under investigation. However, existing fuzzers like syzkaller are fairly complex and may run in to many bugs like this, which still adds additional work to porting even if the bugs identified are eventually fixed.

3.4 Related works

The previous section focused very heavily on the syzkaller project. While I didn't apply this directly to unikernels, it is the closest piece of work to what I wanted to accomplish and many of the decisions I made during the project were inspired by it. On top of this, I also read a selection of other papers, which provided additional background on the OS fuzzing research which has taken place over the last few years.

Kim et al. (2019) [11] as well as Xu et al. (2019) [20] both investigate the fuzzing of file systems, treating the file system as another dimension to be fuzzed. The file system is mutated as well as the input syscalls and this means a great number of situations can be tested.

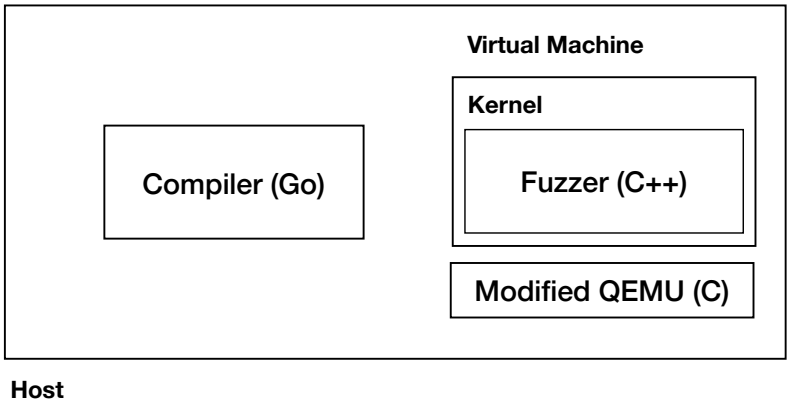
Hazimeh et al's (2020) [9] Magma investigates the evaluation of fuzzers. They insert bugs for a fuzzer to find, and using instrumentation, are able to differentiate between executing code involved with a bug and actually exercising the unintended behaviour.

DIFUZE, the focus of Corina et al. (2017) [1], identifies kernel drivers as a particular area of interest for fuzzing. Communication with these drivers is through the ioctl interface, and often involves complex data structures. The

parsing required for structures like this means complex code may be involved, creating an an unusually large attack surface. The paper investigates ways of determining the data structures used by a driver through static analysis and the result is able to identify more bugs in this context than syzkaller.

4 Design and Implementation

With all of the above in mind, I decided that building a fuzzer from scratch would be easier than trying to port an existing fuzzer to unikernels. I also thought a new fuzzer would be an opportunity to focus more heavily on older languages, producing something applicable to future work more easily.



The fuzzer I built consists of three main components. Firstly, a compiler written in Go converts data about available system calls from the syzkaller Syzlang format to C++ code. Second, a fuzzer written in C++ loads this data and executes random system calls generated using this data. Finally, the fuzzer is compiled as part of a unikernel or loaded alongside Linux, at which point it is run inside of a modified version of QEMU. The QEMU build is modified to collect code coverage data through breakpoint based instrumentation, which is sent back to the fuzzer to guide the input generation process.

4.1 Input generation

To effectively build a fuzzer for unikernels, the first problem I wanted to solve was input generation.

4.1.1 Syzlang

Through my research, I learned that syzkaller uses Syzlang, a "syscall description language". This is a grammar which describes the syscalls available in Linux, the arguments they take, and the values they return. Importantly, this is more detailed than what can be exported directly from the code. For example, the code used to describe the open syscall looks like this:

```
open(file ptr[in, filename], flags flags[open_flags],
     mode flags[open_mode]) fd
```

This returns fd, a "resource" which is defined as follows:

```
resource fd[int32]: -1
```

From this, we can tell that not only is an int returned from the system call, but that this is valid for other system calls which require a file descriptor. This allows the fuzzer to generate inputs which are more likely to be valid, and consequently more likely to trigger coverage in the code.

It was clear that access to the data in Syzlang would be useful, but the compiler in syzkaller loads this in to Go structs. I needed a way to access the data from C instead.

4.1.2 Cgo

One option I explored was using Cgo [3], a feature of Go which allows code to be compiled as a C library. Unfortunately, this produces a library which is dynamically linked against libc, and some of the expected functions will not be found when the code is run inside of a unikernel. This is evident when running the nm command on the output file, as shown in Figure 1. The command shows symbols which are missing and expected to be defined elsewhere with a U. Providing implementations for some of these may be possible, but I did not find a way to provide all of the functions which were needed.

```
U _madvise
U _malloc
U _mmap
U _munmap
U _nanosleep
U _open
U _pipe
U _pthread_attr_destroy
U _pthread_attr_getstacksize
```

Figure 1: Output of nm command showing undefined symbols in library

While this approach didn't work out in the end, I was able to get fairly close. The code available in the `cgo-lib` folder is a proof of concept demonstrating syzlang being passed as a string from C to Go, the file being compiled by syzcaller, and a partial result being returned to the caller in C.

4.1.3 Transpiling to C++

While I didn't want to depend on Go during fuzzing, I decided that I could use the existing Go code beforehand to generate a C++ file containing equivalent types. To do this, I first created a header file defining a number of Vector types that the program could depend on. For example, a system call is defined as follows:

```
typedef struct Syscall {
  int ID;
  uint64_t NR;
  string Name;
  string CallName;
  int MissingArgs;
  struct SyscallAttrs Attrs;
  vector<struct Field> Args;
  int ret;
} Syscall;

extern vector<struct Syscall> SYSCALLS;
```

In my Go code, I produce an output file amd64.cpp which meets this definition. As well as providing information about the available system calls, I also export information about types (which includes numbers with different bit representations) and constants (a map of values such as the bits to set for particular flags).

