Metering the Meter, or How to Efficiently and Deterministically Charge the Execution of Smart Contracts

GEORGE MITENKOV, ETH Zürich, Switzerland

Blockchain systems are rapidly evolving and, fueled by novel use cases and the competition with traditional financial systems, have to be not only secure and decentralized but also performant: processing thousands of transactions per second and having sub-second latency. Under such demanding conditions, the cost models, which are used by blockchains to mitigate attacks on the network and to align the economic incentives between users and node operators, become even more important. On one hand, a cost model has to be a deterministic over-approximation of the real costs of smart contract execution to ensure that even malicious users pay their fair share of the costs. On the other hand, cost models have to be efficiently implementable, so that excessive cost metering during execution does not become itself a burden.

Unfortunately, existing solutions do not always satisfy these properties and are not designed with scalability and performance in mind. In particular, this work shows that naive implementations of cost models and their metering schemes hinder possible performance benefits from techniques such as JIT compilation, while still leaving the blockchain systems open for attacks.

To overcome these limitations, we propose a new approach to the design of cost models and their implementation. While traditionally cost models are predominantly based on resource consumption in the network, the goal of our approach is to additionally align the incentives and requirements for users, developers, and node operators. In particular, we focus on one of such requirements – fast and efficient metering of instruction costs in smart contracts. Inspired by the problems of optimal profiling and code coverage, we define the problem of minimum metering instrumentation. The goal is to identify a minimal set of instrumentation points in a program so that it is possible to calculate the sum of costs of all executed instructions and ensure that the running sum of costs is within a fixed limit While the problem is hard in general, we present an algorithm that solve the problem under certain conditions.

Our evaluation shows that the proposed algorithm only needs to instrument 30.4-37.5% of the basic blocks of popular smart contracts, on average, and can also yield more than 2× run time improvements on selected benchmarks compared to the state-of-the-art approaches.

1 INTRODUCTION

In recent years, with the growing demand for decentralized platforms, numerous blockchain systems have emerged. Abstractly, a blockchain is a state machine replication system, which allows a set of participating nodes (validators) to agree on a sequence of blocks of transactions submitted by other nodes on the network (clients). The blockchain maintains its state and records the history of all executed transactions, each carrying a smart contract bytecode. While traditionally being designed as distributed ledgers supporting only simple peer-to-peer transfers of a digital currency, modern blockchains have a significantly richer spectrum of (web3) applications, thanks to smart contracts.

Blockchains can be used in many domains where sensitive information has to be owned, managed, and transferred, with healthcare being a great example. Storing health records on the blockchain allows patients to protect their data and privacy, as well as to grant access to doctors or organizations as desired [64]. Moreover, blockchains are already being used for fraud prevention, making use of the immutability of transaction history, including but not limited to examples of real-estate marketplaces [74], data management systems, etc.

One of the most influential aspects of blockchains is the ability to create an economic value based on the scarcity of resources. The most obvious example is Non-Fungible Tokens (NFTs), which allow one to create, own, and transfer digital art, gaming, or photography pieces using dedicated marketplaces such as OpenSea.[1]

[1]https://opensea.io

Blockchains are also reshaping the financial industry. Exchanges, centralized such as Binance or Coinbase, or decentralized like Uniswap,[2] allow users worldwide to buy and sell different digital currencies, empowering global e-commerce. Digital currencies, in turn, create new opportunities for traders and market makers.

Even though there are many use cases, there is no wide adoption of blockchains amongst the general public today. While blockchain-based Ponzi schemes and collapses of crypto-industry giants like FTX [70] definitely played their part here, one of the main reasons that hinders mass adoption is simply the low throughput of blockchains and high fees.

For example, at the time of writing, Bitcoin [67] and Ethereum [79] – the two most prominent blockchains, can process 7 and 20-30 transactions per second (TPS), respectively. In comparison, Visa is capable of executing more than 65k TPS. Such drastic differences in throughput impede the usability and the adoption of blockchains.

Low throughput has another implication which affects the popularity of blockchains and makes it hard to replace traditional payment systems like Visa or MasterCard. Executing a transaction on a blockchain is usually not free. A fee is needed so that clients pay for the computational resources they utilize, e.g., to reward validators for the provided service (so they can cover their costs), as well as to protect the network from Denial-of-Service (DoS) attacks. Because the throughput is low, transaction fees go up, e.g., a peer-to-peer transfer on Ethereum can cost up to $10-20 during peak times.[3]

Transaction fees are crucial for blockchains. They are calculated based on a cost model – a mathematical abstraction that regulates how the network resources are priced. For example, most state-of-the-art cost models assign costs to bytecode instructions.

Because it is impossible to capture the true complexity of a blockchain system, the cost models are only approximations. Ideally, a well-designed cost model is a simple and tight over-approximation, which protects it from malicious users and DoS attacks.

A cost model has to be implemented in the blockchain protocol to record the used resources and ensure they are paid for. This process is called metering. Metering must be efficient, otherwise, the system may end up using significant computational resources for the metering process itself.

Often, cost models associate bytecode instructions with costs, which, as of today, contribute the most to the execution costs on popular blockchain networks. The costs are metered and checked at run time to ensure the client who submitted the transaction has enough funds to complete it.

Blockchains use dedicated virtual machines to execute smart contracts with built-in instruction cost metering support. Recently, however, more and more blockchains started transitioning to existing state-of-the-art runtimes, e.g., WebAssembly [48], in order to benefit from existing optimizations, efficient execution pipelines, and large open-source communities.

In these cases, the contract code is usually instrumented with additional instructions, which meter instruction costs. This technique has been particularly appealing because it allows one to JIT compile contracts to native code and to use existing compilers and runtimes without modification.

While there have been many studies on how to scale different parts of blockchain systems and make them more secure, there has been less attention devoted to studying how cost models and metering affect scalability and security. In this work, we study the existing cost models used by a few popular blockchains and how they are implemented, mainly focusing on instruction cost metering.

We use the following set of research questions to guide our study:

- How do the state-of-the-art cost models calculate the cost to execute a smart contract?

[2]https://uniswap.org
[3]https://etherscan.io/chart/avg-txfee-usd

- How are the cost models implemented in existing virtual machines?
- How do cost models and their implementations impact contract execution performance? Also, do they pose a security risk to the blockchain?
- How do the most popular contracts look like? What are their main characteristics, and how do they impact the efficiency of metering?

Guided by these questions, we observe that the existing cost models are not always secure and are also not always efficiently implemented, as we describe in a detailed study in Sections 3 and 4. In particular, we find that the flaws in the existing implementations can lead to smaller profits for validators, incorrect execution behaviour, and even potential attacks on the network. We also discover that instruction cost metering can cause significant performance degradation, sometimes more than 2× compared to unmetered executions.

Another important finding of this work is that smart contracts across multiple chains, even though different, have very similar execution footprints, as we discuss in Section 5. In particular, we find that only a very small fraction of contracts are not executed successfully until the end and an even smaller fraction of contracts run into limits enforced by the cost model. Hence, the existing approaches to meter instruction costs can be significantly improved by optimizing for the common case.

Motivated by the need for safe and efficient ways to meter the contract's execution cost, we formulate the problem of minimum metering instrumentation. The goal of the problem is to instrument the least number of basic blocks in a program for any execution path so that the program can be metered safely and deterministically.

In Section 6, we formally define a set of properties any algorithm should satisfy to solve the problem of minimum metering instrumentation. Our main result is a general bytecode-agnostic algorithm that minimizes the amount of instrumentation for any path in a program while ensuring at run time that there is always enough budget for the execution. Compared to the state-of-the-art approaches for placing the metering instrumentation, our solution is more efficient, yielding up to 2× run time speedup on a selection of microbenchmarks. Moreover, evaluating the proposed algorithm on popular real-world contracts, we identified that on average it is sufficient to instrument only 30.4-37.5% of the basic blocks.

We also show that if the constraints on the metering instrumentation are relaxed, e.g., it is allowed to execute instructions before they are metered, the instrumentation and its overhead can be further reduced. However, the problem becomes hard and does not admit a straightforward solution.

Finally, this work also revisits how to design cost models for blockchains. In Section 8, we present the Cost Model Standard, which, unlike the existing cost models, captures the relationship between clients, smart contract developers, smart contract compilers, and validators. We show that such a design aligns the incentives for the different stakeholders in the blockchain system so that the cost model can be implemented more efficiently and securely. Additionally, more optimal algorithms for the metering instrumentation placement can be used.

2 BACKGROUND

2.1 Blockchains and smart contracts

A blockchain is a distributed state machine that consists of validators and clients. At a high level, validators agree on a block of transactions and execute them, thereby changing the global state of the blockchain and progressing the network forward. Clients, in turn, submit transactions to validators (or intermediate nodes).

All validators maintain the current global state of the blockchain and, typically, a transaction history that records all transactions executed on the blockchain since genesis (the initial state). Validators batch client transactions in blocks and run a consensus protocol to agree on which transactions should be executed. Once agreement on the block content is reached, all transactions in the block are executed in an isolated virtual machine (VM). Executed transactions (also referred to as committed) are appended to the transaction history, and the changes made by successful transactions are applied to the global state.

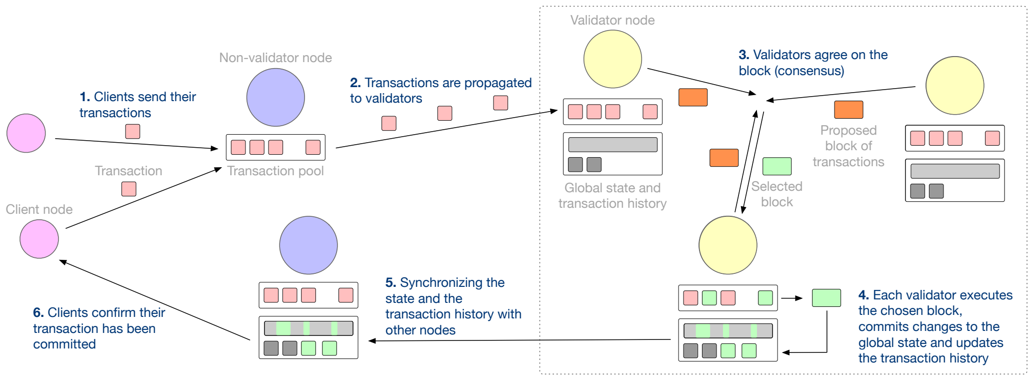

Validator node
Non-validator node
1. Clients send their transactions
2. Transactions are propagated to validators
3. Validators agree on the block (consensus)
Transaction
Transaction pool
Client node
Global state and transaction history
Proposed block of transactions
Selected block
4. Each validator executes the chosen block, commits changes to the global state and updates the transaction history
5. Synchronizing the state and the transaction history with other nodes
6. Clients confirm their transaction has been committed

Fig. 1. Step-by-step interaction between clients and validators.

Clients interact with the blockchain using smart contracts. Smart contracts are usually implemented in domain-specific languages such as Solidity [3], Vyper [8], or Move [34]. However, some blockchains also allow subsets of general-purpose programming languages such as Haskell, JavaScript, Rust, or C++.

Smart contracts are stored on-chain as bytecode, which ensures portability and allows contracts to be executed on any platform irrespective of the hardware vendor. The bytecode used depends on the blockchain or the smart contract language.

Developers can deploy contracts to the blockchain by submitting a transaction with the contract code. Once the contract is deployed, it can be called by any other client (again, via a transaction) or by any other contract. If execution is successful, the state of the blockchain is updated. Otherwise, all changes are discarded. Sometimes, smart contracts can be upgraded by uploading a newer version of backward-compatible bytecode.

Every transaction on a blockchain is associated with a fee the client is willing to pay for the transaction to be executed and committed to the global state. [4] The goal of the fee is to pay for the costs of storing the smart contract code and the data in the global state, and for executing the program.

[4]This is true for most blockchains, but there are some notable exceptions, e.g., Dfinity [49] where the contract developer pays for execution instead.

A)
```
module hello_world {
  use std::{error, signer, string::String};

  // 'key' means this struct is stored on-chain.
  struct Greeting has key {
    value: String,
  }

  // Error codes.
  const ENO_GREETING: u64 = 0;
  const EGREETING_ALREADY_EXISTS: u64 = 0;

  // Returns the greeting stored on-chain.
  #[view]
  public fun get_greeting(
    addr: address,
  ): String acquires Greeting {
    assert!(
      exists<Greeting>(addr),
      error::not_found(ENO_MESSAGE),
    );
    borrow_global<Greeting>(addr).value
  }

  // Set the greeting on-chain.
  public entry fun set_greeting(
    account: signer,
    value: String,
  ) acquires Greeting {
    let addr = signer::address_of(&account);
    assert!(
      !exists <Greeting>(addr),
      error::already_exists(EGREETING_ALREADY_EXISTS),
    );
    move_to(&account, Greeting { value }
  }
}
```

B)
```
import {
  NearBindgen, near, call, view
} from 'near-sdk-js';

// Hello World contract.
@NearBindgen({})
class HelloWorld {
greeting: string = "Hello World!";

  // Return the greeting stored on-chain.
  @view({})
  get_greeting(): string {
    return this.greeting;
  }

  // Set the greeting on-chain.
  @call({})
  set_greeting({ greeting }: { greeting: string }): void {
    this.greeting = greeting;
  }
}
```

C)
```
// Hello World contract.
contract HelloWorld {
  string greeting = "Hello World!";

  // Return the greeting stored on-chain.
  function get_greeting() public view returns (string memory) {
    return greeting;
  }

  // Set the greeting on-chain.
  function set_greeting(string memory input) public {
    greeting = input;
  }
}
```

Fig. 2. Examples of simple smart contracts written for A) Aptos, B) NEAR, and C) Ethereum in Move, JavaScript, and Solidity respectively. The contract allows to store a greeting message on the blockchain and view it.

Validators, in turn, can receive a fraction of the fee to account for the computational resources they used while processing the transaction, e.g., CPU, storage, or network. Similarly, some blockchains distribute a portion of the fees to contract developers [77].

To ensure a predictable maximum latency for executing a transaction, and to protect the network from Denial-of-Service (DoS) attacks, the computational resources used by a transaction are usually bounded. One way to achieve this is to impose hard limits on used resources, e.g., the maximum amount of memory a contract can use or the maximum number of bytecode instructions that can be executed. Alternatively, resource usage can be part of the cost model and included in the transaction fee. In this case, the execution must be halted as soon as the fee the client is willing to pay for the transaction (and has funds for) is not large enough to cover the costs for the used resources.

In this work, we consider four popular blockchains, namely Ethereum [79], Solana [81], NEAR [77], and Aptos [75]. We note that the blockchain space has numerous other different networks. However, we believe that considering these four chains is sufficient to understand the differences and trade-offs of state-of-the-art blockchain execution models, their relationship with the cost models, and resource metering schemes. We briefly summarize the selected chains in Table 1.

In the rest of the section, we describe how blockchains work in detail, focusing predominantly on the execution layer. Throughout, we use examples from real networks, showing how they are organized, how their execution layer is structured, and what runtimes they use to run smart contracts. We discuss the design trade-offs and compare networks with each other.

2.2 Global state of blockchains

All blockchains keep track of some state, which may change every time a transaction is executed. Usually, blockchains use an account model for that, where the state is a map between addresses (unique identifiers) and accounts, which can be mutated by transactions. Depending

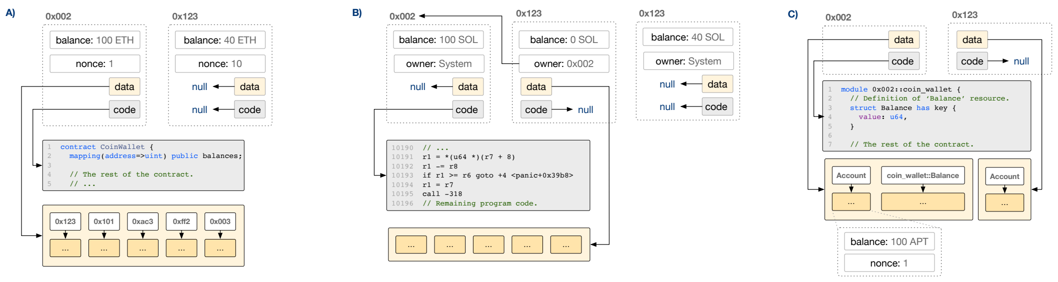
Table 1. A brief description of blockchains that are selected for this work.

Blockchain	Launch date	About the chain
Ethereum	2015	The first blockchain to introduce the concepts of smart contracts and gas. Many modern blockchains, so-called L2 networks, operate on top of Ethereum to improve its scalability and efficiency. It is the most well-studied chain in the academic literature.
Solana	2020	Aims to solve the scalability and performance issues of Ethereum. Supports Rust and C/C++ contracts, and leverages existing infrastructures such as LLVM [55] and eBPF [62]. The first blockchain to introduce parallel execution of transactions, thereby significantly reducing the transaction fees.
Near	2020	Also launched in 2020, NEAR is an important blockchain to study in our work because it stores smart contracts (originally written in JavaScript or Rust) on-chain in WebAssembly (Wasm) [48] and uses existing Wasm compilers and runtimes. Nowadays, Wasm is very popular in the blockchain community, being also used by networks like Polkadot [80] and will also potentially be used by a future version of Ethereum [31].
Aptos	2022	Originates from the Diem blockchain [17] and, like Solana, aims for scalability, performance, and security. Uses the Move language, designed for blockchains from first principles, with built-in formal verification, improving security and user experience. Also supports parallel processing of transactions.

on the blockchain, accounts can store different data as shown in Figure 3. Accounts are typically owned by the users, the system, or smart contracts.

2.2.1 Accounts on Ethereum. Ethereum accounts can be of two types: 1) externally-owned accounts (EOAs), which do not store any code, and 2) contract accounts, which are owned by a smart contract and store the code and the associated data. Additionally, they keep track of a balance in ETH (native token).

2.2.2 Accounts on Solana. Accounts on Solana differ significantly from Ethereum. Solana accounts can be either data accounts or contract accounts. Moreover, every account on Solana is owned by a contract account, either created by a client or owned by the system itself. The direct implication of this is that contract accounts are stateless, and it is the data accounts that store the global state.



```
contract CoinWallet {
  mapping(address=>uint) public balances;

  // The rest of the contract.
  // ...
```

```
10190  // ...
10191  r1 = *(u64 *)(r7 + 8)
10192  r1 -= r8
10193  if r1 >= r6 goto +4 <panic+0x39b8>
10194  r1 = r7
10195  call -318
10196  // Remaining program code.
```

```
module 0x002::coin_wallet {
  // Definition of 'Balance' resource.
  struct Balance has key {
    value: u64,
  }

  // The rest of the contract.
```

Fig. 3. Organization of global state on A) Ethereum, B) Solana, and C) Aptos blockchains.

2.2.3 Accounts on NEAR. NEAR, similarly to Ethereum, stores both code and data under the same account. The difference is that NEAR uses Wasm to store contract code on-chain. Wasm contracts are organized in modules, which contain functions and can be called either locally or from other modules (resulting in cross-contract calls).

2.2.4 Accounts on Aptos. Aptos uses a model similar to Solana, decoupling data from code. However, unlike Solana, Aptos allows the users to store the code of contracts they develop (modules) and the data they own (resources) under the same account but in different namespaces.

Modules store bytecode and can be identified in the global state by the address of the account they belong to and the module name. The code in modules is organized into functions. Like in many mainstream programming languages, functions have different visibility, ranging from public to private, and can be called from other on-chain modules similar to general-purpose programming languages.

Resources are the data accessible from the global state, with semantics inspired by linear logic [46]. This way, resources (e.g., tokens or other digital assets) cannot be copied or dropped implicitly, only moved between different accounts.

The type of a resource (i.e., the Move type in the original smart contract) together with the account address identifies a resource in the global state. It is worth mentioning that generic instantiations are considered to be different types.

2.3 Blockchain fee system and gas

To execute a transaction on a blockchain, the users must pay a fee. Usually, blockchains use the concept of a fee market in which clients compete with each other offering different fees for their transactions. The higher the fee is, the more likely the transaction will be executed and committed on-chain. If the fee market does not exist, the cost of transactions is said to be fixed.

The fee is usually tied to the amount of resources used when a transaction is executed, but there are exceptions. Solana is an example of a blockchain that uses a fixed fee only proportional to the number of signatures (used to verify that the sender has authorized the transaction) in a transaction. [5] This means that the amount of resources used, such as the number of executed bytecode instructions, is not taken into account when computing the fee. Instead, the runtime of Solana defines a compute budget – a small cap on the amount of resources that transactions can use and their size.

If the fees are not fixed, their calculation usually takes into account how many resources a transaction uses during its execution. If this is the case, the utilization of resources is typically measured in gas – a fundamental unit of computation that accounts for the computational work done by the contract and the virtual machine used to execute it.

The concept of gas was first introduced by Ethereum, and many other blockchains also use it. We describe gas in detail later in this section, however, for now, it is sufficient to understand that the amount of gas consumed by a contract is 1) roughly proportional to the computational resources used, e.g., CPU, memory, etc., and 2) deterministic, i.e., do not depend on a particular hardware platform used by validators or wall/CPU time. We also note that Solana uses a similar concept to gas called computational units, although it is not included in the fee at the time of writing and is only used to cap the resource usage.

[5]We note that while the fee is constant in the number of signatures, the cost per signature varies. In addition, there have been ongoing discussions on whether there is a need to account for the computational resources used by transactions when calculating the fee, similar to what Ethereum and other gas-based chains are doing.

Table 2. Calculation and distribution of transaction fees on Ethereum, Solana, NEAR and Aptos.

Blockchain	Fee distribution	Fee calculation
Solana	50% distributed to validators, 50% are burnt.	numSignatures × pricePerSignature
Ethereum (< 2019)	100% distributed to validators. [6]	gasUsed × gasPrice
Aptos	100% burnt. [7]	
NEAR	30% distributed to contracts, 70% distributed to validators.	
Ethereum (≥ 2019)	The base fee is burnt, and the tip is given to a validator.	gasUsed × (basePrice + tipPrice)

Gas usually has a dynamic price in a native blockchain token, which depends on the current load of the network, the demand, or some other factors. Clients then compete on the fee market by offering higher prices for a unit of gas, increasing the chances of getting their transaction included in the block.

On Aptos and an earlier version of Ethereum (before deployment of EIP-1559 [37] in 2019), a simple first-price auction is used to guide the transaction fee mechanism. First, a client specifies the gas limit and the gas price when submitting a transaction. The product of the gas limit and the gas price is the maximum fee the client is willing to pay. When the transaction is completed, the amount of gas used is known, and the final fee is obtained by multiplying the gas price by the gas used.

For Ethereum, this mechanism did not work well which resulted in transitioning to a new transaction fee model that works as follows. The fee is split into 1) a base fee, which is a fixed amount determined by the system and which is always burnt, and 2) a tip that is given to a validator that incentivizes including this transaction in the block. When the fee is burnt, it simply disappears from the system decreasing the total supply of a native token. The tip is specified by the client. While the new model helped to reduce transaction waiting times, the new model did not help with gas fees reduction [59].

Here, NEAR stands out from the other blockchains because it returns 30% of transaction fees to contract developers (i.e., to the accounts where the executed contract is stored). With this, NEAR incentivizes developers to write contracts which will attract more users to the network.

We observe that for validators the best scenario is to maximize the fees they receive while minimizing the usage of computational resources. Hence, if two contracts have identical gas costs, rational validators prefer the contract with the lowest resource consumption to maximize their profit. Indeed, lower resource consumption most likely means that the contract can run faster, which allows the validator to execute more transactions and increase the revenue. Alternatively, lower resource consumption decreases expenses, e.g., less network bandwidth is used, increasing the validator's profit.

[6]Strictly speaking, they were miners instead of validators because the pre-2019 versions of Ethereum were using a proof-of-work consensus protocol.

[7]Based on discussions with developers, Aptos will also distribute a fraction of fees to validators when there is more traffic on the network.

2.4 Transaction validation and execution

A transaction submitted by a client is not executed straight away. It first ends up in a transaction pool – a priority queue of pending transactions. The priority of a transaction is influenced by different factors, e.g., the gas price.

Transactions are validated while in the pool. Validation consists of multiple steps, including but not limited to checking the validity of signatures, whether the transaction has already been executed, or if the transaction's sender has sufficient resources to pay the transaction fee (i.e., maximum transaction fee if it is not known upfront). If validation fails, the transaction is rejected and evicted from the pool.

Eventually, transactions from the pool are forwarded to validators. Validators almost always batch transactions together into blocks. The size of blocks can range from hundreds to thousands of transactions. Operating on blocks instead of single transactions allows validators to agree, execute, and commit at higher throughput, minimizing communication overhead. At the execution layer, the execution of a block is simply an execution of an ordered sequence of transactions. Note that the ordering is fixed according to a consensus protocol.

We now describe how a single transaction from a block is executed by a validator. Before execution, transactions are re-validated. Re-validation typically follows the same procedure as in the transaction pool.

After re-validation, the transaction gets executed using a virtual machine (VM). [8] The VM is initialized with a maximum computational budget. On Ethereum, NEAR, and Aptos, this budget is the maximum amount of gas the contract can use. Additionally, or instead, this budget can enforce bounds on computational resources, such as memory usage or the number of executed bytecode instructions. For example, Solana uses this approach.

Usually, the client specifies which function to call from the contract and with which arguments. The VM then runs the selected function while ensuring the used computational resources are within the budget. The process of ensuring the budget is not exhausted is called *metering*.

During execution, the changes the transaction makes to the global state are not applied immediately and instead are kept separately. If the transaction terminates successfully without exceptions, the changes are applied to the global state. [9] Conversely, contract execution can fail (e.g., not enough gas to finish execution, or an exception is thrown because of a division by zero), in which case the execution is aborted, and the changes made to the state are discarded. Note that failed transactions are still recorded in the transaction history, and the fees are also charged.

2.5 Cost models for blockchains

As already mentioned, all contract executions on blockchains are metered in order to mitigate DoS attacks and to ensure that clients pay for the computational resources they use. What exactly is metered is determined by the cost model.

The goal of a cost model in blockchains is to *approximate* the real cost of executing a contract on validators' hardware under the current protocol implementation. Most importantly, the cost has to be deterministic on any platform and using any protocol client. This is crucial because costs are usually included in the transaction history ledger (e.g., amounts of used gas in Ethereum) and are part of transaction outputs validators agree on when running a consensus protocol. If validators were to compute even slightly different costs for the same transaction, consensus would fail.

[8]The VM can also be used for validation. For example, Aptos uses a smart contract to do certain validation checks, such as checking the account balance of the sender.

[9]Strictly speaking, the changes are applied to the sub-state maintained for a given block. Only when all transactions in the block are executed, these changes are applied to the global state, i.e., persisted into the actual storage backend used by the protocol.

In general, cost models over-approximate the cost to prevent malicious clients from attacking the network. At the same time, the cost model should be as close as possible to the real cost. If not, clients are overcharged and are economically motivated to use a different blockchain that is more cost-effective.

