DeCl: Deterministic and Metered Native Sandboxes

Zachary Yedidia
Stanford University

Geoffrey Ramseyer
Stanford University

David Mazières
Stanford University

Abstract

We present Deterministic Client (DeCl), a software sandboxing system for ARM64 that securely runs untrusted machine code at near-native speeds while guaranteeing deterministic execution and deterministic termination after a specified instruction count.

One key application is a smart contract engine in a digital currency system, which requires both deterministic execution and metered runtime. Today's blockchains, for example, run bytecode in a virtual machine, such as WebAssembly or the Ethereum Virtual Machine, which either incurs the extremely high overhead of a software interpreter or expands the TCB to include an entire just-in-time compiler, while still paying significant runtime overhead.

Instead, DeCl can execute smart contracts as native ARM64 programs, after validating them with a simple and efficient verification pass. The compilation and instrumentation process, to produce code that can pass these verification checks, is entirely untrusted. DeCl incurs a geomean 16% runtime overhead on supported SPEC 2017 benchmarks compared to native code, while WebAssembly interpreters incur nearly 2000% overhead on the same benchmark set, and even state-of-the-art just-in-time WebAssembly compilers incur upwards of 90% overhead.

We integrate DeCl into Groundhog, an existing smart contract engine that uses interpreted WebAssembly, and demonstrate a significant performance improvement for compute-intensive contracts, and minimal performance loss otherwise.

2012 ACM Subject Classification Security and privacy → Distributed systems security; Software and its engineering → Automated static analysis

Keywords and phrases Smart Contract Performance, Sandboxing, Determinism

1 Introduction

Smart contracts serve as the foundation for extensible digital currencies. In order to work on a blockchain, smart contracts must be deterministic and have bounded execution time. Existing systems use custom languages like EVM bytecode and WebAssembly to enforce these properties. For maximum performance, these languages could be compiled to machine code using highly optimizing compilers. However, these compilers typically have bugs and cases of pathologically slow compile times. As a result, it is not generally safe to use optimized compilation for smart contracts, and many existing systems use interpreters instead, which are orders of magnitude slower than optimized native code. Even JIT compilers such as Cranelift that are designed to accept malicious code are large and complex, and face a tradeoff between performance and security: adding more optimizations means increasing the possibility of a vulnerability. No prior systems are able to safely run smart contracts directly as highly optimized native code.

We present Deterministic Client (DeCl), a sandboxing system that makes bare-metal smart contracts possible. With DeCl, smart contracts are expressed directly as ARM64 machine code, which can be executed natively on cloud servers such as AWS Graviton [22]

or Ampere Altra [12, 27], and personal computers such as Apple Macs. DeCl uses a machine code verifier to ensure that the ARM64 code loaded into a sandbox is isolated, deterministic, and metered. This trusted verifier uses a simple linear-time algorithm to determine whether an ARM64 program should be accepted or rejected. The compilation and intrumentation process, to produce code that can pass verification, is entirely untrusted. As a result, highly optimizing ahead-of-time compilers such as LLVM can be used without compromising the security of the system.

DeCl is built on top of a prior sandboxing system called Lightweight Fault Isolation (LFI) [41]. We show how to extend LFI to enforce that untrusted native code is both deterministic and preemptable, with two primary innovations:

1. Position-oblivious code (POC), an extension to LFI that makes programs deterministic no matter where in the address space they are loaded. This extension works by preventing programs from directly reading their own absolute addresses, and only allowing offsets from a base address to be read.
2. Two approaches for deterministic metering, so that programs are preempted before a maximum number of instructions have been executed. Crucially, this preemption is deterministic, meaning that programs always produce the same result, even when preempted.

DeCl's deterministic metering works by keeping a count of the program's remaining "gas" in a reserved register. This register must be updated and checked at specific places in the program (such as before branch instructions). Unfortunately, these checks result in additional overhead. We present several optimizations to decrease this overhead. When running supported benchmarks from SPEC 2017, DeCl incurs a geomean runtime overhead of around 16% compared to native code, when using the most efficient metering scheme. For comparison, all existing approaches use either interpreters or JIT compilers with metering support, which incur overheads of roughly 2000% and 90% respectively (as measured on SPEC 2017 using WebAssembly implementations). On top of the performance benefits, DeCl's machine code verification approach is significantly simpler (and therefore more secure) compared to JIT compilation. We propose two techniques for protecting the control-flow integrity of gas update sequences, either using aligned bundles or hardware-enforced control-flow if available.

We integrate DeCl into Groundhog [32], a scalable smart contract engine that currently uses a WebAssembly interpreter. Using native code for smart contracts allows them to run significantly faster and to have access to hardware SIMD and cryptography intrinsics. Since smart contracts are generally short-lived, we made significant optimizations to achieve a sandbox startup latency of roughly $30\mu s$, and we ensured that loading sandboxes does not interfere with Groundhog's scalability.

The rest of the paper is organized as follows: Section 2 explains LFI, the sandboxing foundation upon which DeCl is built. Section 3 describes the position-oblivious code extension to LFI, which makes programs deterministic no matter where in the address space they are loaded. Section 4 describes our approaches for deterministic metering: branch-based and timer-based metering. Section 5 describes our implementation, and other modifications needed to enforce determinism on top of LFI. Section 6 evaluates DeCl on the SPEC 2017 benchmark suite. Section 7 describes and evaluates DeCl's integration into Groundhog. Finally, Section 8 summarizes related work.

2 Background

2.1 Threat Model

In this work we are concerned with adversarially enforcing determinism. An untrusted party provides a program expressed as ARM64 machine code, and it should only be accepted if it is guaranteed to be deterministic. The system must reject programs that are non-deterministic, and may reject programs if it cannot determine whether or not they are deterministic. Nonetheless, we would like to be able to accept a wide variety of programs, while imposing minimal performance overhead.

We also require that the programs that run inside DeCl are *metered*, and therefore cannot use more than a limited amount of CPU time. This requirement when combined with the determinism requirement requires special design, since the preemption must happen in a deterministic way. The runtime system is initialized with a limit on the number of instructions that can be executed before preemption occurs.

We assume the following properties about the runtime system, which are typical for smart contract engines:

1. The runtime does not provide any support for shared-memory multiprocessing.
2. The runtime provides a set of deterministic "runtime calls" to the sandbox that it may use to interact with the runtime.

As part of enforcing determinism, the sandboxed program must not be able to access data outside its specific memory region, which is deterministically initialized. This requires some form of memory isolation. Since many smart contracts are short-lived programs, startup time is performance-critical and we therefore require a memory isolation scheme that can support fast startup and teardown. For this reason, we have chosen to use software isolation by building on top of LFI, although in principle DeCl could be used in combination with a hardware-based protection mechanism as well.