I first open the file, and write out the required header include. I then start the declaration of the `SYSCALLS` vector and output a single line per system call, containing all of the data needed to populate the struct. C++ was incredibly useful here, as the shorthand notation provided for initialising vectors allowed me to avoid producing code to allocate memory as I would need in C. The resulting file contained 304 system calls, and a range of information about argument types, spread across 979 lines in total. The equivalent C code would have been much longer, as each system call would have been preceded by several lines initialising constants such as strings before the actual system call definition could be included.

```
#ifndef amd64
#define amd64

#include "amd64.hpp"

vector<struct Syscall> SYSCALLS = {
  {
    .ID = 0, .NR = 163, .Name = "acct", .CallName = "acct",
    .MissingArgs = 0,
    .Attrs = {
      .Disabled = false, .Timeout = 0, .ProgTimeout = 0,
      .IgnoreReturn = false, .BreaksReturns = false
    },
    .Args = {
      {
        .name = "filename", .type = 222,
        .hasDirection = false, .direction = 0
      },
    },
    .ret = -1
  }
};

...
```

Figure 2: `amd64.cpp` file produced for a single system call

This file contains some system calls more than once, because the Syzlang file provided by syzkaller allows a single system call to produce more than one return type. For example, the open syscall is included twice, returning a `fd` representing a file in one case and a `fd_dir` representing a directory in the other.

Notably, I made two minor changes to the syzkaller source code to enable the use of the compiler in this way. Firstly, the compiler emits warnings when types are unused, but these were previously emitted as errors which caused compilation to end. I reduced the level which these are logged at to avoid this. Additionally, the required `deserializeFile` function was private to the syzkaller package, so I made it public allowing me to access it externally. Both of these changes are provided as the file syzkaller.patch to be applied on top of a clone of the repository.

This approach was sufficient for generating the data needed to continue development of the fuzzer, and I believe the fact that we rely on code from syzkaller should make adapting to future Syzlang changes fairly easy.

4.2 Building the fuzzer

Once I had information about the syscalls available on OSes with a Linux like interface, I was ready to build the main fuzzer. This is composed of a few main parts.

Firstly, the fuzzer holds a corpus, which is a set of inputs that have previously generated new coverage and can be iterated upon for further fuzzing.

This can be initialised with "seed" data, data known to be valid and which generates notable coverage of the program, but in my case it begins empty and is populated throughout the runtime of the program. The data within this corpus is not individual syscalls, but instead a list of "programs", which are sequences of syscalls. This allows inputs to be generated which perform multiple steps, such as opening a file and then writing some data to it. These programs are represented as structs with a vector of calls and a vector of arguments within them. Arguments can be absolute values, such as flags, or references to the return value of previous syscalls when a resource has been generated.

This corpus is accessed from within a main loop, which decides with random probability if we should perform mutations on an existing input from the corpus or generate a new program entirely randomly. The probability of picking between each of these cases can be changed to alter the behaviour of the fuzzer. I chose to generate new inputs every 100 iterations, which is the same value used by syzkaller.

To generate inputs, the fuzzer needs to be able to produce random arguments which meet the constraints of a given type. For example, the "int8" type represents a number representable by 8 bits, so the fuzzer needs to be able to produce a value in this range. In the case of strings, these may represent a filename, in which case we need to initialise a string with a valid filename and pass a pointer to the first character as an argument to the syscall. Again, resources are a fairly special case. After a random syscall with a resource argument has been selected, the fuzzer checks to make sure at least one previous syscall produces the required resource, moving on to adding a different call if that is not the case. Assuming the syscall can be added, a random call that produces the desired resource is selected from the program's existing calls. The return value from this call is used. A new program is formed by choosing 20 random syscalls and populating them with 20 random argument values. This forms the basis of my generation strategy.

In the case where existing inputs from the corpus are used, these are mutated so they have the potential to generate different coverage to when they were run previously. In my project, I did not focus heavily on the implementation of a mutation strategy, and only implemented a single mutation where a random syscall from the program is removed. This is because I agreed with my supervisors that there was no original work to do here and that it could be easily added at a later date. Usually, a much larger number of mutations are implemented, including combining two programs or inserting a new call. It is also common to mutate individual arguments, for example incrementing a value, applying a shift to an integer, or flipping a bit in a more complex structure.

Finally, I needed to be able to execute system calls with the inputs I had generated. I did this implementation in the file `executor.cpp`.

The first challenge I ran in to was that the macro used to perform system calls is different between OSes. Usually system calls are made through libc and not directly from user code, so the macro is not needed. However, in my case, the executor is an exception to this. In Linux, the file `unistd.h` is included and and the macro syscall is invoked. This takes the syscall number as its first value and any parameters to the call as subsequent arguments. Unikraft was similar, but uses the `uk_syscall` macro. I also wanted to be able to test my fuzzer on my personal macOS machine without worrying about the consequences of invoking arbitrary syscalls, so this was yet another mode I needed to support. To enable

this, I made use of a compiler flag which controlled the inclusion of three different `executeSyscall` implementations. This is passed to `g++` in my Makefile, using `-D platform_$(PLATFORM)` which is based on the environment variable `PLATFORM`. When both `platform_unikraft` and `platform_linux` are undefined, I provide an empty stub implementation. Otherwise, I call the platform specific macro.

4.2.1 Making a Unikraft image

I decided to target Unikraft as my first unikernel to be fuzzed. This is a fairly mature Unikernel, and through my supervisor I was in contact with some of the developers behind it.