It is also important that the cost model is efficiently implementable. If not, then measuring the cost of the resource consumption can become expensive. This means that clients would be undercharged and validators would not profit from execution anymore, not getting paid enough for the resources they use. It is worth mentioning that cost models can accept under-approximations in case the accurate cost is too difficult or too costly to measure.

2.6 Cost model implementation and metering schemes in blockchains

In essence, metering is an implementation of the cost model for the protocol. There are two important kinds of metering: 1) metering to charge for the usage of the validator's computational resources, which we call gas metering, and 2) metering to ensure the usage of computational resources does not exceed some pre-defined limits but does not charge for the usage, also referred to as resource metering. If gas or resource limits are exceeded, the execution must be halted. What are the limits, and what are the costs is determined by the cost model.

Most of the blockchains opt for gas metering schemes: all actions performed by a contract are associated with a gas cost. Using gas is a great example where the cost model simplifies the system by defining a common unit of resource consumption. The costs are typically associated with the bytecode instructions, e.g., executing an addition instruction on Ethereum costs 3 units of gas.

However, the implementation details of VMs sometimes also leak into the cost model and as a result, are included in the gas cost. For example, the VM used by Aptos has a gas cost for generic type instantiations as a protection against DoS attacks and deeply-nested generic types. This is because the implementation uses a recursive traversal over the type, which can be slow and could be used by a malicious user to slow down the network. The metering scheme should also account for costs like these.

We consider resource metering to be a special case of metering because there the exact consumption over time is not important. For example, if there is a limit on the maximum allowed memory the contract can use, it does not matter how much memory was used at run time, as long as it did not go beyond the limit.

For example, Solana has an upper limit on the number of bytecode instructions a contract can execute. In this case, the VM may not have to track the number of executed instructions at any given point in time. Instead, it might be sufficient to check the computational budget occasionally. [10] For gas metering, precision becomes crucial because the amount of used gas determines how much clients pay for their transactions.

Metering and the limits on computational resources are very tightly coupled with the cost model. In Section 3, we describe the cost models used by the state-of-the-art blockchains, and compare the costs and limits. In Section 4, we describe how gas and resource metering is implemented, also discussing trade-offs and possible problems of existing solutions.

[10]Note that this is possible with instruction counting because it is a monotonically increasing function. For memory allocations and de-allocations, such a strategy would not work.

2.7 On-chain code representation and virtual machines

On the chain, the smart contract code is stored as bytecode, which allows greater portability and thus is preferred over native code. The bytecode is executed in a virtual machine, e.g., using an interpreter or a JIT compiler.

Here, we consider the different code representations (or simply instruction sets) and the VMs used by popular blockchains. Table 3 summarizes the VMs and their supported code representations.

Table 3. A brief description of bytecode and VMs used by various blockchains.

Blockchain	On-chain bytecode	VM
Ethereum	EVM bytecode – stack-based bytecode, designed from scratch specifically for EVM.	Ethereum Virtual Machine (EVM) [79] – a specification of a VM for Ethereum protocol with multiple implementations.
NEAR	WebAssembly (Wasm) – stack-based language primarily developed for the browsers and the Web.	NearVM [7] – a wrapper around Wasm runtimes to work under the blockchain setting.
Solana	Solana Bytecode Format (SBF) – a modified version of eBPF bytecode, register-based. An important difference of SBF from eBPF is that loops are allowed.	Solana has its own VM, but in this work we focus on one of the components – RBPF [10], which is a modified version of the eBPF VM used to execute contracts.
Aptos	Move bytecode – stack-based representation, designed from scratch for the Move programming language.	MoveVM [19] – a VM to securely execute Move bytecode. In this work we consider the version used by Aptos.

First, we observe that Ethereum and Aptos use bytecode invented for blockchains and consequently a VM designed from scratch. In contrast, Solana and NEAR opt for existing instruction sets, in order to benefit from existing infrastructure, tools, and broader developer communities.

EVM, NearVM, and MoveVM use stack-based bytecode, which allows contracts to be more compact compared to the register-based representations [73]. In contrast, RBPF used by Solana is a register-based VM.

Next, we focus on the instruction sets supported by EVM, RBPF, NearVM, and MoveVM. We note that we do not describe all kinds of instructions, but rather focus on interesting design decisions that impact the cost model and its implementation, as we will see in Sections 3 and 4.

2.7.1 Simple instructions. All instruction sets we consider mostly comprise of simple instructions: arithmetic, type casts, constants, and stack manipulation (if stack-based). However, there is a noticeable difference in their semantics, which has long-standing effects on both the cost model and its implementations. As an example, we describe a simple integer addition shown in Table 4.

Most of the simple instructions in EVM operate on 256-bit integers, matching the word size of the stack. As a result, manipulating integers of smaller bit widths requires extra instructions (e.g., to zero out the bits) and becomes more computationally expensive. At the same time, simple instructions always terminate, e.g., addition wraps and even division by zero is defined as zero. We see that in such a design, it is the responsibility of the developer and the compiler (e.g., for Solidity) to have checks to avoid wrapping. A clear drawback is that this is error-prone, and in particular, opens the door for vulnerabilities in smart contracts.

MoveVM uses the opposite approach, making most of the simple instructions checked, and therefore arithmetic, type casts, and some other instructions can throw an exception at run time.

Table 4. Comparison of addition in EVM bytecode, SBF, Wasm and Move bytecode.

Language	Description
EVM bytecode	A single instruction that performs an unsigned 256-bit addition, wraps on overflow.
Wasm	Additions can be 32- or 64-bit, and wrap on overflow.
SBF	Additions can be 32- or 64-bit, and operate on both registers and immediates. Wrap on overflow.
Move bytecode	The addition is unsigned and is an overloaded instruction for different bitwidths from 8 to 256 bits (and throws an exception if the bit widths of arguments do not match). Throws an exception on overflow.

Such a design clearly prioritizes security over performance, because all instructions have an implicit branch and also because instructions are overloaded. Moreover, this has implications for the cost model making the instruction costs higher because of the implicit control flow.

NearVM and RBPF use existing instruction sets, with arithmetic, casts (for Wasm) and other instructions having wrapped semantics. Still, division traps on overflow or division by zero.

We conclude that the instructions sets used by different blockchains and VMs have different semantics, prioritizing either performance or security. Even simple instructions can have implicit control flow, which has to be accounted for by cost models and their implementation.

2.7.2 Control-flow instructions. Control flow is particularly interesting in smart contracts. Here, we consider how control flow is handled within a single contract, as summarized in Table 5.

Table 5. Comparison of explicit control flow in popular bytecodes.

Language	Description of local function calls	Description of general control flow
EVM bytecode	Calls to local functions defined in the same contract are implemented as jumps. To resolve the function call special selector code to jump to the right address is used.	Unstructured control flow with conditional/unconditional branches and terminating instructions (e.g., return).
Move bytecode	Supports generic function calls. Local and cross-contract calls are not differentiated.	
SBF	Local function calls are supported using `call` instructions. Arguments are passed via registers (SBF) and/or via stack (Wasm).	
Wasm		Structured control flow is built into the language in addition to classical branches.

We observe that all instruction sets have instructions that transfer control flow or terminate the execution (successfully or not). Moreover, at the time of writing, there is no exception handling. [11]

We highlight that Wasm stands out compared to the other instruction sets because it uses a structured control flow. As we will see in Section 3, structured control flow can be problematic because the code structure can differ significantly from native code, e.g., x86 assembly. For example, Wasm programs can contain deeply nested blocks (a sequence of instructions where branches can

[11] There is some progress on adding exception handling to Wasm, but it is not yet finalized [18], nor supported by Wasm-based blockchain runtimes.

only jump to inner blocks or the end of the block) with almost no real branching. [12] While Wasm interpreters have to execute these instructions, they do not exist in native code.

It is also worth mentioning the difference in function call handling. Because function calls in Move bytecode can be generic, MoveVM has to perform type instantiation at run time. The use of generics simplifies smart contract development and allows for code reuse but also shifts the type instantiation to run time, which can negatively impact performance and adds complexity to the cost model.

At the same time, the Ethereum smart contract code does not have functions at all. This makes porting EVM bytecode to alternative faster representations (e.g., Wasm) challenging, and makes it very difficult to analyze the code structure.

2.7.3 Cross-contract calls. Cross-contract calls are essential for smart contract design and are widely used by web3 developers. In general, there are two different ways of handling cross-contract calls, as we can see in Table 6: by starting a new VM or loading the called contract at run time.

Table 6. Comparison of cross-contract calls in popular VMs.

VM	Description
EVM, NearVM, RBPF	A new VM is instantiated when there is a cross-contract call. The current state (e.g., registers, arguments, or memory) is passed to the new VM instance. When the call completes, the control returns to the caller's VM.
MoveVM	When the contract is executed, its cross-contract dependencies (i.e., imported modules) are loaded in advance. When a cross-contract call is identified, the function is called from one of the loaded modules, identically to how a local function would be called.

We observe that MoveVM follows a more traditional design used by general-purpose VMs and languages: the code is loaded at run time and the VM simply transfers the control flow to it.

This design allows loading the called contract only once, and then calling functions, just like local functions would be called. On VMs like EVM, a new VM is instantiated for each call, and therefore the arguments and the return data have to be passed in and out for the new VM.

At the same time, such a design also has a drawback. For example, all dependencies are loaded before a contract executes, whether or not all of them are used. This can negatively impact performance and even be used as an attack vector (under the existing cost model).

2.7.4 Memory instructions. In all VMs, there is a concept of memory that can be accessed and modified at run time, i.e., a heap space. Table 8 summarizes the memory model for different VMs and describes how memory can be accessed.

We observe that while both EVM and NearVM use linear memory, the semantics of memory expansion are different. NEAR requires memory management to be done by the compiler – to insert instructions to grow memory. In contrast, on Ethereum, every memory access can potentially expand memory. This approach can degrade performance because memory is expanded multiple times instead of performing a bulk allocation.

Interestingly, MoveVM used by Aptos does not have an explicit memory at all. While this has its pros (e.g., automatic memory management, simpler bytecode), it can be very restrictive and might have unpredictable performance. For example, multiple fields in a struct might not be allocated consecutively, making field access latencies different.

[12] https://github.com/bytecodealliance/wasmtime/issues/3414

Table 7. Comparison of memory models and instructions for different bytecodes.

VM	Description
EVM bytecode	Memory is a word-addressed byte array with a word size of 256 bits, zero-initialized and linear, i.e., it is never freed and only new bytes can be allocated. EVM bytecode has three instructions to handle memory to load words from memory or store a single byte or a word. These instructions perform memory expansion implicitly when accessing an unallocated memory address.
Wasm	Memory is linear and can only grow in increments of pages, that is at least 4 KB. Memory can be expanded using a dedicated `grow_memory` instruction, and accessed via standard Wasm load/store instructions. Accesses that are made beyond the current memory size trap.
Move bytecode	There is no concept of memory in Move bytecode. A run-time value representation of data is used instead, where value is just an enumeration, which can be a primitive type, a heap-allocated vector, or a heap-allocated struct.
SBF	Memory is a fixed-size heap space, which can be accessed using different flavours of load/store instructions. Accesses are checked, and they trap if an invalid memory region is accessed.

2.7.5 Storage instructions. Recall that blockchains have a global state where all the data is maintained. The global state can be accessed by smart contracts and modified, and therefore VMs have to handle this interaction. For Ethereum, Solana, NEAR, and Aptos blockchains the storage is organized as key-value pairs, but the way the contracts access it is different, as described in Table 8.

Table 8. Comparison of storage models and instructions in EVM bytecode, SBF, Wasm and Move bytecode.

Language	Description
EVM	Storage access can be either a load or a store, both of 256-bits (EVM word size).
Wasm	Storage accesses in Wasm contracts are modelled as calls to host functions. Host functions are defined in the VM (in the host program, as the name suggests) but can be called from Wasm code. Like in Move bytecode, reading and writing to storage, existence checks, and delete operations are supported.
SBF	Solana accounts are stored in memory-mapped files, which map into the virtual address space of a running process that executes the contract. Storage can be accessed using load/store, and other similar SBF instructions.
Move bytecode	Move bytecode has instructions to move data to storage, borrow it from the global state (returning a mutable or immutable reference), check existence, or delete it.

We observe that for EVM and MoveVM the storage is part of the bytecode specification, and is accessed with special instructions. Move storage instructions are more detailed, potentially allowing for more optimizations. At the same time, they take a generic type parameter that represents the type of data to access from, and thus the type is only known at run time. As a result, it is challenging to implement JIT compilers for Move bytecode as the generic types would need to be monomorphized during execution.

In contrast, Solana and NEAR use existing VMs and therefore use different approaches. NEAR is still similar to Ethereum or Aptos because storage accesses are modelled as host function calls.

Solana maps storage to a memory region, which allows RBPF to treat storage and memory accesses uniformly and keep the VM implementation simple.

2.8 Bytecode verification, interpretation and JIT-compilation in blockchains

Usually, VMs for general-purpose languages employ a variety of techniques in order to improve security and performance. In particular, two major techniques that are important for this work are bytecode verification and JIT compilation.

Table 9. Bytecode verification and execution in existing VMs.

VM	Bytecode verification	Execution
EVM	A single pass to compute the table with jump targets and validate them.	A variety of interpreters, ranging from simple implementations [2] to more optimized ones [15] which use techniques such as jump-threading.
NearVM	Before execution, Wasm code is validated, e.g., type checked.	Mostly JIT compilation via Wasmer [13] (in production) or *wasmtime* [16].
RBPF	A modified eBPF verifier (e.g., it allows loops). Performs a linear pass with simple checks and does not analyze the control-flow graph of the contract.	A simple interpreter and a JIT compiler (working for x86, and Aarch64 under development).
MoveVM	Extensive bytecode verification, which checks the structure of the bytecode and whether it is well-typed, stack sizes, absence of dangling references, etc.	A very simple interpreter with no optimizations.

As shown in Table 9, the VMs for blockchains are still very young and usually lack security features (e.g., no extensive bytecode verification) and are not optimized for performance. We observe that in cases where fast execution is available (e.g., for Solana and NEAR), it comes from re-using existing infrastructures and runtimes (in the case of Wasm).

However, using existing runtimes comes at a cost. The VMs used by blockchains need to integrate deterministic resource metering in order to implement the cost model. We will see in Section 4 that this is a challenging task, and can lead to bugs and security vulnerabilities in blockchains.

2.9 Summary

In this section, we understood what are blockchains and how smart contracts are executed. We learnt that blockchains use cost models to approximate the real execution or storage costs. Cost models are fundamental to protect the network from DoS attacks, to ensure that clients pay their fair share of the infrastructure cost, and to reward validators who operate the network.

The implementation of a cost model is called metering. Metering tracks execution and storage costs, as well as limits certain run-time resources. A well-designed cost model must have an efficient metering scheme.

We also presented a detailed comparison of multiple VMs used by state-of-the-art blockchain networks: Ethereum, Solana, NEAR, and Aptos, as well as their bytecodes. We observed that the semantics of instructions in different chains vary greatly.

3 BLOCKCHAIN COST MODELS: A STUDY

In this section, we describe the cost models used by some of the popular blockchains, focusing on Ethereum, NEAR, Solana, and Aptos. A well-designed cost model is crucial for the cost-effective execution of smart contracts. As we will see later, the existing cost models do not always reflect that.

Cost models have to account for many different aspects of the blockchain system, including but not limited to execution in the VM, the underlying protocol, or long-term storage. In this work, we primarily focus on execution-related aspects of cost models, analyzing what incentives they bring, as well as identifying what they miss.

3.1 Measuring computations on blockchains

Blockchains use a cost model to 1) approximate the real cost of executing a contract, and 2) provide the right incentives to contract developers, infrastructure operators, and clients. Cost models usually take into account the typical hardware that validators use, the VM stack, and the blockchain protocol.

3.1.1 Approximating hardware costs. The executed instructions usually serve as a good metric for calculating the cost of a program and how it utilizes the hardware. For example, there exist instruction tables [43] which help engineers estimate the latency and throughput of different instruction sets on various CPUs. Depending on what kind of instructions are used, programs can be more or less expensive (in terms of the usage of the computational resources) to execute.

Being able to predict the execution cost based on the instruction opcode is particularly attractive for blockchains. A common approach used by the majority if not all blockchains, including Ethereum, NEAR, and Aptos is to associate instructions with costs so that the cost is proportional to how computationally expensive the instruction is. Then, in order to approximate the cost of a sequence of instructions it suffices to sum up their costs.

It is also important to consider the code size of the program in the cost model. It is well-known that large program sizes can degrade performance because programs do not fit into the instruction cache, or because they have to be loaded/compiled at run time by the VM. Hence, blockchains usually have a cost per byte of code.

The second metric which approximates the utilization of hardware is memory. Indeed, memory accesses can impact the execution cost of a program significantly: the latency of an access varies a lot depending if it is a cache hit or miss, and where the data is coming from.

Approximating the real cost of memory is difficult because the cost model has to be deterministic for any hardware while the effects of caching are not. Instead, blockchains opt for using simpler memory models with pricing based on the amount of allocated space, rather than the access patterns.

Lastly, the cost model must account for the data stored in the blockchain's global state, because this consumes disk space of validators. The cost of storage has to take into account 1) the number of accesses and the amount of fetched data from the disk (in bytes), and 2) how long the data is to be stored for.

Note that the cost of long-term storage is crucial: if the amount of active users of the blockchain increases, so does the global state, putting more pressure on validators and forcing them to use larger and larger disks. Hence, in existing blockchain systems the cost model usually incentivizes clients to delete their data or enforces a rent-like mechanism [13] to prune unused disk space.

In summary, the cost of the hardware can be estimated as a weighted sum of:

[13] https://docs.solana.com/implemented-proposals/rent

(1) the costs of executed (bytecode) instructions,
(2) the cost to load the program,
(3) the memory costs when executing a program,
(4) the storage costs to store the global state of the blockchain.

3.1.2 Approximating VM costs. Contracts are executed inside a VM, either via an interpreter or using JIT compilation. Therefore, it is important to consider the costs associated with starting up the VM itself.

In addition, before a contract gets executed, its bytecode may be analyzed. For example, Aptos runs a bytecode verifier before executing any program. We note that to our knowledge there are no blockchains that include such analysis into the cost model. [14]

This can lead to potential attack vectors on existing networks. For example, if the algorithm uses a fixed-point iteration to perform an analysis, e.g., compute some data flow, then it is vulnerable to adversarial inputs. In particular, it is possible to come up with an instance of a program where the run time becomes at least quadratic in the input size.

In summary, the cost of the VM can be estimated as a weighted sum of:

(1) the cost of launching the VM,
(2) the cost of analyses (e.g., bytecode verification) performed by the VM.

3.1.3 Approximating protocol costs. Finally, it is important to consider the cost of the actual protocol a blockchain uses. Based on the protocol specification, a transaction has a certain size in bytes and carries one or more signatures that have to be verified. Existing cost models typically place a limit on the maximum size a transaction can have (so as not to saturate the network) and include the cost for signatures attached to the transaction.

Moreover, every blockchain protocol validates transactions before execution by specifying the number of checks transactions should pass. These checks, as we have seen, are agnostic of the actual program and usually involve checking the sender's balance or the sequence number of the transaction. As some computational resources are used for validation, they also have to be included in the cost model.

We note that for convenience, existing blockchains combine certain VM and protocol fixed costs under a single *base* cost – the minimum cost of executing any program on a blockchain. Such an approach makes the cost model more concise and easier to analyze.

In addition, a cost model should also take into account the execution of transactions within the block. For example, Figure 4 shows the runtime of Block-STM [44] – a parallel transaction processing engine used by the Aptos blockchain, on different workloads: peer-to-peer transfers between 2, 4, and 100 accounts, selected randomly.

We observe that the larger number of accounts implies lower conflict rates between any pair of transactions. As a result, the execution time can increase by order of magnitude if there are many conflicts.

While similar trends are observed in other chains that support parallel processing of transactions, it is interesting that no blockchain has yet included block execution into its cost model. Instead, different attempts to reduce the conflicts have been proposed, e.g., by providing sets of read-write locations [15] or shuffling transactions [51].

In general, scalability improvements such as parallel transaction processing have to be accounted for in the cost model. Another such example is blockchain sharding. With sharding, the blockchain is split into multiple sub-chains which allows the network to process transactions in parallel within

[14]There exists a proposal to charge for the `JUMPDEST` analysis on Ethereum [32]. However, it has not yet been accepted.
[15]https://docs.solana.com/developing/programming-model/transactions#overview-of-a-transaction

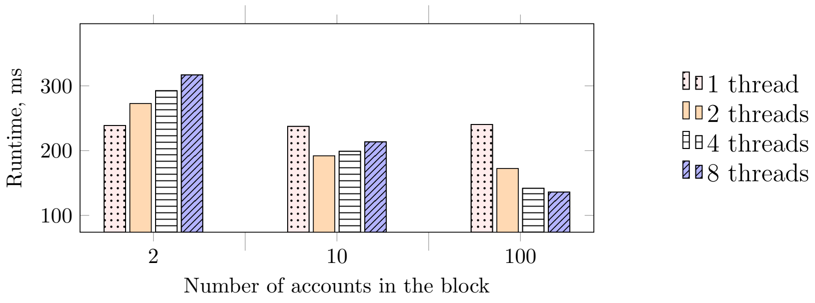


Fig. 4. Runtimes for executing a block of 1000 peer-to-peer transactions using Block-STM with a different number of threads. On the x-axis – the number of accounts in the system, used to randomly generate peer-to-peer transactions. The data is obtained using Apple M1 Pro CPU with 8 cores and 16 GB of memory, with multi-threading and frequency scaling disabled. Aptos node version 1.7.2 is used.

each shard. However, sharding is susceptible to communication overhead, as it essentially sacrifices intra-shard parallelism for cross-shard communication. To date, there has been little research done on how sharding affects cost models.

In summary, the cost of the protocol can be estimated as a weighted sum of:

(1) the cost for a transaction to reach execution inside the VM, including validation, signature checks, etc.,
(2) the cost for executing a block of transactions,
(3) the cost of sharding.

In the rest of the section, we analyze the aforementioned costs, primarily focusing on execution inside a VM. We first look at the base costs used by popular blockchains and analyze how they compare to the average cost of a transaction. We also discuss different upper bounds used by the networks to limit resource consumption. Finally, we present a detailed analysis of how bytecode instructions are priced for the selected blockchains, focusing on simple, control-flow and memory/storage instructions. Where applicable, we mention the limitations of the current cost models and discuss potential problems or attacks.

3.2 Gas and computational units

In order to charge for on-chain computations, blockchains use the concept of *gas*. A notable exception is the Solana blockchain, where the term *computational units* (CUs) is used instead for the same concept. Solana is also the only blockchain in our study that uses a flat-fee model, charging the users for the number of signatures per transaction and not for the computations.

Even within the gas-based cost models, the costs in units of gas differ by orders of magnitude. For example, on Ethereum instructions usually have single-digit costs, NEAR uses tera-gas (10^{12} gas) when describing the costs, and Aptos uses fractions.

As a result, it becomes very difficult to compare the cost between different blockchains if using raw gas units. In order to allow comparisons across chains, we convert gas costs to the equivalent costs expressed in terms of US dollars (USD), as shown in Table 10. For example, if some computation has a gas cost of C_g, then its cost in USD C_{USD} is computed as

$$C_{USD} = C_g \cdot C_u \cdot C_t,$$

where C_u is the cost of a single gas unit in the native token and C_t is the exchange rate of the token to USD.

In this work, we use the exchange rates for the native tokens calculated based on the 7-day average from 05.09.23 to 11.09.23 taken from CoinMarketCap.[16] In order to calculate the gas prices in USD, we additionally took 7-day average gas prices for each network.

Table 10. Exchange rates for native tokens, the units of gas, and addition instruction for Ethereum, NEAR, and Aptos blockchains. For Solana, the cost for a single signature is shown.

	Ethereum	Solana	NEAR	Aptos
Token price	\$1,631	\$17.5	\$1.1	\$5.5
Gas unit (signature) cost	$\$23.0 \cdot 10^{-6}$	$\$87.5 \cdot 10^{-6}$	$\$0.11 \cdot 10^{-15}$	$\$7.7 \cdot 10^{-6}$
Cost of addition	$\$69 \cdot 10^{-6}$	\$0	$\$0.09 \cdot 10^{-12}$	$\$24.6 \cdot 10^{-9}$

We can already observe how the difference in the bytecode influences the costs. NEAR uses Wasm, a relatively low-level bytecode, and so its instructions are significantly cheaper than instructions in EVM or Move bytecodes.

In the rest of the section, we use the costs from Table 10 to compare prices across chains for different instructions or actions on the blockchain. We also refer to addition instruction as a baseline, as it is sometimes preferable to express the cost in the number of additions.

3.3 Base costs

Every blockchain includes a base cost in the cost model in order to account for fixed VM and protocol costs, as discussed before. Table 11 shows the base costs for some of the blockchains, both in terms of gas and in terms of USD.

Table 11. Comparison of base costs on Ethereum, NEAR, and Aptos blockchains. For NEAR, we use the base cost for calling a method of a smart contract. Again, Solana is different and has no base cost.

	Ethereum	Solana	NEAR	Aptos
USD price	\$0.48	\$0	$\$0.255 \cdot 10^{-3}$	$\$11.6 \cdot 10^{-6}$
Equivalent to # additions	7,000	-	$2.82 \cdot 10^{6}$	469
Average transaction cost	\$0.84	-	$\$1.40 \cdot 10^{-3}$	$\$3.14 \cdot 10^{-3}$

The base cost of Ethereum is equal to the cost of a transfer between two externally-owned user accounts (EOAs). According to Ethereum's website, this amount covers the cost of cryptographic primitives to get the address of a sender from the signature, as well as an estimate of the cost of storing the transaction. We observe that on Ethereum the base fee accounts for approximately 50% of an average user transaction cost. This means that the cost of executing the bytecode of smart contracts does not dominate the total cost. Another possible explanation is that transactions on Ethereum are mostly transfers, and therefore have zero execution cost, thereby skewing the average transaction cost closer to the cost of a peer-to-peer transfer.

We observe that Aptos has a significantly smaller base cost which is closer to the cost of bytecode instructions, unlike Ethereum, if expressed in terms of the number of additions. This can also be

[16] https://coinmarketcap.com

seen when comparing the base cost with the average cost of a transaction on Aptos. We deduce that on Aptos bytecode execution dominates the final cost.

Another possible explanation for such a discrepancy between the base cost and the cost of execution on Aptos is the inefficient VM implementation. Because MoveVM is not performant, the cost of processing a single instruction has to be increased (indeed, addition on Aptos is at least a million times more expensive than addition in Wasm).

In contrast, the base cost on NEAR for calling a function in a contract is an order of magnitude larger than a single Wasm addition. The average cost of transaction on NEAR is 5.5× greater than the base cost. It means that for NEAR the execution cost dominates the total cost of an average user transaction, similar to Aptos.

