École Polytechnique Fédérale de Lausanne

Parallel Experiments Lène

by Solène Husseini

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Prof. Cono D'Elia Daniele
External Expert

Luca Di Bartolomeo
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

July 12, 2024

No computers were harmed during the course of our experiments.

Acknowledgments

I want to thank my advisor, Prof. Mathias Payer, for his supervision, guidance and feedback. I enjoyed the time I spent working with his group in HexHive, the people there gave me the support I needed to pursure this project.

I also want to thank my family, my friends, and my girlfriend Nicole, who helped me a lot to work through this project.

Lausanne, July 12, 2024 Solène Husseini

Abstract

Out of all the techniques that can be used to find bugs, fuzzing has earned its reputation as one of the more reliable. However, it requires considerable resources to be performed effectively. In particular, we explore the use of GPUs to run the fuzzers, since their hardware capabilities seem like they could match our problem: running many times the same program with different inputs, a classically parallelizable problem. Taking inspiration from Trail Of Bits [1]'s preliminary efforts, in which they use recompilation to let programs run on the GPU, we propose to instead write an emulator. Our approach aims to palliate some of the more fundamental issues we could identify in the previous work, while providing a more extensible and introspectable platform for fuzzing on GPUs. Our preliminary results show that our approach has comparable performance to a CPU fuzzer in some emulated workloads. We then identify a number of potential optimizations that show promising performance improvements.

Contents

Chapter 1

Introduction

Fuzzing has proven to be a very effective strategy to find bugs in open-source software. When combined with compiler-inserted instrumentation and sanitizers, it reliably uncovers new issues. Compared to static analysis, it also has the advantage of generating proofs-of-concepts, simplifying reproducibility and bug-fixing. However, this comes at the cost of running the program over and over, which necessitates a lot of infrastructure, power and time to work at scale.

A particularly interesting aspect of fuzzing is that the target program is always the same, and often with very similar inputs from run to run, making the whole process very amendable to parallelization. We hypothesize that the parallel-first nature of the architecture of graphics processing units (GPUs) could contribute to fuzzing efforts, through performance, metrics collection, and running programs in different ways. One obstacle to this effort is that conventional CPU programs have not been designed around the architecture of GPUs, hence ignoring their limitations and their advantages. As a clarification, this is not about fuzzing GPU programs, but about using GPUs to fuzz CPU programs.

GPUs' execution model is different in multiple ways. First, they are optimized for data-parallel workloads, that is running the same program but on different data. The control-flow of the programs can still differ between threads, but this divergence can heavily penalize performance if not taken into account. On top of this, their memory hierarchy is optimized for graphical tasks, hence it differs from CPUs. For instance, most of their caches are write-through, threads do not have a notion of a program stack, and they have access to *shared memory*, a low-latency but limited-capacity random-access memory. In light of these discrepancies, the execution strategy used to run CPU programs on GPUs is of particular interest.

The first approach taken by a related work by Trail Of Bits [1] is to lift programs from assembly to *LLVM IR*, and then compile them back to *PTX*, Nvidia GPU's assembly language. Recompilation theoretically preserves the performance of the original program which is very desirable for fuzzing. Furthermore, their use of lifting allows them to recompile any binary without requiring its source

code. However, direct recompilation does not give much room to relieve the disadvantages of running a CPU program on a GPU. For instance, they do not make use of shared memory, and they cannot modify the control-flow of the program, which will almost certainly lead to performance loss from divergence. Their evaluation circumvents these issues by running on a 100 lines function with few memory accesses and no syscalls which is a good fit for recompilation, even if it would not scale well to the size of a real program.

Our answer to the multiple issues encountered when running conventional programs on the GPU is to implement a CPU emulator, which allows us to harness the parallelism of GPUs while keeping control of the code that will run on it. This trades off the overhead of running programs in an emulator with the ability to take advantage of shared memory, mitigate control-flow divergence, and run conventional CPU binaries. This last point is particularly relevant because, although it does not contribute to better performance, it ensures compatibility with most of the already existing toolchains and allows us to reproduce crashes on real-hardware and other emulators like QEMU, and lets us debug them using conventional tools like GDB.

We make the case that this approach is more promising because it also simplifies the incorporation of other features. For instance, instrumenting specific instructions (branches or memory operations) can be done directly in an emulator, and does not require altering the target program. This flexibility also lets us implement *fork* as a single memcpy, which is very inexpensive. We can hence reproduce the functionality of a *fork server* on our GPU emulator with minimal effort, greatly speeding up the fuzzing process.

Overall, our main contribution is a fuzzer that runs on the GPU while taking much more advantage of its hardware features and mitigating the inefficiencies of the previous approach. Our fuzzer is designed to accept programs used in production today, and shows promising performance.

Chapter 2

Background

2.1 GPU architecture and terminology

Our work, while being at the intersection of both fuzzers and GPUs, is most innovative in the way it enables a different way of running programs. Hence, we will start by describing the high-level architecture of GPUs.

In order to keep a consistent nomenclature throughout, we will employ the terminology of the graphics API we choose to use, *Vulkan Compute*. As shown in Figure 2.1, a program is run by performing a *dispatch*. When issuing a dispatch, not only is a *kernel* passed (i.e. a GPU program), but also the dimensions and the number of *workgroups* that will run it. In order to use the maximum amount of resources available, we choose in our case to use 240 workgroups of 128 *invocations*, giving 30720 invocations in total. An invocation can be conceived of as a GPU thread: it has its own register file and follows its own path through the kernel, although with some limitations, as we will discuss in the next section. One important property of workgroups is their access to *shared memory*. It is a small but low-latency random-access memory shared between all invocations in the same workgroup. It can both be used to communicate data between invocations within the workgroup, or as a cache for often needed data. On our machine, it spans 48KiB per workgroup and is used as a performance optimization. There are many more subtleties to take into consideration when designing a program, but this high-level overview should be enough for our purposes.

On top of this terminology, we also introduce the notion of *emulated thread*, or *ethread* for short. This refers to a thread of execution run by the emulator, by opposition to an invocation, which corresponds to an actual GPU thread. An ethread is not necessarily confined to single invocation: by storing its relevant data in global memory, ethreads can be metaphorically "moved" between carrier invocations.