To compile my fuzzer in to a Unikraft image, I used the kraft command line tool. This reads a `kraft.yaml` file specifying the entry point of the program, modules to enable, and other libraries which should be included. This, along with the files `Makefile` and `Makefile.uk`, allow for an image to be built for any of the supported architectures. My `kraft.yaml` was fairly simple, with a few libraries added for C++ support.

```
specification: '0.4'

unikraft:
  version: 'staging'
  kconfig:
    - CONFIG_LIBSYSCALL_SHIM=y
    - CONFIG_LIBPOSIX_SYSINFO=y
    - CONFIG_POSIX_USER=y
    - CONFIG_LIBNEWLIBC=y
    - CONFIG_LIBPOSIX_PROCESS=y
    - CONFIG_LIBUKMMAP=y
    - CONFIG_LIBUKTIME=y
    - CONFIG_LIBVFSCORE=y

architectures:
  x86_64: true
  arm64: true

platforms:
  linuxu: true
  kvm: true
  xen: true

libraries:
  libunwind: 'staging'
  compiler-rt: 'staging'
  libcxx: 'staging'
  libcxxabi: 'staging'
  newlib: 'staging'
```

I also enabled a number of modules such as `UKMMAP` and `POSIX_PROCESS`. These register additional syscalls in the syscall shim layer and allow more of

the Unikraft codebase to to be fuzzed. Despite this, some system calls were still unsupported and this meant the fuzzer was making calls that immediately returned `ENOSYS`, a value indicating that no matching syscall was found. To work around this, I updated my fuzzer to call `uk_syscall_name_p` for each call on startup. This usually returns the name of a system call, but returns `NULL` if no matching call is found. This allowed me to automatically disable those calls.

The last issue I tackled before successfully running the fuzzer on Unikraft was that the linker failed to find the `uk_syscall` symbol. I explored this with Dr. Razvan Deaconescu, an Assistant Professor at University Politehnica of Bucharest. We discovered using `nm` that this was due to some function name mangling that C++ performed. Surrounding the include statement in an `extern "C" {` block was sufficient to fix this.

4.3 Code Coverage

The next problem I needed to solve was code coverage. As mentioned previously, traditional operating system fuzzers use kernel features such as kcov which were not available. Consequently, I experimented with a few different options.

4.3.1 Performance counters

An early idea I had was that the information provided by performance counters may correlate with code coverage. These are stored in special CPU registers and contain values such as the number of instructions executed or the number of CPU cycles which have taken place. A concern was that constructs such as loops may break the relationship, but due to the relative ease of accessing performance counters across kernels, this was an idea that I wanted to spend some time investigating.

To understand if there was a correlation between code coverage and performance counter values, I wanted to obtain both from the same code executed with the same inputs. To find code to use, I looked in to a number of benchmarking suites, eventually settling on the SNU NPB 2019 Suite [2], which is an implementation of the NAS Parallel Benchmarks (NPB). These are usually used for benchmarking the performance increase resulting from parallelisation, but I ran just the serial implementation so I could avoid the code running on multiple CPU cores.

After choosing a benchmarking library, I needed to be able to access performance counters during the benchmarks. I did this by implementing a small library perf.c, which uses the perf_event_open syscall to record values from the CPU. For my initial experiments, I observed a single performance counter, `PERF_COUNT_HW_BRANCH_INSTRUCTIONS`.

Before running the experiments, I set the isolcpus boot parameter in my kernel, and updated the perf_event_open syscall to restrict measurements to the selected core. This guaranteed that other processes would not interfere with the measurements I got.

I adapted two of the benchmarks in the benchmarking library to add performance counter instrumentation, BT and CG. In each case, the benchmark performs a number of iterations, and I calculated the average number of branches across all of them. I also ran the same experiment removing the performance

counter instrumentation, compiling with gcov and outputting coverage information to a file. This gave me two data points showing the number of branch instructions taken and the number of basic blocks executed. The results are shown below.

Benchmark	**Branches (perf)**	**Blocks Executed (gcov)**
BT	444819	330
CG	13943298	50

This initial experiment was not promising, but I realised that without a much larger corpus of benchmarks, I would not have enough data to produce a graph showing how the two variables were correlated. I then began looking for other projects that may provide a large corpus of code to be tested. A possible option was the test suite of an open source project, as each test is likely to cover a small but different part of the code, hopefully exercising different code constructs in frequencies similar to what would be found in most codebases. However, due to the complexity of updating a test harness to add instrumenation, I chose against this.

As well as developing syzkaller, Google also run the FuzzBench [6] project, which regularly tests a variety of fuzzers on a range of applications. Usefully, as part of these test runs, the corpus of interesting inputs generated by fuzzers is archived. A small wrapper program is also provided around the application being tested, providing a single entry point that runs the application with a given input. After some research, I realised I would be able to run the entire corpus, with either gcov or performance counter based instrumentation added to the wrapper application.

For this set of experiments, I chose to use the libxml benchmark, which is based on libxml v2.9.2. The archived corpus I downloaded contained 7,705 files, each containing a small input which generated new coverage during fuzzing with AFL. I recorded the value of six performance counters, and then repeated the runs recording the percentage of code blocks covered.