In conclusion, we observed that the cost of VM execution (i.e., costs of bytecode instructions) is the main contributing factor to the cost of an arbitrary user transaction, on average. As a result, it is crucial to ensure that bytecode instructions are well-priced.

3.4 Run-time limits

Blockchains enforce different limits at run time in order to ensure the resource consumption is bounded to protect against DoS attacks and to ensure deterministic execution. For example, NEAR and Aptos blockchains have a maximum length a function name can have in a contract so that malicious code does not slow down the network by using huge identifiers. For this work, it is worth mentioning a pair of limits: 1) code size, and 2) stack height for stack-based bytecode. We show the limits in Table 12.

Table 12. Limits on the maximum code size and the maximum stack height.

	Ethereum	Solana	NEAR	Aptos
Maximum code size (KB)	24	1.2	4096	64
Maximum stack height	1024	-	262,144	1024

We observe that the code size varies a lot. Interestingly, Solana uses a very small code size (rough estimation – 153 SBF instructions because each instruction is at least 8 bytes). This impacts how contracts are deployed – instead of deploying the contract all at once, developers have to send multiple transactions to put the code on-chain in chunks. This is because Solana does not price the code size and therefore keeps a very small free limit.

We observe that Aptos allows one of the biggest code sizes. However, it is worth mentioning here that the size of the code is priced: there is a free cap of 600 bytes, after which users have to pay on a per-byte basis. In contrast, Ethereum does not have that.

Wasm is also interesting. We see that it allows the largest programs, but this is because Wasm is low-level and so programs have many instructions, which increases the size significantly, unlike Solana.

The second limit is the stack size, which is not used for Solana because SBF is a register-based instruction set. Again, the same observation holds: Wasm code is expected to have more items on the stack but for Ethereum and Aptos – heights are the same.

Limiting the stack height is important for Wasm because the maximum number of elements a stack can hold varies across operating systems and different CPU architectures. At the same time, the limits have to be deterministic, and therefore the runtime that executes the contract has to limit its stack size as well.

In this work, however, we do not focus on enforcing stack size, or other forms of run-time limits. The problem of stack sizes is mostly applicable for Wasm, and stack-based instruction sets. Here, we consider instead a more general problem of metering of instruction costs and memory. The main conclusion is that certain limits require very expensive metering.

3.5 Costs for bytecode instructions

In this section, we study how different kinds of instructions are priced on Ethereum, NEAR, Solana, and Aptos. This is a particularly interesting study because one can observe how the semantics of bytecode instructions influence the cost.

While Solana uses a flat-fee model and does not take executed bytecode instructions into account when computing the cost of a transaction, it still defines a limit on the maximum number of instructions that can be used in a program. As we will see, such a setting simplifies the cost model but can be exploited.

3.5.1 Simple instructions. For selected blockchains, the majority of instructions have a constant integer cost. Here, we consider simple instructions such as arithmetic, casts, etc. In particular, we focus on arithmetic instructions – their costs (relative to each other) and their costs in USD (across chains).

We observe that all blockchains with an exception of Ethereum assign the same cost for simple instructions. For example, Table 13 shows how the costs for multiplication and division compare to the cost of addition.

Table 13. Comparison of costs of multiplications and divisions on Ethereum, Solana, NEAR, and Aptos. All costs are expressed in terms of the cost of addition instruction or USD.

Multiplication or division	Ethereum	Solana	NEAR	Aptos
# additions	1.67	1	1	1
USD	$\$115 \cdot 10^{-6}$	$0	$\$0.09 \cdot 10^{-12}$	$\$24.6 \cdot 10^{-9}$

It is interesting that while division is more expensive than addition or multiplication on modern CPUs [43], it is still assigned the same cost (or almost the same in the case of Ethereum). To understand if this approach is robust, we set up a small experiment. We construct SBF programs which consist only of additions or divisions and execute them using Solana's RBPF JIT compiler (without metering their costs). We use Intel Core i7-7700HQ (Kaby Lake) for the experiments with multi-threading and frequency scaling disabled.

We observe that programs with divisions only, on average, are 30× slower. In addition to large latencies of division instruction compared to addition, the slowdowns are due to the JIT compiler adding checks for division by zero and division overflow.

Such a significant run-time overhead has an impact on how many transactions a chain can process and the fees validators receive. We recall that Solana prices transactions using the flat-fee system, equivalent to $\$87.5 \cdot 10^{-6}$ per signature (i.e., per transaction). Given that Solana can operate at a few thousand transactions per second (TPS), assuming an approximation of 5000 TPS, in order to process 1,000,000 transactions a validator needs $1,000,000/5,000 = 200$ seconds, approximately.

Given that Solana has 2,421 validators in total, with 50% of fees being distributed to the validator, the daily revenue of a validator can be calculated as

$$24 \cdot 3,600 \cdot 0.5 \cdot 87.5 \cdot 10^{-6} \cdot 5,000 \cdot \frac{1}{2,421} = \$7.8$$

in expectation.

Hence, processing 30× less transactions results in only earning \$0.3 daily, if all transactions were to contain only divisions.

We conclude that having the same cost for simple instructions can be a good idea, however, outliers such as division have to be taken into account. One can say it is not realistic to have contracts consisting of division only, but there can be instances of other instructions also with high variance in latency, or malicious users aiming to reduce validators' profits and slow them down.

Curious if other blockchains are affected by this, we looked into solutions by Ethereum, NEAR, and Aptos. Only Ethereum has a higher cost for division than for addition, with it being 1.7× more expensive. At the same time, a recent study [68] shows that Ethereum opcodes are not well-calibrated in terms of costs.

Authors observed that additions and multiplications have the same running time in their experiments, but division instructions are 5× slower than addition, on average. This is because according to Ethereum EVM bytecode specification, division by zero is defined as zero, and so division bytecode instruction has an implicit branch. We observe that the factors between the costs and execution times are different, and so it is easy to construct an adversarial example that would not cost proportionally to the work done by a validator node.

On Aptos, division has the same cost as addition. Divisions are slower than additions, and therefore the program can have the same problem as in Solana. However, we do not observe that in our experiments because the VM implementation is not optimized for performance, and the cost of the interpreter's dispatch loop and internal data structures dominates the total running time.

At the same time, NEAR benefits from using Wasm, which has a better solution thanks to its specification. In Wasm, division operation traps on division by zero or overflow, and therefore the Wasm code must also include checks for division by zero or overflow. These checks are also priced, and therefore, the latency due to branches on overflow is implicitly taken into account.

In conclusion, we observed that

- instructions can have very different latencies which, however, are not always taken into account by modern blockchains;
- instructions that can throw an exception, such as division, require additional handling, which prevents validators from earning more and also decreases the throughput of the network.

The latter also adds complexity to metering, as we will see in Section 4.

3.5.2 *Control-flow instructions.* In all cost models we study, control flow is never taken into account. Instead, the billing is based on the instructions that lead to a change in control flow. Table 14 shows the costs for branch instructions on Ethereum, Solana, NEAR, and Aptos.

Table 14. Cost of control-flow branching instructions on Ethereum, Solana, NEAR, and Aptos in terms of the number of additions and USD.

Instruction	Ethereum	Solana	NEAR	Aptos
unconditional branch (# additions)	2.67	1	1	0.33
unconditional branch (USD)	$\$184 \cdot 10^{-6}$	\$0	$\$0.09 \cdot 10^{-12}$	$\$18.5 \cdot 10^{-9}$
conditional branch (# additions)	3.33	1	1	0.67
conditional branch (USD)	$\$230 \cdot 10^{-6}$	\$0	$\$0.09 \cdot 10^{-12}$	$\$27.7 \cdot 10^{-9}$

Based on the data, we observe that branches are cheaper on Aptos than on Ethereum. Such counter-intuitive results can be due to the semantics of Move bytecode. We recall that arithmetic is always checked in Move, and therefore there is always an implicit conditional branch. This aligns with the cost of addition being smaller than the cost of the conditional branch.

We also see that branches are cheaper in Move when compared to branches in EVM bytecode, relative to the cost of addition. A possible explanation for this is that EVM models function calls to functions within the same contract as branches, and as a result, their cost is increased.

In general, the main question is whether branches should be more expensive or not. In practice, branches have very small latencies thanks to branch predictors in hardware. However, on misprediction, the latency increases by 10-20 clock cycles and stalls the execution pipeline of a CPU.

Because the cost model has to be an over-approximation to prevent DoS attacks, assigning higher costs to branches is reasonable as it accounts for the worst case. However, Ethereum is the only blockchain that does that.

To support our claim, we conduct a small experiment again using the JIT compiler in RBPF. We compare programs which use only additions with programs which use only branches (with different offsets and patterns), with results presented in Figure 5.

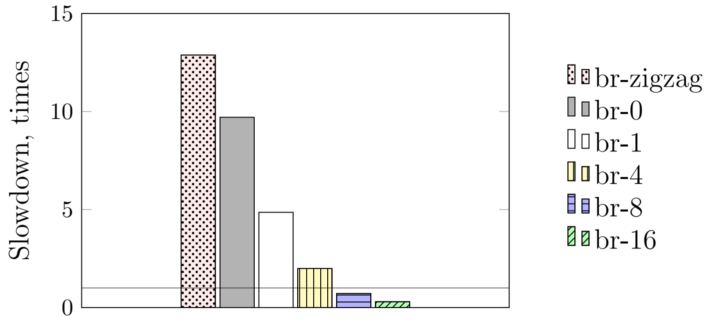


Fig. 5. Slowdowns of programs that consist of branches only compared to the program with only additions. Here, *br-N* programs unconditionally jump with offset *N*, and *br-zigzag* jumps back and forth until converging in the middle of a program. The data is obtained using Intel Core i7-7700HQ (Kaby Lake) with multi-threading and frequency scaling disabled. RBPF version 0.7.1 is used.

We observe that the run time of programs with branches is significantly slower. For example, when the branch always jumps to the next instruction (*br-0*), the program is 9.7× slower than the program with additions. Similarly, the programs *br-1* and *br-4* which execute 2× and 4× less instructions instead of being faster, end up being 4.9× and 2× slower. We see that *br-zigzag* has the worst running time and is 12.9× slower than the equivalent program with additions (while executing the same number of instructions).

The experiment shows that while existing cost models might be acceptable for interpreter-based execution, it is not the case for JIT compilation. Branches are significantly slower than simple instructions such as additions, if executed in native code, and should have higher costs. Moreover, branch offsets and access patterns play an important role in affecting the running time.

Another interesting aspect of existing cost models is that control flow impacts the code size and can lead to higher costs for the clients if the code is JIT compiled and not interpreted.

For example, in EVM bytecode, there is a special instruction `JUMPDEST` which costs 1 gas unit ($\$23 \cdot 10^{-6}$ or 0.33 additions) and marks a valid jump target. Similarly, Wasm uses structured control-flow, enforced by `block` and `loop` bytecode instructions (which cost $\$0.09 \cdot 10^{-12}$ or 1 addition). In particular, branches can only jump to the end of the `block` or the start of the `loop`.

Just like `JUMPDEST`, these instructions are only needed to enforce the validity of the control flow and otherwise are no-ops. We refer to such instructions as *markers*.

Firstly, markers increase the code size. If we consider Ethereum for instance, every `JUMPDEST` instruction adds a byte to the size of the code. Having 1024 such markers in the contract code already uses 4.2% of the code size limit, which, given that Ethereum models local function calls as branches, can be very limiting to the users.

Secondly, if the code is interpreted, markers are still executed. If they are not priced, a malicious program consisting of `JUMPDEST`s or a nested sequence of Wasm `block`s can be executed for free.

At the same time, if priced, the cost of execution of JIT compiled contracts over-approximates the true execution cost: markers do not appear in the generated native code. As a result, users can end up paying for the nested sequence of Wasm `block`s which are never executed. While this scenario is particularly attractive for validators (higher transaction fees while doing less computations), it can be bad for end users.

During our study, we found an interesting difference in existing cost models used by NEAR and Polkadot on how to price `loop` instruction. According to NEAR, the instruction is executed at every iteration of the loop and therefore has to be metered and charged for at each iteration. In contrast, on Polkadot, the `loop` instruction is executed only once when the loop is entered, and so also has to be metered only once. Essentially, the `loop` instruction is treated as a jump to the loop head, and so the control flow returns to the first instruction in the loop body.

This observation results in different costs for the same Wasm programs on different blockchains. On NEAR, loops turn out to be more expensive. Also, as a result, unrolling the loop on NEAR is more beneficial because the cost for `loop` is avoided.

To conclude our investigation about the costs of control-flow instructions, we note the following:

- existing cost models do not take control-flow structure into account (e.g., vectorizable and non-vectorizable loops can have the same costs);
- branching instructions are mostly under-priced in existing cost models, in particular if contracts are JIT compiled;
- using bytecode instructions to mark valid control flow may lead to significant code size increase, higher costs for end users but also higher profits for validators;
- semantics of the language can be interpreted by different cost models differently, affecting execution costs.

In Section 4, we will further observe that branches also complicate metering.

3.5.3 Memory instructions. Another important part of the cost model usually considered when executing smart contracts is memory. First, we consider how different blockchains price memory allocations. On Ethereum, its linear memory is expanded using the equation

$$3a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

where a is the number of new 32-byte words allocated. In contrast, Wasm grows its memory in increments of pages and so NEAR assigns a per-page cost. Solana has a free 32 KB heap, and Aptos does not charge for memory consumption at the time of writing.

In Table 15, we compare the costs of bulk allocations on different blockchains. In particular, we compare how much it costs to allocate 4 KB of memory (a typical page size).

We observe that allocating that much memory on Ethereum is extremely expensive. This is because allocations of the size of 4 KB make the cost equation quadratic in the number of words.

The other downside of Ethereum's cost model, apart from high prices, is the fact that memory can be expanded on every store to memory. As a result, this has to be enforced by the metering

Table 15. Costs for bulk memory allocations (4 KB) on Ethereum, Solana, NEAR and Aptos.

	Ethereum	Solana	NEAR	Aptos
Cost for 4KB	\$32	\$0	$\$0.11 \cdot 10^{-15}$	-

scheme, as we discuss later in Section 4. Other blockchains opt for more metering-friendly cost models, where allocations are either not tracked (Aptos), are done in advance (Solana), or are performed in larger increments by special instructions (NEAR).

We now consider the cost models used to price memory accesses, summarized in Table 16. We observe that accessing memory is very cheap on Ethereum, Solana, and NEAR, with the costs comparable to the cost of addition. However, it is worth mentioning that because EVM can expand memory on access, the cost can be significantly higher.

Table 16. Costs for memory accesses on Ethereum, Solana, NEAR, and Aptos.

	Ethereum	Solana	NEAR	Aptos
Memory access (#additions)	1	1-2[17]	1	-
Memory access (USD)	$\$69 \cdot 10^{-6}$	\$0	$\$0.09 \cdot 10^{-12}$	-

Aptos does not have the concept of memory but instead has instructions to access data from structs and vectors which are first-class types in Move bytecode. The costs of these instructions are usually between 2-3 additions in Move.

It is interesting to analyze how caches impact the latency of memory accesses. Because these instructions have costs similar to arithmetic, we expect to observe a very similar behaviour to the one we have seen with branches: in practice, accessing memory can be very slow and instructions are under-priced.

However, in this work, we focus on the other fact – all memory accesses have an implicit branch and throw an exception if the access is out-of-bounds. Just like with divisions, this alone has two implications: 1) implicit control makes metering more difficult, which we discuss in Section 4, as well as 2) JIT compilers have to emit strictly more code than for additions, and interpreters need more checks. We conclude that the cost of memory access has to be higher to account for that.

3.5.4 Storage instructions. While the cost model for persistent storage is out of scope for this work, it is important to at least mention it here because accessing storage still impacts the run time and metering.

In general, storage accesses are priced based on the number of bytes loaded at run time. Typically, the price is reduced if the access is cached by internal data structures (deterministically), or the access is cheap. For example, writing to a new storage location is expensive on Ethereum, but changing existing items is not.

A general cost function associated with storage access is

$$C_{base} + C_{key} + C_{per\_byte} \cdot n$$

where C_{base} is the base cost, C_{key} is the cost for the number of bytes in the key (storage is a key-value database), n is the number of loaded bytes, and C_{per_byte} is the cost per-byte.

While existing pricing is rather detailed and complicated, for this work the important takeaway is that the cost of storage access cannot be determined in advance, in general, and is dynamic in the

[17]Earlier versions of SBF bytecode have `lddw` instruction which occupies two words, and therefore costs 2 CUs.

number of loaded bytes. We note that because Solana treats storage accesses the same as memory accesses, such a problem does not exist.

3.5.5 Cross-contract and local calls. Existing blockchains handle local and cross-contract calls very differently. For example, Ethereum represents local calls as branches and uses a very complex model for cross-contract calls. In addition, EVM, RBPF, and NearVM instantiate a new VM for each cross-contract call.

Such an approach is particularly disadvantageous for efficient metering. Before the cross-contract call, the exact cost for the currently executed instructions has to be known so that the new VM instance can precisely meter the callee's costs.

In this work, we only consider local and cross-contract calls on Aptos and local calls on Solana and NEAR so that we can focus on the aspects relevant to metering and use a more conventional setting where the VM loads new functions when they are called.

Table 17 shows the base cost of a function call on different blockchains. We observe that both Solana and Wasm price calls the same way as simple instructions. While we have not conducted experiments similar to the ones with branches and control flow, it is clear that having the same price for a function call and addition may result in malicious programs slowing down the network, while not being charged accordingly.

When a call is made even in native code, there is always an additional cost for saving registers, pushing function arguments to the stack, jumping to the start of the new code, and restoring the saved registers. Clearly, a reasonable cost model has to account for this additional work.

Table 17. Base costs of a function call on Solana, NEAR, and Aptos in terms of the number of additions and USD.

	Solana	NEAR	Aptos
# additions	1	1	6.25
USD	\$0	$\$0.09 \cdot 10^{-12}$	$\$153.8 \cdot 10^{-9}$

On Aptos, in contrast, the cost of a function call differs from the cost of simple instructions. It can be written as

$$C_{base} + C_{arg} \cdot n$$

where C_{base} is the base cost for the call, C_{arg} is the cost for each argument, and n is the number of arguments. An extra cost is used for each type parameter if the function is generic. Such an approach allows Aptos to scale the cost for more expensive functions that use many arguments.

We observe that the base cost of a function call on Aptos corresponds to approximately six additions. While it is hard to judge if the cost is too small or too large, it definitely shows that simple instructions on Aptos are not efficiently implemented and are very expensive.

For cross-contract calls, another crucial aspect is code loading. Recall that when a function from another contract is called, MoveVM loads the code module where the callee is defined and its transitive dependencies. Additionally, the loaded code is re-verified. It is clear that unless cached, code loading can be very computationally expensive. Surprisingly, the cost model of Aptos does not account for that.

In conclusion, we found that the cost of function calls is calculated using a simple model and is mainly constant. While it is reasonable to assume that such simple models can be attacked, a more in-depth study is needed to evaluate the performance and the economic impacts of calls on validators' profits and metering. However, this is out of the scope of this work.

3.5.6 Summary. In this section, we explored existing cost models used by Ethereum, Solana, NEAR, and Aptos. We found that, in general, blockchains use very tight limits for resources such as code size or stack height (for stack-based bytecode) to ensure that transactions are quickly propagated through the network and to enforce deterministic execution.

We also discovered that existing models are not well-calibrated. While simple instructions such as arithmetic have similar and low costs, some instructions produce side effects and can trap. In these cases, execution can be significantly slower, especially if the contracts are JIT compiled. Moreover, as we will see in Section 4, such instructions are difficult to meter efficiently, making existing metering schemes slow and susceptible to malicious inputs.

Cost models used by the state-of-the-art blockchains also do not take control flow into account. This is a surprising fact, in particular, given that it is well-known that unpredictable control flow can slow down execution significantly.

Lastly, we observed that the cost of bytecode instructions dominates the total execution cost. Moreover, these costs are mainly constant, with storage and memory being the only exceptions.

4 METERING SCHEMES: A STUDY

A cost model is implemented as part of a VM or a protocol. In the latter case, the implementation usually does not have a significant overhead. For example, given the maximum size of a transaction in bytes, a transaction can be rejected at the validation stage while sitting in a transaction pool. In contrast, the implementation of a cost model inside the VM, and the metering schemes in particular, directly impact security and performance.

In this section, we study how blockchains meter resource usage. In particular, we focus on the state-of-the-art metering schemes used for:

(1) metering instruction execution cost,
(2) metering memory and storage costs.

These metering schemes already cover a large portion of the cost model's implementation and are sufficient for our study.

4.1 Instruction cost metering

Recall that the role of instruction cost metering is to ensure that the sum of the costs of executed instructions does not exceed a pre-defined budget, and to compute the final execution cost. If the budget is not sufficient, the execution must be halted.

Under this specification, it is straightforward to see how to implement instruction cost metering. For any instruction, one has to 1) calculate the cost of instruction, 2) subtract the cost from the current budget, and 3) if the budget is negative, halt the execution. Otherwise, the next instruction is executed.

In practice, there are a few options to meter instruction costs, as summarized in Table 18. They depend on the blockchain network and the underlying VM.

Table 18. Different approaches to meter instruction costs in VMs. Here, we also list some of the blockchain networks that use it.

Metering type	Example	Used by
Software	Integer counter, updated on every iteration of the dispatch loop of an interpreter.	Aptos, Solana, Ethereum
Bytecode	Dedicated bytecode instruction which updates the counter.	NEAR, Polkadot
Native code	The counter is stored in a pinned register or memory location and is occasionally updated by the program.	Solana

Instruction cost metering in software is the easiest to implement, and it is the go-to approach for the networks that use interpreter-based VMs, e.g., Aptos and Ethereum. Solana also uses integer counters in software for its interpreter in RBPF.

The metering in the bytecode is used by Wasm-based blockchains – NEAR and Polkadot. NEAR instruments contract with calls to host functions and then JIT compiles the resulting Wasm code. Polkadot translates Wasm into its own custom bytecode with a special metering instruction and uses an interpreter instead. Both blockchains aim to minimize the amount of the metering instrumentation (the number of host calls or metering instructions). Later we will see how these two approaches compare.

If the contract code is JIT compiled, the metering is injected into the target program. While NEAR does this at the Wasm level, before the native code is generated, Solana is an example of a blockchain that produces metered native code directly. RBPF's JIT compiler uses a fixed register and an algorithm to minimize the number of instructions used for metering.

Next, we consider three existing approaches to metering in turn. We guide our study by answering the following questions:

(1) How does metering impact the run time and the code size?
(2) How does metering impact the security of the blockchain?
(3) What are the economic implications of metering for validators, if they collect a fraction of the fees?

We use an Apple M1 Pro CPU with 8 cores and 16 GB of memory for all experiments. The only exception is RBPF because its JIT compiler only supports x86 backend. For the experiments with RBPF, we use an Intel Core i7-7700HQ (Kaby Lake). In both cases, we disable multi-threading and frequency scaling.

4.1.1 Instruction cost metering in software. Instruction cost metering in software can only be used with interpreters. Here, we consider four different interpreter-based VMs: *evmone* [15] and *geth* [2] for Ethereum, the interpreter of RBPF for Solana, and MoveVM for Aptos. They all maintain an integer counter, which is updated and checked at each iteration of the dispatch loop.

First, we analyze the performance implications of instruction cost metering in software. We compare the cases where VMs use a counter or not. To ensure VMs do not track instruction costs, we comment out the necessary parts of the code. For all experiments, we use the built-in benchmark suites of selected VMs. We describe our observations and their implications in Table 19.

Table 19. Instruction cost metering in software: observations and implications.

Observation	Implication
The metering with a counter does have a run-time overhead of at most 0.2–0.5% for all VMs on all benchmarks. Disassembling the interpreter's code, we find that metering only takes a few x86 instructions (see example in Figure 6). In comparison, an implementation of the addition bytecode handler, e.g., in *geth*, consists of 63 x86 instructions.	The metering has no performance penalty in the *current* implementations of the VMs because the run time is dominated by other parts of the system. Metering overhead is negligible. As a result, there are no implications on security or validator profits.
All VMs but *evmone* use an unsigned 64-bit integer for the counter, which results in less optimal native code.	Using signed integers should be preferred to reduce the number of assembly instructions. For example, using the USD price of a gas unit on Aptos, a straightforward calculation shows that with a signed 64-bit counter the execution cost can be at most $\$7 \times 10^{13}$, which is already an astronomical number.

We conclude that the overhead of instruction cost metering in software is fairly low. This is because existing interpreters used by popular blockchains are not implemented for speed.[18] As the interpreters get better over time, we believe that the metering problem will be more apparent.

[18]It is well-known that the most efficient interpreters have to be hand-written in assembly, as discussed here: http://lua-users.org/lists/lua-l/2011-02/msg00742.html.

```
  ; meter instruction cost
  cmp    rax, rsi
  jb     .out_of_budget
  sub    rax, rsi
  ; continue execution
.out_of_budget
  ; handle running out of budget
```

```
  ; meter instruction cost
  sub    rax, rsi
  jb     .out_of_budget

  ; continue execution
.out_of_budget
  ; handle running out of budget
```

Fig. 6. Assembly code (x86) for instruction cost metering when the budget is an unsigned (on the left) or signed (on the right) counter. The budget is stored in the `rax` register and the instruction cost is stored in the `rai` register.

4.1.2 Instruction cost metering in bytecode. In order to implement counter-based metering, an interpreter with metering being a first-class citizen has to be designed from scratch (such as EVM or MoveVM) or an existing one has to be adapted (Solana's RBPF) and maintained, which can be difficult, and time-consuming. Also, it is not possible to JIT compile the contracts or interpret them in some fast interpreter because then the execution is not metered.

In order to overcome these issues, metering instructions can be injected at run time into the contract's bytecode prior to interpretation or JIT compilation. There were early attempts to compile contracts to LLVM IR [4], but modern blockchains use Wasm runtimes, as they offer great portability, decent performance, and extensive tooling. For a blockchain VM, it can therefore be sufficient to instrument a Wasm contract with metering code, and then use an existing runtime to execute it.

Here, we consider two libraries for instrumenting Wasm code: *wasm-instrument* [9] (previously used by NEAR, Polkadot and FileCoin [54]) and *finite-wasm* [20] (currently used by NEAR in production). These libraries are used to inject metering instrumentation into Wasm contracts, which can then be executed by state-of-art interpreters or JIT compilers.

Additionally, we consider *wasmi* [12] – an interpreter for Wasm contracts used by Polkadot in production. Instead of re-using Wasm infrastructures, it translates Wasm into its own intermediate representation and interprets it.

One must consider two things when injecting metering instrumentation:

(1) the instrumentation itself: which instructions are injected, and how do they track the costs,
(2) the instrumentation placing algorithm: where to inject the instrumentation, and how often.

Table 20 describes three different kinds of instrumentation Wasm programs can be used.

Table 20. Metering instrumentation kinds for Wasm. Additionally, we list which instrumentation libraries and Wasm runtimes use these approaches to instrumentation.