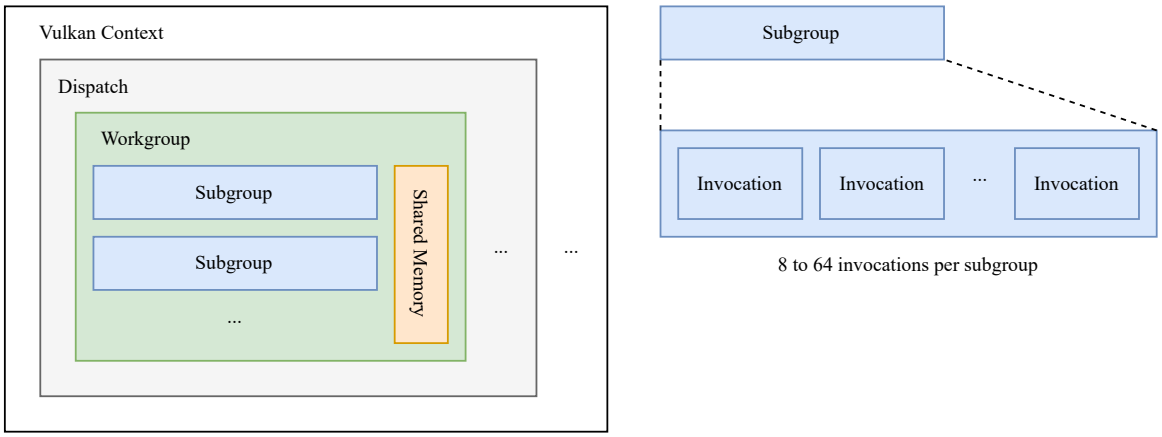

Figure 2.1: GPU architecture overview

2.2 Divergence

One of the most critical performance aspects of writing GPU program is the management of divergence. Most modern GPUs use a *Single Instruction Multiple Threads*, or *SIMT* architecture. To give a rough mental model as an approximation, it means that invocations are grouped in *subgroups* of 8 to 64, depending on the manufacturer (ours uses subgroups of 32). Each invocation does not have a distinct program counter and decoding logic: instead, it is shared by all invocations within a subgroup. This means that, in spite of having each their own compute units (to some extent) and register files, invocations must go through the same control-flow.

GPU programs contain conditional branches nonetheless. They are executed using different mechanisms depending on the brand of GPU, such as setting or clearing an *enable* flag for each invocation, or having dedicated branch and loop instructions. Most of these mechanisms rely on transforming the control-flow into data-flow, since data can vary between invocations. The following example in Listing 2.1 demonstrates that every invocation in the same subgroup will step through every instruction that any other invocation needs to run. In the worst case, where every invocation chooses a different branch, like in Listing 2.2, performance may be divided by up to the subgroup size (32 in our case). For instance, this may happen because of a large switch statement, or multiple nested branches. Every nested *if* block in the control-flow graph exponentially contributes to the divergence problem, i.e. n levels of *if* blocks will generate 2^n distinct control-flow paths in the worst case.

Listing 2.1: Divergence Demonstration

```
// Only one invocation in the subgroup takes the branch,
// yet every other invocation in the subgroup must still step through foo,
// although without actually executing the instructions.
if (condition_only_true_for_one_invocation()) {
        foo();
```

```
} else {
        bar();
}
```

Listing 2.2: Worst case divergence demonstration

```
// Performance divided by 32 (worst case scenario)!
switch (gl_SubgroupInvocationID) { // <-- divergence
        case 0: foo0(); break;
        case 1: foo1(); break;
        ...
        case 31: foo31(); break;
} // <-- reconvergence
```

In typical GPU programs, only two kinds of branches are useful to distinguish, the ones that never diverge, and the ones that can diverge. For the second kind, the likelihood that a single invocation takes the branch does not need to be taken into consideration. Indeed, even with a very low probability to take the branch, once we consider a full subgroup, it is almost certain that the some invocations will diverge. For instance, given 32 invocations per subgroup, a branch with 90% chance of being taken ($P(branch) = 0.9$) will diverge $1 - (0.9^{32} + 0.1^{32}) = 96\%$ of the time. We can expand the computation like so, taking inspiration from a very through blog post discussing divergence in depth [4]:

$$
\begin{aligned}
P(\text{divergence}) &= 1 - P(\text{no divergence}) \\
&= 1 - (P(\text{all invocations take the branch}) + P(\text{no invocation takes the branch})) \\
&= 1 - (P(\text{branch})^{32} + (1 - P(\text{branch}))^{32}) \\
&= 1 - (0.9^{32} + 0.1^{32}) \\
&\approx 0.96
\end{aligned}
$$

Loops are also a form of branching, hence they can cause the same issue. Specifically, any invocation which leaves a loop will have to wait for the others in the subgroup to also leave it to make progress. Effectively, the time to execute any given loop depends on the maximum number of iterations that any invocation takes in the subgroup, compounding on the performance loss.

CPU programs do not take divergence into consideration, hence compiling them to run on the GPU directly will very likely result in high divergence and severely under-utilize the available resources. We contrast the potential impact of divergence between recompilation and emulation in section 3.3.

2.3 State of the art

Before introducing our own design, it is worth looking at the most prominent attempt to fuzz on GPUs that we could find online. This is a summary of the work of Trail of Bits [1], and it is of particular interest because the shortcomings of their approach motivate our own. Their strategy allows them to fuzz arbitrary binaries, and works as follows:

1. Lift the ELF program from assembly to LLVM IR using Remill [2], one of the tools they wrote to lift binaries.
2. Implement some special intrinsics in the LLVM IR to remap memory reads and writes to work on the multithreaded environment of the GPU.
3. Compile the modified IR to CUDA, Nvidia's GPU framework.