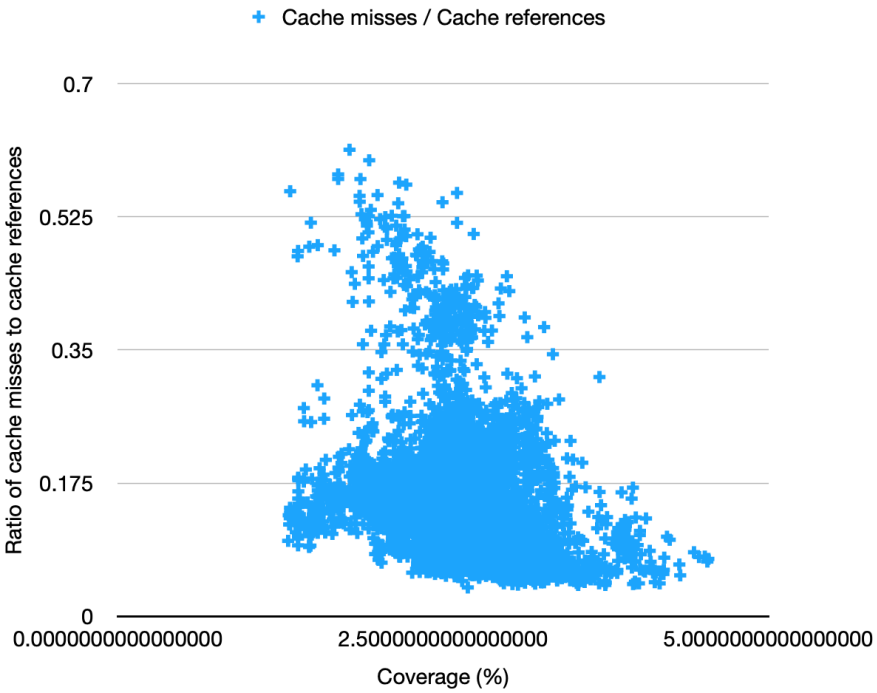


Figure 3: Graph showing the cache hit rate against code coverage

Both of these runs were performed using a small bash script I wrote. This executed each test sequentially, outputting the instrumentation data to a CSV file. In the case of gcov, I used the llvm-profdata and llvm-cov commands to read coverage information as JSON, piped to `jq` to extract just the percentage coverage.

With this much larger pool of data, I was ultimately able to rule out the use of performance counters.

4.3.2 Breakpoints

In Nagy et al. [16], the idea of Coverage-Guided tracing is introduced. Instead of running coverage tracing throughout the entire fuzzing process, so it can be known when new coverage is obtained, coverage information is only obtained once it is already known that an input generates new coverage. To achieve this, an "interest oracle" is made. This is a version of the binary which has had breakpoints added at the start of each basic block using a tool such as Dyninst. Consequently, when the breakpoint is hit, this is an indication that new code has been reached. Breakpoints are removed when they are hit to make sure that they only run the first time each basic block is executed.

To add a breakpoint to a program, the first byte of the instruction which the breakpoint should be set on is replaced with the `INT3` assembly instruction. INT generates a software interrupt, and `INT3` in particular has some properties

which make it desirable for debugging. Firstly, it only takes up a single byte, meaning instructions of any length can be overwritten. The byte which has been overwritten can be stored and this can then be easily put back once the breakpoint is hit. Second, it bypasses permission level checks which usually happen in userspace and would prevent the interrupt from being called.

This approach was extremely promising for the fuzzing of unikernels, because breakpoints can be added using a hypervisor like QEMU without any coverage features needing to be implemented in the kernel. Additionally, there were possible performance benefits. This was the focus of the Nagy et al. paper, which found that over 90% of the time spent fuzzing is related to coverage instrumentation even though less than 1 in 10,000 test cases are coverage increasing.

To take a similar approach for unikernels, I first needed to identify where to place breakpoints within the kernel. For my initial prototype, I chose to use the Dyninst project's SymtabAPI, which reads a binary's symbol table and provides an interface through which the offset of both functions and variables can be found. Using this, I was able to build a small C library which populates a struct with the absolute address of each function within the source of a kernel built with debugging symbols. Currently, this happens at the function level. However, for more precision, I believe it would be fairly easy to integrate with a library that provides the location of basic blocks. This additional precision may impact performance, so it would be necessary to perform some experiments to see if this extra code coverage was worth the overhead.

4.3.3 QEMU

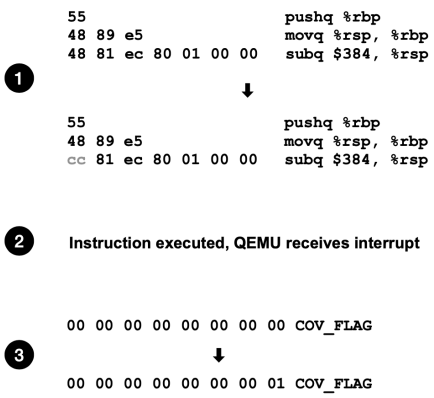


Figure 4: Process of adding an INT3 instruction and capturing the interrupt

To actually instrument the program, the Unfuzzer tool built in the Nagy et al. paper has a parent process which performs the modification process. Then, one or more child forkservers are started, which execute the program. Interrupts are caught by the parent process, which collects coverage information, unmodifies the binary, and restarts the forkservers. In my case, I wanted to keep the code running inside of the unikernel as simple as possible, and I did not have

access to fork. Consequently, I chose for the parent process to be QEMU itself.

QEMU supports GDB through a remote debugging server referred to as "gdbstub". Fortunately, I was able to reuse many of the functions here for inserting breakpoints and preserving the values which had been overwritten. For example, when I first run the program, I iterate over each of the addresses provided by my C library and insert a software breakpoint. This is done by calling `kvm_insert_breakpoint`.

```
static void instrument(CPUState *cpu) {
    *functions = getAllFunctions(get_kernel_filename());

    printf("Found %lu functions, instrumenting...\n",
           functions->count);

    for (unsigned long i = 0; i < functions->count; i++) {
        // Add a breakpoint at the start of this function
        kvm_insert_breakpoint(cpu, functions->offsets[i], 0,
                              GDB_BREAKPOINT_SW);
    }

    printf("Kernel instrumented with %lu breakpoints!\n",
           functions->count);
}
```

I initially called this as soon as the virtual machine started, within the internal function `kvm_vcpu_thread_fn`. However, I found that my breakpoints were not hit, which I think may be because they get overwritten as the kernel is loaded in to memory by the bootloader. To address this, I made use of a hardware breakpoint, and inserted the software breakpoints needed for fuzzing only when this breakpoint was hit. I chose the start of `print_banner`, but any kernel function called sufficiently late in the boot process should work. Finally, I made a small change to `kvm_handle_debug`, to make sure it does not skip over hardware breakpoints which it doesn't recognise.