Mutable global	Host function	Dedicated bytecode instruction
The Wasm module stores a global variable that represents the cost counter. The instrumentation is a few Wasm instructions that check and update the global.	Metering is done using a host function call. The cost counter is kept on the host side. The instrumentation is a call to the host function, which updates the counter.	Metering is encoded as a bytecode instruction, which also carries the cost. The bytecode handler has to be added to the VM.
Used by *wasm-instrument*.	Used by *wasm-instrument* and *finite-wasm*.	Used by *wasmi* interpreter.

All three approaches have different trade-offs. Using host functions is preferred over mutable global because the counter is kept in sync if Wasm code calls functions from other modules. In the case of a mutable global, it would have to be synchronized and linked correctly. [19] At the same time, using a dedicated instruction may be more performant than both of these approaches, because the metering then happens purely on the VM side. The drawback of this solution is that a bytecode handler (and code generation if JIT compilation is supported) needs to be implemented and maintained.

We now consider the second problem – how to inject metering instrumentation. The strawman solution would be to instrument every instruction in the source program, but this incurs a significant run-time overhead. Instead, an instrumentation placement algorithm is used to identify a better way of injecting metering code.

First, we consider *wasmi* and study the impact of the metering instrumentation on the run time of the programs executed by the interpreter. For that, we use a set of simple Wasm programs from the *wasmi* benchmark [20] as well as two programs from LLVM's testsuite [21] (quicksort and matrix multiply) compiled to Wasm with Emscripten. [22]

For the placement, *wasmi* uses a simple algorithm that sums up the costs of instructions if there is no control flow, and injects a special bytecode instruction that charges the accumulated cost. As it can be seen in Figure 7, instrumenting at every instruction brings significant performance regression, with slowdowns ranging from 7% to 120%. With a more careful metering employed by *wasmi*, the slowdown is decreased to at most 25%. In particular, we observe that on compute-intensive programs with long straight-line sequences of instructions such as `float_mm`, the instrumentation overhead is insignificant. At the same time, programs with control flow incur the highest overheads.

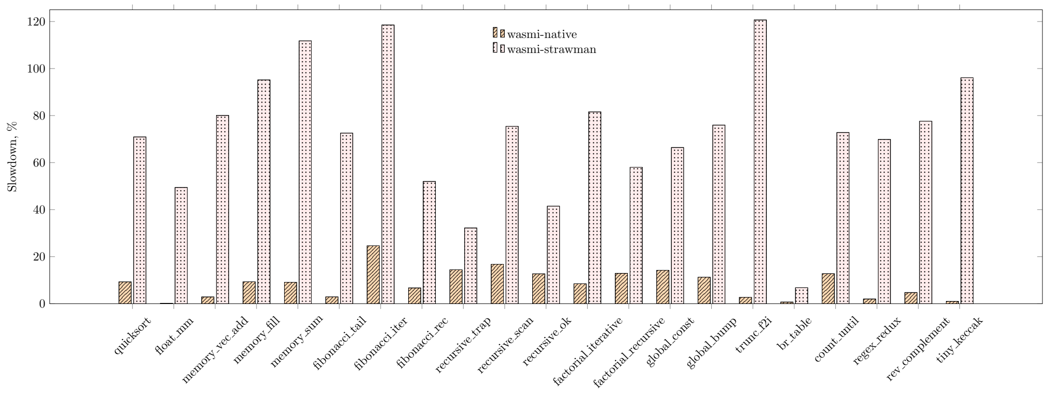


Fig. 7. Slowdowns of different Wasm programs when 1) metering is done for every instruction (*wasmi-strawman*) and 2) metering is done only on diverging control flow (*wasmi-native*). Experiments are conducted with *wasmi* version 0.20. The lower is better.

Next, we consider *wasm-instrument* and *finite-wasm* instrumentation libraries. Figure 8 shows an example of a Wasm program instrumented with *wasm-instrument* and *finite-wasm*. We observe that all these algorithms accumulate the costs over a sequence of instructions, and place the metering instrumentation only occasionally.

[19]There was also a proposal to wrap calls to other Wasm contracts so that they take one more argument (the cost counter) and return one more value (the remaining cost) as described in https://github.com/paritytech/wasm-instrument/issues/11. Essentially, the proposal aimed at defining an ABI for the contracts with metering taken into account.
[20]https://github.com/paritytech/wasmi/tree/master/crates/wasmi/benches
[21]https://github.com/llvm/llvm-test-suite
[22]https://emscripten.org

A)

```
(func $original (result i32)
  i32.const 10
  i32.const 5
  i32.div_s
  i32.const 2
  i32.add
  (block $bb1
    call $foo  ;; returns i32
    br_if $bb1
    call $bar ;; returns nothing
  )
  return
)
```

B)

```
(func $wasm_instrument (result i32)
  call $meter (i64.const 9)
  i32.const 10
  i32.const 5
  i32.div_s
  i32.const 2
  i32.add
  (block $bb1
    call $foo  ;; returns i32
    br_if $bb1
    call $meter (i64.const 1)
    call $bar ;; returns nothing
  )
  return
)
```

C)

```
(func $finite_wasm (result i32)
  call $meter (i64.const 3)
  i32.const 10
  i32.const 5
  i32.div_s
  call $meter (i64.const 4)
  i32.const 2
  i32.add
  (block $bb1
    call $foo  ;; returns i32
    call $meter (i64.const 1)
    br_if $bb1
    call $meter (i64.const 1)
    call $bar ;; returns nothing
  )
  call $meter (i64.const 1)
  return
)
```

Fig. 8. An example of a simple Wasm program, which divides two numbers, adds 2 to the result, calls function `foo`, and conditionally calls function `bar`. From left to right: A) not instrumented, B) instrumented by *wasm-instrument*, and C) instrumented by *finite-wasm*.

We highlight that none of the algorithms is formally defined, or well-documented. As a result, we analyze the code and examples to understand how they work. We observe that *wasm-instrument* tries to accumulate costs for all instructions that are guaranteed to be executed together. Essentially, the algorithm exploits the tree-like structure of Wasm to identify dominance relations between different sections of Wasm code and then places the instrumentation. In contrast, *finite-wasm* instruments all instructions first, and then moves them around to combine costs. The main difference from *wasm-instrument* is that it does not allow for costs to move across instructions with potential side effects, such as calls or division.

Next, we consider the impact of instrumentation on the run time. We again use the same benchmarks from *wasmi* and its runtime, but instrument the Wasm code using *wasm-instrument* and *finite-wasm*. The metering is done using host calls. Figure 9 shows the slowdowns we observe.

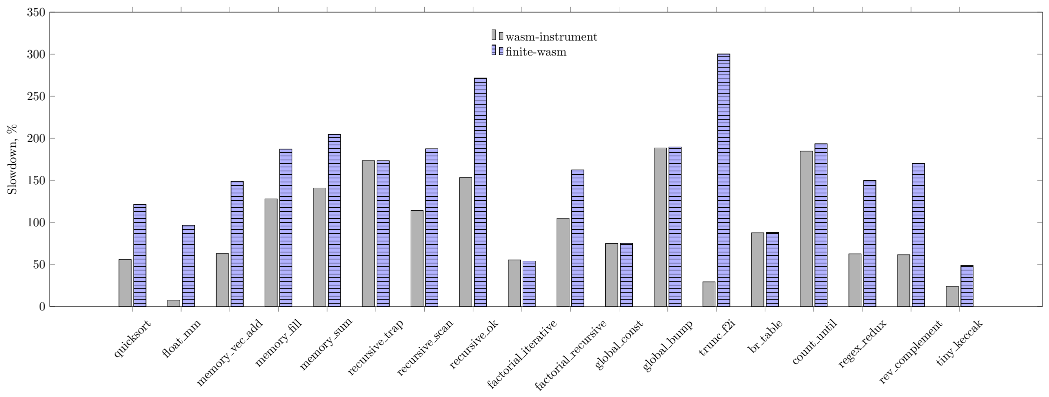
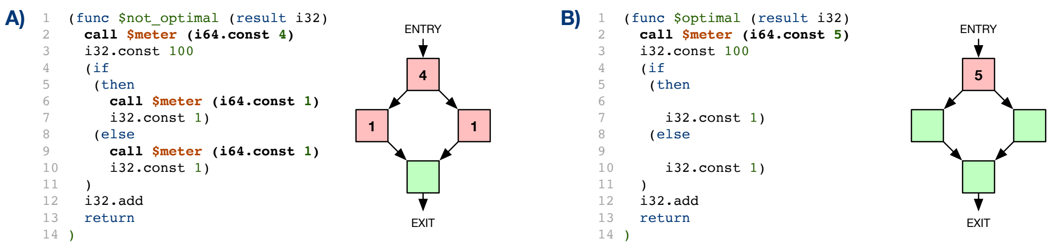


Fig. 9. Slowdowns of Wasm programs when different instrumentation strategies are used (*wasm-instrument* version 0.4.0 and *finite-wasm* version 0.5.0). The lower is better.

We see that instrumentation in Wasm code leads to even higher slowdowns compared to the approach used by *wasmi* (using custom bytecode with special instruction for metering). We believe that the observed run-time overhead is because *wasm-instrument* and *finite-wasm* use host calls for metering.

We also observe that using *finite-wasm* slows down execution 3 – 4× on some of the benchmarks. Because *finite-wasm* instruments on every trapping instruction, `trunc_f32_i` experiences the largest slowdown. We conclude that the state-of-the-art metering of instruction costs in bytecode is not efficient at all.

It is also not hard to see that existing placement of metering instrumentation is not optimal. In Figure 10, the costs of the bodies of `if` and `else` statements are identical, and therefore their instrumentation can be avoided.

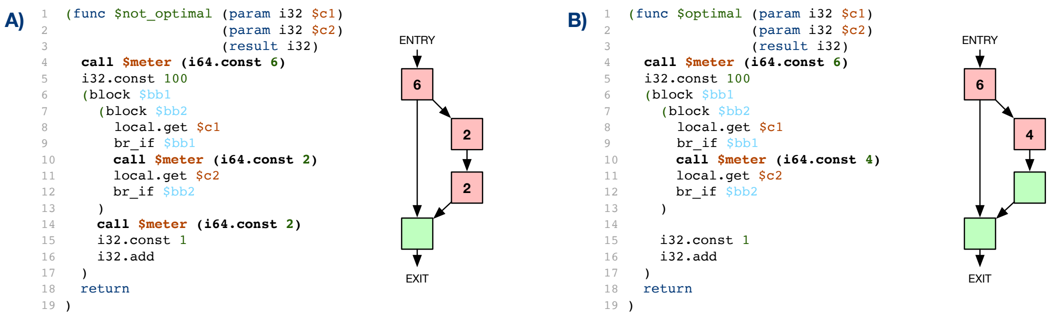


A)
```
1  (func $not_optimal (result i32)
2    call $meter (i64.const 4)
3    i32.const 100
4    (if
5     (then
6       call $meter (i64.const 1)
7       i32.const 1)
8     (else
9       call $meter (i64.const 1)
10      i32.const 1)
11   )
12   i32.add
13   return
14 )
```

B)
```
1  (func $optimal (result i32)
2    call $meter (i64.const 5)
3    i32.const 100
4    (if
5     (then
6
7       i32.const 1)
8     (else
9
10      i32.const 1)
11   )
12   i32.add
13   return
14 )
```

Fig. 10. Example of non-optimal metering instrumentation in *wasm-instrument* for a simple program which computes 100+1 = 101: A) the original instrumented program, B) program with more efficient instrumentation placement. In both cases, we also show the control-flow graphs with basic blocks with costs highlighted in red, and basic blocks without instrumentation highlighted in green.

In Figure 11 we observe that the algorithms do not take information about dominance into account. Here, if the first jump to $a is taken, the cost is indeed 6 for the execution. But, the second branch is taken all the time, which means that instructions at lines 15-16 are always executed if that path is taken. As a result, having one metering call is sufficient.



A)
```
1  (func $not_optimal (param i32 $c1)
2                     (param i32 $c2)
3                     (result i32)
4    call $meter (i64.const 6)
5    i32.const 100
6    (block $bb1
7      (block $bb2
8        local.get $c1
9        br_if $bb1
10       call $meter (i64.const 2)
11       local.get $c2
12       br_if $bb2
13     )
14     call $meter (i64.const 2)
15     i32.const 1
16     i32.add
17   )
18   return
19 )
```

B)
```
1  (func $optimal (param i32 $c1)
2                 (param i32 $c2)
3                 (result i32)
4    call $meter (i64.const 6)
5    i32.const 100
6    (block $bb1
7      (block $bb2
8        local.get $c1
9        br_if $bb1
10       call $meter (i64.const 4)
11       local.get $c2
12       br_if $bb2
13     )
14
15     i32.const 1
16     i32.add
17   )
18   return
19 )
```

Fig. 11. Example of non-optimal metering instrumentation in *wasm-instrument* because the algorithm does not use the dominance information. The program takes two boolean constants (c1 and c2) as inputs, which are used to jump between different blocks. Importantly, the branch on line 12 jumps to the same location (line 14) whether taken or not. A) The instrumentation placed does not account for lines 11-12 dominating lines 15-16. B) Better placement of instrumentation which leverages the dominance relation.

The inefficiency of instruction cost metering directly impacts the revenues of validators. We observe that branching instructions, blocks and loops – all require metering. Slowing down the contract by 2× means that revenues of validators are also 2× lower than they could have been.

While performance is important, the security of the network and the correctness of smart contracts have always been a priority. As already seen, *finite-wasm* conservatively places instrumentation for any instruction that may trap or produce side effects, sacrificing execution speed. Indeed, implementing an algorithm to place the metering instrumentation correctly is not trivial.

For example, Wasmer [13], a well-known Wasm runtime, has its own implementation of instruction cost metering. During our research, we identified multiple bugs in the implementation,[23] which makes Wasmer's metering unusable in practice.

An example of a malicious Wasm program exploiting Wasmer's implementation is shown in Figure 12. The problem with the existing implementation is that it does not account for implicit control flow when an instruction traps.

```
(func $bomb (param i32 $a)
            (param i32 $b)
            (result i32)
  local.get $a
  local.get $b
  i32.div_u

  return
)
```

```
(func $bomb_instrumented (param i32 $a)
                         (param i32 $b)
                         (result i32)
  local.ge $a
  local.ge $b
  i32.div_u   ;; division by 0 traps!
  call $meter (i64.const 4)
  return
)
```

Fig. 12. Example of an attack on Wasmer instruction cost metering. On the left, an original malicious program. On the right, the same program but with metering instrumentation. When b is zero, division traps, but the instruction cost is not yet metered.

Such a program can be used for DoS attacks on a blockchain. The attacker can craft a very large contract with no control flow and ending in a division by zero. Then, the contract is called many times. As a result, validators will be executing a large amount of instructions, but the attacker will pay nothing (or a small base fee only).

It is crucial to mention that implicit control flow has further implications than just security. If there is a trap during program execution, then the existing algorithms for placing the metering instrumentation meter either larger costs (e.g., *wasm-instrument*, *finite-wasm*) or smaller (e.g., Wasmer). This has two consequences.

First, it is undesirable for users because they can be charged for non-executed instructions if there is a trap during execution. However, one can argue that such design incentivizes writing good and well-tested smart contract code.

Second, it creates a problem of reaching an agreement at a consensus level. Honest validators running VMs with different algorithms for instrumentation placement can produce different outputs. For example, a contract can trap for one validator but run out of budget for the other. We note that this problem can be avoided by making the metering instrumentation placement algorithm part of the VM specification.

We conclude that state-of-the-art instruction cost metering in bytecode is not yet efficient, nor secure:

- Existing algorithms do not take into account cost and dominance information, which can result in high metering overheads, indirectly affecting the liveness of a blockchain network and the profits of validators.
- Existing algorithms are either too conservative in the presence of trapping instructions (e.g., *finite-wasm*) which causes significant run time overhead, or simply do not take it into account, which is a security vulnerability for the blockchain.

[23] https://github.com/wasmerio/wasmer/issues/4219 and https://github.com/wasmerio/wasmer/issues/4256

- Metering costs are not always accurate and differ for various implementations. It can affect transaction fees for users, as well as introduce challenges for the consensus.

4.1.3 Instruction cost metering in native code. Finally, we consider how to meter instruction costs if compiling to native code. To our knowledge, Solana is the only blockchain that has its own JIT compiler (adapted from eBPF JIT compiler, but a substantial amount of work has been added), and therefore faces the problem of generating the metering instrumentation together with native code.

Table 21 describes how the counter for metering can be implemented in native code. We observe that storing the counter in a register is preferred, as it avoids memory access instructions unless in-memory increments are supported. As we have already seen, the instrumentation only needs two assembly instructions (subtraction and comparison). Interestingly, Solana uses an unsigned one, which requires three assembly instructions.

Table 21. Approaches for metering instrumentation in native code.

In-register counter	In-memory counter
Fix a register for accumulating the cost of a running program. If selected carefully, the calling convention allows the use of the register as an argument to the function, essentially defining an ABI for the metering. As a result, metering can be done across function and contract calls. Solana fixes `rdi` register during code generation.	Store the cost counter in memory. Can still be efficient on x86 platforms which allow adding a 32-bit immediate to a a 64-bit variable stored in memory. [24] As we have discussed, the total instruction cost can never overflow 32 bits.

Similarly to Wasm-based VMs, for JIT compilation it is important to minimize the amount of instrumentation in the generated code. To place instructions optimally, Solana uses a very interesting approach which uses the fact that every SBF instruction costs 1 computational unit. In this setting, the cost of executed instructions can be rewritten as the difference between the original value of a virtual program counter (PC) and its final value. During code generation, on every branch/call, the current virtual PC is subtracted from the counter (which marks the end of the metering section), and the target virtual PC is added to the counter (which marks the start of the next metering section). In order to avoid long unmetered sequences of instructions, metering instrumentation is also inserted every 10K SBF instructions. Figure 13 shows some intuitive examples of how metering works.

In order to see the effect of instrumentation in native code, we consider the ten most used Solana contracts, according to *solscan*. [25] We observe that the emitted instrumentation increases the code size by at least 30% (Figure 14). Large code size can negatively impact performance, e.g., more instruction cache misses can happen, or even page faults.

However, the code size on its own does not show how the performance of a contract is affected. We additionally calculate the number of x86 instructions generated for metering each contract and compare it with the number of basic blocks in the control flow graph of the same contract. As discussed before, Solana uses comparison and a conditional branch to check that the current cost is within the limits, subtraction to meter it, and addition to undo the metering for one of the targets of a conditional branch.

[24] https://www.felixcloutier.com/x86/add
[25] https://solscan.io

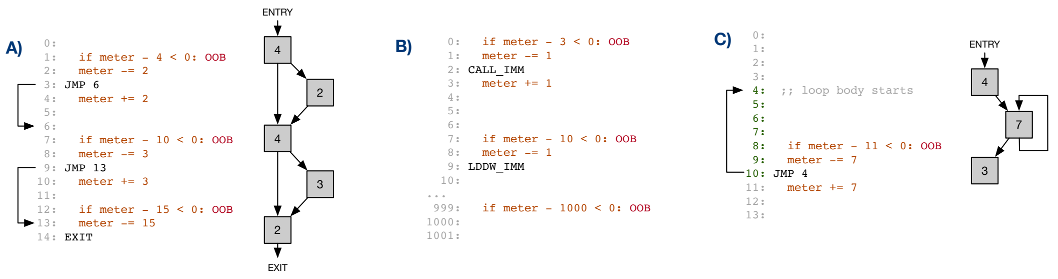

A)
```
 0:
 1:   if meter - 4 < 0: OOB
 2:   meter -= 2
 3: JMP 6
 4:   meter += 2
 5:
 6:
 7:   if meter - 10 < 0: OOB
 8:   meter -= 3
 9: JMP 13
10:   meter += 3
11:
12:   if meter - 15 < 0: OOB
13:   meter -= 15
14: EXIT
```
B)
```
   0:   if meter - 3 < 0: OOB
   1:   meter -= 1
   2: CALL_IMM
   3:   meter += 1
   4:
   5:
   6:
   7:   if meter - 10 < 0: OOB
   8:   meter -= 1
   9: LDDW_IMM
  10:
...
 999:   if meter - 1000 < 0: OOB
1000:
1001:
```
C)
```
 0:
 1:
 2:
 3:
 4:  ;; loop body starts
 5:
 6:
 7:
 8:   if meter - 11 < 0: OOB
 9:   meter -= 7
10: JMP 4
11:   meter += 7
12:
13:
```

Fig. 13. Example of metering (pseudocode in red) added by RBPF's JIT compiler. Here, OOB represents a label to the error handler. A) Example of a program with control flow, with its CFG shown in the middle. The CFG is annotated with instruction costs. We only show jump instructions to make it more readable. B) Example of metering for calls, the `lddw` instruction, and when not metered instruction sequences become large. C) A do-while loop.

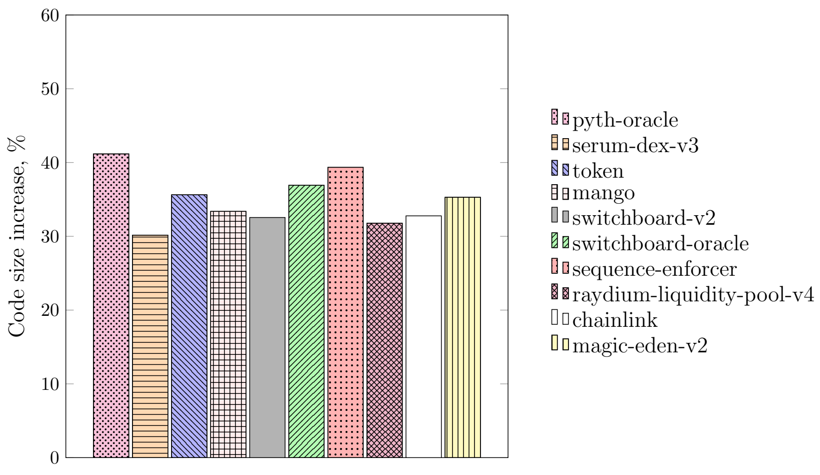


Fig. 14. Increase in the code size (generated x86) of metered Solana contracts. Contracts are ordered based on their popularity, from the most popular (on the left) to the least (on the right). RBPF version 0.7.1 is used.

The results are shown in Figure 15. We observe that there are on average 3-4 x86 instructions generated for each basic block for metering. This fact is particularly important because it highlights that the algorithm that produces the instrumentation is essentially instrumenting every basic block.

Solana uses instruction costs only to limit the computation, and not to calculate the execution cost. Still, there can be attacks on the liveness of the blockchain network by exploiting the instrumentation placement.

For example, the first version of SBF bytecode supports the `lddw` (load) instruction, which takes two words instead of one, unlike the other instructions. The instruction is deprecated in the most recent version of the bytecode but still exists on-chain. For example, between 2% and 4% of instructions used by the ten most popular Solana contracts are `lddw`.

The issue with `lddw` is that it occupies two words which breaks the assumptions of the metering algorithm used by Solana. In order to make the metering correct the developers decided to add metering instrumentation immediately before every `lddw` occurrence. It is then possible to create

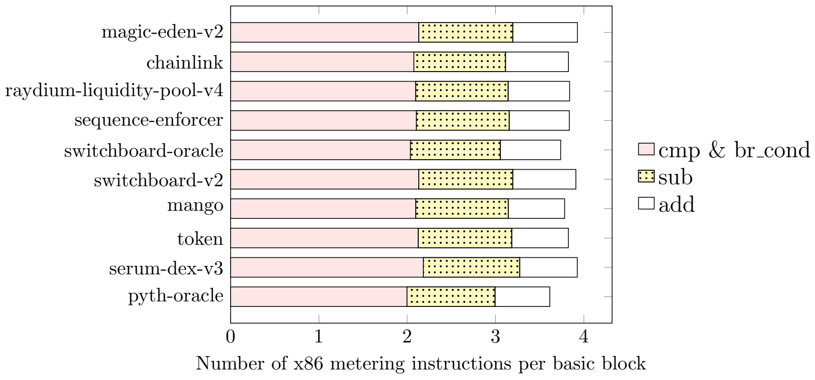


Fig. 15. The ratio between the number of x86 instructions produced for metering instrumentation and the number of basic blocks in a contract. RBPF version 0.7.1 is used.

an adversarial program consisting of only these load instructions, which based on our experiments can lead to 2× − 5× slower execution of such programs.

We conclude that low-overhead metering in native code is very important, as otherwise the code size of a metered program doubles. It is crucial that the implementation of such an algorithm accounts for corner cases (like `lddw`) in a general way, to avoid performance regressions.

4.2 Memory and storage costs metering

Memory models are always taken into consideration when designing a cost model. We saw in Section 3 that blockchains usually associate a per-byte or per-page cost to account for the memory usage, as well as cap it at a certain value. Under the existing cost models, the goal of memory metering is to ensure that the contract does not use excessive amounts of memory and that the cost of memory allocations (which are typically expensive) are recorded.

Metering memory allocations depends on how they are done. Table 22 describes two ways: static memory allocation performed in advance, and dynamic memory allocation which happens at run time.

Table 22. Different ways of allocating memory, and their impact on metering.

Static memory allocation	Dynamic memory allocation
No metering is required because memory is pre-allocated based on the client-requested size and is therefore pre-paid. Used by RBPF.	The amount of allocated bytes or pages has to be tracked for each allocation, and it is used to calculate the final cost. Used by EVM and NearVM.

We observe that static memory allocation, used by Solana, simplifies metering a lot, allowing the RBPF's JIT compiler to avoid injecting any additional instrumentation. For the rest of the section, we therefore focus on VMs where memory is allocated dynamically.

4.2.1 *Metering memory in software.*

EVM implementations such as *geth* or *evmone* meter memory usage in software, just like the instruction costs. Again, we observe that for the selected interpreters, the overhead of memory metering is invisible, similar to the instruction cost metering we described before. Indeed, memory allocation can take from hundreds to thousands of instructions, so a few instructions added by metering are negligible.

4.2.2 Metering memory in bytecode or native code. In order to meter memory in bytecode or native code, instrumentation has to be injected into the contract's code. For Wasm, this accounts for instrumenting every occurrence of `memory_grow` with costs proportional to the number of allocated pages. We observe that the additional instrumentation is not a real bottleneck because the time taken to allocate memory still dominates the processing time for extra instructions.

It is interesting to see how the semantics of the language can affect memory metering. In EVM, we recall that memory expansion is not decoupled from memory access. A write to a new memory location results in memory expansion and incurs a cost. This means that if Ethereum contracts are JIT compiled, every memory access has to be metered to adhere to the semantics of the memory model.

As a result, memory metering for Ethereum contracts, if done in bytecode or native code, can have a significant impact on performance. In this way, the approach used by Wasm is better: only explicit allocations have to be metered. However, we must say that having explicit `memory_grow` instructions shifts the job to the compiler or a contract developer: they have to generate these instructions, which is nearly impossible to do optimally.