2.2 Lightweight Fault Isolation

Lightweight Fault Isolation (LFI) is a sandboxing system for ARM64 that uses software-based fault isolation (SFI). Programs that execute within LFI are restricted to a 4GiB region of contiguous virtual memory that they may access, and may not execute system call instructions, or other instructions deemed "unsafe." These restrictions are enforced via a static verifier that analyzes the machine code of untrusted programs. Importantly, in the ARM64 encoding every instruction is a fixed 4-byte value and must be aligned, meaning machine code can be accurately disassembled without needing to protect against malicious programs that might jump into the middle of an instruction.

The basic LFI scheme allocates each sandbox at an address aligned to 4GiB. Several registers are reserved for special use:

- `x21`: stores the base address of the sandbox.
- `x18`: always contains an address within the sandbox.
- `x30`: always contains an address within the sandbox.
- `sp`: always contains an address within the sandbox.

Note that ARM64 has 32 64-bit integer registers (`x0`-`x30` and `sp`), which can also be accessed via names that only access the bottom 32 bits (`w0`-`w30` and `wsp`). When `wN` is written, the top 32 bits of `xN` are set to zero.

The static verifier will reject any program that contains instructions that modify reserved registers without maintaining their invariants. For example, the only way to modify `x18` is via an instruction that guarantees that the resulting value stored in `x18` is within the sandbox (i.e., has the same top 32 bits as those in `x21`). The following instruction can be used for this purpose:

```
add x18, x21, wN, uxtw
```

This instruction takes the 32-bit value `wN` (namely the low 32 bits of `xN`), treats it as an **u**nsigned 32-bit value before e**xt**ending it to 64 bits (specified by the `uxtw` modifier), adds the extended value to register `x21`, and stores the result in `x18`. Since `x21` contains the 4GiB-aligned base address of the sandbox region, the sum stored to `x18` is guaranteed to be within the sandbox, regardless of the value of `xN`.

Since `x18` is always guaranteed to hold a valid address within the sandbox, it is safe to perform loads or stores such as `ldr xN, [x18]`. LFI includes multiple optimizations not discussed in detail here. The most important is the use of the 32-bit addressing mode, that effectively allows the safe `add` from above to be performed as part of a load/store instruction with no additional overhead. The instruction `ldr xN, [xM]` can be transformed into the safe equivalent `ldr xN, [x21, wM, uxtw]`, which incurs no overhead.

Each sandbox is allocated 4GiB of virtual memory, with 80KiB guard regions (unmapped pages) placed at the start and end of the sandbox. These guard pages are necessary to prevent loads and stores from neighboring sandboxes, since `ldr xN, [x18, #i]` is a legal load of the address `x18 + i`. Due to the instruction encoding, the immediate, `#i`, can be at most 65536, and thus such a load cannot access beyond a neighboring sandbox's guard region.

The LFI static verifier is a simple program that performs a single linear pass over a compiled binary to determine if it is safe to run. LFI is a particularly attractive sandboxing system compared to language-based systems for smart contract or other security-critical settings because the simplicity of the verifier reduces the size of the trusted code base (TCB). Existing smart contract systems leave a significant amount of performance on the table, because they generally rely on either slow (but simple) interpreters, or JIT compilers specifically hardened against malicious inputs [4] that, by design, apply only simple optimizations.

3 Position-Oblivious Code

Static machine code is linked to execute at a particular address. Position-independent code is code that can be executed regardless of the address where it was loaded. Typical LFI programs are position-independent. However, these programs may still determine the position at which they were loaded by reading addresses of functions, heap variables, or stack variables. This means that the load address of a position-independent program is a source of non-determinism: a position-independent program may behave differently depending on where it was loaded. Since all sandboxes are loaded in the same address space in LFI, this is a problem for supporting multiple deterministic sandboxes.

To solve this problem, we introduce *position-oblivious code* (POC): programs that cannot determine their load address. A position-oblivious program may be loaded at any address, and its result is guaranteed to have no dependence on that load address. A program can be verified to be position-oblivious by a static verifier before it is loaded.

Position-oblivious code works by building on the LFI technique. In LFI, every memory access is already preceded by a guard that forces the address to be in-bounds by inserting a constant into the top 32 bits of the address. Valid addresses are stored in reserved registers to ensure they are not tampered with. We take this further with position-oblivious code by statically verifying that absolute addresses are never directly observed. The verifier ensures that reserved registers, which may contain absolute addresses, are only ever accessed via their bottom 32 bits (i.e., `w30` instead of `x30`). Thus, the program may only directly observe offsets from the base address. Since the absolute sandbox base address can never be observed, the program is oblivious to where it was loaded.

For example, the following sequence is used to store the return address (`x30`) into the address offset stored in `x0`. This is usually performed as `str x30, [x0]`, but with LFI and position-oblivious code it becomes:

```
mov w22, w30
str x22, [x21, w0, uxtw]
```

As another example, moves from the stack pointer, such as `mov x0, sp`, are rewritten to read only the 32-bit subset of `sp`:

```
mov w0, wsp
```

Absolute addresses that contain the correct top 32 bits identifying the sandbox location are only stored in reserved registers that are verified to never be directly observed. Before accessing a memory location, the top 32 bits are always set to the correct value, and only the bottom 32 bits of a reserved register may be read.

The verifier also ensures that any instructions that produce an address dependent on the program counter (the `adr` and `adrp` instructions) place the result in a reserved register. If the application would like to use the result, it be required to read from the bottom 32 bits of the reserved register. For example the typical operation to load the address of a global via a PC-relative offset, `adr x0, foo`, would instead be expressed as:

```
adr x18, foo
mov w0, w18
```

These additional constraints primarily affect only three situations: storing the return address on the stack, loading the address of a global, and loading the value of the stack pointer to perform arbitrary computation. Compared to the frequency of loads/stores, these cases are relatively rare, and as a result the additional overhead imposed is minor.

When implementing a runtime with POC support, care must be taken so that the runtime does not reveal a sandbox's true addresses. For example, pointers returned from runtime calls must have their top 32 bits zeroed. Since POC programs behave the same no matter where they are loaded, they are compatible with a POSIX `fork` API, even though they all exist within the same hardware address space.