I then needed to find a way to communicate coverage information back to the program running inside of the unikernel. To do this, I decided that the simplest approach would be a global variable, defined within the C code. In QEMU, I use Dyninst to read the location of this from the symbol table, and I then set the "flag" variable to 1 if new code has been covered. The fuzzer can reset this back to 0 before fuzzing with new input. One limitation of this approach is that it is restricted to a single thread. To fuzz in parallel, more than one virtual machine can be started. Fuzzing with multiple threads in the same virtual machine would need multiple flag variables, but the effectiveness of this may still be limited as it allows syscalls being made on the different threads to interfere with one another.

4.3.4 Porting coverage measurements to Linux

While developing the fuzzer, a problem I ran in to was that it was hard to test the effectiveness on only less mature OSes. While unikernels were ultimately the focus of my project, I realised that there were several benefits to being able to test the fuzzer on Linux. Firstly, this would allow me to see how it performs on a much larger number of syscalls. Secondly, it would provide me with the

opportunity to directly compare the performance of my code coverage strategy against existing approaches like kcov. Finally, it would allow me to demonstrate that the fuzzer I had made was generic, being applicable to OSes other than the ones I had in mind when developing it.

The first part of doing this was to find a way of running my fuzzer source code on Linux. A very different approach was needed to the one taken for unikernels, as while unikernels are built to boot in to a particular program, the Linux operating system is a multitasking operating system which usually loads programs dynamically. This presented a problem, as even if I could load a program on boot, I would not know where in memory it would reside.

To solve the problem of running my fuzzer at boot, I used the initramfs scheme. This is a feature in Linux which allows a compressed archive created using cpio to be uncompressed at boot, at which point the binary at /init will be executed. Usually, this is a simple image which performs tasks like disk decryption and the loading of drivers before mounting the real file system. However, I was able to create an image containing my fuzzing binary, and pass this as an initrd argument to QEMU. This fairly simple solution helps to minimise the steps required to build the fuzzer.

The other problem to solve was the loading of symbols for the purpose of adding breakpoints. When running a unikernel, the kernel I pass to QEMU is a normal Linux binary and Dyninst is able to extract symbols from that directly. However, when running Linux, I initially wanted to avoid rebuilding the Linux kernel. I achieved this by using the open source vmlinux-to-elf [15] tool, which reconstructs debug symbols using a compressed symbol table available in kernels called "kallsyms". I then updated my QEMU build so an optional environment variable could be set to override the location that symbols are loaded from.

By default, the Linux kernel has a feature called Kernel Address Space Layout Randomisation, or KASLR. This randomises the base address where the kernel is loaded, adding a constant but unpredictable offset to the base address where the image is loaded to. This is a useful security protection, as it prevents attacks where malicious processes jump to known locations in memory, for example by exploiting a buffer overflow to overwrite memory at a location which is known to be significant. However, this makes instrumentation with breakpoints harder. Consequently, I pass the the `nokaslr` argument to the kernel which disables this behaviour.

Finally, I needed to be able to share coverage information between the QEMU hypervisor and the init process running the fuzzer. Since I wasn't confident that we could rely on where the init process was loaded, I worked alongside another student who was following the project to implement a new system call, which would allow a program in userspace to access a global variable stored within the kernel. Jules Irenge provided me with a small patch to the Linux kernel that adds a new global variable to the file `kernel/sys.c`, where syscall handlers are implemented. The patch also updated the file `unistd.h` to add constants for the new system call number, and both `syscall_64.tbl` and `syscall_32.tbl` to actually register these handlers in the lookup table. I took this patch and modified it slightly. Firstly, I switched from storing a struct to a single integer, as I only needed a single flag indicating if new coverage had been obtained. Second, I updated the call handler to clear the flag when it is read, so it is only set on future calls if coverage has been obtained since we last checked. I built the kernel after applying this patch and was able to use this in

place of the official kernel build I was using when running QEMU.

4.3.5 Porting to HermiTux

During the project my fuzzer was also ported to HermiTux, a unikernel which can run native Linux binaries. This was done by supervisor, Dr Pierre Olivier. Since the unikernel is binary-compatible, this was reported as being a fairly straightforward process with the fuzzer simply being built for Linux. Some small changes were made including fixing the implementation of `clock_gettime` so the random number generator used by the fuzzer could be seeded.

5 Evaluation

5.1 Effectiveness of fuzzing

5.1.1 Bugs Identified

A number of bugs were identified by my fuzzer, which can be broken down in to two categories. General kernel issues refers to bugs found while porting the fuzzer. Implementation bugs in syscall handlers refers to bugs which caused a crash while the fuzzer was running signalling an issue in the implementation of the handling code for that particular system call. The full list of issues can be found in the Appendix.

Category	**Bugs Identified**
General kernel issues	2
Implementation bugs in syscall handlers	5

5.1.2 Performance Overhead

Since fuzzing relies on executing a program many times, reducing the performance overhead of instrumentation is a key part of increasing a fuzzer's efficiency. Towards the end of my project, I wanted to see what the overhead of inserting breakpoints was. The UnTracer fuzzer which this technique was inspired by claims below 1% average overhead, so I was hoping to see only a small change in program execution time.