Finally, we note that the size of allocated memory is usually dynamic, and therefore it is not possible to optimize the placement of instrumentation to meter it. However, having fewer allocations is likely to minimize the metering overhead, if any. This way, the problem of metering is shifted to the memory allocator.

4.2.3 Metering storage. For storage metering, the same concept as for memory metering can be used. For example, on the Aptos blockchain loading a resource from storage has two costs: 1) to load the bytes into memory, and 2) to serialize the bytes into internal in-memory representation which is used by MoveVM. In particular, every time there is storage access, the VM has to meter the cost for the fetched data. We observe that this metering is identical to the metering for memory allocations or expansion.

4.3 Summary

Existing cost models for blockchains either require a flat fee for execution or use per-instruction costs. In the latter case, instruction costs have to be metered at run time. The metering has to be implemented correctly and efficiently to yield deterministic and accurate costs, to not open an attack vector on the network, or to not degrade the performance.

We learnt that state-of-the-art runtimes used by blockchains are mostly simple, and the straightforward metering implementations in existing VMs achieve both correctness and do not have a visible performance overhead.

However, for more advanced VMs that use JIT compilation or leverage existing performant infrastructures metering instruction costs can become a problem. Particularly, straightforward metering implementations can slow down the execution by 2×.

We presented a detailed comparison of how multiple VMs used by state-of-the-art blockchain meter run-time instruction costs. We identified that existing metering schemes are not ideal:

- they can cause significant run-time regressions in existing systems;
- they can lead to a significant increase in the code size, sometimes more than 30%;
- they lack any formal descriptions, which sometimes leads to incorrect implementations, and allows malicious users to halt the network for some time.

Lastly, we briefly considered how other resources (memory and storage) are metered. In particular, we observed that instead of dynamic metering there are chains that statically pre-allocate resources. There are pros and cons for both solutions.

5 SMART CONTRACTS ANALYSIS & STATISTICS

In this section, we investigate the structure of popular smart contracts deployed on different chains. The goal is to understand the real execution behaviour, in particular:

- how often the execution of contracts fails, and why,
- what bytecode instructions are most commonly used,
- how does the control flow of contracts look like, etc.

The behaviour and the structure of real contracts can help us to understand the requirements for the cost model and the metering. As a result, we can develop more efficient metering schemes, optimized for the common case based on the gathered statistics.

5.1 Frequency of transaction failures

As we have already seen, metering is an important defense mechanism against DoS attacks. It is interesting to investigate how many contracts violate the run-time limits enforced by the cost model or are submitted with small costs insufficient to cover execution. Additionally, it is worth analyzing what proportion of contracts terminate abnormally in general.

We first look into the Aptos blockchain and analyze all transactions submitted to it between 01.01.23 and 31.05.23 (in total of 5 months). These are the transactions starting at version 60641618 and ending at version 151268548 (inclusive).

We observe that on Aptos for the selected period, 93.0% of transactions are successful, as shown in Figure 16. From the failed transactions, 2.5% run out of gas, which contributes only 0.2% when compared to the total number of transactions. Such an insignificant fraction highlights that out-of-gas exceptions are rare and are not frequently encountered by regular users.

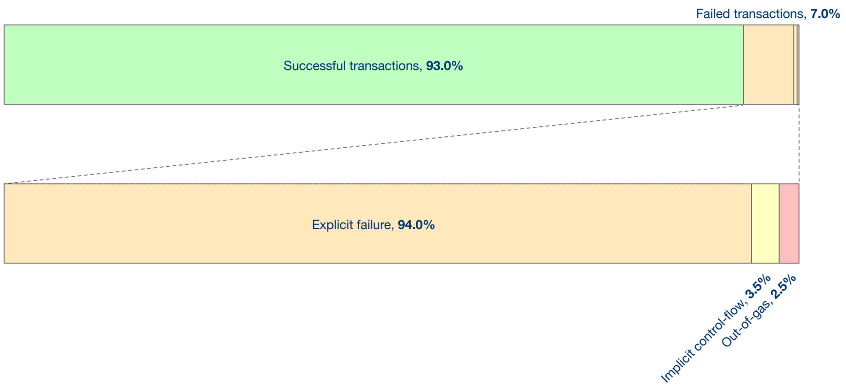


Fig. 16. The distribution of successful and failed transactions on Aptos over the selected time interval.

In addition, we investigate how many transactions failed due to implicit control flow leading to an exception (e.g., arithmetic underflow or stack overflow). We find that only 3.5% of failed transactions did not explicitly abort. This means that 94.0% of transactions explicitly failed, by going into the basic block with `abort` instruction.

While we have not been able to gather such detailed data for other blockchains, we still observe a very similar trend – failures are rare. On Solana, the most popular contracts have a relatively high success rate, around 80-90% for the five most popular contracts, as shown in Table 23. These contracts account for more than 50% of traffic on Solana.

[26]https://analytics.solscan.io/overview

Table 23. Success rates of popular Solana programs, based on *solscan* analytics tool. [26] Programs are ordered by their popularity from left to right. The data is based on 04.10.2023.

pyth-oracle	serum-dex-v3	token	mango	switchboard-v2
84.00%	84.85%	82.35%	80.91%	98.33%

At the same time, on Ethereum, a recent study [58] showed that running out of gas is not common either, and if the execution is aborted, it is done with special instructions, e.g., `REVERT`. The study suggests that from all selected transactions that did not terminate successfully, only 20% were due to out-of-gas exceptions. The main causes for these exceptions are misunderstanding the transaction mechanism and low gas limits.

We conclude that the execution of smart contracts is predictable from the first glance. Exceeding metering limits has very low probability and primarily occurs due to 1) clients not knowing how to estimate the resource or gas consumption, and 2) occasional adversarial behaviour. While it is important to ensure that resources are metered correctly for all possible executions, the gathered data suggests that optimizing the metering for successful execution is reasonable.

5.2 Distribution of instruction opcodes

Next, we investigate how often the different kinds of instructions are used in smart contracts.

5.2.1 Distribution of instructions on Aptos. We analyze all existing contracts on the Aptos blockchain using the recent version 280140423 (which corresponds to 30.09.23). In total, there are 10.8K contracts with 67.9K functions and 2.6M bytecode instructions. On average, a single function has only 38.9 instructions.

As a result, we analyze the distribution of different Move opcodes globally, i.e., for all contracts together, rather than focusing on specific functions. Otherwise, due to the small number of bytecode instructions per function, the statistics we gain can be not that insightful.

We show the distribution of different kinds of opcodes on Aptos in Table 24. We observe that the fraction of all storage instructions combined is smaller than 2.4%, which means that, on average, there is a single global storage access per function. Because calculating the cost of the global instruction is not common, most of the instructions in Move contracts have statically known costs.

Table 24. Distribution of different opcodes for all Move modules stored on Aptos on 30.09.23. We consider global storage instructions (checking existence, loading or storing data), control-flow instructions (calls, branches, and explicit aborts), and simple instructions (addition, casts, constants).

Total	Global existence, %	Global loads, %	Global stores, %	Calls, %	Branches, %	Aborts, %	Simple, %
2643004	0.46	0.52	1.32	12.10	7.87	1.97	16.29

Another insight from the data is that function calls are 1.5× more common than branching instructions. On one side, this highlights that the code on Aptos is very modular. At the same time, the Move bytecode stored on Aptos is not optimized, and it is possible that many opportunities where a compiler should inline the call are unused. For example, in Figure 17, we show a snippet of Move code stored on-chain that clearly benefits from inlining.

Finally, we observe that the high-level nature of the Move bytecode significantly impacts the distribution of opcodes. In Move, structs and vectors are first-class citizens of the bytecode, and

```
public exists_at(arg0: address): bool {
bb1:
  MoveLoc[0](arg0: address)
  Exists[0](Account)
  Ret
}
```

Fig. 17. An example of a Move function that should be inlined by the compiler, but is currently not. The function checks if Account resource exists at a particular address. If inlined, a single bytecode instruction is used to check for the existence of the resource. If not inlined, significantly more instructions are executed: passing the arguments to the call, setting up a new function frame, etc.

therefore there is a variety of instructions to access their fields and elements. Based on the data presented in Table 24, we deduce that these instructions must account for up to 60% of the total number of instructions.

5.2.2 Distribution of instructions on Solana. Next, we look at different kinds of instructions that exist in Solana contracts. Unlike Aptos, Solana uses low-level SBF bytecode, and so contracts contain thousands of instructions. We therefore consider only the ten most popular contracts for our analysis, presented in Table 25.

Table 25. Distribution of different opcodes for the ten most popular Solana contracts (ordered in their popularity from top to bottom). We consider arithmetic instructions (addition, multiplication, shift, etc.), memory instructions (load/store), and control-flow instructions (call/jump).

Program	Total	Arithmetic, %	Loads, %	Stores, %	Calls, %	Jumps, %
pyth-oracle	9122	42.7	21.3	12.8	5.5	16.9
serum-dex-v3	24439	37.3	26.3	18.1	4.8	12.8
token	13597	40.7	21.6	16.0	6.2	14.5
mango	15899	37.4	22.5	19.4	5.5	14.2
switchboard-v2	176762	49.2	16.9	14.3	10.6	7.9
switchboard-oracle	66054	42.6	19.4	15.6	9.1	11.8
sequence-enforcer	20338	46.1	17.5	13.7	9.7	11.0
raydium-liquidity-pool-v4	74516	40.5	23.2	16.5	7.2	12.1
chainlink	57388	43.3	20.0	16.5	8.0	11.4
magic-eden-v2	101564	50.5	15.8	12.7	11.9	7.6

First, we see that Solana contracts are large, sometimes comprised of more than 100K SBF instructions. We observe that nearly half of the instructions in all contracts are simple arithmetic, and a third are memory accesses. It is worth mentioning that because there is no difference between memory and storage in SBF bytecode, it is very challenging to understand the real access patterns.

We also observe that while the number of control-flow instructions varies from contract to contract, on average, on average 19.9% of instructions are responsible for the control flow. We note that this number is comparable with the average proportion of control-flow instructions on Aptos, which is 21.8% on average.

5.2.3 *Distribution of instructions on Ethereum.* Lastly, we look at Ethereum and the distribution of different EVM opcodes there. Luckily, there has been some prior research done in that area. Bistarelli et. al. [33] show that on Ethereum stack manipulation opcodes and stores to memory (`MSTORE`) are the most popular opcodes. The reason why `MSTORE` is so popular is potentially due to Solidity compilers: when a Solidity contract is compiled, a scratch space is always allocated in memory. Moreover, the authors claim that all contracts access storage and cross-contract calls occur in 11% of all contracts. At the same time, the study does not account for different types of instructions, e.g., if it is arithmetic or control flow, which makes it less useful for our purposes.

5.2.4 *Summary.* To sum up, we looked at the distribution of different kinds of bytecode instructions in multiple blockchains, focusing closely on Aptos, Solana, and Ethereum networks. We observed that storage accesses are not common in contracts (for chains where they can be distinguished from memory accesses). This means metering does not have to process variable instruction costs often. Predominantly, simple instructions such as arithmetic are used with statically known costs.

We also discovered that performing such an analysis is a non-trivial task, as one has to account for the semantics of the bytecode of interest, whether it preserves the domain-specific information (e.g., storage access), or missed optimizations in the compiler, amongst other things. Moreover, the difference in high- and low-level instruction sets makes it harder to compare the distribution of a particular opcode across multiple chains.

5.3 Control-flow structure

As discussed above, analyzing what kinds of instructions are used on-chain is hard. Here, we analyze the control flow of the contracts instead. The motivation is that 1) the control flow is chain-agnostic, which allows one to compare contracts on different blockchains better, and 2) the control flow directly impacts the metering of instruction costs.

We note that control-flow analysis is also not trivial in existing systems. For example, on Ethereum, due to the bytecode structure it is hard to distinguish between *if-else* or *while* statements [33].

Furthermore, it can be somewhat meaningless to analyze the control flow if contracts are not optimized, or if functions have small code sizes and use relatively high-level bytecode. This is particularly the case for Aptos, where functions, on average, are under 40 bytecode instructions. As a result, function calls and the call graph hide the complexities of the control-flow graphs of Move contracts.

Therefore, we consider Solana contracts to analyze the control flow in smart contracts. We note that the SBF bytecode is also very close to the native code and hence can give a good estimation of how the control flows through the contract when it is executed by a validator. Our analysis is summarized in Table 26.

First, we observe that the size of a basic block on Solana is around 4 SBF instructions only. Considering the metering instrumentation, this is very significant. Because SBF instructions almost always map one-to-one to native assembly, adding extra instrumentation per block would result in up to 2× code increase. We note that this conforms with what we have observed while studying the metering scheme used by Solana in Section 4.

Table 26. Statistics about the control-flow structure of Solana contracts. Here, V is the set of basic blocks, and E is the set of edges. By $\overline{b}$, we denote the average number of instructions per basic block. We use $E_c \subseteq E$ and $E_b \subseteq E$ to describe critical- and back-edges, respectively. Lastly, V_l and D are the sets of leaves and diamonds.

Program	$\lvert V\rvert$	$\overline{b}$	$\lvert E\rvert$	$\frac{\lvert E_c\rvert}{\lvert E\rvert}$ (%)	$\frac{\lvert E_b\rvert}{\lvert E\rvert}$ (%)	$\frac{\lvert V_l\rvert}{\lvert V\rvert}$ (%)	$\lvert D\rvert$
pyth-oracle	2456	3.7	3318	21.7	0.8	7.3	52
serum-dex-v3	5116	4.8	6859	20.8	0.4	8.2	129
token	3263	4.2	4274	19.4	0.4	7.9	82
mango	3758	4.2	4965	18.6	0.4	8.5	97
switchboard-v2	36902	4.8	42204	8.4	0.1	6.6	265
switchboard-oracle	16125	4.1	19855	17.2	0.7	8.2	206
sequence-enforcer	5032	4.0	5931	15.9	0.4	11.1	111
raydium-liquidity-pool-v4	16710	4.5	21870	18.6	0.2	7.0	250
chainlink	12771	4.5	16428	18.3	0.3	6.8	220
magic-eden-v2	22477	4.5	25115	7.8	0.2	8.0	186

Second, it is interesting that programs have many critical edges. An edge is critical if its source has multiple successors and its destination has multiple predecessors. We conclude that contracts must have been often using conditionals similar to *if* statements. In contrast, all contracts have a low fraction of edges which are back-edges. Back-edges are edges of the control-flow graph that are not part of the Depth-First-Search traversal of the graph. In particular, back-edges allow one to find the lower bound on the number of cycles in the graph. We conclude that loops are not common in popular Solana contracts.

Finally, we analyze how many leaves and diamonds control-flow graphs contain. Leaves are vertices with no successors. In the control-flow graph, these vertices signify termination of execution. Diamonds are simple graphs on four vertices, where one vertex has two successors and one vertex has two predecessors.

We find that on average 8.0% of all vertices are leaves. Because the contracts we study define many functions, the graph is a collection of strongly connected components, and so there has to be at least one leaf for each function, which can explain these numbers. At the same time, we observe that diamonds are not common, and so the control flow rarely diverges and re-converges almost immediately after.

Lastly, we discuss the control-flow structure of Ethereum contracts, again based on the study by Bistarelli et. al. Instead of EVM bytecode, they analyzed the source code of contracts. The authors claim that *for* and *while* loops occur in 24% and 6% of Solidity contracts they analyzed, respectively. Hence, about a third of all Ethereum smart contracts have at least one loop. Conditionals, such as *if-else*, are used by most of the Ethereum contracts.

In conclusion, we looked at the control-flow structure of smart contracts, predominantly focusing on Solana. We observed that contracts have a large number of basic blocks, which are small, on average. This can negatively impact the running time and the code size if contracts are not metered efficiently. We also observe that loops are not common in smart contracts. Instead, there are many critical edges (particularly in Solana contracts), which is significant for metering as we will see in the later sections.

5.4 Summary

Blockchains are yet to find the right set of applications they are useful for. To this day, contracts are mainly related to economic activity: decentralized exchanges, trading, transfers, swaps, and other similar workloads. All these tasks do not require complicated control flow, nor numerous loops to do an expensive computation, which is evidenced by the data we obtained when looking at the distribution of instruction opcodes and control-flow graphs of contracts.

While control flow is not complicated, it is still widely used by the contracts, predominantly via conditionals such as *if-else* statements. This means that smart contracts can have many basic blocks, so efficient and more optimal metering schemes are needed.

Lastly, the execution behaviour of smart contracts is predictable: in general, a contract is executed successfully within a budget. Statistics show that exceptions due to implicit control flow or due to running out of budget are rare. This means that existing metering schemes can be improved if optimized to be efficient for smart contracts that terminate successfully.

6 MINIMUM SAFE METERING INSTRUMENTATION

In Section 4 we have observed that the existing approaches to metering are not always efficient, in particular when the metering instrumentation is injected into the smart contract. Moreover, placing the metering instrumentation carelessly can create security vulnerabilities, like we have seen with Wasmer's instrumentation, and can affect the liveness of the blockchain network.

In this section, we formally develop an efficient algorithm for computing the placement of the metering instrumentation for a given program, which is also safe. The goal of this algorithm is to reduce the run-time overhead of instrumentation, as well as to ensure safety – the program has to terminate within the allowed limit. Moreover, our goal is to design an algorithm that can be applied to arbitrary instruction sets with fixed costs, so that they are not bound to particular semantics such as structured control flow in Wasm.

The algorithm should also have a linear or sub-linear complexity in the number of basic blocks in the control-flow graph of a program. Such a requirement is necessary because the instrumentation is computed at run time by validators executing the code. For example, if the complexity is quadratic, the algorithm can become an attack vector and can be exploited by a malicious program.

The algorithm we present can be used for any metering scheme that tracks the cost of instructions, e.g., gas costs on Ethereum, NEAR, and Aptos, or computational units on Solana. We consider a setting where instruction costs are statically known as opposed to variable or dynamic costs. Such a setting is sufficient for the following reasons:

(1) As observed in Section 3, many popular blockchains use a fixed cost when pricing instructions. Non-constant costs associated with memory expansion (in Ethereum or NEAR cost models) or the cost of accessing a resource in Move can sometimes be computed at compile time. Figure 18 shows such an example. Moreover, there are cost models like the one used by Solana which pre-allocate resources to avoid dynamic costs.

(2) Lastly, consider a cost that depends on some expressions $e_1, \dots, e_n$ defined by the program. First, we observe that the metering instrumentation for this cost cannot be moved before the definition of any of the expressions $\{e_i\}_{i=1}^n$. If the metering is placed significantly after the definitions, then $\{e_i\}_{i=1}^n$ have to stay live until the metering instrumentation. Such a setting is usually considered to have poor performance. For example, in native code, this creates higher register pressure (there should be a register to hold every variable the dynamic cost depends on) or creates unnecessary spilling to memory. As a result, placing the metering code immediately after the definitions of $\{e_i\}_{i=1}^n$ might be preferable. In any case, there is no straightforward solution, and doing more work in non-linear time is not needed.

```
 1  use std::signer;
 2
 3  struct Foo has key {
 4    value: u64,
 5  }
 6
 7  public fun foo(user: signer): u64 acquires Foo {
 8    let addr = signer::address_of(user);
 9    let foo = borrow_global<Foo>(addr);
10    return *foo.value
11  }
```

```
12 // The cost of global borrow (line 9) can be
13 // calculated according to the formula:
14 //   cost = base_cost + sizeof(T) * cost_per_byte
15 //
16 // Assume base_cost = 100, cost_per_byte = 10.
17 //   cost = 100 + sizeof(T) * 10
18 //        = 100 + 8 * 10
19 //        = 180
20 //
21 // because the size of T is statically known to be
22 // 8 bytes.
```

```
23 // Size is statically known for any instantiation.
24 struct Bar<phantom T> has key {
25   value: u64,
26 }
25 // Sizes can change or depend on instantiation.
28 struct Fizz<T: store> has key {
29   value: T,
30 }
31 struct Bazz has key {
32   value: vector<u64>,
33 }
```

Fig. 18. On the left, an example of Move code where the non-constant cost for resource Foo is not a problem for the metering and can be computed by the compiler. On the right, examples of Move resources that have statically known or unknown costs.

The rest of this section is organized as follows. First, we define the properties that any metering instrumentation placement should satisfy. We then consider a simple approach of placing metering instrumentation at the start of each basic block in a program and study how such a placement can be implemented efficiently and robustly even in the presence of exceptions. We then describe an algorithm that computes a more optimal placement for metering instrumentation by analyzing a program's control-flow graph. We also present the algorithmic framework we have used, along with the proofs of correctness.

6.1 Properties of metering instrumentation

It is crucial to define a set of properties that any algorithm for placing metering instrumentation has to enforce. Before describing the properties, it is important to mention that we assume that the algorithms cannot change the instructions of the original program. In particular, this guarantees that the cost of execution of a user program cannot suddenly change at run time. The algorithms can only add instrumentation to meter the instruction costs, as well as insert new basic blocks but containing the instrumentation only.

Informally, there are three properties the metering instrumentation must satisfy:

- Validity: We want the instrumentation not to change the behaviour of programs that terminate successfully.
- Consistency: We may want to make sure that the execution is correct even if it fails. For example, if the program results in an arithmetic error, the instrumented program should not suddenly return an error saying it run out of budget.
- Safety: we want to make sure no work done by the runtime goes unpaid. Safety ensures we meter resources on time.

We now develop more formal definitions of these properties. Before that, we have to cover some preliminary basics.

In this section, we use a mathematical model to reason about instructions and their costs. We let $\mathcal{I}$ be the set of instructions and let $c : \mathcal{I} \longrightarrow \mathbb{Z}_{>0}$ be a cost function. Recall that we assume that all instructions have a fixed cost. Without loss of generality, we can say that the costs are always positive.

The programs we consider are organized as control-flow graphs (CFGs) where instructions belong to basic blocks. A basic block is a sequence of instructions without explicit control flow. The cost of instructions is calculated using the cost function c. For any instruction sequence, its cost is the sum of the costs of instructions in the sequence. For simplicity of notation, we use the same cost function c when talking about the cost of instruction sequences, to avoid using summations.

$$c(I) = \sum_{i \in I} c(i), \text{ for a sequence of instructions } I.$$

We consider an arbitrary cost model C. Let $\mathcal{A}_C$ be an algorithm that computes the placement of the metering instrumentation under C. For any program P, we define $\mathcal{A}_C(P)$ as the metered version of the program after running the instrumentation algorithm. By P_C we denote the ground truth – a metered instance of P instrumented under the specification of the cost model C. A simple way to obtain P_C is to place instrumentation before each instruction in P.

We consider that any program is executed given a budget $B > 0$ and the current global state of the blockchain, σ. The execution of a program updates the budget based on the costs of the executed instructions and can also modify the state. We call the sequence of instructions T executed with budget B and at the state σ a trace $T_{\sigma,B}$. The result of any metered execution, if it terminates, is a triple (S, B', σ') where S is the status of execution (described in Definition 6.1), B' is the final budget, and σ' is the final global state of the blockchain.

Definition 6.1. *Consider a metered instance of an arbitrary program P under the cost model C, P_C. Let its execution be a transition from (B, σ) to (S, B', σ').*

We say the program execution has status $\mathcal{T}$ if it terminates successfully under the provided budget and cost model. More formally, if non-metered P transitions the global state from σ to σ' using a trace of instructions T, then the following holds. If $B \geq c(T)$, then the execution status is $S = \mathcal{T}$, the final budget is $B' = B - c(T)$, and the final state is σ'.

We say the program execution has status $\mathcal{F}_E$ if it terminates with an exception E originating from one of the executed instructions. Formally, assume non-metered P keeps the global state σ because executing a trace of instructions T results in exception E. Then if $B \geq c(T)$, the execution status is $S = \mathcal{F}_E$, the final budget is $B' = B - c(T)$, and the final state is still σ. Moreover, the same exception E is still produced.

Finally, the program execution has status $\mathcal{F}_{<0}$ if it terminates because it runs out of budget. Formally, consider any trace of instructions T which P can execute. Then, if $B < c(T)$, then the execution status is $S = \mathcal{F}_{<0}$, the final budget is $B' = 0$, and the final state is still σ.

Lastly, given an executed trace of instructions T of a terminating metered program P, we say that the execution cost is $c(T)$.

We now present the three properties the metering instrumentation must satisfy, namely validity, consistency, and safety. Note that in all cases we only consider programs that terminate under the cost model. If the cost model is not sound and allows non-terminating programs, the metering instrumentation is undefined.

Definition 6.2 (**Validity**). *Assume that P_C terminates with status $\mathcal{T}$ and final budget B. Then $\mathcal{A}_C(P)$ terminates with the same status equal to $\mathcal{T}$ and the same final budget B.*

Validity ensures that the program is metered according to the cost model. Validators can choose different algorithms to place instrumentation, but as long as these algorithms produce valid instrumentation and the execution succeeds, the execution results and final budgets are equivalent. This is required for consensus, which usually does not admit approximated results.

Note that in the definition of validity, we do not mention that the final states of P_C and $\mathcal{A}_C(P)$ are the same. However, this holds implicitly because the same trace of instructions is executed.

Definition 6.3 (**Consistency**). *Assume that P_C terminates with status $S \in \{\mathcal{F}_E, \mathcal{F}_{<0}\}$ and final budget B. Then $\mathcal{A}_C(P)$ terminates with the same status equal to S and with an identical budget equal to B.*

Consistency guarantees that no matter what instrumentation algorithm is used, the execution status and the final budgets are the same. For example, consider two instrumented versions of a program P under the cost model C, called P_1 and P_2 which do not satisfy consistency with respect to P_C. Then, it can happen that P_1 terminates with status $\mathcal{F}_E$ and a non-zero final budget, while P_2 terminates with $\mathcal{F}_{<0}$ and zero budget. In practice, this can mean that one instance of the program has terminated with an integer overflow error and the other has run out of the supplied budget.

The discrepancy in execution results in case of abnormal termination (either due to exceptions or being out-of-budget) is again important for consensus, similarly to validity. Validators can select any algorithm that produces valid and consistent instrumentation, which in turn guarantees agreement amongst honest validators.

We should mention that one can ignore consistency (and validity, strictly speaking). Such an approach might be easier to realize, but at the same time, it limits the system significantly. For example, the changes to the algorithm have to be versioned because two versions can now yield different execution costs. Moreover, it precludes validators from running different instrumentation algorithms, making the network weaker because it depends on a single algorithm implementation.