Their strategy to perform recompilation aims to ensure good performance, while also not requiring the source-code of their fuzzing targets. This limits the use of off-the-shelf sanitizers, and would require them to reimplement common ones like *ASan*. They also implement a some optimizations that dramatically increase the performance of their fuzzer:

1. Stripe memory on a 4 byte basis, by groups of $32 * 4 = 128 KiB$, to optimize the memory latency in case multiple invocations in the same subgroup perform a memory load at the same address in their emulated address spaces. Indeed, in that case, the memory accesses will land in the same cache line, hence only requiring a single memory transfer instead of 32 (the subgroup size in our cases).
2. Use a copy-on-write strategy to reuse as much memory as possible. This is quite important as on-device memory is scarce on GPUs.
3. Asynchronously transfer crashing inputs to the CPU to minimize downtime.

While these optimizations greatly improve the performance of their fuzzer, we do not consider them essential in our own. Indeed, these techniques are orthogonal to the difference in approach between recompilation and emulation. There are no hard constraints that would prevent their implementation, hence we consider them to be future work, and discuss them in more detail in chapter 6.

Their work is not directly comparable to ours because their fuzzer is designed to work on arbitrary binaries, whereas our own requires the source code to be efficient. We take inspiration from their work to implement our own way of fuzzing CPU programs on a GPU.

Chapter 3

Design

Before describing the inner workings of our system, we first motivate the use of GPUs as a viable alternative to CPUs in the context of fuzzing, alongside the trade-offs that this platform presents. Then, we explain how our novel approach to GPU fuzzing through emulation overcomes some of the challenges faced by recompilation, while providing a system that is easier to integrate with existing programs used in production.

3.1 Fuzzing on GPUs

Fuzzers' mode of operation consists in running a single program over and over, with varying inputs. Each of these executions is independent, since the programs do not communicate with each other. Hence, fuzzing lends itself very well to parallelization, and most existing fuzzers are designed to exploit it. Given that GPUs are machines designed to run programs in parallel, they seem to be a natural fit for such a task.

However, most programs are designed to run on CPUs. If we want to use a GPU to fuzz a CPU program, there are a few key challenges that must be kept in mind:

1. **Memory.** Consumer GPUs come with dedicated on-device memory to improve performance. This memory is soldered-on cannot be expanded like on a typical server, and hence may limit the total amount of ethreads we run at once. If the memory is too small to fit enough ethreads to fully saturate the compute side of the GPU, a lot of performance will be wasted.

   Furthermore, GPU memory is designed for throughput first, not latency. This makes sense in the context of computer graphics, but may hinder the performance of CPU programs, which expect low-latency random-access to memory.

2. **Syscalls.** Running programs on the GPU necessarily requires a layer of translation to let the fuzzed program interact with its environment. Multiple approaches can be taken: we can forbid syscalls entirely, like in Trail of Bits's experiment, we can translate syscalls to perform them on a CPU-side helper process in a similar way to user-mode *QEMU*, or we can emulate syscalls on the GPU directly. Each of these approaches has drawbacks, either in the capabilities offered to the fuzzed process, or in the performance and resource usage of translating syscalls.

3. **Sanitizers.** Sanitizers are essential to ensure that bugs found during fuzzing produce a crash. However, conventional sanitizers may not always work in a GPU context since the hardware environment is different, or may require patching to work properly.

These factors restrict the range of applicability of our fuzzer. However, we posit that our solution could be very beneficial for some classes of programs that require small inputs and have no interactions with the rest of the system. For instance, regular expression matchers, text and binary parsers, compressors, or cryptographic programs all fit this description. The programs our fuzzer would work best with is similar to the domain of applicability of *LibFuzzer*, which is itself widely used. Hence, we expect fuzzers running on GPUs to be an effective technique in the long run, with comparable utility to current fuzzers for specific workloads.

3.2 Comparison of execution models

As motivated above, we aim to fuzz CPU programs on a GPU. However, these programs were not designed to run on them, hence a layer of adaptation is needed. We contrast here the approach taken by the previous related work by *Trail of Bits* with our own, and show that the choice of translation greatly influences the usefulness of the fuzzer.

Trail of Bits used recompilation to run binaries on the GPU. By lifting them to LLVM IR and recompiling them to PTX, the programs run in a straightforward way on the device, essentially turning a CPU program into a GPU program. However, this transformation limits how much the program can be adapted to take advantage of the GPU's characteristics. For instance, the GPU's low-latency *shared memory* is unused and nothing can be done to mitigate the performance impact of control-flow divergence. Therefore, while recompilation can work, we propose a different solution.

We write an emulator capable of running CPU programs on the GPU. This strategy allows us to remediate many of the issues of recompilation:

1. **Control over GPU code.** Our emulator is written in *GLSL* and runs through the *Vulkan Compute* API. Since it is written in a language designed for GPUs, we can use all the hardware features that are available, such as shared memory, subgroup operations and workgroup

synchronization. Furthermore, our emulator is small and self-contained, which lets us control and work around divergence directly. In the recompilation approach this would not be possible without examining and potentially patching every target program.

2. **Reuse existing toolchains.** A fuzzer ought to run many programs; using existing toolchains lets us leverage the tools already at our disposition and ease the inspection of the resulting programs. In our project, we use a version of the GNU *binutils*, linked with the *musl* libc, cross-compiled to the *RV32ima* instruction set architecture, all without any patching. Since the toolchain targets a conventional CPU, we expect it to yield similar code to what could be run in production, unlike the recompilation approach which must transform the code to fit the constraints of a GPU ISA.

3. **Fuzz programs used in production.** Using an existing platform allows us to fuzz programs already in use, without requiring porting, patching, or significant modification to the program's semantics. For instance, we can run the *Lua* interpreter, which is a software libraries used in a production environments. On the other hand, CUDA toolchains are not designed to compile conventional C programs and may lead to issues when attempting to build C libraries with it.

4. **Reproducibility.** We can reproduce crashes found by our fuzzer by running the crashing input on the same binary on a real machine, or through another CPU emulator like QEMU. This ensures that crashes are not due to bugs in our fuzzer, and allows more thorough examination of the crash using off-the-shelf debuggers and other tools. Crashes on a recompiled CUDA binary cannot be run as is however: debugging must be performed on the original CPU binary, which may not trigger the same bugs in the same way, and can complicate triaging and bug-fixing.