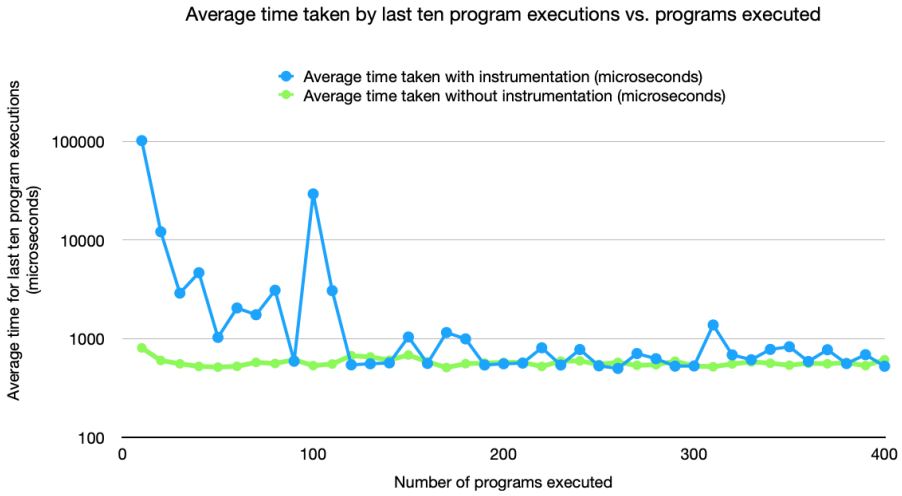


Figure 5: Graph showing the performance overhead of breakpoint based instrumentation, on a logarithmic scale

As shown in Figure 5, I ran my fuzzer on the Linux kernel twice, recording how the average time for the last ten program executions changed. Initially, I did this without breakpoints to get a control for the average execution time of a program. As expected, this remains constant, taking just over 500 microseconds for each 20 syscall program. I then added breakpoints at the start of each function using the features in my modified version of QEMU. Programs initially took much longer to run, but after around 100 programs had been executed, the performance of the two trials is similar, with only rare, small spikes in the time taken by the instrumented kernel. This gives me confidence that as breakpoints are removed, the overhead decreases, making the overall overhead of this approach fairly low.

5.2 Planning

Figure 6 is a Gantt chart that I made at the start of Semester 1, showing when I intended to work on each part of the project. The initial research followed the schedule quite closely, and I did a presentation on what I had learned about syzkaller on the 22nd October. I had then planned to research gcov for the last week of October so I could begin implementation in November. This didn't quite go to plan, however, as I spent longer researching the various parts of the project until making my first commit on the 15th November.

Implementation followed the plan quite closely, but delayed by a few weeks due to the extra time spent on research. For example, I planned to finish the Syzlang compiler in November, but committed this at the start of December instead. I would have benefitted from revising this plan as the scope of the project changed. For example, I planned to spend about two months working

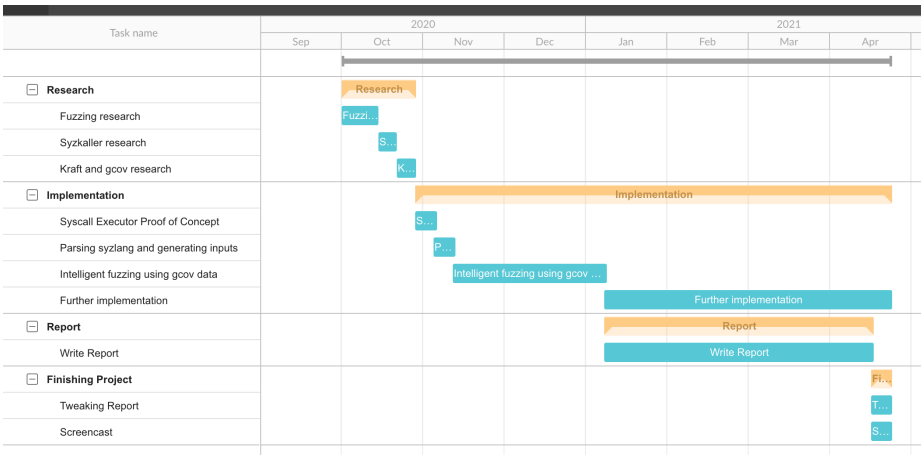


Figure 6: Gantt chart made at the start of the project

to use gcov data in the fuzzer. However, this development time instead went on investigating other ways of generating code coverage information.

Work on my report started later than planned. While I had written a small amount in March, I switched focus to this at the start of April. Again, it would have been beneficial to revise my plan to reflect these adjusted deadlines.

5.3 Project goals

At the start of my project, I set out three key goals that I wanted to accomplish over the year.

Firstly, I wanted to study fuzzing concepts and existing tools. This would be useful as it was not an area I had worked in before, and I wanted to understand the existing research before doing my own. My first part of the success criteria for this goal was to be able to demonstrate where I have found approaches in existing work that I could adapt. I believe that there are several places where I can demonstrate this. For example, I compile a specification of available syscalls from syzkaller's Syzlang, and many of the implementation decisions for the fuzzer were based on the main fuzzing loop used by existing fuzzers like AFL. I also wanted to be able to identify any gaps in existing research that I have taken steps towards addressing. I believe my major contribution here is showing that breakpoints can be used as a viable alternative to traditional code coverage techniques when fuzzing kernels that do not have features like kcov available. Adding breakpoints for code instrumentation is not a new technique in itself, but I have shown that it can be applied in a new context. I have also built a number of components which are less complex than existing fuzzing tools and allow a greater range of kernels to be fuzzed.