Definition 6.4 (**Safety**). *Assume that P_C terminates with status $\mathcal{F}_{<0}$ and executes a trace of instructions T with the total cost $c(T)$ equal to the initial budget.*

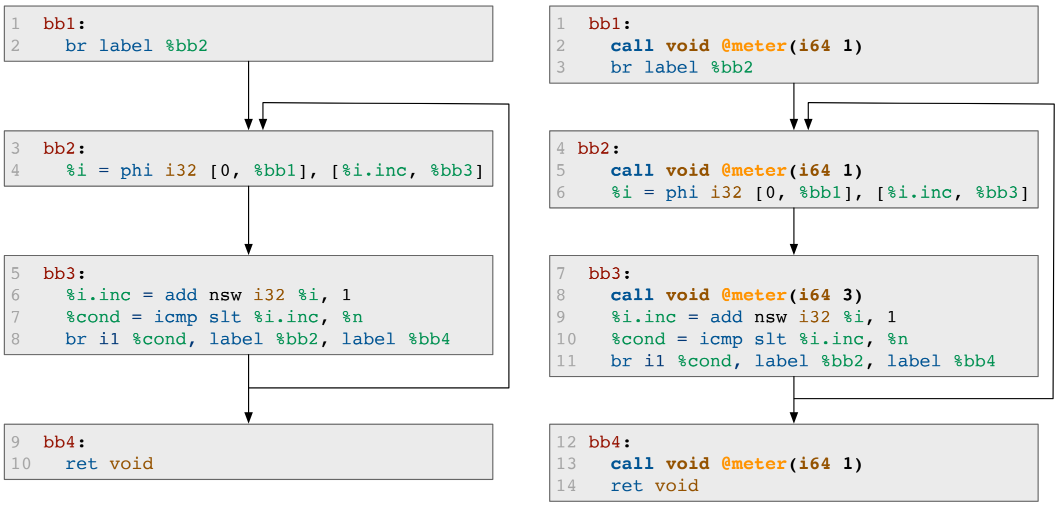
We say that $\mathcal{A}_C$ is k-safe if for some $k \geq 0$ the following holds. $\mathcal{A}_C(P)$ terminates with status $\mathcal{F}_{<0}$ and executes a trace of instructions R such that $|c(R) - c(T)| \leq k$. If $c(R) \leq c(T)$ for any T, we say the algorithm is safe. Otherwise, we say the algorithm is generous and can tolerate costs up to k to go unmetered.

Informally, safety ensures that the metering placement eventually terminates the program which is executed with an insufficient budget. The role of the parameter k is to allow a more relaxed placement of instrumentation while still guaranteeing termination. For example, for programs running out of budget, it is acceptable to execute a few more instructions than under the specification of the cost model.

6.2 Per-block placement of metering instrumentation

Now, we define the first algorithm which places the metering instrumentation for each basic block in the program, which we call *PB*. Since the algorithm is straightforward, we do not present the pseudo-code, and instead opt for a textual description. Figure 19 shows an example of such a placement in LLVM IR. [27]

(1) For each basic block, compute the sum of the costs of all instructions in that block.
(2) The instrumentation is placed at the beginning of the basic block.



```
bb1:
  br label %bb2

bb2:
  %i = phi i32 [0, %bb1], [%i.inc, %bb3]

bb3:
  %i.inc = add nsw i32 %i, 1
  %cond = icmp slt %i.inc, %n
  br i1 %cond, label %bb2, label %bb4

bb4:
  ret void
```

```
bb1:
  call void @meter(i64 1)
  br label %bb2

bb2:
  call void @meter(i64 1)
  %i = phi i32 [0, %bb1], [%i.inc, %bb3]

bb3:
  call void @meter(i64 3)
  %i.inc = add nsw i32 %i, 1
  %cond = icmp slt %i.inc, %n
  br i1 %cond, label %bb2, label %bb4

bb4:
  call void @meter(i64 1)
  ret void
```

Fig. 19. Example of metering instrumentation as computed by the PB algorithm. For clarity, the metering is shown as a call to the `@meter` function which takes a single integer argument – a cost to update the budget of the program. We assume all LLVM instructions have unit cost.

It is clear that the algorithm takes linear time in the number of instructions in the program. Moreover, the instrumentation placed by PB satisfies validity and safety, as proved below.

Proof. Consider a program P and its metered instance P_C which terminates under C.

[27] We use LLVM IR here as an example because 1) its syntax is well-known, and 2) it is high-level enough to avoid complicating the figures with unnecessary information.

Safety: Assume P_C terminates with status $S = \mathcal{F}_{<0}$ when executed with initial budget B. This means that there exists a trace T of instructions in P which is executed successfully, and which costs $c(T) = B$. Let U be the sequence of basic blocks visited by executing the trace T. Then

$$B = c(T) \leq \sum_{u \in U} \sum_{i \in u} c(i) = \sum_{u \in U} c(u).$$

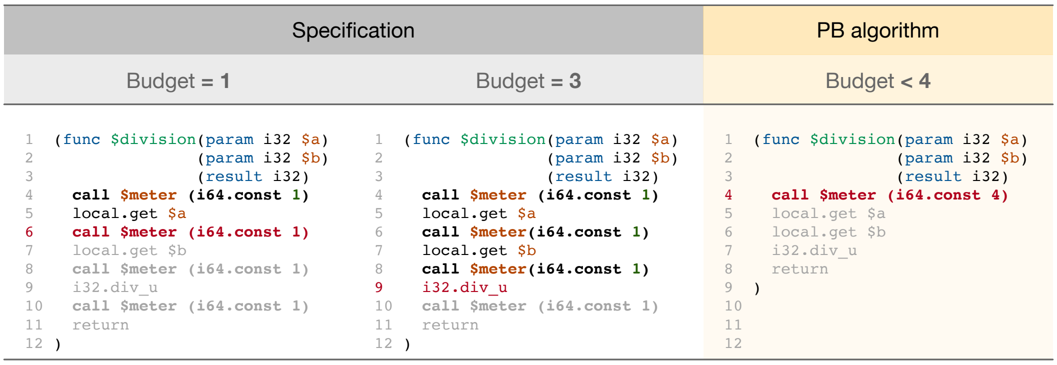
That means that the metering instrumentation placed by PB observes that the budget is not sufficient having executed a trace $R \subseteq T$. Hence, it also terminates with the same status S.

Validity: Assume P_C terminates with status $S = \mathcal{T}$. Then, each executed basic block b has cost $c(b)$. At the same time, PB meters precisely $c(b)$ for each basic block b as well. Hence, the final costs of execution are the same. Moreover, the final budget, status and states are also the same because the metered costs are equivalent and the instructions in the original program are not modified by PB. □

There are two important observations about the PB algorithm we can make. First, we observe that the placement of the metering instrumentation at the beginning of the block is optimal (we do not consider placements across blocks yet). Any other placements within the basic block would keep the number of instrumentations the same. However, placing the instrumentation at the kth position in the block only guarantees k-safety which is strictly worse than 0-safety as guaranteed by PB. [28]

Second, the instrumentation placed by the PB algorithm does not satisfy consistency. To see that, one can consider a simple counterexample as shown in Figure 20. Here, we use a Wasm program comprised of a single function that divides two inputs, and which has a single basic block. We assume that all instructions have unit cost and the input b is zero.

If the function instrumented by the PB algorithm is executed with a budget of at most 3, the execution terminates with $\mathcal{F}_{<0}$ because the metering call detects that the budget is too small. At the same time, if executed with a budget of 3 under the specification, the execution terminates with $\mathcal{F}_E$ because the division in Wasm traps on division by zero.



Specification		PB algorithm
Budget = 1	Budget = 3	Budget < 4

Budget = 1:
```
(func $division(param i32 $a)
               (param i32 $b)
               (result i32)
  call $meter (i64.const 1)
  local.get $a
  call $meter (i64.const 1)
  local.get $b
  call $meter (i64.const 1)
  i32.div_u
  call $meter (i64.const 1)
  return
)
```

Budget = 3:
```
(func $division(param i32 $a)
               (param i32 $b)
               (result i32)
  call $meter (i64.const 1)
  local.get $a
  call $meter(i64.const 1)
  local.get $b
  call $meter(i64.const 1)
  i32.div_u
  call $meter (i64.const 1)
  return
)
```

Budget < 4:
```
(func $division(param i32 $a)
               (param i32 $b)
               (result i32)
  call $meter (i64.const 4)
  local.get $a
  local.get $b
  i32.div_u
  return
)
```

Fig. 20. Wasm-based example illustrating that the instrumentation placed by the PB algorithm does not satisfy consistency. The first two boxes (on the left) show the execution under the specification (i.e., every instruction is metered individually) with budgets of 1 and 3. The third box (on the right) shows the execution under the PB algorithm for any budget smaller than 4.

[28]We note that this statement is questionable in the presence of dynamic costs. For example, we can combine dynamic costs to avoid extra metering instrumentation. The effect of this on performance is very low, however, and is not considered in this work.

Note that the violation of consistency comes from the implicit control flow originating from instructions which may throw an exception. In these cases, when a program instrumented by the PB algorithm is executed, the algorithm optimistically assumes that the budget is exceeded and then terminates. However, this might not be the case because the budget can still be sufficient to reach an instruction that implicitly transfers the control flow.

It is worth mentioning that the instrumentation is placed by the PB algorithm at the beginning of each basic block, and therefore the lack of consistency is not a security issue. Indeed, the metered cost always covers the whole block, so even if the program must terminate with an erroneous status $\mathcal{F}_E$, it does not consume a higher budget than it is supposed to. However, this is a problem on the consensus layer if different instrumentation algorithms are used.

One way to solve this problem is to be conservative and meter before each instruction with implicit control flow. This solution is used by *finite-wasm*, which is used by the NEAR blockchain.

In this work, we opt for an optimistic approach instead. As we have observed in Section 5, it is not common for contracts to terminate by implicitly throwing an exception such as divide-by-zero. It is even less common for contracts to run out of their budget. As a result, our goal is an efficient algorithm that calculates the placement of the metering instrumentation so that it satisfies validity, and, separately, a mechanism to ensure consistency.

6.3 Consistency recovery mechanisms

Instead of focusing solely on the PB algorithm, we consider an arbitrary cost model C and an arbitrary algorithm $\mathcal{A}_C$ which places valid metering instrumentation at the beginning of basic blocks. Such a relaxation is useful because it can be applied to any valid algorithm (and particularly to the more efficient algorithms we describe later in this section). Figure 21 shows an example CFG instrumented by different algorithms.

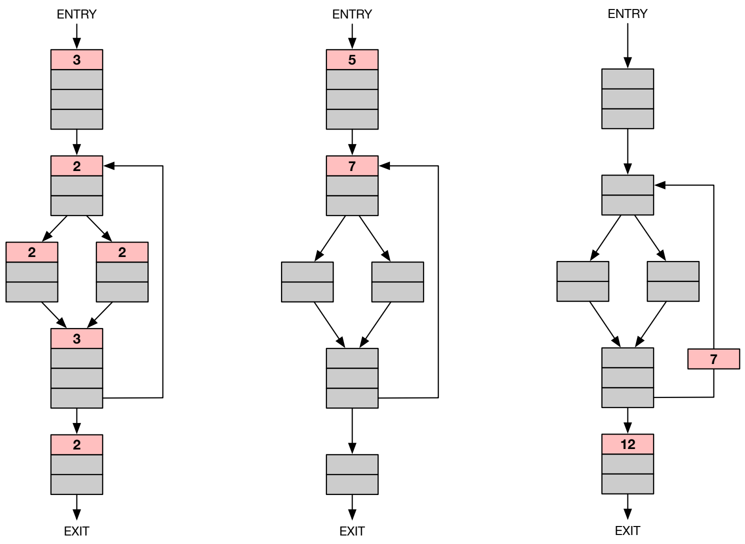


Fig. 21. Example of a simple CFG instrumented by three different algorithms. All instructions (in grey) have unit cost. The metering instrumentation is depicted in red with the cost written in bold. On the left, the PB algorithm is used. It is easy to see that the instrumentations shown in the middle and on the right satisfy validity (but not necessarily safety), despite not all blocks having metering instrumentation.

We consider an extended instruction set $\mathcal{I}_m$, which is simply the union of the original instruction set $\mathcal{I}$ and instruction m used for the metering. We note that while in reality, a single metering instruction can take 2-4 assembly instructions, in this model we consider it to be a single (pseudo-) instruction. We use the following classification of instructions in $\mathcal{I}_m$.

- *AlwaysReturn*: Instructions in this class are guaranteed to return and therefore it is safe to combine their costs. For example, wrapping addition belongs to this class.
- *MayThrow*: Instructions in this class may throw an exception. These instructions cause consistency problems if the costs are optimistically combined. If they throw an exception, status $\mathcal{F}_E$ must be produced. For example, division is usually such an instruction.
- *Meter*: Instructions in this class are inserted by metering instrumentation placement algorithms. Every instruction $i \in Meter$ subtracts the metered amount $m(i)$ from the current budget. If the budget is not sufficient, the execution must halt with status $\mathcal{F}_{<0}$. Trivially, $c(i) = 0$ for $i \in Meter$.

Consider any trace of instructions T such that the execution terminates with status $\mathcal{T}$. Let $M \subseteq T$ be the set of all instructions in T that belong to the $Meter$ class. Then for the instrumentation to be valid it must hold that:

$$\sum_{i \in T} c(i) = \sum_{i \in M} m(i)$$

Informally, the sum of metered costs must be equal to the cost of the execution according to the cost model when the program terminates successfully. With this setting, it is not difficult to see where the consistency problems can arise. In total, there are four sources of inconsistencies depicted in Figure 22.

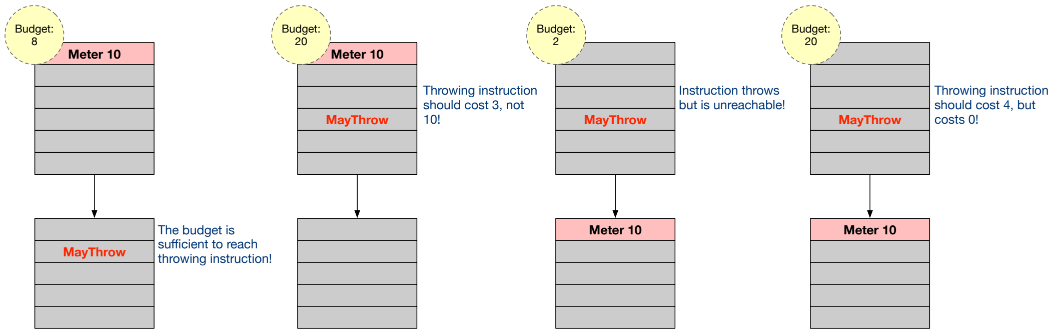

Budget: 8
Meter 10
MayThrow
The budget is sufficient to reach throwing instruction!
Budget: 20
Meter 10
MayThrow
Throwing instruction should cost 3, not 10!
Budget: 2
MayThrow
Instruction throws but is unreachable!
Meter 10
Budget: 20
MayThrow
Throwing instruction should cost 4, but costs 0!
Meter 10

Fig. 22. Four examples where the instrumentation placement does not satisfy consistency.

We consider two instances of a program P: the specification P_C and the instrumented version $\mathcal{A}_C(P)$ under the cost model C. Let B be the initial budget. Denote the trace of the executed instructions by an unmetered program as T and let the last instruction in T always throw an error E and be the only instruction with implicit control flow. Assume P_C terminates with status $S \neq \mathcal{T}$. Again, let $M = \{i \in T \mid i \in Meter\}$ and let c_M be the total cost metered by instrumentation in M.

The four sources of inconsistency can be formally described as follows.

- Assume $S = \mathcal{F}_{<0}$ and $c_M \leq B < c(T)$. In this scenario, the program under the cost model specification does not have enough budget. Because the metered amount is within the budget to execute the whole trace T, $\mathcal{A}_C(P)$ terminates with $\mathcal{F}_E$ instead.

- Assume $S = \mathcal{F}_E$ and $c(T) \leq B < c_M$. Because the metered amount exceeds the budget, $\mathcal{A}_C(P)$ terminates with a different status $\mathcal{F}_{<0}$, while under the specification the program runs into an exception.
- Assume $S = \mathcal{F}_E$ and $c(T) < c_M \leq B$. In this case, because the metered amount is still within the budget, $\mathcal{A}_C(P)$ terminates with the same status S throwing the same exception E. However, consistency is not satisfied because the final budgets are not equal and $B - c(T) > B - c_M$.
- Assume $S = \mathcal{F}_E$ and $c_M < c(T) \leq B$. Similarly to the case above, $\mathcal{A}_C(P)$ terminates with the same status S, but different final budgets. This time, the instrumented program meters fewer instructions than necessary, which can be a security problem.

We observe that in order to ensure consistency of $\mathcal{A}_C(P)$, we must guarantee that 1) we can select the correct status for execution in a deterministic way, either $\mathcal{F}_{<0}$ or $\mathcal{F}_{<E}$, and 2) if the program terminates with erroneous $\mathcal{F}_E$ status, the final budget must match the one obtained under the specification P_C.

Firstly, observe that (1) means that on running out of budget when processing some instruction $i \in Meter$ the execution should not immediately terminate. It is possible that the budget is sufficiently large for the execution to reach an instruction $j \in MayThrow$ which halts the execution with a different status. At the same time, it is not enough to check if j is reachable within the budget, because the fact that j throws can depend on the state of the program just before j. Alternatively, there can be divergence in the control flow and without executing the rest of the trace it might not be possible to know which branch will be taken.

Secondly, we observe that if execution encounters an instruction $i \in MayThrow$ which throws an exception E, the execution status can be determined in a simple way:

- If there is not enough budget to reach i, the execution terminates with the status of $\mathcal{F}_{<0}$ and the final budget of zero.
- Otherwise, the execution terminates with status $\mathcal{F}_E$. Moreover, one may need to compute the precise budget needed to reach i.

Using these two observations we define a simple mechanism to ensure consistency, presented as Algorithm 1. The mechanism consists of two functions. The first one, `on_meter(a)`, is executed for any instruction $i \in Meter$ which meters the amount a. The second one, `on_throw(i)`, is executed when an instruction $i \in MayThrow$ results in an exception. Both functions can be viewed as bytecode handlers in an interpreter loop, one to process the metering instructions and one to be used for error handling.

We observe that this design does not impact successful executions at all, allowing them to work at full speed. If there are no exceptions, then `on_throw` is never called. If the program never runs out of budget, the branch on line 6 in Algorithm 1 is never taken, no matter whether the recovery mechanism is used or not.

It remains to show how `on_meter` and `on_throw` functions can be integrated into an interpreter or a JIT-compiler, as well as how to implement `calculate_budget_to_reach`.

We observe that `on_meter` and `on_throw` are trivial to implement both as a part of an interpreter/compiler or in bytecode. For example, one can fix a memory location to track the recovery counter. Note that fixing a register is not required: the counter is accessed only when the program is in the failing state, and there the efficiency is not very important.

There are two ways to implement `calculate_budget_to_reach`: either it recomputes the cost online, or the algorithm has to do it beforehand.

First, we consider the case when the costs are calculated in advance. For that, we use the concept of *refund tables*. We associate each instruction with a refunded amount which should be subtracted

Algorithm 1 Consistency recovery mechanisms.

```
1: budget ← $B$                                     ▷ Counter to keep track of the program budget
2: recovery ← 0                    ▷ Counter that is positive when in consistency recovery mode
3:
4: function on_meter(amount)
5:     budget ← budget - amount                       ▷ Update the budget with metered amount
6:     if budget < 0 then
7:         if recovery > 0 then
8:             return $\mathcal{F}_{<0}$       ▷ No exceptions before the program runs out of budget
9:         end if
10:        recovery ← amount                                         ▷ Enter the recovery mode
11:    end if
12: end function
13:
14: function on_throw(instruction)
15:    b ← calculate_budget_to_reach (instruction)  ▷ Obtain the budget to reach the instruction
16:    if budget + recovery - b < 0 then               ▷ Compare with the last positive budget
17:        return $\mathcal{F}_{<0}$
18:    end if
19:    return $\mathcal{F}_E$
20: end function
```

from the current accumulated cost to obtain the correct value. We show examples of refund tables in Figure 23.

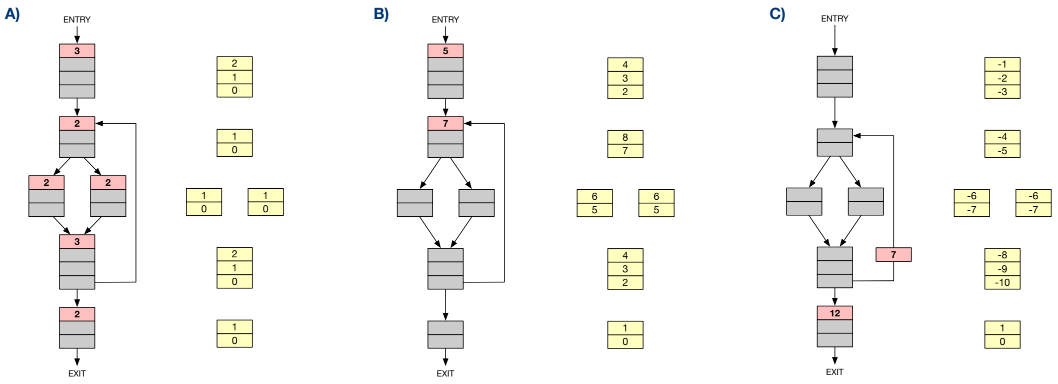


Fig. 23. Three examples of a CFG metered differently. For each, in yellow, refund tables are shown. The exact cost to reach a particular instruction can be simply calculated by subtracting the corresponding refund from the current accumulated cost (which is in turn the original budget minus the current budget).

Refund tables can be calculated by performing a single pass over all instructions in the program. Each entry is simply the difference between the currently metered amount and the actual cost. Observe that there is no need to consider loops.

We note that this approach works even in the presence of calls, although in a slightly more complicated manner. If there is an exception inside a function, we can use the refund tables to calculate the cost to reach the throwing instruction from the start of the function. Because the function was called, there might be some cost that has not yet been refunded in the caller. This extra refund can be calculated by unwinding the call stack: the callee is removed from the stack and the refund is calculated for the previous caller. This process is shown in Figure 24.

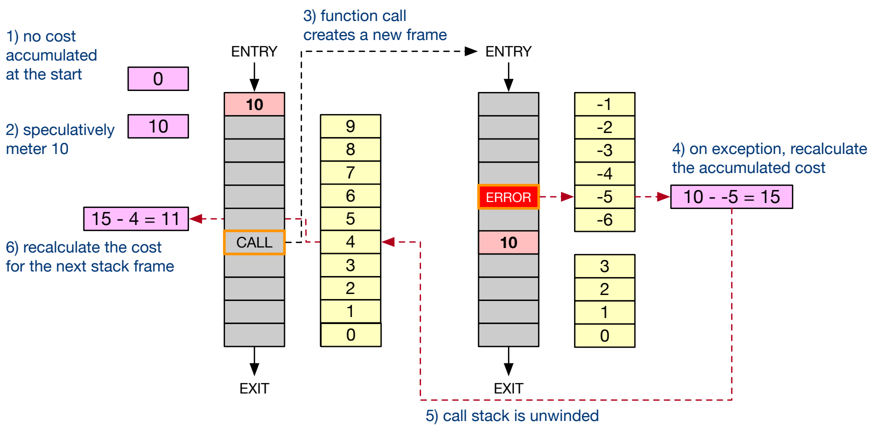

1) no cost accumulated at the start
0
2) speculatively meter 10
10
15 - 4 = 11
6) recalculate the cost for the next stack frame
ENTRY
10
CALL
EXIT
9
8
7
6
5
4
3
2
1
0
3) function call creates a new frame
ENTRY
ERROR
10
EXIT
-1
-2
-3
-4
-5
-6
3
2
1
0
4) on exception, recalculate the accumulated cost
10 - -5 = 15
5) call stack is unwinded

Fig. 24. Example of using refund tables to recalculate the actual cost of execution if exception happens inside a function call.

While being faster than online calculation, refund tables add additional complexity and require more space. While each entry can be potentially a single byte, the amount of needed memory still increases by n bytes if there are n instructions in the contract. In theory, this can be optimized by storing refunds only for instructions which may throw an exception.

An alternative approach is to calculate refunds online, trading extra space for refund tables with the additional processing time. In addition, the other benefit is that we calculate the refunds on the fly only for the functions which are currently on the call stack, and not for every function in the program.

With a consistency recovery mechanism, validity and safety are the only necessary properties an efficient instrumentation placement algorithm must satisfy. We next look at such an algorithm. However, first, we develop an algorithmic framework we will be working in.

6.4 Algorithmic framework

Our next step is to develop a more optimal algorithm, using the PB algorithm as a starting point. We note that, in general, the algorithm for the placement of the metering instrumentation only has to satisfy validity and be k-safe for some $k \geq 0$. Consistency can be enforced using a consistency recovery mechanism.

Before describing a safe algorithm we develop, we need to formally define the setting we are using one more time, as we will not be operating on instructions anymore and instead analyze the vertices of a control-flow graph.

We consider CFGs denoted as $G = (V, E)$ where V is a set of vertices and E is a set of directed edges. G does not have any parallel edges.

There is a unique *entry* vertex $s \in V$ and a unique *exit* vertex $t \in V$. The exit vertex is reachable by any other vertex in V and any vertex in V can be reached from s. This condition means that any loop in the program corresponding to G has an exit and an entry.

We use $N_{out}(v) = \{u \in V \mid (v, u) \in E\}$ and $N_{in}(v) = \{u \in V \mid (u, v) \in E\}$ to represent the sets of successors and predecessors of a vertex $v \in V$, respectively.

Definition 6.5. *A vertex $v \in V$ is called a predicate if $|N_{out}(v)| > 1$. A directed edge $(u, v) \in E$ is called a critical edge if $|N_{out}(u)| > 1$ and $|N_{in}(v)| > 1$.*

The significance of critical edges in a CFG is that in order to enable certain compiler optimizations they have to be split by introducing a new vertex. The most obvious example of such an optimization is the partial redundancy elimination (PRE) [65].

In PRE, there are expressions which can be redundant on some but not all paths. For example, in Figure 25 an expression E is redundant on the paths containing the basic block a but is not for the paths containing the blocks b and c. Moreover, E is never computed on any path through the blocks c and d.

PRE optimization splits the edge (c, b) and introduces an empty basic block v. Then, E can be moved to blocks a and v, without introducing non-existent computations on the path going through the blocks c and d. Such a technique is also useful for the algorithms that place metering instrumentation, as we will see later on.

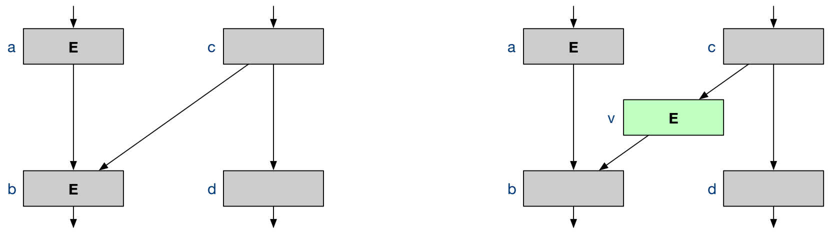


Fig. 25. Partial redundancy elimination for the expression E. On the left, an original CFG where E is computed twice on the path $a - b$. On the right, a CFG obtained after running the PRE algorithm.