5. **Syscall support.** While syscalls are always second class citizens on a GPU, emulation offers more flexibility. In our emulator, we treat syscalls like special instructions, and try to handle all of those that allow it in the emulator directly, without falling back onto the CPU. This strategy can also work with recompilation. However, crucially, our emulator decouples the execution of the emulator from the execution of the target program. This lets us start, pause, move and resume ethreads on arbitrary invocations transparently. This mechanism could allow offloading syscalls that need it to the CPU without blocking all the other invocations in the same subgroup.

Overall, emulation offers a consequent trade-off compared to recompilation. In exchange for all of these benefits, we expect emulated code to be much slower compared to code directly compiled for a GPU due to the emulation overhead. However, some of this performance loss can be compensated by a better use of the available hardware, notably by using shared memory and mitigating divergence. We first show how an emulator can bound the influence of divergence in section 3.3, then discuss the actual performance impact that divergence may incur in section 5.3.

3.3 Mitigating divergence using emulation

One of the most prominent advantage of emulation in running CPU programs on the GPU is that it lets us decouple the control-flow of the program from the control-flow of the emulator. Indeed, we cannot control the divergence of an arbitrary recompiled program, as its size would prohibit it and its characteristics would change between targets. On the contrary, when running a program through our emulator, only the divergence of the emulator matters, isolating the divergence of the target program.

To elaborate, the structure of our emulator is a simple loop fetching the next instruction, then switching on its opcode to emulate the relevant functionality, as sketched in Listing 3.1. This switch necessarily diverges if the ethreads within a subgroup execute instructions with different opcodes. However, this divergence is much less impactful performance-wise than the divergence of a CPU program compiled to GPU directly. Specifically, each opcode is simple to emulate, hence the size of each branch is very short. This ensures that all invocations reconverge promptly after diverging, which minimizes the impact of divergence. On top of this, preliminary measurements show that the dynamic distribution of instructions in our tests skews towards the *lw* and *addi* instructions (~30% of all instructions executed). This naturally leads to less divergence, which compounds on the performance benefit.

In summary, our emulator prevents the divergence of the target program from impacting the performance of our fuzzer to too great an extent. Instead, our emulator does diverge, but in a controlled and consistent way, bringing consistent and predictable results regardless of the workload.

Listing 3.1: Sketch of the emulator structure

```
while (!error) {
        uint instruction = load_instruction(pc);
        switch (opcode(instruction)) {
                case OP_ADDI: ...;
                case OP_LW: ...;
                ...
        }
}
```

3.4 Megafork

A fuzzer will, by nature, run the same code many times, even if a given code path is not affected by the generated inputs. One common solution to this, as employed by the AFL fuzzer, is to install a *fork server* that runs the target program up to a point, before repeatedly forking with a different value at a given place.

This strategy can be replicated in a very simple way on our GPU emulator. Indeed, the entirety of the state of an emulated thread is contained in a single contiguous segment of a few hundred kilobytes at most, and a simple memcpy suffices to implement a fork. Hence we can replicate AFL's fork server, which can greatly speed up the fuzzing process. We call this forking functionality a megafork, because it can be highly parallelized. Each invocation can perform a megafork in parallel, without contention, making the exploration of many seeds faster.

Another fuzzing strategy, used by LibFuzzer, is to write a test harness which will be called in a loop by the fuzzing library. This technique already tries to minimize work duplication by running initialization code only once at the start, but we can remove even more work using a well placed megafork. It can combine the non-repetition of the fork server with the speed of a looping fuzzer, at the cost of modifying harnesses to add this special call.

Chapter 4

Implementation

The system, as described by Figure 4.1, consists of two major parts: the driver, running on the CPU, and the emulator, running on the GPU. Before delving into the implementation itself, it is important to review the technologies used in our project. We chose to use the Vulkan Compute API instead of NVIDIA's CUDA because it is an open standard and would not limit the kind of GPUs this project could target. Most modern GPU features are now supported through Vulkan, hence there were no extra features we would have needed by using CUDA instead. While CUDA has a more complete ecosystem, especially in profiling and debugging tools, we are committed to use open standards.

4.1 The driver

The driver is a conventional Rust binary whose role is to parse the program to be fuzzed, to set up the GPU context using the Vulkan Compute API, before delegating the rest of the work to the GPU. The program extracts the segments, entry point, and memory permissions from the *RISC-V* statically linked Linux binary. Then, it allocates buffers accessible by both the CPU and the GPU, and initializes the memory. After this, a Vulkan API call passes execution to the GPU emulator to perform the actual fuzzing. Once finished, the driver reads back the results of the fuzzing run and prints out a summary.

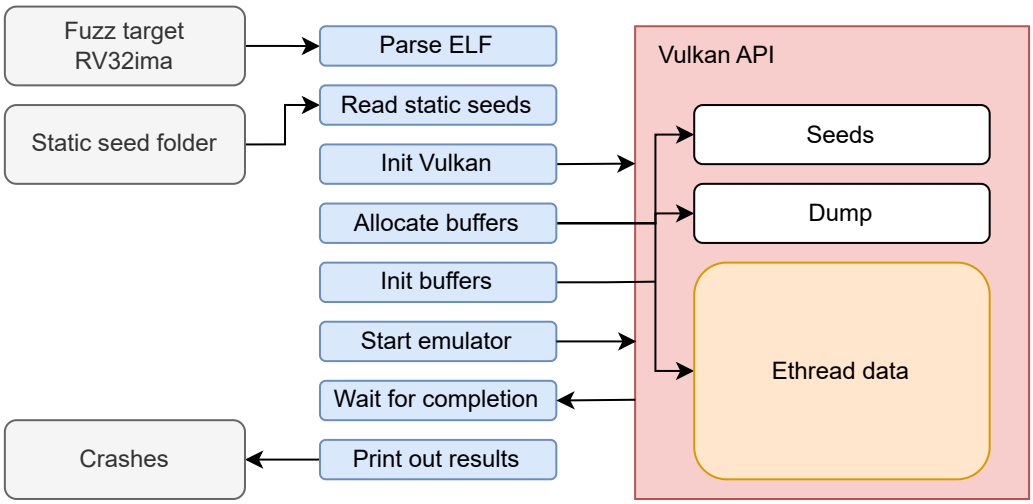

Figure 4.1: Driver high-level overview

4.2 The emulator

4.2.1 Memory layout and access

In order to describe how the emulator works, we start by discussing the memory layout of the data stored in the GPU memory buffers. This data is written by the driver before starting the emulator, and is read back once the emulator is done. It is laid out in three buffers:

1. **The seeds buffer.** Our fuzzer does not take care of mutating the inputs for implementation simplicity. Instead, we assume that an external tool would provide a corpus of seeds to run. These seeds are provided in this read-only buffer, which is split in two: first an array of *(offset,length)* tuples gives the position and length of the seeds in the buffer, followed by the seeds themselves.
2. **The dump buffer.** When encountering a crashing seeds, the emulator appends the corresponding *(offset,length)* tuple in the dump buffer. The CPU will then read it and deduce which input triggered the crash. This buffer is write-only.
3. **The ethreads buffer.** The main buffer in which the data of each ethreads is stored. We discuss its purpose below.