4 Deterministic Metering

By default, there are no constraints on the amount of CPU time that may be consumed by an LFI program. To prevent an LFI program from taking over the CPU, we use a notion of "metering," also known as "gas." The program continues running until there is no more gas in the meter, at which point it is preempted. A typical LFI runtime would implement this by using timer interrupts. However, this is not directly applicable for use-cases that

require determinism. When using timer interrupts, the particular instruction that will be interrupted next is not deterministic because it depends on factors such as the precision of the timer and the execution time of each instruction. If two replicas run the same program but use timers for preemption, one replica might halt the program before a side-effecting instruction (such as a runtime call), while the other might halt it afterwards, causing the replicas to see different results from the same program.

A deterministic alternative to timing is to keep an instruction count. By generating an interrupt after N instructions have executed, programs can be deterministically preempted. The performance monitoring unit (PMU) in Arm processors can be used to generate interrupts based on instruction counts. However, complete determinism is not guaranteed by the architecture. The architecture states that a "reasonable degree of inaccuracy in the counts is acceptable." Furthermore, the term *reasonable degree of inaccuracy* is explicitly left undefined, though the architecture does give guidelines stating that under normal operations the counters must be accurate. While individual microarchitectures may have deterministic instruction counts, this guarantee is not strong enough to rely on the PMU for deterministic preemption. In our testing, the PMU instruction counter cannot be used to deterministically generate signals within a Linux process running on Apple silicon.

Rather than rely on the PMU for deterministic metering, we propose an extension to LFI that manually tracks metering via additional instructions that must be included in a sandboxed program. At a high level, these extra instructions debit a gas counter register (`x23`) according to the number of instructions executed and use this gas counter to cause deterministic termination, either via an indirect branch into the LFI runtime or via preemption from a timer. We will refer to these two mechanisms as *branch-based* and *timer-based* metering respectively. While timer-based metering makes use of a non-deterministic timer, it is used in combination with the gas counter to make preemption deterministic.

Furthermore, since we are enforcing metering directly on machine code programs, the metering schemes are designed so that a static verifier can efficiently verify that the proper instructions are included in the program to allow for preemption. The verifier must be certain that there is no way to execute a branch instruction without also executing the immediately preceding metering instructions. This could happen if a previous branch or jump skipped over the metering instructions and directly targeted a branch. There are two mechanisms that we consider to prevent this scenario: aligned bundles and branch target identification (BTI). Since aligned bundles do not require any modern hardware features (unlike BTI), we primarily discuss and evaluate aligned bundles as the preferred approach. These schemes can be easily adapted to use BTI if it is available.

4.1 Aligned Bundles

The aligned bundles approach splits a program into bundles of N bytes, each starting at an address divisible by N. Padding is introduced into the binary so that every basic block begins at an aligned address. All direct branches in the program are verified to target the beginning of a bundle, and all indirect branches are forced by the verifier to target the beginning of an aligned bundle via the use of a reserved register (`x24`) that must contain a bundle-aligned address. The verifier will only accept indirect branches that target `x24`, and will only accept instructions that modify `x24` by zeroing the bottom $\log(N)$ bits from `x18` (which must contain a valid address). Since all branches must target the beginning of a bundle, it is impossible to execute an instruction at the end of a bundle without executing the instructions at the beginning.

For branch-based metering we use 16-byte (4-instruction) bundles, and for timer-based

metering we use 8-byte (2-instruction) bundles. Since ARM64 is a fixed-width instruction set, it is always desirable to make bundles as small as possible to reduce padding.

Aligned bundles are convenient because they don't require any special hardware support and allow for an efficient verifier. However, they introduce padding, and cause some runtime overhead for the additional alignment instruction needed for indirect branches.

4.2 Branch Target Identification

Another approach for control-flow enforcement is to use Arm's Branch Target Identification (BTI) extension. BTI requires that all indirect branches target a special `bti` instruction. If an indirect branch lands on a non-`bti` instruction, the processor traps. The BTI extension does not apply to return instructions, even though they are a type of indirect branch, since call stacks are meant to be validated with Arm's pointer authentication extension instead. However, since pointer authentication only provides a probabilistic guarantee of correctness, we instead rewrite all return instructions into indirect branches, and place `bti` instructions at return sites.

This approach has the benefit of not introducing alignment instructions for indirect branches or additional padding (except for additional `bti` instructions), but involves rewriting return instructions, which comes at a slight performance cost (we measured around 4%). This approach also requires hardware support for the BTI extension, which is available on Apple M2 processors, but not yet on most cloud servers.

A hybrid approach is also possible, where aligned bundles are still used, but only for the targets of direct branches, and BTI is used for indirect branches. This keeps direct branches easy to verify (just check the alignment of the target), and avoids having to use an alignment instruction for indirect branches.

4.3 Branch-based Metering

Branch-based metering keeps an instruction meter in a reserved register (`x23`). The verifier enforces that every basic block ends with instructions that decrease the meter by the number of instructions executed in the block, and check that the meter has not reached zero.

In general, detecting basic blocks in machine code is not possible because indirect branches may jump anywhere in the program. However, we do not need fully precise basic block analysis in order to provide metering, so long as any imprecision is conservative and deterministic. Consider a program without indirect branches; in this case all basic blocks are known statically, and we can charge the correct amount of gas at the end of each basic block. Now, if the program includes indirect branches, those branches may arrive somewhere within an existing basic block. The program will then be charged as if the entire basic block executed, even if it was only partially executed due to an indirect branch arriving in the middle. This may deterministically overcharge the program, but that is safe and only means the program terminates sooner.

To remove this imprecision, the program could split its basic blocks at indirect branch targets. Splitting a basic block into two is always legal, and allows the program to avoid overcharging itself even with indirect branches.

As shown in Figure 1, a 3-instruction metering epilogue must be inserted at the end of every basic block. This epilogue decreases the instruction meter and then checks if the top bit of the count has become 1. If so, the meter has underflowed and an indirect branch to the LFI runtime entrypoint (stored in reserved register `x25`) is executed. Due to the encoding of

```
.bundle_lock
sub x23, x23, #n
tbz x23, #63, .OK
blr x25
.OK: <branch>
.bundle_unlock
```

(a) Direct branches.

```
bic x24, x18, 0xf
.bundle_lock
sub x23, x23, #n
tbz x23, #63, .OK
blr x25
.OK: br x24
.bundle_unlock
```

(b) Indirect branches.

Figure 1 Instruction sequences for branch-based metering, where `n` denotes the number of instructions in the immediately preceding basic block.

the subtract instruction, it is impossible to both underflow and have bit 63 of `x23` equal to zero (the immediate cannot be large enough).