Definition 6.6. *When talking about cycles in a directed graph G, we use the terminology defined by Ball and Larus [28]. Let u, v and w be three consecutive vertices on a cycle. We say there is a fork at u if there are edges (u, v) and (u, w). We say there is a join at u if there are edges (v, u) and (w, u). We say there is a pipe at u if there are edges (v, u) and (u, w) or (w, u) and (u, v). A diamond is a cycle with exactly one fork and one join. A directed cycle is a cycle consisting of pipes only.*

Definition 6.7. *Let C be a cycle and let $G[C]$ be the subgraph of G induced on C. An edge that connects two non-consecutive vertices of $G[C]$ is called a chord of C.*

Lemma 6.8. *Consider a graph $G = (V, E)$ such that it has no critical edges. Then any directed cycle $X \subseteq V$ has no chords.*

Proof. For contradiction, assume there is a chord $(a, b) \in E$ in some directed cycle X. Take $c \in X \cap N_{out}(a)$ and $d \in X \cap N_{in}(b)$. Note that it is possible that $c = d$. Clearly, $|N_{out}(a)| > 1$ and $|N_{in}(b)| > 1$, so (a, b) is a critical edge, which is a contradiction. □

Any vertex in a CFG has a cost, determined by a cost function $c : V \longrightarrow \mathbb{Z}_{>0}$. Moreover, for any set of vertices $X \subseteq V$ we define the cost of X as the sum of the vertex costs. For convenience, we again abuse the notation and use the cost function throughout: for costs of vertices, paths, or arbitrary sets of vertices.

In our model we do not allow having explicit costs for edges. One can observe that this is not needed because edge costs can be simulated: an edge can be split by introducing a new vertex with a cost equal to the cost of that edge.

We define a metering function $m : V \longrightarrow \mathbb{Z}_{\geq 0}$. For any vertex $v \in V$, we call it *metered* if $m(v) \neq 0$ and *unmetered* otherwise. The metering instrumentation algorithm can then be reduced to identifying such a function m – an assignment of zero and non-zero integer values to vertices of a CFG. The metering instrumentation can then be injected into all basic blocks b with $m(b) \neq 0$. A single instrumentation which meters the amount $m(b)$ and placed at the beginning of the basic block is sufficient.

Definition 6.9 (**Validity**). *The metering function is called valid if for any path $\pi = s, \dots, t$ it holds that*

$$\sum_{v \in \pi} m(v) = \sum_{v \in \pi} c(v).$$

We note that the validity of m ensures the validity of the corresponding instrumentation. Assume a sequence S of basic blocks is successfully executed. The execution cost is simply $c(S)$. Hence, the metering function produces a valid instrumentation.

Definition 6.10 (**Safety**). *We say that for a vertex $v \in V$ is covered if for any entry-exit path $\pi = s, \dots, t$ it holds that*

$$\sum_{v \in \pi} m(v) \geq \sum_{v \in \pi} c(v).$$

Next, let

$$k = \min_{\forall \pi \in \mathcal{P}} |m(\pi) - c(\pi)|$$

where $\mathcal{P}$ is a set of all possible paths starting at entry $s \in V$. Then, we call such metering function k-safe. If $k = 0$, i.e., the costs of all vertices in a CFG are covered, then the metering function m is called safe. Otherwise, it is called generous.

We observe that the safety of a metering function implies the safety of the corresponding instrumentation. If the metering function is safe, then when a basic block b is about to be executed the cost of b has either been already metered or will be metered using the metering instruction placed at the beginning of b and which is large enough to at least cover the instructions in b. Hence, running out of budget is always caught early or at least triggering the recovery mode if a consistency recovery mechanism is used.

Corollary 6.11. *The metering function $m : V \longrightarrow \mathbb{Z}_{\geq 0}$ so that for each vertex in V it holds that $m(v) = c(v)$ is both valid and safe.*

Recall that we are interested in the properties of the metering instrumentation for a program P only when the program terminates under the specification of the cost model. Hence, it is convenient to define the following lemma already.

Lemma 6.12. *Consider an arbitrary program P represented by the control-flow graph $G = (V, E)$ and an arbitrary cost model C. Then if P_C terminates, any directed cycle in G which is also reachable from the entry $s \in V$ has a non-zero cost.*

Proof. For contradiction, assume that there is a directed cycle $X \subseteq V$ such that it has a zero cost and is reachable from $s \in V$. Hence, there is a path from the entry vertex s to some $u \in X$. Since the cost of X is zero, it can be executed infinitely many times. That is, one can select an execution path $s, \dots, X, X, \dots$, and so P_C does not terminate, which leads to a contradiction. □

Onward, we assume that for all control-flow graphs G we consider, it holds that for any directed cycle X, $c(X) > 0$. Such an assumption does not limit the generality of the instrumentation placement algorithms. We only consider programs which terminate under the specification of the cost model C. But then Lemma 6.12 implies the condition.

Next, we look at the safe per-path placement (SPP) algorithm which aims at minimizing the number of instrumented basic blocks along any entry-exit path in a CFG, while ensuring safety.

6.5 Computing valid and safe metering function which is path-minimal

So far we have seen the PB algorithm which operates on basic blocks. It is not hard to see that there exist programs where the PB algorithm does not produce the optimal placement. For example, in Figure 21, the program instrumented with the PB algorithm goes through 5 metering instructions, while it can be reduced to just 1 or 2.

We therefore look for an algorithm that optimizes the placement of the metering instrumentation over paths in a CFG of a program. We formulate it as an optimization problem using the algorithmic framework we have previously defined.

Given a CFG $G = (V, E)$, our goal is to find a linear-time algorithm to compute the metering function $m : V \longrightarrow \mathbb{Z}_{>0}$ such that m is valid and safe, and so that the number of metered vertices is minimized along all paths (and so m is called *path-minimal*).

6.5.1 Locally path-minimal safe and valid metering function. Before we describe the algorithm, it is important to make a few observations about the metering function we are looking for. We consider two simple structures, also shown in Figure 26: a straight-line sequence of vertices and a predicate vertex. For each, we find a metering function which is locally optimal.

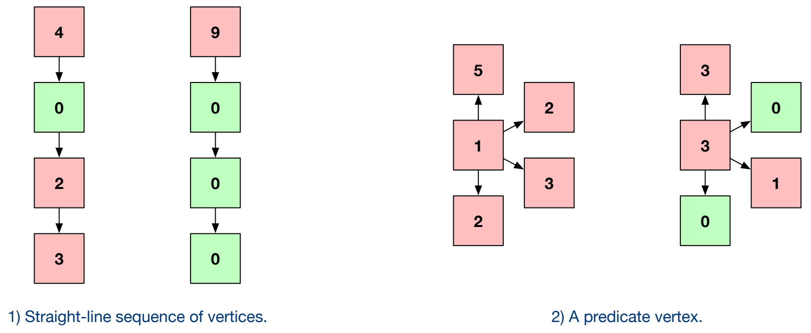


Fig. 26. Optimal and safe metering function for two local structures: 1) a straight-line sequence of vertices (on the left) and 2) a predicate vertex (on the right). The vertices' labels are the metered amount.

Lemma 6.13. *Consider a graph on vertices $V = \{v_1, \ldots, v_n\}$ with edges $\{(v_i, v_{i+1}) \mid 1 \leq i < n\}$. Suppose we are also given a valid and safe metering function $m\colon V \longrightarrow \mathbb{Z}_{\geq 0}$. Then m can be refined into $m'\colon V \longrightarrow \mathbb{Z}_{\geq 0}$, which is valid, safe, and minimizes the number of metered vertices. Moreover, m' is defined as*

$$m'(v_i) = \begin{cases} m(V) & \text{, if } i = 1 \\ 0 & \text{, otherwise} \end{cases}$$

Proof. We observe that m' is trivially a valid metering function. Moreover, it is safe because every vertex is covered by the metering in the first vertex v_1. This metering is also minimal because a single vertex is metered (and having no metered vertices would violate validity). □

Lemma 6.14. *Consider a graph with vertices V such that $V = \{u\} \cup N_{out}(u)$ (the graph is a directed star with a predicate vertex u). Also, let m be a valid and safe metering function on V. Then m can be refined into $m' : V \longrightarrow \mathbb{Z}_{\geq 0}$, which is valid, safe, and minimizes the number of metered vertices in V.*

Denote $c = \min_{v \in N_{out}(u)} m(v)$. Then the optimal and safe metering function m is defined as

$$m'(x) = \begin{cases} m(u) + c & , \text{if } x = u \\ m(v) - c & , \text{if } x = v \in N_{out}(u) \end{cases}$$

Proof. It is trivial to refine m to m' using its definition, so it suffices to show that m' is valid, safe, and minimizes the number of metered vertices.

Take any $v \in V \setminus \{u\}$. Then $m'(u) + m'(v) = m(u) + c + m(v) - c = m(u) + m(v)$ and so m' is both valid and safe by definition.

Now it remains to show optimality. Let $n = |\{v \in V \mid m(v) = c\}|$ be the number of unmetered vertices. Now, for contradiction assume that m' is not optimal and hence it should be possible to adjust the cost along the path $u, \ldots, v$ for some $v \in V \setminus \{u\}$ to make n bigger.

Since $c = \min_{v \in N_{out}(u)} m(v)$, there exists a vertex $v \in V \setminus \{u\}$ such that $m(v) = 0$. Hence, it is not possible to increase $m(u)$ as it must decrease the metered amount for v.

We also cannot decrease $m(u)$ by an amount greater than or equal to $m(u)$ (recall that m' must be safe). Hence, decrease $m'(u)$ by some $0 < t < m(u)$. However, this makes all vertices $v \in V$ such that $m'(v) = 0$ to be metered, decreasing n instead. Note that such v always exists, because the minimum metered amount has been taken. □

6.5.2 Globally path-minimal safe and valid metering function. Lemma 6.14 is particularly powerful and can be applied to arbitrary CFGs on a vertex set V if critical edges are split. In that case, the successors of any predicate vertex $v \in V$ would have only a single predecessor – v itself. This means that we can refine the metering function across G to obtain a locally optimal function around v and its successors.

Inspired by Lemmas 6.13 and 6.14, we can define an algorithm to find the metering function that satisfies both safety and validity by repeatedly computing the locally optimal metering function. The hope is that the safety and validity would be preserved during the iterations of the algorithm, and it will terminate reaching the global optimum for m.

The pseudo-code for the algorithm is present in Algorithm 2. The algorithm first splits all critical edges in the control-flow graph and starts with the metering function m_0 equal to the cost function c. Then, it iterates until it reaches a fixed-point and at each iteration applies Lemma 6.14. Note that Lemma 6.13 can be trivially expressed as multiple applications of Lemma 6.14. The algorithm terminates when the metering function m_i cannot be further improved. Finally, empty blocks are removed. A step-by-step walk-through of the algorithm for a simple CFG is shown in Figure 27.

We now show that that the algorithm terminates with a valid and safe metering function. We then argue that the function is optimal. It is worth mentioning that the SPP algorithm shown in Algorithm 2 is not efficient, and its running time can be further improved. Nevertheless, we use this version of the algorithm for our proofs.

Lemma 6.15. *Consider $G = (V, E)$ – a CFG with all critical edges split. For any directed cycle $C \subseteq V$, $m_i(C) = c(C)$ at each iteration of the algorithm.*

Proof. We use induction on the number of iterations. Trivially, the statement holds at iteration 0. It remains to show that a single iteration of an algorithm preserves the total metered amount for the cycle.

For inductive step, assume this is true and $m_i(C) = c(C)$. We now prove that this implies that $m_{i+1}(C) = c(C)$ as well.

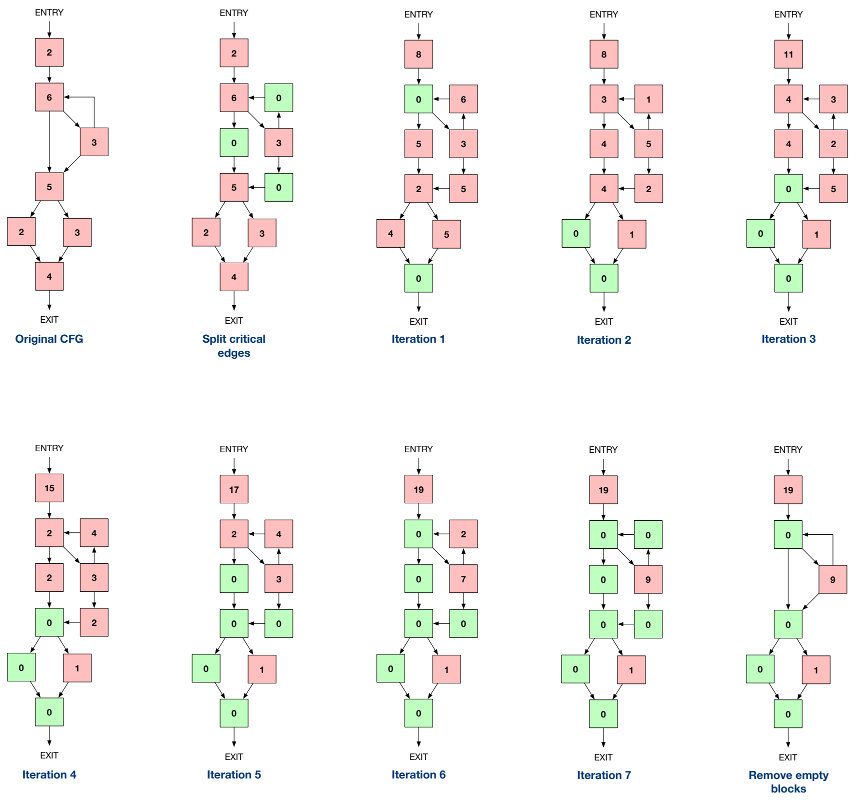


Fig. 27. Execution of the SPP algorithm on a simple graph which has a single cycle with multiple exits. Metered vertices are represented in red, and the current values of the metering function are written in bold.

Because the algorithm splits all critical edges, by Lemma 6.8, there are no chords in C.

Let $W \subseteq V \setminus C$ be a set of vertices which have edges going to C, and let E_W be the collection of such edges. We observe that it is not possible for $w \in W$ to have multiple outgoing edges, because this would contradict an assumption that there are no critical edges. Hence, there is a unique edge for each $w \in W$ and $|N_{out}(w)| = 1$.

At iteration $i + 1$, for $(w, v) \in E_W$, $m_i(v)$ is propagated to w, and also to a single $x \in C$, as v lies on a cycle. Note that there are no chords, which results in uniqueness of x, and so the metering amount in the cycle does not change.

We now consider the opposite scenario: can more metering flow into the cycle? Let $U \subseteq V \setminus C$ be a set of vertices which have edges pointing from C, and let E_U be the collection of such edges. Like in the case above, it is not possible for $u \in U$ to have other incoming edges – this would create a critical edge. Also, for each $(v, u) \in E_U$, there is a unique edge (v, y) with $y \in C$. By the same reasoning, having multiple edges creates chords.

At iteration $i + 1$, we move the metered amount from successors of v to u. The amount is always bounded by the largest metered amount amongst them. In particular, we can never move more than $m_i(y)$. Moving less, will keep some fraction of metered amount with y, leaving the total metered amount unchanged. □

Theorem 6.16. *The metering function computed by the SPP algorithm is valid and safe.*

Algorithm 2 The SPP algorithm which computes safe and valid metering function.

```
function spp($G$)
    split_critical_edges ($G$)
    $i \leftarrow 1$                                      ▷ Tracks the number of iterations of the algorithm.
    $m_0 \leftarrow c$                                    ▷ Initially the metering function is the cost function.
    $processed \leftarrow \lambda v\ .\ 0$
    while $true$ do
        $total\_update \leftarrow \sum_{v \in V} min_{u \in N_{out}(v)} m_{i-1}(u)$
        if $total\_update = 0$ then                       ▷ If the metered function cannot be improved, terminate.
            remove_empty_blocks ($G$)
            return $m_i$
        end if
        for $v \in V$ do
            $c' \leftarrow min_{u \in N_{out}(v)} m_{i-1}(u)$
            $m_i(v) \leftarrow m_{i-1}(v) + c'$              ▷ Each vertex takes the minimum cost of its successors.
            for $u \in N_{out}(v)$ do
                if $processed(u) \neq i$ then            ▷ Ensures we do not subtract twice.
                    $m_i(u) \leftarrow m_{i-1}(u) - c'$      ▷ Successors give a fraction of their costs.
                    $processed(u) \leftarrow i$
                end if
            end for
        end for
        $i \leftarrow i + 1$
    end while
end function
```

Proof. The metering function is trivially valid and safe before iteration when $i = 0$. Assume that after iteration i the computed metering function m_i is valid and safe. Showing that m_{i+1} is valid and safe completes the proof by induction.

Observe that the algorithm is simply an application of Lemma 6.14 to all vertices in V, and each application is locally valid and locally safe. Hence, m_{i+1} must be safe.

Moreover, m_{i+1} is also valid. The reason why local application allow this is because critical edges are split. As a result, every successor of some vertex v dominates v, and therefore shifting the cost there does not change the per-path sum, which gives the validity. □

Next, we show that the SPP algorithm terminates. We will prove that by associating a ranking function with the algorithm, which is bounded from below by zero and decreases at each iteration.

Definition 6.17. *A vertex v is called killed at iteration i, if one of the following holds.*

(1) *$m_i(v) = 0$ and $N_{out}(v) = \emptyset$*
(2) *$\exists u \in N_{out}(v)$ such that $m_i(u) = 0$ and u is killed.*

For any killed vertex v we define a witness path $W(v)$ as follows:

(1) *if $N_{out}(v) = \emptyset$, then $W(v) = \emptyset$.*
(2) *otherwise, $W(v) = \{u\} \cup W(u)$ for arbitrary $u \in N_{out}(v)$ with $m_i(u) = 0$.*

Corollary 6.18. *If a vertex $v \in V$ is killed by the algorithm in iteration i, it remains killed for all iterations $j \geq i$.*

Corollary 6.19. *The exit vertex $t \in V$ is always killed in the first iteration.*

Theorem 6.20. *There exists a constant $c \in \mathbb{Z}_{>0}$ such that every c iterations at least one vertex is killed by the algorithm.*

Proof. Consider an arbitrary iteration i of the algorithm, and the set of killed vertices so far $K \subseteq V$. By Corollary 6.19, $|K| \geq 1$. We let $W := \{v \in V \mid (v, u) \in E \land u \in K\}$ be the set of all vertices with at least one of their successors killed.

If there is a vertex $w \in W$ such that $N_{out}(w) \subseteq K$, then w gets killed by a single algorithm step. Indeed, all its successors are killed, the update computed by the algorithm makes the metered function to be 0 in at least one vertex, and so w is killed by definition.

Otherwise, for all $w \in W$ there exists $u \in N_{out}(w)$ such that $u \notin K$. In this case, some vertices in W lie on a directed cycle. This holds because for any such a vertex u there is a path $\pi = \{u, \dots, t\}$, since t has to be reachable from any vertex. As $t \in K$, $\pi \cap W \neq \emptyset$. This holds for all $w \in W$ and so there must be a directed cycle.

Moreover, there is a directed cycle C such that for all $w \in C \cap W$, for all $u \in N_{out}(w)$ either $u \in K$ or u lies on some directed cycle (it can be C or some other cycle). This follows from the reachability of the exit vertex t. If there is some successor $u \notin K$, there is a path to t which goes through W again. This either creates a directed cycle which satisfies the property or it goes to some other w' outside of this cycle. Because W is finite, at some point the desired cycle is found.

We now take any such cycle C and let $B := C \cap W$ and let $N_b := \{u \in N_{out}(b) \mid b \in B \land u \notin K\}$. In essence, B are all vertices on C which also have successors in K and N_b are all successors of $b \in B$ which are not yet killed.

We claim that there exists a constant $c \in \mathbb{Z}_{\geq 1}$ such that after c steps of the algorithm all vertices in N_b are metered (the metering function evaluated at these vertices is non-zero). Therefore, the next algorithm step either:

- kills b if its successor with the smallest metered value lies in K, or
- decreases the metered amounts for vertices in $N_{out}(b) \cap K$.

Even if b is not killed, more steps of the algorithm eventually drain $N_{out}(b) \cap K$ so that there is vertex $x \in N_{out}(b) \cap K$ such that $m_i(x) = 0$, which kills b. It remains to show why the cost is decreasing every c steps, for some positive constant c.

For each not yet killed successor $n \in N_b$, we find another not killed vertex f ($m_i(f) \neq 0$) such that there is no vertex with a non-zero metered amount which is closer for any possible path to reach f from n. If $m_i(n) \neq 0$, then $f = n$.

Such a vertex always exists because directed cycles preserve their cost as shown in Lemma 6.15, and are also initialized with at least one no-zero cost according to Lemma 6.12.

Finally, we select this longest path $\pi = \{n, \dots, f\}$ and let $c' := |\pi| - 1$. Then in c' steps, the metered amount for the vertex n becomes non-zero. We repeat this for all vertices in N_b and take $c - 1$ to be the largest of the c's. Then, the cth iteration computes the minimum of all vertices in N_b which is non-zero, and so the cost of vertices in $N_{out}(b) \cap K$ decreases. □

It is now trivial to show termination using Theorem 6.20. We define a function $g(i)$ as the number of non-killed vertices at iteration i. Assume there are n vertices before we start the algorithm, and so $g(0) \leq n$. We observe that the first iteration is guaranteed to kill the exit vertex, and so $g(1) \leq n - 1$.

By Theorem 6.20, it is sufficient to take $c \cdot (n-1)$ more iterations to get all the remaining vertices killed, i.e., obtain $g(c \cdot (n-1) + 1) \leq 0$, which means that the algorithm eventually terminates.

Theorem 6.21. *The SPP algorithm computes a valid and safe metering function which is path-minimal: the number of the metered vertices across any path is minimized.*

Proof. Suppose that the algorithm operates on some graph $G = (V, E)$. For contradiction, assume that the computed function m by the algorithm is not path-minimal. Suppose that $v \in V$ is a vertex such that $m(v) \neq 0$ and can be shifted to other vertices in V to reduce the number of metered vertices per path while keeping the metering function valid and safe.

Because the new function has to be safe, then $m(v)$ can only be moved through the incoming edges of v. Because critical edges are split, in order to reach v one must go through the predecessors of v. Hence, moving $m(v)$ is equivalent to performing a step of the SPP algorithm. □

6.5.3 Complexity of computing a safe and valid metering function which is path-minimal. With the SPP algorithm, one can compute the placement of the metering instrumentation which is valid and safe, as well as minimizes the number of instrumented basic blocks across all paths in the program. While the algorithm also achieves termination, it uses a fixed-point iteration which can have unpredictable run-time effects.

In particular, the Algorithm 2 for a CFG with n vertices and m edges has a time complexity of $O(\alpha(n + m))$ where α is a non-zero constant. We note that α depends on the diameter of the graph – the costs have to be propagated from the vertices the furthest away from the entry vertex. Moreover, cycles add even more complexity. In the presented algorithm, the metering cost can be propagated through the cycle multiple times. As a result, the run-time complexity becomes highly data-dependent.

6.5.4 Linear-time computation of a safe and valid metering function which is path-minimal. In the setting where the code is instrumented by a validator at run time, having data-dependent time complexity can be a security issue. A malicious user can craft a program that would require numerous iterations during the instrumentation, affecting the liveness of the network.

In order to avoid that, we propose a technique to speed up the computation of the metering function for structured control flow. The main idea is to process every vertex once, ensuring that the order in which they are processed yields a valid metering function.

First, we require additional analyses. In particular, we

- compute the topological ordering of the vertices of the control-flow graph in $O(n + m)$ time where n is the number of vertices and m is the number of edges, and
- identify all loops in the graph, in linear time using dominance [27, 35, 36].

We note that the topological order is not defined for graphs with cycles, however, it is not a problem because we additionally find all cycles as well.

Then, a valid and safe metering function can be computed as follows. First, we order all vertices in reverse topological order, to ensure that every vertex precedes its predecessors. Then, for each vertex v:

(1) If v does not belong to the cycle, update $m(v)$ using Lemma 6.14, essentially performing an update step of the iterative SPP algorithm.
(2) If v lies on a cycle, record it and continue.

Essentially, this construction allows us to process and record all loop exits. Once all exits for some loop are recorded, we can compute the values of the metering function for the loop and its entries, "fast-forwarding" the iterative construction employed by the SPP algorithm we presented before. The cost of the vertices inside the loop can be computed by processing previously recorded vertices in their reverse topological order. After the loop has been processed, we continue with the vertex processing defined above.

6.6 Exploring valid and generous metering functions

We observe that the algorithm presented before computes the safe metering function such that the number of metered vertices per path is minimized, or simply *path-minimal.* However, if we forgo the safety property, the metering function is no longer path-minimal.

This is particularly easy to see with diamonds, as shown in Figure 28. The safe metering function is non-zero where control flow diverges. As a result, the metering function can have many non-zero values along a path, and many zero values along the other. Instead, for this example, a better metering function with a smaller number of metered vertices per path would only meter a single vertex for any path.

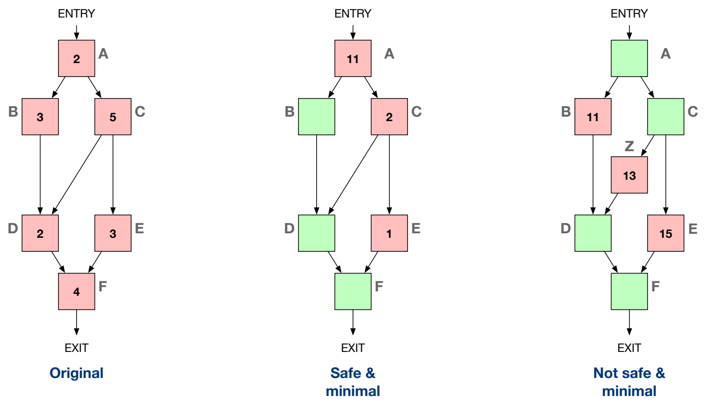


Fig. 28. On the left, an example CFG annotated with costs. In the middle, a valid and safe metering function which is path-minimal. We can see that the path $A - C - E - F$ has 3 non-zero values, while the path $A - B - D - F$ has only one. On the right, a metering function that does not satisfy safety but is minimal. We observe that all paths traverse only a single non-zero block. Note that we had to split an edge with an artificial vertex Z in order to obtain a valid function.

In Section 5 we observed that it is not common for smart contracts to run out of budget. Hence, the safety requirement for these “good” scenarios can be relaxed – it might be acceptable not to identify immediately that the execution has used all the budget. Therefore, in order to compute the placement of the metering instrumentation one has to find a valid and generous metering function, instead of a safe one, in order to ensure that the amount of instrumented basic blocks in the program is indeed minimized for any path in the control-flow graph. While it is up to the cost model to decide whether executing over the budget is allowed or not, here we focus solely on the algorithmic aspects of computing the generous metering function.

In order to compute a path-minimal generous metering function, one can first compute a safe metering function, and then refine it in a step-by-step fashion into a generous one. However, such an approach does not always yield minimality.