Ethread data layout

The ethread buffer stores all the data used by the emulated threads. As described by Figure 4.2, this buffer starts with a shared segment, and is followed by many private segments, each storing the data

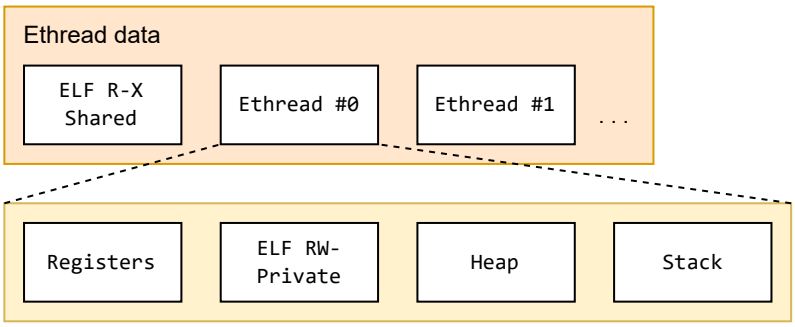

Figure 4.2: Ethread data memory layout

of a single ethread. In order to simplify and speedup the memory access of each ethread, the private segments are all statically allocated to the same size by the driver.

The shared segment holds the data stored in the *R-X* segment of the ELF program we are fuzzing. It contains notably the *rodata* and *text* sections of the executable. Since this segment is read-only, it can be shared across all ethreads. Furthermore, in most of our test cases, the shared segment contributes to around half of the memory footprint of the program. While not as fine-grained as a copy-on-write approach, this simple technique is still very effective and straightforward to implement.

The private segments of each ethread are split into multiple zones, storing their current context and private memory:

1. **Emulated registers.** The first zone stores the emulated registers of the corresponding ethread. It is initialized either by the CPU before the start of the GPU kernel, or by the emulator during a megafork. Unlike CPUs, the global memory found on GPUs is optimized for throughput, not latency. Hence, we cache the emulated registers in shared memory. We take advantage of the fact that although it is small, shared memory has comparable latency to L1 cache, letting us emulate an order of magnitude more instructions per second.

2. **ELF RW- segment.** Since this ELF segment is writeable, we need to provide a copy of the ELF *RW-* segment to every process. It is important to ensure that the size of the RW- segment remains small, since the size of this zone is not adjustable by the driver, unlike the stack and heap.

3. **Stack.** The stack is of a fixed size, chosen by a command line argument to the driver. It is initialized to match the layout that Linux would use: when starting the program, the stack pointer points to *argc*, followed by the array of *argv* pointers, the array of *envp* pointers, and finally by the *auxiliary vector*. Emulating the auxiliary vector proved to be essential, as both *glibc* and the *musl* libc read some of these vectors to initialize their datastructures, and will crash if initialized incorrectly.

4. **Heap.** In order to simplify the emulator and minimize fragmentation, we do not implement the mapping of arbitrary *VMAs* (Virtual Memory Areas). However, most programs use a heap, hence we allocate a single zone of memory to fulfill that purpose. In order to access it, we rely on the programs using the *sbrk* syscall to allocate memory. This is facilitated by the fact that we use *musl*'s old memory allocator, which uses sbrk for all memory allocations under a few megabytes. The *brk* of the ethread is tracked in a special extra register and is to check the validity of memory accesses during address translation.

Address translation

Our emulator uses a fixed set of VMAs to represent memory. Hence, memory translation is simple and fast. We first perform a range check against the start and end virtual addresses of each segment, and then perform simple arithmetic to convert the virtual pointer into a raw offset into the ethread data buffer. Address translation is a very frequent operation, performed both during instruction fetch and the emulation of memory instructions, such as *lw* and *sw*. Given that we are emulating a processor, we also have to perform all address translation in software. While this provides more flexibility with respect to the memory layout, its implementation has to remain simple to be fast. Hence why we do not implement arbitrary VMA allocation and page mapping.

4.2.2 State machine

The emulator is implemented by a kernel written in GLSL. It is started by the CPU, as shown in Figure 4.1, and emulates an ethreads' execution. Figure 4.3 shows a high-level diagram of its execution. The seeds are managed globally between all invocations, to ensure that each seed is only run once. A single atomic operation is needed to pick a seed while preventing other threads from picking the same again. It is important to keep operations that communicate between threads to a minimum to ensure that the scaling between performance and number of invocations remains linear. This is also the reason why we cache the emulated register of the current ethread in shared memory. Global memory on GPUs is designed to maximize throughput, at the cost of latency, hence keeping data used very often as close as possible to its invocation is crucial for performance.