The second objective for this project was to select an appropriate tool, define a fuzzing strategy, and select a target unikernel. I believe I met this by choosing to build a new fuzzer and giving justification for taking this approach. Additionally, I chose Unikraft as a unikernel which was more mature than the alternatives, and which had a syscall shim layer that I could use as the entry

point for fuzzing. I used Whitebox fuzzing as looking at the existing literature convinced me that this would be the most effective form of fuzzing.

Finally, I wanted to be able to show evidence of applying the fuzzing strategy and either reporting or fixing any potentially uncovered bugs. I found two bugs in Unikraft, and my fuzzer enabled the identification of bugs in HermiTux too. These are listed in the Appendix.

5.4 Future work

While overall I am proud of the work I achieved during the project, there remain experiments to be performed and parts of the implementation to be completed. This is partly due to the fact that it was a research project exploring the application of techniques in new ways, and in some cases building a proof of concept was prioritised over a complete implementation of any given component.

5.4.1 Type generation

During the implementation of both my compiler and fuzzer, I prioritised support for the types that appeared most frequently in system call arguments, such as integers, flags and filenames. However, syscalls take a much greater range of arguments than this, such as pointers to larger structs. Based on the types I currently have support for, the fuzzer is able to generate inputs for 179 of the 305 syscalls documented by Syzkaller.

Improving support here would allow a greater number of system calls to be fuzzed, which has the potential to significantly increase the amount of kernel code covered. Additionally, there is scope to generate a wider range of values for given arguments. For example, currently only valid flags are generated. While this should ordinarily be the case, to avoid generating inputs which are immediately discarded by the kernel, it would be useful to infrequently generate other values to check they are correctly handled. File descriptors are another example of this. Before I had correctly implemented resources, I was incorrectly passing a syscall index as an argument to subsequent syscalls instead of the syscall's return value. These values would almost certainly have been invalid, but this lead to the identification of a bug in HermiTux where arbitrary file descriptors passed to `close` would cause internal descriptors to be closed. This sort of bug can only be identified if the fuzzer intentionally generates input which violates the values it knows are allowed.

5.4.2 Enabling features like UBSan and ASan

Compilers such as GCC and Clang have support for features such as the undefined behaviour sanitizer (UBSan) and address sanitizer (ASan). These can be enabled at compile time and add additional checks within the assembly code to detect bugs that can usually go undetected. For example, the undefined behaviour sanitizer in Clang can detect when an an array whose bounds can be statically determined has had an out of bounds access. The address sanitizer can detect similar bugs such as the use of memory after it has been freed. Currently I haven't tried using these with the fuzzer but they may allow additional bugs to be caught.

5.4.3 Support for other kernels

During the project a few changes were identified that would be needed for some kernels.

Firstly, the fuzzer currently relies on the `syscall` macro, which Unikraft has an equivalent for. However, other unikernels like OSv provide Linux compatibility through the libc interface, relying on functions such as `read`. To be able to support these OSes, the fuzzer would need to be able to call any one of these functions without knowing the particular sequence of calls at runtime. This should be possible but would likely involve directly manipulating registers or a large switch statement over all possible function calls.

Second, the breakpoint based instrumentation I implemented was done by modifying QEMU, a hypervisor used by many unikernels. However, this is not the case for HermiTux, so all fuzzing performed on this kernel was with entirely random system calls. Providing coverage information when fuzzing HermiTux would likely require modification of the custom hypervisor it uses. This should be feasible, as the HermiTux hypervisor follows a similar architecture to QEMU and the modifications I made to QEMU were fairly simple. However, it would require additional work.

5.4.4 Persisting fuzzing state after a crash

In a successful run of the fuzzer, the kernel would eventually be expected to crash after running in to undefined behaviour caused by an implementation bug. Currently, all state is lost at this point, including the programs in the corpus and the instructions which have not yet been reached and should still be instrumented. The fuzzer does print the system calls it is executing to stdout, allowing manual intervention in an attempt to reproduce the crash and find the root cause. However, in the future, it would be useful to persist this state to the host to avoid losing it entirely. There are several ways in which this could be implemented. In syzkaller, the syz-fuzzer process communicates with syz-manager over a network socket, and similar could be done using `virtio`. This is a virtual network interface supported by some unikernels such as Unikraft as well as hypervisors like QEMU. Alternatively, the data could be written to a shared filesystem. An example of this would be using the 9pfs support in Unikraft to share a network file system with the application running with KVM.

Once the syscall history could be persisted to the host, it would be possible to explore creating reproducers for crashes. This is a process implemented by other fuzzers where the program is slowly reduced in size to find the smallest number of system calls that reliably reproduce the crash. Manual work is still required to identify the bug but this significantly reduces the amount of work required.

5.4.5 Fine-grained disabling of system calls

As mentioned in Section 3.3.3, unikernels are currently an area of active research and are usually less mature. This means that it is much more likely to encounter crashes than when fuzzing a traditional operating system like Linux. While the `disabled` attribute in the exported Syzlang data can be used to disable an entire system call, this means an entire part of the kernel may no longer be able to be fuzzed. A better approach would be if more specific conditions could be

specified, such as avoiding the use of a particular argument range or preventing two system calls from being called one after the other. This would allow frequent crashes to be avoided even before it is possible to land a patch fixing the issue in the upstream kernel repository.

5.4.6 Identifying common system call sequences

Finally, to seed the fuzzer, it would be interesting to record sequences of syscalls made by applications which are commonly deployed in the cloud. This should allow the fuzzer to cover more of the kernel source more quickly, meaning bugs can be identified after a shorter session of fuzzing. Additionally, this would give greater assurances than the unikernel being fuzzed is compatible with that application, which may be of interest to kernel developers.