Indirect branches must use register `x24`, which is guaranteed to contain a valid address that is bundle-aligned. As a result, all branches are guaranteed to target the beginning of a bundle. To maintain `x24`'s invariant, the only legal way to modify `x24` is with the instruction `bic x24, x18, 0xf`, which transfers a value from `x18` (already guaranteed to be a valid sandbox address) to `x24` while also masking the bottom 4 bits to make the address bundle-aligned.

4.3.1 Gas check elision

The expensive part of the basic block epilogue is the branch instruction that checks for underflow. Fortunately, we can optimize away this check if the basic block does not end with a backwards branch. Consider a program of size N where a basic block that ends with a forwards branch exhausts all remaining gas, and no check is performed. Within at most N instructions, the program must either execute a backwards branch, causing a gas check, or terminate because no instructions remain. Thus, if the basic block exhausts all gas, the program will terminate after at most N further instructions.

In fact, the program must terminate within at most $N - I$ further instructions, where I is the address of the end of the basic block. Thus, if the size of N is a concern, the verifier could choose to only allow gas check elision for forwards branches that are close to the end of the program—i.e., ones with high values of I. Alternatively, the verifier could require at least one branching gas check to occur every M instruction for some $M \ll N$. However, our basic approach is just to cap the value of N at a reasonable and small amount such as 10M instructions (40 MiB of code) and elide all gas checks for forwards branches. An M2 processor can typically execute this many instructions in under 5 milliseconds.

4.4 Timer-based Metering

Timer-based metering uses a timer interrupt in combination with the gas counter to enable deterministic preemption. This metering scheme relies on the insight that a program running in a DeCl sandbox can only have externally visible effects when it makes a runtime call. As long as the program does not make any externally visible changes after it runs out of gas, it does not need to be immediately preempted. Thus, timer-based metering works as follows:

- A timer interrupt is configured to fire frequently.
- When the program makes a runtime call, it is terminated if its gas is negative.

```
.bundle_lock
sub x23, x23, #n
<branch>
.bundle_unlock
```

(a) Direct branches.

```
bic x24, x18, 0x7
.bundle_lock
sub x23, x23, #n
br x24
.bundle_unlock
```

(b) Indirect branches.

Figure 2 Instruction sequences for timer-based metering.

- When a timer interrupt occurs, the program is terminated if its gas is negative.

Since the runtime always checks the gas before applying the effects of a runtime call, it is impossible for a program to have any effect after it runs out of gas, even though it may continue running for a non-deterministic amount of time. To an external observer, the program behaves deterministically. Of course, this scheme assumes that a runtime call is the only way to cause an externally visible effect—if instead the memory state of the program were also externally visible, this would no longer work.

This scheme also assumes the time slice is short enough such that after running out of gas it is impossible to underflow the gas counter back to a positive number before the next timer interrupt occurs. This would require executing 2^{63} instructions before the next time slice, which would take roughly 29 years at a rate of 10 billion instructions per second, making this assumption safely satisfied.

Since gas checks are only performed after interrupts or runtime calls, basic block epilogues only need to update the gas counter, and can always omit the expensive gas check. This makes it possible to use a bundle size of 8.

Limitations

Timer-based metering relies on the ability to configure a timer interrupt, for example by using signals on Linux. This scheme also allows the program to use up to N seconds of CPU time after running out of gas, where N is the size of a time slice. While timer-based metering allows all gas checks to be elided, it can cause overhead if the time slice is too short. As a result, branch-based metering may still be better for systems that give programs only a small amount of gas, or that do not want to pay the cost of a trap when gas runs out.

5 Implementation

The implementation of DeCl consists of the standard set of LFI components: a program generator, a static verifier, and a runtime implementation. This section gives an overview of the concrete modifications to LFI needed for determinism, and how those modifications are implemented in the system components.

5.1 Modifications to LFI

Implementing deterministic, metering sandboxes using LFI requires some minor modifications, in addition to the position-oblivious code and metering discussed in previous sections.

Dedicated runtime call register

LFI reserves the first page of the sandbox to store metadata about the process, including the entrypoint for runtime calls. This allows the system to reuse `x21` as the address of this metadata page instead of reserving another register for this purpose. However, the contents of this page are not deterministic, so the sandbox cannot be allowed to read it. Thus, DeCl cannot use this page for metadata (DeCl simply leaves it unmapped) or the take advantage of the associated optimization, and instead stores the runtime call table outside of the sandbox, while reserving a a separate register (`x25`) to contain the address of the runtime call table page. The verifier enforces that this register is only ever used via the instruction `blr x25`, and is therefore never read by the program.

Redundant guard hoisting

Guard hoisting is an optimization used by default in the original implementation of LFI, which reserves two registers. To avoid reserving these two additional registers, we use a more limited form of this optimization, that simply eliminates redundant guards of the same address without any intervening modifications to `x18`.

Summary

DeCl reserves 6 registers in total: the sandbox base address (`x21`), gas counter (`x23`), runtime call table (`x25`), data addresses (`x18`), control addresses (`x24`), and a temporary register for use by various sandboxing sequences (`x22`). The return address register (`x30`) and stack pointer (`sp`) are also special registers that can only be modified and accessed via their bottom 32 bits.

5.2 Program Rewriter

Programs are compiled for DeCl using an assembly rewriter, which consumes the output of an optimizing compiler and inserts additional guards and other instructions. The new metering extension inserts sequences before branch instructions. Two create aligned bundles, we use the `.bundle` directives supported by Clang and GCC, including `.bundle_align_mode` to set the bundle size, and `.bundle_lock`/`.bundle_unlock` to force sequences of instructions into the same bundle. Our implementation in GCC requires a minor change to the GNU assembler to enabling bundling support on ARM64, which is disabled by default.

At assembly rewriting time the direction of a branch (forwards or backwards) is not necessarily known, since the branch may target a location in another compilation unit. As a result, we just insert a stub sequence of two/four instructions, which can apply to forwards or backwards branches. After linking, a "post-linker" is applied, which looks for these two/four-instruction sequences and replaces the stub with a proper `add`/`sub` instruction depending on the direction of the branch. Note the post-linker cannot have false positives, as the program rewriter leaves no uninstrumented branches. In addition, for forwards branches the post-processor removes the `tbz`/`brk` instructions that perform the gas check and replaces them with `nop`s.

5.3 Static Verifier

DeCl uses an additional static verifier on top of the existing LFI verifier. This additional verifier enforces that code is position-oblivious, by verifying that reserved registers are never

read directly (only via their bottom 32 bits), except when the target of the instruction is also a reserved register. Additionally, `adr`/`adrp` instructions must target a register meant for holding the addresses that they generate (i.e., a register whose top 32 bits cannot be read). This is implemented as a single pass over the machine code. The verifier must also restrict programs to a deterministic subset of ARM64, and must enforce metering, described below.