4.2.3 Special syscalls

It is useful to define an interface allowing communication between the emulated program and the emulator. For instance, we wanted to have a way for the program to request a new seed to be written in a given buffer. This is useful as it allows wrapping the test program in a loop which fetches a new seed before each execution, skipping the startup and teardown cost of a new ethread instance every time (in some ways similar to in-process fuzzing, like LibFuzzer). We settled on implementing these

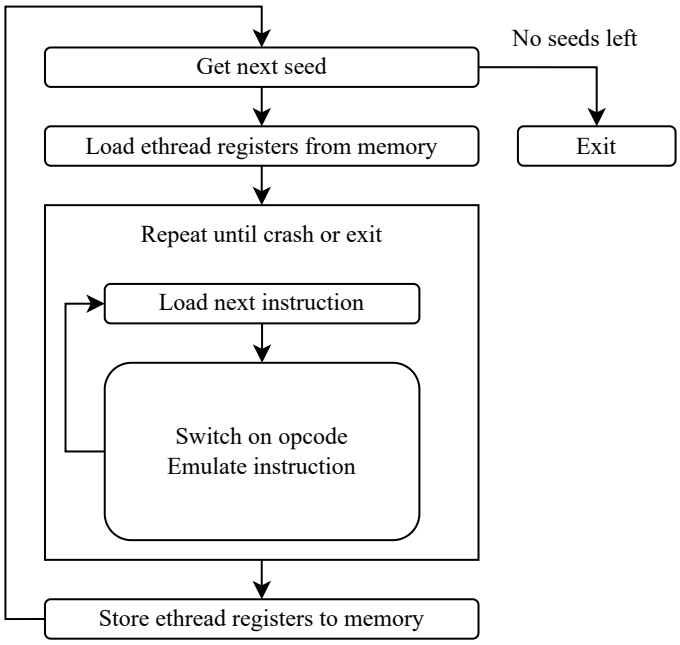

Figure 4.3: Emulator high-level overview

functionalities using custom syscall numbers, since this communication mechanism was already implemented and trivial to use for the target program. Hence, we define a few extra syscalls, using the following numbers:

1. 1337 returns the current ethread id.
2. 1338 returns the length of the current seed.
3. 1339 writes the current seed to a given pointer.
4. 1341 fetches a new seed from the global seed pool, if any are still left.

4.3 Target program compilation

In order to run a program on our GPU emulator, it is necessary to compile it in a compatible way. In particular, because we chose the RV32ima ISA to simplify the implementation of the emulator, we need to compile the target programs with a cross-compiler. Furthermore, we must statically link all libraries the program may need, including libc. We do not support dynamic linking for two reasons: first, our emulator allocates a static set of VMAs; second, dynamic linking is very complex and poorly documented, and would add significant complexity to the driver. Originally, we compiled our programs with *glibc*, to ensure our emulator would be compatible with it. However, we now

favor the *musl* libc because it used less syscalls, has a much smaller code size footprint, starts faster and has a simpler allocator, making it a better fit for our uses.

Chapter 5

Evaluation

5.1 Experimental setup

Our experiments will all be performed on the same test program, displayed in Listing 9. The test is inspired from LibFuzzer testing harnesses present for the *Lua* interpreter in the *oss-fuzz* repository. Given that our fuzzer does not support compiling LibFuzzer harnesses directly, we compile the tests with a helper program which loads seeds from files and runs the target harness in a loop. The GPU hardware we use in the following experiments is an Nvidia 4070ti card, with 7680 CUDA cores and 12GiB of on-device memory. The CPU used in section 5.4 is an Intel i7-8565U, with 16GiB of dual channel DDR4 clocked at 2400MT/s. All tests are performed on an up-to-date Arch Linux system. Given our usage of recent Vulkan Compute features, a distribution with up-to-date packages is needed.

5.2 Multithreaded performance scaling

Before any comparison to other systems, we check that the basic assumptions we have about our system hold. Given that we design our emulator for a highly parallel system, we first ensure that the performance of the emulator scales linearly with the number of invocations allocated to it.

Our test case is a modified version of a Lua LibFuzzer harness, taken from oss-fuzz. We removed unnecessary memory allocations to make the workload more compute bound. The seed corpus is taken directly from oss-fuzz as well. The exact code for the experiment is illustrated in Listing 9.

As shown in Figure 5.1, on the workload specified in Listing 9, the number of instruction emulated per second does scale linearly with the number of invocations. The scale chosen stops at 30720 because it is the maximum number of invocations that could be spawned at once. Hence, the

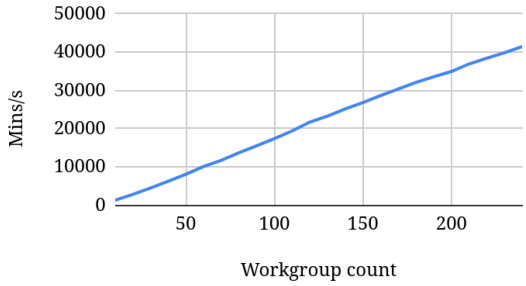

Figure 5.1: Millions of instructions emulated per second relative to workgroup size

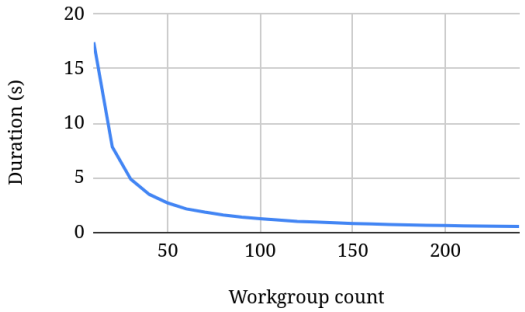

Figure 5.2: Execution time relative to workgroup size

performance of our emulator scales as expected with the performance of the underlying GPU.

5.3 Impact of divergence

As discussed in the design and implementation sections, we postulate that our emulator allows restricting the performance impact of divergence. In order to validate that claim, we setup the following experiment: we run the same test program as the previous experiment, but with a custom corpus of seeds. They are generated from a pool of 2^N distinct seeds, with N from 1 to 7. The seeds are then replicated until a given common number of total seeds is met, 3072000 total seeds in our experiment.

Our experiment shows that with $N = 0$, all seeds are be identical, hence divergence is eliminated and we get the maximum performance out of the GPU. With $N = 1$, two seeds are mixed into the pool, and the performance predictably drops. As the number of distinct seeds grows, performance drops, before plateauing at 32, which is the subgroup size of our card.

5.4 Performance comparison with CPUs and QEMU

One of the main motivation for our research was to assess the viability of using GPU emulation from a performance perspective. We present here a three-way comparison between a classic CPU implementation of our test case, a RISC-V cross-compiled version ran on our GPU emulator, and this version run in the RISC-V32 mode of QEMU user. A comparison with Trail of Bits's recompilator would have been appreciated, but we could not find the source code of their work and experiments online.

