cartesi

Running Linux in ZK

There are crazier things than the EVM to run in ZK

cartesi.io

Carsten Munk - CTO @ Zippie, also a Board Director @ Cartesi Foundation - zkLinux is work done under a public goods grant from the Cartesi Foundation. This talk was constructed in an augmented intelligence fashion with ChatGPT (GPT-4), Midjourney and feedback from other team members.

zkWarsaw Day
29 / 8 / 2023

cartesi

What is the Cartesi Machine?

An open source RISC-V emulator implementing a fully deterministic RV64GC ISA (read: runs unmodified Ubuntu)

- Written in C++, LGPL 3.0 licensed
- 64-bit memory addressability
- Developed since 2018 and still actively developed
- All machine state is in memory and can be merkle-ized at 64-bit word level
- Can persist state at any step and resume execution
- Optimistically provable in a verification game manner
- IMNSHO: The ultimate runtime for ZK apps is a general purpose ISA/processor that can run a full 64-bit OS

https://github.com/cartesi/machine-emulator

Table 2: *The processor state. Memory-mapped to the lowest 512 bytes in physical memory for external read-only access.*

Offset	State	Offset	State
0x000	x0	0x160	misa
0x008	x1	0x168	mie
...	...	0x170	mip
0x0f8	x31	0x178	medeleg
0x100	pc	0x180	mideleg
0x108	mvendorid	0x188	mcounteren
0x110	marchid	0x190	stvec
0x118	mimplid	0x198	sscratch
0x120	mcycle	0x1a0	sepc
0x128	minstret	0x1a8	scause
0x130	mstatus	0x1b0	stval
0x138	mtvec	0x1b8	satp
0x140	mscratch	0x1c0	scounteren
0x148	mepc	0x1c8	ilrsc†
0x150	mcause	0x1d0	iflags†
0x158	mtval		

†Cartesi-specific state.

https://cartesi.io/cartesi_whitepaper.pdf

cartesi

What is RiscZero?

The RISC Zero zkVM is a verifiable computer that works like a real embedded RISC-V micro-processor, enabling programmers to write ZK proofs like they write any other code.

- Apache2 license and includes a full proving and verification system.
- Supports Rust for writing ZK proofs. Any language that compiles to RISC-V can be supported.
- 32-bit RISC-V RV32IM ISA; ~192mb RAM in guest
- Continuations enable not having to worry about instruction count (splits execution over multiple proofs)
- Recursion and STARK→SNARK→EVM through online service (Bonsai)
- https://risczero.com

```rust
#![no_main]
#![no_std]

use risc0_zkvm::guest::env;

risc0_zkvm::guest::entry!(main);

pub fn main() {
    // Load the first number from the host
    let a: u64 = env::read();
    // Load the second number from the host
    let b: u64 = env::read();
    // Verify that neither of them are 1 (i.e. nontrivial factors)
    if a == 1 || b == 1 {
        panic!("Trivial factors")
    }
    // Compute the product while being careful with integer overflow
    let product = a.checked_mul(b).expect("Integer overflow");
    env::commit(&product);
}
```

cartesi

The game plan

- Proof: Execution from machine state X until C RV64GC cycles becomes state Y
- Include sufficient amount of Cartesi Machine code into a RiscZero guest so it can execute full RV64GC cycles
- Start up a real Cartesi Machine and RPC with it in the host to retrieve memory + machine state hash at the starting cycle
- Have the host provide the memory contents in full at that step to the guest*
- Then see how many cycles we can fix in one RiscZero segment (proof)
- Then prove we iterated from initial machine state X with several iterating proofs of execution between states, to a final state Y (recursable)
- Then prove from initial machine state X of a machine containing a Linux kernel and root file system until halt of machine
- And then add memory commitments (machine started from state X and we're now in state Y after Z cycles) [not done yet]