5.3.1 Enforcing Instruction Determinism

The programs accepted by the DeCl verifier must be guaranteed to execute deterministically. Therefore they must consist only of instructions known to be deterministic, which are part of a subset of the valid Armv8.0 instructions accepted by the existing LFI verifier. According to our analysis of the Arm reference manual and machine-readable specification, the following cases of non-determinism exist at the instruction level and must be excluded from our subset:

- Instructions encoded with malformed SBZ/SBO (should-be-zero/should-be-one) fields are CONSTRAINED UNPREDICTABLE.
- Some instructions have explicitly non-deterministic semantics, such as instruction pairs for atomic loads and stores. For example, the instruction `stxr` returns different values depending on the status of the exclusive monitor, which can change based on factors such as timer interrupts generated for the host kernel.
- Unallocated instructions and instruction encodings that are UNDEFINED are excluded.

According to our analysis of the Arm ISA, the floating point instructions are deterministic. This is corroborated by program fuzzing, which we have run on several Arm architectures to check for determinism: Apple silicon, Arm Neoverse, Ampere Altra, and QEMU. Our fuzzer generates multi-megabyte snapshots of deterministic instructions (random instructions that pass verification) without branches, and ensures the end-state of the program (memory and registers) is consistent across microarchitectures. We intend to expand the capabilities of the fuzzer in the future.

5.3.2 Enforcing Metering

The third part of the extended verifier checks for metering. In order to validate direct metering, the verifier must first determine the program's basic blocks. It does so using the typical linear-time algorithm for constructing leaders. A leader is one of:

1. The first instruction.
2. Any instruction following a branch.
3. Any instruction that is the target of a direct branch.
4. Any `bti` instruction.

A structure recording all leaders can be constructed with a linear pass over the machine code.

Next, the verifier iterates through all the leaders and enforces that the correct gas update sequence appears before the leader, or before the branch instruction if the prior basic block ends with a branch. For branch-based metering, the gas update sequence must consist of:

```
    sub x23, x23, #n  (d1nnn2f7)
    tbz x23, #63, end (b6f80057)
    blr x25           (d63f0320)
end:
```

The value `n` is calculated by the verifier based on the location of the current instruction and the previous leader, and is verified against the the immediate in the subtraction instruction provided by the program.

The verifier also tracks the locations of these instructions, and in a final pass ensures that no modifications to `x23` or `tbz x23` instructions occur outside these areas, and that branches do not target instructions within gas update sequences by making sure that branches correctly target aligned bundles.

Some "basic blocks" may in fact be unreachable—for example, the padding between the end of one function and the beginning of another. The assembly rewriter does not know about this padding, and thus does not insert gas update sequences there. However, at verification time, the verifier will expect this block to be metered. Our solution is to allow basic blocks to be merged: if the verifier sees a basic block that ends without a gas update sequence and has no terminating branch (it just falls through), then the verifier allows this and instead forces the count of the next basic block's gas update sequence to include these additional instructions. This means the first basic block of a function will charge for any additional padding between it and the previous function. This results in a very minor loss of precision. This also allows the verifier to transparently work with versions of direct metering of varying precision. A less precise implementation will merge more basic blocks, to gain some performance at the cost of precision.

Additional considerations

When using BTI, the verifier must also ensure that the program contains no `ret` instructions. Similarly, with aligned bundles `ret x24` is the only legal return instruction. The verifier also ensures that `x25` is never accessed, except via special runtime call sequences (e.g., `ldr x30, [x25]; blr x30`).

Complexity

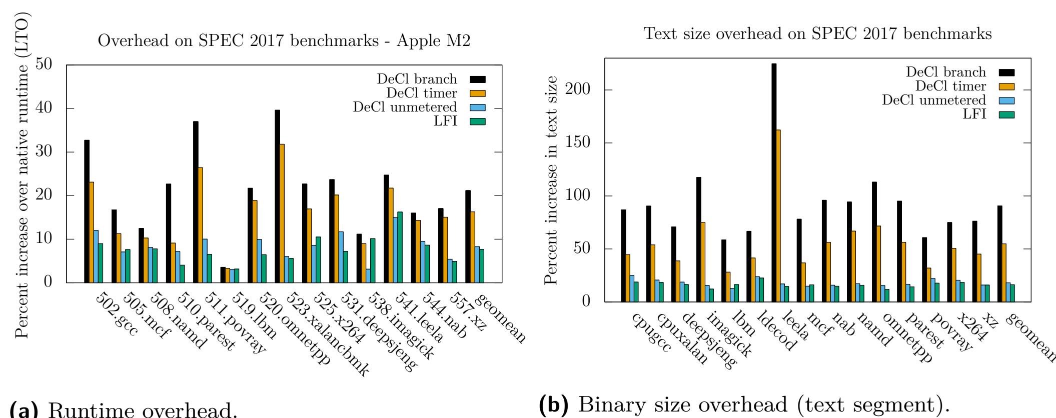
Our implementation is built using the Capstone disassembler [11] and consists of 358 lines of C. This is in addition to the existing LFI verifier, which is 323 lines of Rust. Thanks to its small size, the static verifier is significantly simpler than a JIT compiler (especially one with optimizations), and is also possibly simpler than an interpreter implementation, which must understand and correctly implement the semantics of all instructions. By contrast, our verifier only needs partial understanding of most instructions (e.g., is it a branch? does it modify a register?).

6 Evaluation

We evaluate DeCl on a Mac Mini M2 with 16GB of memory running Asahi Linux 6.6.0. In our evaluation, we measure runtime overheads caused by:

- DeCl unmetered: LFI extended with position-oblivious code (POC).
- DeCl: LFI extended with POC and metering.

We evaluate on the SPEC 2017 benchmark suite. Benchmarks are limited to ones that compile with LFI: they must be written in C or C++ and be compilable with a Musl/LLVM toolchain. We also use the SPECrate benchmarks rather than SPECspeed due to the 4GiB memory restriction imposed by LFI (SPECspeed requires 16GiB RAM per benchmark). This



(a) Runtime overhead.

(b) Binary size overhead (text segment).

Figure 3 Overheads of basic LFI, DeCl unmetered (position-oblivious code), and DeCl with metering.

restricts the suite to 14 benchmarks. All benchmarks are compiled with LLVM 16.0.6, with link-time optimization (LTO) enabled.