6 Conclusion

This project was a great learning experience for me and I thoroughly enjoyed my time working on it. My supervisor, Dr Pierre Olivier, was incredibly helpful throughout. Through sharing his experience building the HermiTux unikernel, I learnt a lot about how unikernels are run in the KVM. He also shared a lot about the research process and guided me while I researched existing projects, and developed my own work. I learnt a lot over the year, going from knowing nothing about fuzzing or unikernels to understanding how they work in some level of detail. I also learnt a lot of new skills, like how debug symbols are stored in a Linux binary and how to read them. I even worked with QEMU for the first time and began to understand how QEMU communicates with the underlying hardware to start a virtual machine in KVM mode.

I was able to build a new fuzzer and demonstrate its effectiveness fuzzing the syscall interface of unikernels, which as far as I know has not been done before. This identified a number of bugs that I was able to share with project authors and highlighted a new way that these types of systems can be tested in the future.

References

[1] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 21232138, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi.org/10.1145/3133956.3134069`, `doi:10.1145/3133956.3134069`.

[2] Youngdong Do, Hyungmo Kim, Pyeongseok Oh, Daeyoung Park, and Jaejin Lee. Snu-npb 2019: Parallelizing and optimizing npb in opencl and cuda for modern gpus. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 93–105, 2019. `doi:10.1109/IISWC47752.2019.9041954`.

[3] Go. Cgo. URL: `https://golang.org/pkg/cmd/cgo/`.

[4] Patrice Godefroid. Fuzzing: Hack, art, and science. *Commun. ACM*, 63(2):7076, January 2020. URL: https://doi.org/10.1145/3363824, doi:10.1145/3363824.

[5] Google. Afl. URL: https://github.com/google/AFL.

[6] Google. Fuzzbench: Fuzzer benchmarking as a service. URL: https://github.com/google/fuzzbench.

[7] Google. syzkaller. URL: https://github.com/google/syzkaller.

[8] Google. syzkaller/setup.md - linux kernel. URL: https://github.com/google/syzkaller/blob/master/docs/linux/setup.md#linux-kernel.

[9] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020. URL: https://doi.org/10.1145/3428334, doi:10.1145/3428334.

[10] Antti Kantee and Justin Cormack. Rump kernels: No os? no problem! *login Usenix Mag.*, 39, 2014.

[11] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147161, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3341301.3359662, doi:10.1145/3341301.3359662.

[12] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[13] Simon Kuenzer, Vlad-Andrei Bdoiu, Hugo Lefeuvre, Sharan S̃anthanam, Alexander Jung, Gaulthier Gain, Cyril S̃oldani, Costin Lupu, tefan Teodorescu, Costi R̃ducanu, Cristian Banu, Laurent Mathy, Rzvan D̃eaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. EuroSys'21, New York, NY, USA, 2021. ACM. doi:10.1145/3447786.3456248.

[14] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGARCH Comput. Archit. News*, 41(1):461472, March 2013. URL: https://doi.org/10.1145/2490301.2451167, doi:10.1145/2490301.2451167.

[15] Marin. vmlinux-to-elf. URL: https://github.com/marin-m/vmlinux-to-elf.

[16] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802, 2019. doi:10.1109/SP.2019.00069.

[17] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 5973, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3313808.3313817, doi:10.1145/3313808.3313817.

[18] OSv. Osv linux abi compatibility. URL: https://github.com/cloudius-systems/osv/wiki/OSv-Linux-ABI-Compatibility.

[19] Unikraft. Application development and porting - syscall shim layer. URL: http://docs.unikraft.org/developers-app.html#syscall-shim-layer.

[20] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834, 2019. doi:10.1109/SP.2019.00035.

7 Appendix 1: Bugs identified during the project

7.1 Unikraft

These issues were discovered by both me and Dr. Razvan Deaconescu while building the fuzzer for Unikraft.

7.1.1 Page fault when executing `uk_syscall(0, 0)`

This is the first issue I opened after realising that passing 0 the `open` syscall caused a crash.
Issue URL: https://github.com/unikraft/unikraft/issues/89

7.1.2 Defining a large global vector of strings results in page fault

When populating a large, statically initialised vector in C++, Unikraft crashes during boot.
Issue URL: https://github.com/unikraft/lib-libcxx/issues/4

7.2 HermiTux

These issues were discovered by supervisor Dr. Pierre Olivier while porting the fuzzer to HermiTux.

7.2.1 Bad parameter sanitization for prlimit64

The prlimit system call is used for getting and setting resource limits. The old_limit parameter, where the old limit value is stored, was being dereferenced before checking to see if it was `NULL`.
Issue URL: https://github.com/ssrg-vt/hermitux/issues/19

7.2.2 clock_gettime does not return the time offset from the epoch

The call `clock_gettime` was always populating the result struct with the value zero.
Issue URL: https://github.com/ssrg-vt/hermitux/issues/20

7.2.3 bad error code returned from sys_creat

The call `sys_creat` was returning a value based on what is usually returned by the libc wrapper, instead of the value returned when calling the syscall on Linux.
Issue URL: https://github.com/ssrg-vt/hermitux/issues/21

7.2.4 Sanitize close() parameter

File descriptors passed to the `close` call were not validated, allowing internal file descriptors used as part of the KVM VCPU to be closed.
Issue URL: https://github.com/ssrg-vt/hermitux/issues/22

7.2.5 Page fault in Glibc's malloc

This fault has not yet been debugged but causes a crash within glibc.
Issue URL: https://github.com/ssrg-vt/hermitux/issues/23