cartesi

The simple part for an embedded Linux engineer

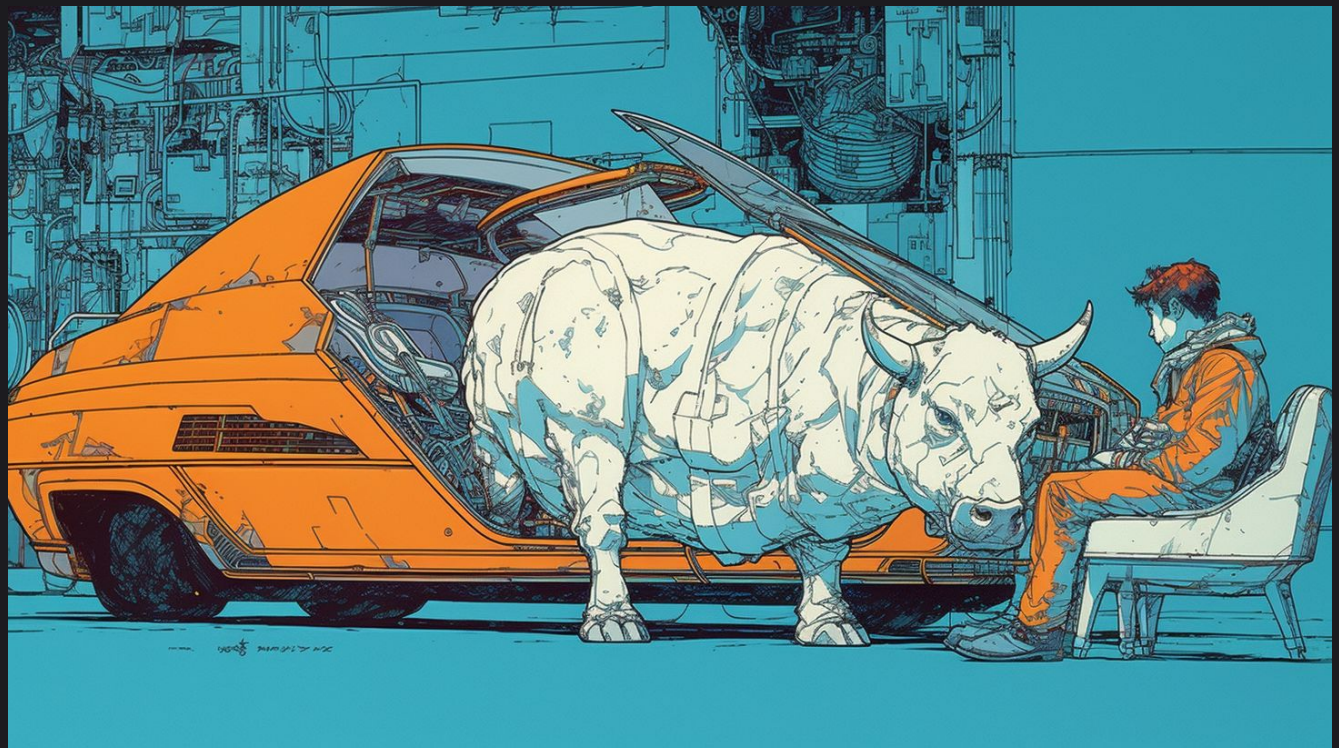
- Build Cartesi Machine for RV32IM ISA
- No libc, stdlib nor allocator - only stack. Cartesi Machine doesn't need allocation as all state is in the RAM of the machine (or static variables)
- Merge Cartesi Machine .o files into one .o with 'ld -relocatable' and use 'cc' Rust crate to bridge RiscZero guest code
- Export some symbols to the C++ code and import some

```rust
#[no_mangle]
pub extern "C" fn printout(c_string: *const c_char) {
 let s = unsafe { CStr::from_ptr(c_string).to_string_lossy().into_owned() };
 println!("printout: <{:?}>", s);
}
```

```rust
 extern {
     fn run_uarch(mcycle_begin: u64, mcycle_end: u64) -> u64;
 }
```

cartesi

So how exactly do we fit a 64-bit address space into a 32-bit guest?



cartesi

Paging on RISC-V TLB level!

- Cartesi Machine implements a TLB which all physical memory accesses are resolved through on a page level
- Each TLB entry of 4K (page size) is given an offset between the physical address in-machine and in-emulator to speed up accesses in futures
- Always dirty TLB and processor state pages
- Upon hitting a page we've not accessed before we would through host-guest communication page/copy it in and commit to its original hash
- And then commit hashes of dirty pages in the guest upon completion