While the programs in SPEC are compiled to be deterministic, in order to run the benchmarks, the runtime must provide non-deterministic functions, such as system calls that return the time. Our runtime for running SPEC provides these non-deterministic functions for evaluation purposes. We are only running these SPEC programs with DeCl to quantify the runtime overhead of metering. A system that actually uses DeCl for full determinism would need to implement a custom runtime with a deterministic API of runtime calls. We evaluate such a system in the next section (§7).

Figure 3a shows the overheads of all three configurations of DeCl: with branch-based and timer-based metering, and unmetered. Table 1 provides a summary of the geomean overheads. The unmetered configuration performs very similarly to unmodified LFI (8.3% versus 7.7% overhead), since the only change is the addition of position-oblivious code, which in general only affects code that saves return addresses or loads globals. Timer-based metering is the most efficient form of metering, with 16.3% overhead, since it is able to omit all gas checks, but branch-based metering is not far behind at 21.2% overhead.

Figure 3b shows the percent increase in text size. This increase is primarily caused by the additional instructions that have been inserted, and to a lesser extent additional padding to enable bundling for the metered versions. This measurement only includes the text segment, so the overall binary size overhead is smaller. Geomean overheads are 16.3% (LFI), 18.1% (DeCl unmetered), 54.9% (DeCl timer), and 90.7% (DeCl branch).

Next, we compare DeCl with the following WebAssembly engines:

- Wasmtime Cranelift (fuel), version 24.0.0 [5]: a WebAssembly JIT compiler designed for running untrusted code, with support for some optimizations and deterministic metering, called fuel.
- `iwasm` interpreter, version 1.3.2 [6]: a WebAssembly interpreter developed as part of the WebAssembly Micro Runtime project. The version we run is unmetered.

We enable WebAssembly's 128-bit SIMD extension for Wasmtime, but not for `iwasm` since it is not supported in the interpreter.

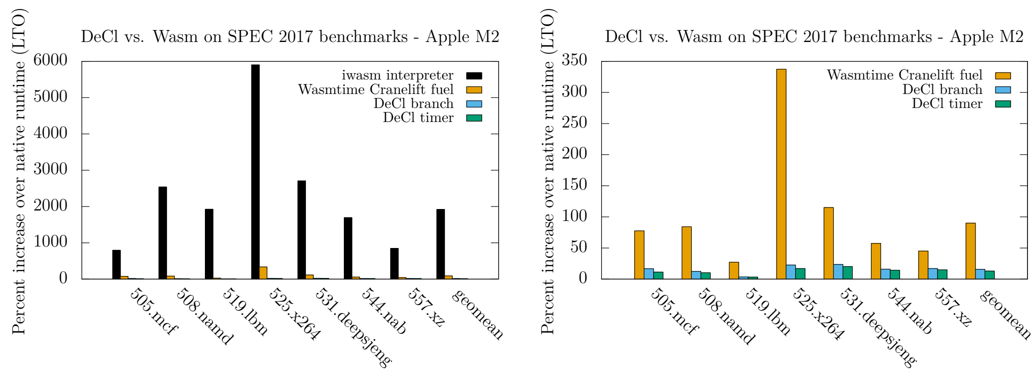
These results are shown in Figure 4. DeCl significantly outperforms Wasmtime, with fuel enabled, with over 5× lower overhead. As expected, both Wasmtime and DeCl are over an

System	Fig. 3a	Fig. 4
iwasm (interpreter)	-	1,920%
Wasmtime	-	90.0%
DeCl branch	21.2%	15.9%
DeCl timer	16.3%	12.9%
DeCl unmetered	8.3%	-
LFI	7.7%	-

Table 1 Summary of geomean overheads from Figure **??** (full set of supported benchmarks) and Figure 4 (only benchmarks supported by WebAssembly).

order of magnitude faster than the WebAssembly interpreter.

Geomean results from both sets of experiments are summarized in Table 1.



(a) `iwasm` included in the plot.

(b) `iwasm` not included in the plot.

Figure 4 Comparison between DeCl (position-oblivious and metered LFI) and various WebAssembly engines.

7 Integration with Groundhog

We integrate DeCl within the existing smart contract engine Groundhog [32], replacing the WebAssembly interpreter (wasm3 [21]) it previously used. This integration required only minimal changes to Groundhog, as both DeCl and the WebAssembly interpreter have approximately the same input-output behavior. Smart contracts interact with the blockchain's environment (for example, accessing persistent storage, or requesting metadata such as the current block number) via a set of specific functions; DeCl replaces imported WebAssembly functions with runtime calls.

7.1 Optimizing Startup and Teardown

A key challenge with the integration is that Groundhog is designed to scale over many CPU cores via concurrent execution of smart contracts. Maintaining this concurrent execution is possible with DeCl precisely because DeCl supports many separate sandboxes in the same process. However, maintaining the scalability requires care when setting up and tearing down sandboxes, so as to avoid contention on kernel resources. Many existing smart contracts are also very short programs. Therefore it is additionally imperative that the process for

launching and removing a sandbox be extremely fast, since the program might run for only a hundred microseconds or less.

In the existing LFI runtime, sandboxes are loaded from ELF files and mapped into the address space using `mmap`. The code and data segments are protected using `mprotect`. This approach is not suitable for Groundhog, since it has high startup overheads (multiple system calls), and is not scalable because the use of `mprotect` causes TLB shootdowns and possibly other cross-core synchronization within Linux.

We solve this problem by preallocating sandboxes with code and data regions. Programs are given 128KiB of code, with read-execute permission, and 128KiB of data, with read-write permission. These regions are mapped only the first time a sandbox is used, and subsequently can be reused for future sandboxes without the need for any system calls (only a `memset` to zero the memory).

When a sandbox is loaded, we must write to the code region (read-execute) to load its text segment. However, this region is not writable, and we cannot pay the cost of using `mprotect`. Instead, we create an in-memory 128KiB file using Linux's `memfd` API, and map it both at the sandbox's code region as read-execute, and also at a separate location within the runtime as read-write. This means the runtime can write to a sandbox's code region without needing to change any memory protections to do so. After writing to the aliased region, the runtime must execute an instruction cache flush on the executable region. This cache flush is performed in usermode by running the `ic ivau` instruction on each cache line in the range (performed automatically by the compiler's `__builtin___clear_cache` intrinsic).

Splitting a contract's code and data into two separate 128KiB regions requires a slight modification to the default linkerscript used by GNU LD. We have chosen sizes for these regions that are as small as possible, while still being usable, because these regions must be cleared after the sandbox terminates. The throughput of the `memset` operation to perform this clear can become a bottleneck if the regions become too large (such as 1MiB).