The choice to compare with QEMU-user might seem inappropriate, but it is an important data

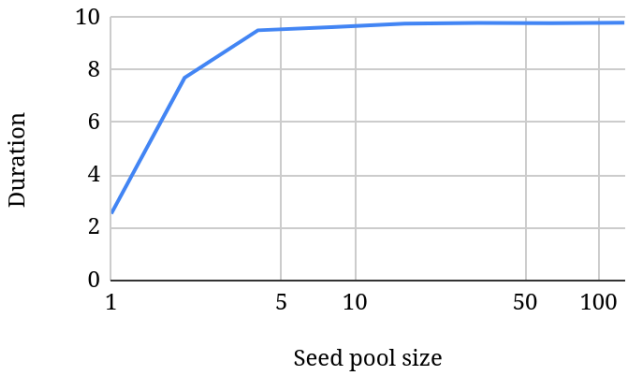

Figure 5.3: Execution time relative to seed pool size

point to quantify. The first reason is that the compiled output of the same C program depends on the compiler's target ISA. In particular, our emulator only emulates RV32ima, that is, it does not support floating point instructions. This is not a fundamental restriction of the emulator, since GPUs also have many hardware floating point units, but rather a conscious decision designed to simplify our system. Because of this, GCC replaces all floating point operations with software reimplementations, which can get very expensive depending on the workload. Hence, running the test through QEMU-user ensures that we are comparing the exact same set of instructions.

QEMU also incurs a significant performance overhead, which ought to be taken into consideration when reading the results. However, this comparison can also inform the utility domain of our system. Indeed, fuzzing some software, like embedded systems firmware for instance, would often rely on emulation anyway. On top of this, these applications typically require very little memory, making them an ideal fit for our emulator. By comparing with QEMU's performance, we can get a meaningful comparison of the performance of our systems in the context of systems which necessitate emulation.

Configuration	**Time**
Native X86	0m13.340s
Emulated rv32-qemu	3m15.346s
GPU rv32	0m5.092s

Table 5.1: Performance Comparison

As we can see from Table 5.1, our emulator and the native execution by our processor are in the same order of magnitude. However, the performance difference with QEMU is much higher, letting us believe that our emulator can be very competitive in fuzzing scenarios requiring emulation.

Chapter 6

Future work

6.1 Optimizations

While our implementation is very capable of running programs, many performance optimizations have not been used, in the name of correctness and simplicity. Hence, there are a few optimizations and features that we could implement to improve the fuzzer's performance.

1. **Memory striping.** We could do the same memory striping optimization that Trail of Bits did to improve memory locality. There is nothing that prevents us from implementing that, and it could dramatically help performance since memory latency appears to be one of the biggest bottlenecks of our implementation.

2. **Copy-on-write (CoW) memory.** Although we already do some memory deduplication by sharing the non-writeable segments of the target program, copy-on-write memory could still improve the memory usage of the fuzzer. This could be particularly critical since GPUs come with a limited amount of on-device memory. Furthermore, megaforking could also benefit from it since CoW could considerably reduce the amount of memory to copy at every megafork.

3. **Port to CUDA.** Our use of Vulkan Compute on Nvidia GPUs leaves us outside of their closed CUDA ecosystem, which restricts how performance can be measured. Reimplementing the project in CUDA could be the path forward, as it would allow more fine-grained performance metrics. It could also allow running our fuzzer on Nvidia's datacenter GPUs like the A100, which do not support Vulkan.

4. **Better syscall support.** Our emulator supports a minimal number of syscalls to simplify the implementation. The ones that are implemented are just enough to let glibc and musl libc start a program and perform memory allocation.

5. **Bigger memory allocation.** The emulator is written in GLSL, which limits the maximum size of a buffer to 4GiB. Hence, we do not use more than 4GiB of on-device memory to simplify buffer allocation and the mapping of ethreads to buffer offsets. This is not a real limitation of our implementation, but only a simplification that can be overcome with a bit more engineering. Targeting CUDA may also relax the maximum size restriction.

6.2 Scheduler

The importance of mitigating divergence is only as relevant as it is the bottleneck of performance. Our measurements show that our implementation is only 5 times slower because of divergence, mitigating the freedom with which to tradeoff execution speed for divergence mitigation. Nonetheless, we still document our attempts to write a scheduler, and why it may not be relevant to implement in our particular situation.

The design of our emulator allows us to conceptually move an ethread between different invocations. By storing its full state in memory, an ethread can be paused and continued in a different invocation, and crucially, in a different subgroup. Indeed, divergence is only a concern at the subgroup level, hence the idea to move ethreads such that all invocations within a subgroup only execute ethreads which are about to run an emulated instructions with the same opcode. For instance, a list of ethread could keep track of all those that will run a *addi* next, such that a subgroup would just need to pick all of these ethreads at once and avoid divergence.

Unfortunately, our experiments show that implementing a scheduler does not improve performance in our case. This in part because the performance impact of divergence is not as important as we hypothesized: in our experiment, divergent workloads are only at worst 5 times slower than non-divergent ones. Hence the tradeoff between the performance improvement of scheduling and its cost does not make sense in our situation.

6.3 A different bug-finding technique

This highly parallel approach to emulation may be used to other ends than exclusively fuzzing. In particular, the combination of the emulator and the megaforking could allow us to check the invariants of a function that mutates global or expensive-to-build state by exhaustive search. This is not already directly feasible in current fuzzers. Fork servers have their own inefficiencies because they must create entire processes, while looping fuzzers like LibFuzzer require building and destroying the input state on every iteration since they do not automatically roll-back state between seeds. While such bug-finding may prove impractical in real-world codebases, it could be used in a different way, for instance to check that a given patch effectively fixes a bug. If the function being

patched is deep inside a call tree or mutates global state, which would be difficult or complex to rollback, our implementation could prove useful. We leave the exploration of such uses to future work.

Chapter 7

Related Work

7.1 Parallel fuzzing

Normally, fuzzing is a strictly single-threaded process, as it revolves around the tight loop of execution, coverage collection, and mutation. Parallel fuzzing instead is the process of running multiple instances of a single fuzzer to maximize the computation capabilities of the available machine. Many parallel fuzzing approaches have been proposed before [3, 7, 9].