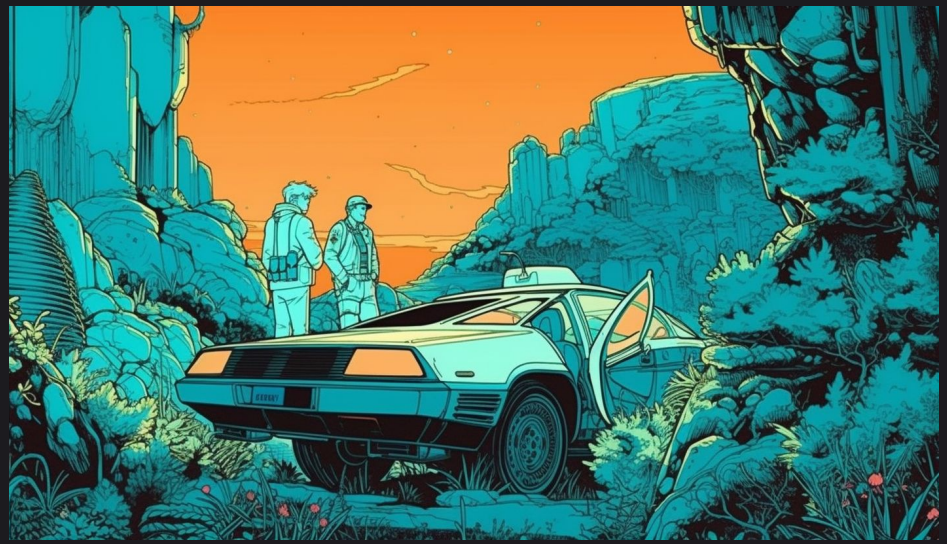
cartesi

It’s alive!

```
  --> host/src/main.rs:23:20
   |
23 | use sha2::{Sha256, Digest};
   |                    ^^^^^^

warning: `host` (bin "host") generated 6 warnings (run `cargo fix --bin "host"` to apply 5 suggestions)
    Finished release [optimized + debuginfo] target(s) in 58.70s
warning: the following packages contain code that will be rejected by a future version of Rust: rstest v0.11.0
note: to see what the problems were, use the option `--future-incompat-report`, or run `cargo report future-incompatibilities --id 1`
     Running `target/release/host 'http://127.0.0.1:50051''
starting at mcycle 7200000
. mcycle 7200000 total segments this session 0
tty: [    0.000000] Linux version 5.15.63-ctsi-2 (
tty: developer@buildkitsandbox) (riscv64-cartesi-linux-gnu-gc
tty: c (crosstool-NG 1.24.0.199_dd20ee5) 10.2.0, GNU ld (cross
tty: tool-NG 1.24.0.199_dd20ee5) 2.35.1) #1 Thu Mar 30 12:58:
tty: 59 UTC 2023
[    0.000000] OF: fdt: Ignorin
tty: g memory range 0x80000000 - 0x80200000
[
tty:    0.000000] Machine model: ucbbar,riscvemu-bare

tty: [    0.000000] Zone ranges:
[    0.000
tty: 000]   DMA32    [mem 0x0000000080200000-0x000000083fefff
. mcycle 7300000 total segments this session 54
tty: f]
[    0.000000]   Normal   empty

tty: [    0.000000] Movable zone start for each node

tty: [    0.000000] Early memory node ranges

tty: [    0.000000]   node   0: [mem 0x000000008020000
tty: 0-0x0000000083feffff]
[    0.000000] Initme
```

Performance

- Ridiculously unoptimized: Around 2000-3000 RV64GC cycles per 2^20 RiscZero cycles (1 proof segment) depending on a few factors
- 1 proof segment on GPU ~17 seconds (execution of RV32IM included)
- 31 million RV64GC cycles for Linux bootup ~=2 days on single GPU
- on 64 GPUs ~= 2744 seconds
- Future work: Upstream emulator patches, Update to Risc0 0.17, Predecoding, leveraging ‘hints’ from host for more performant behaviour, proven JIT to RV32IM of code pages, seeing how many RV64GC cycles per 2^20 RiscZero we can get in, inspirations from UTM (iOS). Specialized device drivers in Linux for acceleration in custom ZK circuits.



https://cartesi.io/cartesi_whitepaper.pdf

cartesi

Usecases

- zkPHP
- zkDoom
- zkMongoDB
- zkMinecraft
- zkWindows ME
- zk-'cargo build' (or anything)
- Start from an already ZK proven Linux boot and replace memory areas ('disks') before root file system is mounted and never have to re-compute Linux boot again unless upgrading kernel
- Proof of exploit on anything we can execute in a RISC-V Linux (this includes X86-64 emulators)
- Some parts of a Cartesi Machine execution public, some parts private

cartesi

https://github.com/stskeeps/poc-cartesi-r0

https://github.com/stskeeps/machine-emulator/tree/sts/experimental/risczero-uarch

Thanks.

Twitter/Telegram/Bluesky: @stskeeps
Further discussion: cartesi.io discord

Special thanks go to Yelyzaveta Dymchenko for running the first cycles of a RV64GC using rvemu on RiscZero in October 2022