With these optimizations, the time to load, execute, and exit from an empty program is $30\mu s$, measured on the M2 processor.

Code caching

As a further optimization, it is possible to cache a sandbox's code and reuse it if the same contract is launched again. This is effective for commonly used contracts like popular ERC20-like tokens, and makes it possible to entirely skip the `memset` operation for the code region. At the moment, our system does not implement this optimization.

7.2 Native Cryptography Primitives

One major benefit of DeCl over WebAssembly sandboxes for practical smart contracts is that DeCl enables users to efficiently implement their own cryptographic primitives. Even basic cryptography, like verifying a signature, is sufficiently expensive that most blockchains provide runtime calls for common cryptographic operations to the smart contract environment. This allows the expensive operations to run at the speed of native code, bypassing sandbox overhead, but limits the operations available to smart contracts.

Ethereum [39] for example, implemented the BN254 curve [8, 28] within its smart contract environment [33, 10]. Improvements to the cryptanalysis [20] have given this curve less than 128-bit security, and users may wish to use a stronger cryptographic primitive, such as BLS12-381 [7, 9, 35]. Yet applications building on Ethereum have no option to change their cryptographic tools, precisely because deploying a new special-cased operation in an

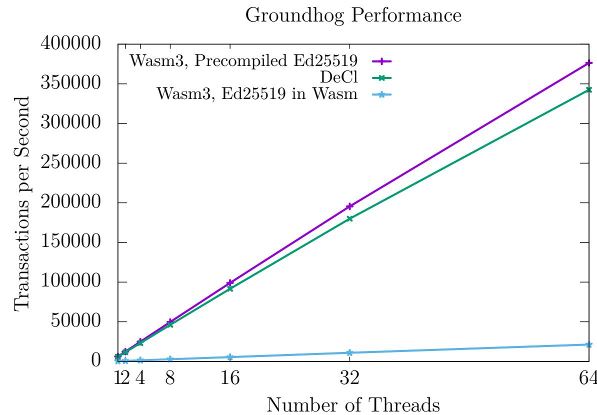


Figure 5 Groundhog throughput on varying numbers of threads, showing that DeCl does not impede scalability. DeCl is much faster than Wasm when both include Ed25519 in the smart contract, and is competitive with a version that exports the calculation to a native implementation provided by the runtime.

already-running blockchain requires a difficult, coordinated upgrade. By contrast, DeCl allows smart contracts to implement their own cryptographic functions inside the sandbox, and run fast enough that little-to-no performance is lost.

Figure 5 plots the throughput of Groundhog using various sandbox environments. Each transaction chooses uniformly at random two (of 10,000,000) accounts and sends a payment from one to the other. Throughput is measured on batches of size 100,000. Varying this workload (by varying the batch size, or varying contention by varying the number of accounts) gives similar performance trends on Groundhog when using either DeCl or WebAssembly with a precompiled Ed25519. These experiments are run on one c7gd.metal instance in an Amazon Web Services datacenter. The system has one 64-core Graviton 3 processor and 128 GB of memory, with no hyperthreading.

Each of these transactions verifies one Ed25519 signature, and microbenchmarks show that approximately 80 to 90% of each transaction's runtime is spent on that computation. Yet while the WebAssembly requires access to an Ed25519 implementation in native code in order to achieve competitive performance, DeCl has no such requirement—all of the cryptographic operations run within the sandbox. A smart contract developer can implement their own signature scheme or alternative cryptographic primitive on their own, and still achieve competitive performance. The small performance gap between DeCl and WebAssembly (with a precompiled Ed25519) comes from the cost of initializing DeCl sandboxes. Wasm interpreters initialize much faster.

These benchmarks use very short-lived smart contracts. DeCl also enables longer-running contracts that perform much more computation, since they will run at least an order of magnitude faster than in an interpreter-based system.

8 Related Work

Fast & Secure Sandboxing

Any replicated state machine that runs untrusted programs (i.e., any blockchain that uses a smart contract system) requires a sandbox environment that both runs deterministically and guarantees termination not just in finite time but after a deterministic number of program

instructions. Bitcoin [26] achieves deterministic termination, for example, with a scripting language without loops [3], while many other blockchains run more complex virtual machines like WebAssembly [16], eBPF [24], or the Ethereum Virtual Machine [39]. Our approach is a variant of Software Fault Isolation [42, 38], which instruments native code and verifies that the code cannot escape the sandbox. We build on top of the existing Lightweight Fault Isolation project [41].

These sandbox designs face three key challenges in practical systems, beyond the absolute requirements of deterministic, metered execution. First, they must run as fast as possible; any speed overhead reduces the overall throughput of the system, leading to higher fees for end-users. Executing smart contracts is a key throughput bottleneck in some blockchains today [17]. Techniques like optimistic concurrency control [15], speculative execution [13], or selective transaction (re)ordering [36, 34, 31, 23, 40] can provide throughput improvements on many workloads, but these approaches are complementary to making a faster sandbox.

Second, any sandbox must execute code securely; any possibility of reading or writing data outside of the sandbox can introduce nondeterminism, which can cause two replicas of a state machine (two blockchain nodes) to disagree on system state. But translating the bytecode of a virtual machine like WebAssembly into efficient machine code is a complex task. Security relies on the correctness of key elements of the toolchain, like the eBPF verifier or a WebAssembly compiler toolchain, which are complex software systems that have previously contained serious bugs [18, 1, 2]. Approaches include software bytecode interpreters, to minimize complexity (at the cost of runtime overhead), simplifying a just-in-time compiler [4], and formal verification of the compiler [37]. Our approach, by contrast, moves most of the code instrumentation and sandboxing into an untrusted code translation pass, leaving only a small portion of trusted code to verify sandbox security.

Finally, the sandboxes must startup quickly; typical smart contract use-cases invoke many separate sandboxes in sequence, and the overall system pays at runtime the cost any compilation steps that translate sandbox bytecode to machine code. Production blockchains rely on interpreters or just-in-time compilers [4]. Additionally, any compiler must be hardened against malicious input that could cause excessively slow compilation times, which limits the set of implementable program optimizations. By contrast, our system pushes all compiler optimizations out of the critical path and out of the trusted codebase.

Program Metering

A bytecode interpreter is easy to meter in the natural way, at the cost of high overhead. WebAssembly programs can be metered more efficiently by instrumenting a program to update a counter only once per basic block of the program, akin to the metering approach in §4.3. [25] implement an instruction counter in software, using a reserved counter and instrumenting only the backwards branches of a program. This is similar to our metering approaches, but does not track basic block sizes since their goal was only to uniquely identify every instruction with a pair of (address, counter). In addition, our setting is adversarial since it must be possible to enforce that programs correctly include and track the counter.