Typically, these methodologies initiate multiple instances of fuzzing in separate processes, with periodic synchronization of the seed corpus. Each instance adheres to the fundamental logic of the original single-instance fuzzer, which is inherently designed for serial execution. This involves a sequential fuzzing loop where a test case is selected from the input queue, mutated to produce new test cases, and executed with the new input while collecting feedback. Each instance independently maintains its fuzzing state, including a code coverage bitmap and a corpus of interesting test cases.

A critical aspect of effective parallel fuzzing is the synchronization of states: all instances must collaboratively contribute to the exploration of the program state and the detection of bugs.

7.2 Fuzzing via emulation

A number of studies have been performed on running a fuzzer on an emulated program, instead of directly on a native environment. The most common use case is due to dealing with complex software environments, where direct instrumentation of the target binary is not feasible For instance, AFL-QEMU [6] mode leverages the QEMU user-mode emulation to enable fuzzing of binaries without the need for source code or binary modification. QEMU, a generic and open-source machine emulator, can emulate the CPU instructions of the target binary, allowing AFL to monitor and

manipulate its execution. This approach is particularly advantageous for closed-source software, legacy binaries, or environments where recompilation is impractical.

7.3 Fuzzing targeting the GPU (and relevant drivers)

Fuzzing for GPU drivers and APIs has gained significant attention in recent years due to the increasing reliance on GPUs for both graphical and computational tasks. Traditional fuzzing techniques, primarily designed for CPU-based applications, often fall short when applied to the unique architecture and execution models of GPUs. One such example is CUDAsmith [5], which automatically generates valid CUDA kernel code, extending CSmith and targeting the CUDA compiler and runtime suite.

7.4 GPU memory management

It has been increasingly more common to use GPUs for general-purpose computing. However, due to the limited memory available for graphic cards, and the expensive performance cost in accessing main memory, extensive research was done to assess the applicability of various memory management strategies for GPUs.

For example, the work by Suzuki et al. [8] illustrates how clever batching of memory accesses, in addition to a custom GPU *shadow page table*, enables for efficient data transfer and memory management for GPU general purpose computation.

Chapter 8

Conclusion

To conclude, we demonstrated that running CPU programs on the GPU using emulation is possible. Furthermore, our approach shows promising performance while tackling many unresolved issues of other similar systems. We believe that some future work remains to be done to bring its utility up-to par with current state-of-the-art fuzzer, but that it will find its niche as an emulation accelerator.

Bibliography

[1] Trail of Bits. URL: https://blog.trailofbits.com/2019/09/02/rewriting-functions-in-compiled-binaries/.

[2] Trail of Bits. URL: https://github.com/lifting-bits/remill.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1032–1043.

[4] Faith Ekstrand. URL: https://www.gfxstrand.net/faith/blog/2020/10/does-subgroup-wave-size-matter/.

[5] Bo Jiang, Xiaoyan Wang, Wing Kwong Chan, TH Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. "Cudasmith: A fuzzer for cuda compilers". In: *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2020, pp. 861–871.

[6] Romain Malmain, Andrea Fioraldi, and Francillon Aurélien. "LibAFL QEMU: A Library for Fuzzing-oriented Emulation". In: *BAR 2024, Workshop on Binary Analysis Research, colocated with NDSS 2024*. 2024.

[7] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin IP Rubinstein. "Towards systematic and dynamic task allocation for collaborative parallel fuzzing". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1337–1341.

[8] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. "{GPUvm}: Why Not Virtualizing {GPUs} at the Hypervisor?" In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 109–120.

[9] Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos, and Jun Xu. "Facilitating parallel fuzzing with mutually-exclusive task distribution". In: *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II 17*. Springer. 2021, pp. 185–206.

Chapter 9

Appendix

```
/// File: luatest.c

#include <stdio.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>

int LLVMFuzzerTestOneInput(char* data, size_t size) {
  lua_State* L = luaL_newstate();
  if (L == NULL) exit(3);
  data[size - 1] = '\0';
  size_t sz = lua_stringtonumber(L, data);
  lua_settop(L, 0);
  lua_close(L);
  return 0;
}

/// File: main.c

#include <stdint.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
```

```
#if defined(__cplusplus)
extern "C" {
#endif

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size);

struct list {
        struct list* next;
        long len;
        char* data;
};

int main(int argc, char** argv, char** envp) {
#ifdef a__riscv

        uint8_t* data = (uint8_t*)malloc(4096);

        while (1) {
                size_t size = syscall(1338);
                if (size > 4096) return 1;
                if (data == NULL) return 2;
                if (syscall(1339, data, size) != 0) return 3;

                int ret = LLVMFuzzerTestOneInput(data, size);
                if (ret != 0) return ret;
                if (syscall(1341) != 0) return 0;
        }

#else

        struct list* head = NULL;
        int count = 0;
        for (int i = 1; i < argc; ++i) {
                FILE* f = fopen(argv[i], "rb");
                if (f == NULL) return 1;

                struct list* node = (struct list*)calloc(1, sizeof(struct list));
                node->next = head;
                head = node;

                fseek(f, 0, SEEK_END);
                node->len = ftell(f);
                node->data = (char*)calloc(node->len, 1);
                if (node->data == NULL) return 3;
                fseek(f, 0, SEEK_SET);
```

```
                long bs = fread(node->data, 1, node->len, f);
                fclose(f);
                if (bs != node->len) {
                        printf("error: expected %li but got %li", node->len, bs);
                        return 2;
                }
                count += 1;
        }
        char* targets = getenv("SEED_REPEAT_COUNT");
        long seed_rep = 1;
        if (targets) {
                char* endp;
                errno = 0;
                seed_rep = strtol(targets, &endp, 10);
                if (errno != 0 || *endp != '\0') {
                        return 1;
                }
        }
        printf("Done loading.\n");
        while (head != NULL) {
                for (int i = 0; i < seed_rep; ++i) {
                        LLVMFuzzerTestOneInput((const uint8_t*)head->data, head->len);
                }
                head = head->next;
        }
        return 0;
#endif
}

#if defined(__cplusplus)
} /* extern "C" */
#endif
```