The deterministic metering in the sandbox runtimes should approximate wall-clock time. Mismatches introduce potential denial-of-service attacks, if adversarially exploited [30]. Good metering metrics require precise profiling of sandbox bytecode. While wall-clock times will always be nondeterministic on hardware, our system, by using native machine code, minimizes the difference in abstraction between the units of metering in the sandbox and execution time on hardware.

Determinism

Several applications are motivated by the need for determinism. One example is for efficiently distributing computation across many machines, as performed by `gg` [14]. Another example is the Exokernel file system, XN [19], which allowed applications to supply their own code for parsing file system data structures as untrusted deterministic functions or UDFs. Determinism is required to ensure that an inode always claims ownership of the same disk blocks—otherwise, a maliciously crafted inode could point to the correct blocks when initially written to disk, then later appear to own blocks belonging to a different file. DeCl provides a more efficient alternative to UDFs.

Record and replay systems, such as RR [29], can run unmodified programs in a deterministic way. These work by running a program and capturing the results of any non-deterministic operations. Then the program can be deterministically replayed. However, this setting is non-adversarial and does not provide bounded termination or careful isolation. These systems typically use `ptrace`, which can significantly hinder performance due to high system call overheads.

9 Conclusion

This work presents Deterministic Client (DeCl), a software sandboxing system that enforces that sandboxes are deterministic, metered, and run at near-native speeds. We explain how DeCl is built on top of Lightweight Fault Isolation (LFI) by providing two novel extensions: position-oblivious code, and deterministically metered native code. DeCl uses a machine code verifier that enforces these properties, which is implemented as a linear-time algorithm in less than 500 lines of code. Thanks to the machine code verifier, the compiler used to generate binaries is an untrusted component, allowing us to use LLVM without compromising security. This allows for applications to run inside DeCl extremely efficiently yet also securely. These properties make it possible for DeCl to run smart contracts that are written as ARM64 programs, making bare-metal smart contracts a reality. We demonstrate the feasibility of this approach by integrating DeCl into Groundhog, a scalable smart contract engine.

References

1 CVE-2021-32629. `https://www.cve.org/CVERecord?id=CVE-2021-32629`, May 2021.

2 CVE-2023-26489. `https://www.cve.org/CVERecord?id=CVE-2023-26489`, March 2023.

3 Bitcoin wiki: Script. `http://web.archive.org/web/20240401132513/https://en.bitcoin.it/wiki/Script`, 2024.

4 Cranelift. `http://web.archive.org/web/20240309013343/https://cranelift.dev/`, 2024.

5 Bytecode Alliance. Wasmtime, 2023. URL: `https://wasmtime.dev/`.

6 Bytecode Alliance. Webassembly micro runtime, 2023. URL: `https://bytecodealliance.github.io/wamr.dev/`.

7 Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks: Third International Conference, SCN 2002 Amalfi, Italy, September 11–13, 2002 Revised Papers 3*, pages 257–267. Springer, 2003.

8 Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International workshop on selected areas in cryptography*, pages 319–331. Springer, 2005.

9 Sean Bowe. Bls12-381: New zk-snark elliptic curve construction. `https://electriccoin.co/blog/new-snark-curve/`, mar 2017.

10 Vitalik Buterin and Christian Reitwiessner. Eip-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. Technical report, Ethereum Improvement Proposals, 2018.

11 Capstone. Capstone - the ultimate disassembler, 2024. URL: `https://www.capstone-engine.org/`.

12 Subra Chandramouli and Jamie Kinney. Expanding the tau vm family with arm-based processors, 2022. URL: `https://cloud.google.com/blog/products/compute/tau-t2a-is-first-compute-engine-vm-on-an-arm-chip`.

13 Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 570–587, 2021.

14 Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. URL: `http://www.usenix.org/conference/atc19/presentation/fouladi`.

15 Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.

16 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3062341.3062363`.

17 Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, and Roger Wattenhofer. Defi and nfts hinder blockchain scalability. In *International Conference on Financial Cryptography and Data Security*, pages 291–309. Springer, 2023.

18 Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 150–157, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3593856.3595892`.

19 M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *16th*, pages 52–65, Saint-Malo, France, October 1997. ACM.

20 Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Annual international cryptology conference*, pages 543–571. Springer, 2016.

21 Wasm3 Labs. Wasm3, 2024. URL: `https://github.com/wasm3/wasm3`.

22 Michael Larabel. Amazon graviton3 vs. intel xeon vs. amd epyc performance, 2022. URL: `https://www.phoronix.com/review/graviton3-amd-intel`.

23 Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. 2020.

24 Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

25 John M Mellor-Crummey and Thomas J LeBlanc. A software instruction counter. *ACM SIGARCH Computer Architecture News*, 17(2):78–86, 1989.

26 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

27 Paul Nash. Azure virtual machines with ampere altra arm–based processors—generally available, 2022. URL: `https://azure.microsoft.com/en-us/blog/azure-virtual-machines-with-ampere-altra-arm-based-processors-generally-available/`.

28 Yasuyuki Nogami, Masataka Akane, Yumi Sakemi, Hidehiro Kato, and Yoshitaka Morikawa. Integer variable χ–based ate pairing. In *International Conference on Pairing-Based Cryptography*, pages 178–191. Springer, 2008.

29 Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan`.

30 Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in evm. *arXiv preprint arXiv:1909.07220*, 2019.

31 Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

32 Geoffrey Ramseyer and David Mazières. Groundhog: Linearly-scalable smart contracting via commutative transaction semantics. *arXiv preprint arXiv:2404.03201*, 2024.

33 Christian Reitwiessner. Eip-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. Technical report, Ethereum Improvement Proposals, 2018.

34 Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 543–557, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3318464.3389693`.

35 Yumi Sakemi, Tetsutaro Kobayashi, Tsunekazu Saito, and Riad S. Wahby. Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-11, Internet Engineering Task Force, November 2022. Work in Progress. URL: `https://datatracker.ietf.org/doc/draft-irtf-cfrg-pairing-friendly-curves/11/`.

36 Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 1–12, 2012.

37 Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. Lightweight, modular verification for webassembly-to-native instruction selection. ASPLOS, 2024.

38 Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Andrew P. Black and Barbara Liskov, editors, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 203–216. ACM, 1993. `doi:10.1145/168619.168635`.

39 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

40 Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: a lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.

41 Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 649–665, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3620665.3640408`.

42 Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.