For example, it can be tempting to use a simple local refinement heuristic for every vertex in the graph and a metering function m:

- if the vertex v is a predicate vertex with $m(v) = c$, then we refine $m(v) = 0$, and set $m(u) = m(u) + c$ for every successor u of v;
- otherwise the vertex is untouched.

While the local refinement can improve the metering function computed by the algorithm in the case shown in Figure 28, it is not always so. A counterexample can easily be constructed by adding a long straight-line sequence of vertices in front of a predicate vertex, as shown in Figure 29.

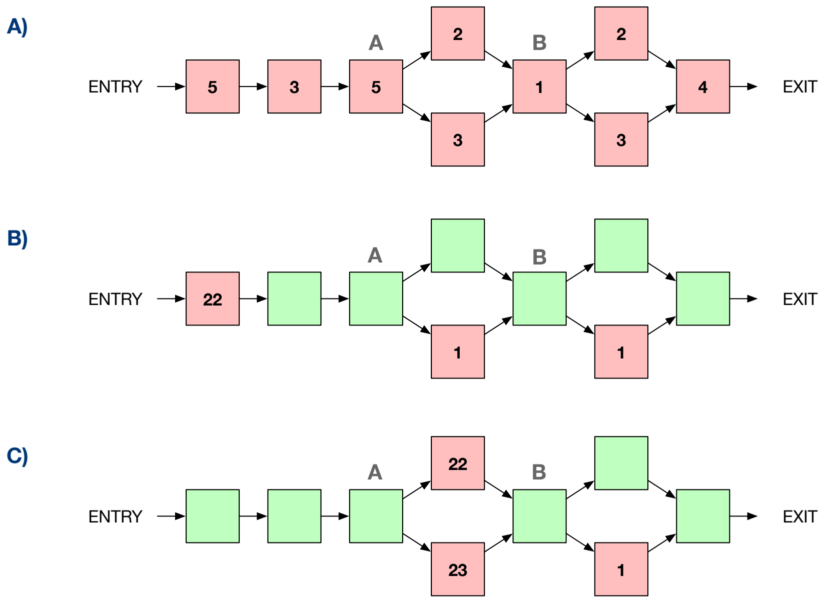


Fig. 29. An example where propagating metering computed by safe metering function to successors does not work. A) An original cost function over a CFG. B) A path-minimal safe metering function. Note that applying the simple local refinement for predicate vertices (*A* and *B*) does not help because the predicate vertices have zero values in the safe metering function. C) A path-minimal generous metering function. Here, the metering from the first vertex is shifted into the diamond at *A* which reduces the number of non-zero vertices.

As a result, it is clear that the algorithm has to use stronger properties than the relations between vertices and their successors. We next consider dominance and post-dominance relations.

6.6.1 Dominators and post-dominators. First, we describe what dominators and post-dominators are. Given a control-flow graph $G = (V, E)$ with a single entry $s \in V$ and a single exit $t \in V$, we define the dominator and post-dominator relation as follows.

Definition 6.22. *A vertex v dominates another vertex u, written as v dom u, if every path from s to u must go through v. A vertex v post-dominates another vertex u, written as v post u, if every path from u to t must go through v.*

If a vertex has more than one dominator, there is always a unique "closest" dominator called an *immediate dominator.* Similarly, there is unique *immediate post-dominator.*

Definition 6.23. *A vertex v* strictly dominates *vertex u, written as v sdom u, if and only if v dom u and $v \neq u$. A vertex v is an* immediate dominator *of vertex u, written as v idom u, if and only if the following holds for any $x \in V$:*

$$v \textit{ sdom } u \land x \textit{ sdom } u \Rightarrow x \textit{ dom } v$$

Similarly, for post-dominators, a vertex v strictly post-dominates *vertex u, written as v spost u, if and only if v post u and $v \neq u$. A vertex v is an* immediate post-dominator *of vertex u, written as*

v ipost u, if and only if the following holds for any $x \in V$:

$$v \; spost \; u \wedge x \; spost \; u \Longrightarrow x \; post \; v$$

Using the concept of immediate dominators and post-dominators, one can define a *dominator tree* (*post-dominator tree*) in which there is an edge between $v, u \in V$ if and only if v *idom* u (v *ipost* u). Figure 30 shows an example of a CFG, as well as its dominator and post-dominator trees. For a dominator (post-dominator) tree T, we say that the vertex $v \in T$ is a leaf if it has no outgoing edges.

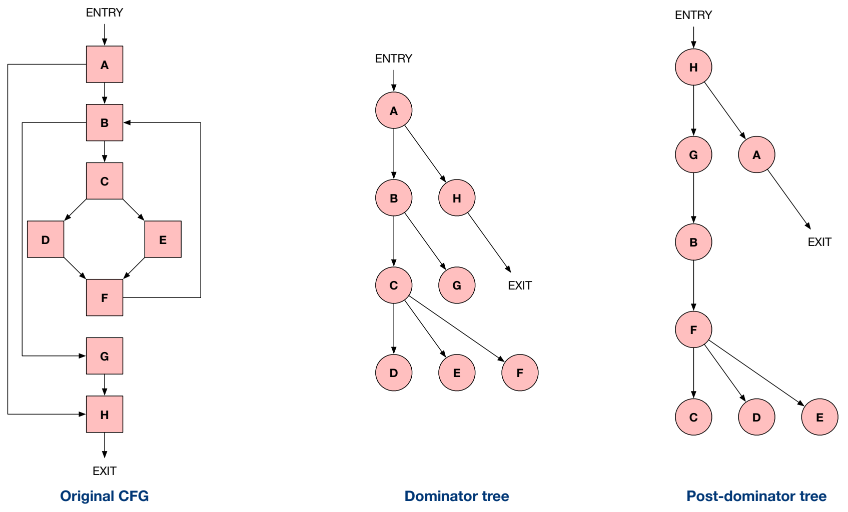


Fig. 30. A simple CFG with corresponding dominator and post-dominator trees. The dominator tree has leaves H, G, D, E and F. The post-dominator tree has leaves A, C, D, and E.

6.6.2 Using dominators and post-dominators to find a valid and generous metering function. We now describe how a valid generous metering function can be computed based on the dominator or post-dominator trees. While the metering function we present here is not optimal, we believe that using dominance and post-dominance relations is the right way to compute a path-minimal valid and generous metering function. At the same time, the metering function which we present here is useful to evaluate the SPP algorithm.

Our main observation is that it is sufficient to instrument the leaves of dominator and post-dominator trees, as long as the critical edges are split, in order to obtain a valid and generous metering function. Splitting critical edges is essential for correctness. In particular, it ensures that all paths are captured by leaves in the dominator/post-dominator trees, with an example shown in Figure 31.

We observe that the new vertex which is the result of the split is always a leaf in the dominator tree. Indeed, the source of the edge, say A, dominates the new vertex X and is its parent in the dominator tree. At the same time, X can only be reached from the entry via A, making it a leaf.

We now prove the main result, showing that it is sufficient to meter the leafs of dominator or post-dominator trees.

Theorem 6.24. *Consider a control-flow graph $G = (V, E)$, with entry and exit $s, t \in V$ and with all critical edges split. Then, metering the leaves of the dominator tree is sufficient to capture any $s - t$-path in the control-flow graph.*

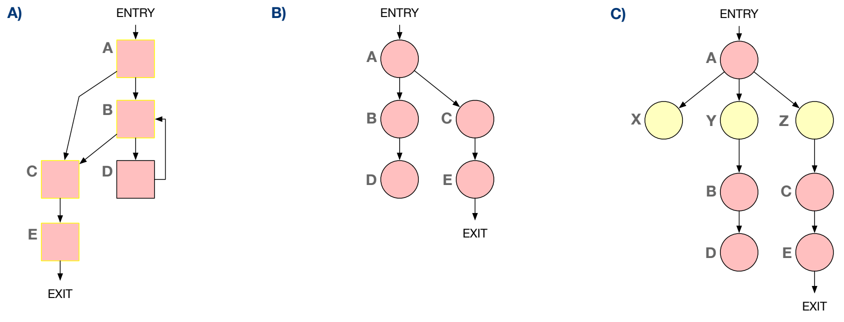


Fig. 31. A) A simple control-flow graph with three critical edges $A - C$, $A - B$, and $B - C$. B) A dominator tree when the critical edge is not split. We observe that the paths $A - C - E$ and $A - B - C - E$ cannot be distinguished by the leaf vertices D and E of the dominator tree. C) A dominator tree when the critical edge is split creating new vertices X (between A and C), Y (between A and B), and Z (between B and C). Now, an extra leaf at X can distinguish between the two aforementioned paths.

Proof. For contradiction, assume that there exist two $s - t$ paths π_1 and π_2 which cannot be distinguished by dominator leaves, i.e., the same subset of leaves is visited by both paths.

Because $\pi_1 \neq \pi_2$, there must be some divergence in control flow at vertex y so that paths are different, possibly a cycle. Let x be the vertex where control flow converges, which must be the case because both paths end in t. Let $S_1 \subseteq \pi_1$ and $S_2 \subseteq \pi_2$ be the corresponding $x - y$ paths which differentiate between π_1 and π_2.

We observe that $|S_1| > 0$ and $|S_2| > 0$ as otherwise the divergence is because of a single critical edge, which is a contradiction.

Now we consider two cases. First, suppose that the diverging control flow does not create a cycle. But then, the predecessors of y are leaves in the dominator tree and belong to different paths, which is a contradiction to our initial assumption. If the divergence in control flow creates a directed cycle, then we observe that predecessor of y which lies on a cycle must be a leaf in the dominator tree. Moreover, y belongs to only one of the paths, which again contradicts our initial assumption. We illustrate both cases in Figure 32. □

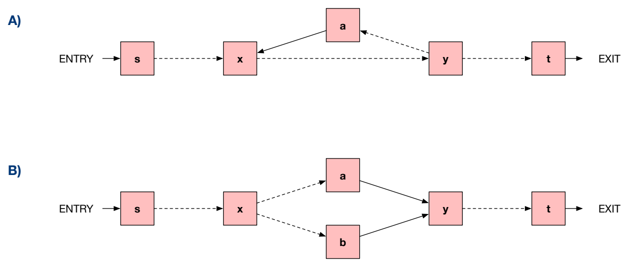


Fig. 32. Two cases considered by Theorem 6.24. Dotted arrows represent paths, and are edges otherwise. A) Suppose π_1 is *s-x-y-t* and π_2 is *s-x-a-y-t*, taking the cycle. But then a must exist to ensure there are no critical edges and is also a leaf. B) Suppose π_1 is *s-x-a-y-t* and π_2 is *s-x-b-y-t*. Both a and b must exist to ensure there are no critical edges, and both are leaves in the dominator tree because the control flow converges at y.

This result also applies to the post-dominator tree, as it is a "reverse" of dominance relation.

Corollary 6.25. *Consider a control-flow graph $G = (V, E)$, with entry and exit $s, t \in V$ and with all critical edges split. Then, metering the leaves of the post-dominator tree is sufficient to capture any $s - t$-path in the control-flow graph.*

While sets of leaves are sufficient, they are not necessarily minimal. Figure 30 provides an example: metering the vertex F is not required and its cost can be merged into the costs of D and E.

6.6.3 Summary. While we have not constructed an optimal generous metering function in this work, we identified an alternative construction which still provides a sufficient instrumentation. The metering function we constructed is particularly useful for evaluating the SPP algorithm in Section 7, and comparing how the safe and valid metering function compares to the sub-optimal generous metering function which can be easily computed.

6.7 Loop Invariant Metering Motion

So far we considered an algorithm that computes the placement of metering instrumentation relying on 1) the structure of the control-flow graph of a program, and 2) its initial costs. However, similarly to many other compiler optimizations, the program may contain information that can help improve the placement of the instrumentation. Here, we describe an optimization for loops that can be used to further minimize the metering overhead. Additionally, we discuss its advantages and disadvantages.

Recall that the metering instrumentation always has to be placed inside loops, or other kinds of cycles in the control-flow graph if we also allow for irreducible control flow. It is easy to see why by considering a simple example of an infinite loop, which, if not instrumented, runs forever.

While loops are not very common in smart contracts, as demonstrated in Section 5, they do occur with relative frequency. At the same time, instrumentation inside loops may cause substantial run-time overhead. Therefore, it can be important to optimize the instrumentation for these scenarios. Here, we discuss such an optimization, which we call *Loop Invariant Metering Motion* (LIMM).

The goal of LIMM is to hoist metering instrumentation out of loops. Such an optimization also allows one to efficiently meter tail-recursive functions: a function that uses tail-recursion can be transformed into a loop first, which can then be further optimized using LIMM. Figure 33 shows a pair of examples where LIMM can be applied.

A)
```python
# original program
while i < 100:
  meter(4)
  i = i + 1

# the total cost is 4 * 100
meter(400)
while i < 100:
  i = i + 1
```

B)
```python
# original program
while i < n:
  meter(4)
  i = i + 1

# the total cost is 5 * n
if n > INT_MAX / 4:
  # out-of-budget handling
meter(4 * n)
while i < n:
  i = i + 1
```

Fig. 33. Examples of how metering instrumentation can be hoisted out of loops. For simplicity, we use Python and model the metering instrumentation as a function call to `meter` function. A) A loop with a constant bound, B) A loop with a statically unknown number of iterations.

It is important to mention that LIMM has to be efficiently implementable. Just like the algorithm for the instrumentation placement itself, this optimization is performed at run time and therefore has to be linear or sub-linear in the size of the program. As a result, it is reasonable to specialize on particular types of loops, preferring frequent and simple patterns.

Here, we describe two cases in particular: loops with statically known bounds and loops with dynamic bounds. In the first case, the metering instrumentation can be moved outside of the loop and, usually, is the product of the loop bound and the cost of the loop body, as shown in Figure 33A.

In the second case, hoisting the metering instrumentation is not as obvious anymore. Because the number of iterations is unknown, the product may overflow, resulting in the incorrect (and very small, if overflow wraps the integer value) cost. Hence, it is crucial to insert an overflow check prior to metering the product expression.

We observe that even such a simple optimization as LIMM has to be carefully implemented to avoid security vulnerabilities. Moreover, we note that hoisting metering instrumentation also makes consistency recovery mechanisms more complicated. Particularly, to ensure consistency one needs to know at which loop iteration the exception has happened. As a result, LIMM may only be profitable if the loop has no implicit control flow.

We conclude that optimizations like LIMM can be useful as the blockchain ecosystem grows and demands more performance. However, they also introduce complexity and security risks. Unless extremely fast execution is necessary, an optimal placement of the metering instrumentation is sufficient.

6.8 Summary

In this section, we formally defined the properties any metering instrumentation should satisfy, namely validity, consistency, and safety. We observed that consistency limits the efficiency of the placed instrumentation, in particular requiring metering instrumentation for instructions with side effects or instructions that trap. With consistency, even simple per-block placement of the metering instrumentation is not possible under the blockchain setting.

However, we found that consistency can be ignored by the instrumentation placement algorithm as long as there is a consistency recovery mechanism – a post-processing step that ensures that if the program execution fails, the correct execution cost is calculated. Using such a mechanism allows one to develop more efficient algorithms to place the metering instrumentation.

We then presented the SPP algorithm and the algorithmic framework we used. The algorithm minimizes the amount of the metering instrumentation for any entry-exit path in a program, while ensuring safety, i.e., that there is always a sufficient budget to cover for the execution cost. Because the SPP algorithm we presented uses a fixed-point iteration, we additionally discussed how its run time can be optimized to be linear.

We also explored how the safety constraint can be relaxed and its impacts on the metering instrumentation. We found that without safety, the problem of minimum metering instrumentation becomes significantly harder. Still, we presented techniques that can be used to obtain a generous metering instrumentation, though sub-optimal.

Lastly, we explored how the metering instrumentation can be optimized further, focusing on hoisting it outside the loops. We observed that such an optimization is not easy to implement correctly, and so it might be preferable to use it only for simple loops.

7 EVALUATION

In this section, we evaluate the SPP algorithm for metering instrumentation placement and compare it with the state-of-the-art approaches, as well as with algorithms that place the metering instrumentation generously. In addition, we discuss how the cost model can interact with the algorithm to prevent attacks on the metering instrumentation.

Below, we highlight the most important observations we made during the evaluation.

- It is crucial to take into account the dominance and cost information for the metering scheme. For example, we observed that the SPP algorithm outperforms state-of-the-art approaches on selected programs, often reducing the slowdown due to metering by a factor of 2. Moreover, in some programs, the instrumentation computed by the SPP algorithm has nearly zero overhead.
- While splitting critical edges is an essential construction step in the algorithm we developed, the newly introduced basic blocks may lead to slightly worse run time compared to less optimal solutions.
- In practice, the iterative SPP algorithm quickly converges, taking around 30 iterations to obtain nearly optimal placement of the metering instrumentation.
- Safety can be very limiting if the CFGs have many leaves. In particular, we observed that on tree-like control-flow graphs, the SPP algorithm yields no improvement.
- Only 30.4% of all basic blocks are instrumented by the SPP algorithm on average (if critical edges are split, and 37.5% otherwise) to obtain a safe and valid metering function.

7.1 Benchmarking methodology

As observed in Section 4, existing VMs used by blockchain systems are not yet performant, instead prioritizing security. Hence, to evaluate the SPP algorithm close to the real-world scenarios but also make the evaluation meaningful we consider two settings:

- *Real setting*, using a set of Wasm microbenchmarks. First, the overhead of metering instrumentation placed sub-optimally is high, as seen in Section 4. Second, there are existing metering implementations that we can compare to.
- *Theoretical setting*, using weighted graphs and analyzing the metering function computed by the SPP algorithm – focusing on the number of metered vertices as a metric. Additionally, we compare how other approaches compare to our algorithm. To make this setting more realistic, we use control-flow graphs extracted from the most popular Solana contracts.

7.1.1 Wasm microbenchmarks. We use Wasm microbenchmarks to evaluate the instrumentation placement algorithms on particularly interesting program instances. In total, we selected 9 programs, summarized in Table 27.

7.1.2 Solana contracts. We consider the ten most popular contracts on Solana, as of 23.09.23. We previously used these contracts while studying existing approaches to metering in Section 4 and analyzing on-chain smart contracts in Section 5. They cover a range of applications, including the NFT marketplaces, token standards, decentralized exchanges, liquidity pools, and oracles to feed real-world data such as asset prices.

7.1.3 Set-up. For all experiments in this section, we use an Apple M1 Pro CPU with 8 cores and 16 GB of memory. Additionally, we disable multi-threading and frequency scaling.

Table 27. Selected Wasm microbenchmarks for evaluation and their description.

Name	Description
`add`	A straight-line sequence of 64-bit additions. Wasm has wrapping semantics for addition, so these instructions never trap.
`div`	A straight-line sequence of divisions, which can trap.
`tree`	A sequence of branches that terminate early. The implementation is a sequence of nested *if-else* statements with a return in the *then* branch, and another *if-else* in the other branch. The CFG of this program is a tree.
`diamonds`	A sequence of branches forming a chain of diamonds. We use a sequence of *if-else* statements, not nested – so that diamonds are formed. We place arithmetic instructions on every branch (depending on the experiment, branches can have an equal or different number of instructions).
`fibonacci`	Computes the nth Fibonacci number using a while loop.
`factorial-iter`	Computes the factorial of the input using a do-while loop.
`factorial-rec`	Computes the factorial of the input using recursion.
`memset-loop`	Sets all elements in the input array to some arbitrary constant using a loop.
`memset-loop-const`	Sets all elements in the input array of fixed size to some arbitrary constant using a loop.

Existing algorithms to instrument Wasm code, as well as runtimes, are all implemented in Rust. To keep benchmarks simple, we use *Criterion.rs* [29] – a statistics-driven library for microbenchmarks. All measurements are taken after the warm-up of 1 second, over a period of 60 seconds.

For executing Wasm programs, we use *wasmi* [30] (version 0.20) – an interpreter-based VM. This runtime is not the most efficient, but it is chosen because it allows low-effort integration of the algorithms and libraries for computing the placement of the metering instrumentation, as well as the microbenchmarks. Because we are primarily interested in the relative performance of programs when instrumented differently, the real running time is less important.

7.2 Analysis of the run-time overhead of metering instrumentation

In this section, we consider Wasm microbenchmarks from Table 27. For each program, we investigate the slowdown caused by the metering instrumentation placed by different algorithms.

We compute the slowdown using the following formula

$$slowdown = \frac{T_{metered} - T_{unmetered}}{T_{unmetered}} \times 100\%,$$

where T denotes the running time of the program.

For comparison, we use the metering instrumentation placed by *wasm-instrument* (version 0.4.0) and *finite-wasm* (version 0.5.0) libraries and described in Section 4. We recall that *wasm-instrument* instruments multiple basic blocks at once, but the algorithm is based on Wasm structure, rather than dominance relationships. In contrast, *finite-wasm* uses a slower but safer approach to metering, instrumenting every instruction that may trap or may have side effects. It still tries to merge the instrumentation for multiple instructions together.

[29] https://bheisler.github.io/criterion.rs/book/criterion_rs.html
[30] https://github.com/paritytech/wasmi

In addition, we also consider a theoretical instance of the metering instrumentation algorithm which places the instrumentation generously, which we call the *GPP* algorithm. Essentially, the GPP algorithm represents an ideal scenario, where safety can be ignored, and only the placement of the metering instrumentation has to be minimized. While we have not developed such an algorithm, it is still interesting to compare the safe algorithm we developed to its generous alternative.

We compare the instrumentation produced by the aforementioned libraries and the GPP algorithm to the SPP algorithm presented in Section 6. For the GPP algorithm, we manually instrument microbenchmarks to obtain an optimal placement of the metering instrumentation without safety. The metering is implemented as a host function that updates the counter.

7.2.1 Slowdowns for simple programs with and without implicit control flow. We consider two simple straight-line workloads – `add` and `div`, varying the number of arithmetic instructions from 64 to 1024. The goal of this microbenchmark is to compare how different instrumentation placement algorithms perform if there is implicit control flow in the program. Figure 34 shows the slowdowns we observe.

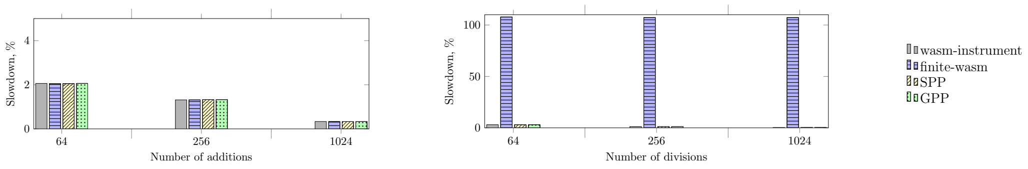


Fig. 34. Slowdowns for `add` and `div` programs, for a different number of instructions. Lower is better.

Because the `add` program consists of instructions that always return, all algorithms yield the same instrumentation – placing a single metering instruction at the start of the program. This leads to the same slowdowns across all algorithms.

Moreover, in such a case the overhead is nearly insignificant, particularly if the the size of the un-instrumented program is large enough. The overhead becomes observable (2%) only when the program consists of a few instructions.

When there is an implicit control flow in the program, e.g., due to possible traps, metering has to be done carefully. We observe that the programs instrumented by *finite-wasm* are 2× slower. This is because *finite-wasm* conservatively instruments at every instruction which traps, which leads to instrumenting the whole program. We conclude that for the NEAR blockchain, the possibility of such a performance regression is not important, in particular when compared to the security aspects of deterministic metering.

In contrast, we see that the SPP algorithm places the instrumentation optimally: a single metering instruction at the start of each sequence. Such an optimistic instrumentation allows to have only 1-2% run-time overhead. However, the algorithm must still be consistent, and as seen in Section 4 *wasm-instrument* fails to do so even though it yields similar run times.

7.2.2 Slowdowns for trees and diamonds. Next, we consider the slowdowns caused when the programs have explicit control flow. First, we consider the `tree` benchmark which is a program consisting of many early returns, with its CFG forming a tree. The program, as well as the instrumentation produced by the two algorithms, is shown in Figure 35.

In our experiments, when the whole tree is traversed, we observe that the existing solutions (*wasm-instrument* and *finite-wasm*) have significant run-time overhead because the instrumentation is placed at every basic block in CFG. Moreover, the SPP algorithm also does not improve the running time. This is because the algorithm moves the instrumentation to the start of the program, leaving the leaves of the CFG empty.

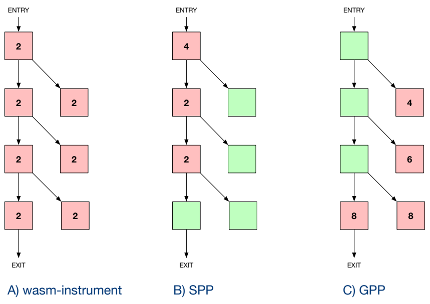


Fig. 35. Instrumentation for `tree` program. Metered basic blocks are shown in red, with costs written in bold. Not metered blocks are green. We observe that without safety, the metering function for the CFG has significantly less metered vertices (C).

In contrast, the GPP algorithm places the instrumentation better – in the leaves of the CFG, as shown in Figure 35C. This means that there is a single metering instrumentation per path in the program, which makes the performance very close to the performance of non-instrumented code. Hence, we conclude that safety limits the optimality of the placed instrumentation when the control-flow graph has many leaves.

Next, we consider four instances of the `diamond` microbenchmark, called `equal`, `cheap`, `random`, and `expensive`. All of them have 1024 diamonds in sequence. The `equal` program has equal costs for all branches in the diamond. The other three programs have non-equal costs between branches, and differ in the condition that transfers control flow: either a branch of the lowest cost is chosen, a branch is chosen uniformly at random, or the most expensive one is selected. We present the slowdowns observed due to instrumentation in Figure 36.

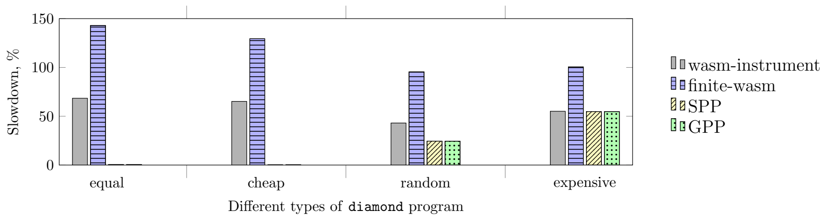


Fig. 36. Slowdowns for the `diamond` program with 1024 diamonds. Lower is better.

We observe that *finite-wasm* is not even optimizing the instrumentation placement for blocks that are always dominated, which causes more than 2× slowdown on all instances.

We also observe that both SPP and GPP algorithms outperform state-of-the-art approaches on `equal` and `cheap` instances because they take the costs of branches into account. In both cases, the execution path has a single instrumentation placed at the start of the program. As a result, the slowdown is reduced to almost 0%.

On the `expensive` instance, the instrumentation placed by the SPP and the GPP algorithms yields the same slowdowns as when using *wasm-instrument*. This is because, during execution, each taken branch always contains the instrumentation.

We see a similar trend on the `random` instance, however, this time using the algorithms we developed lead to smaller performance regressions of 24% compared to 43% of *wasm-instrument*. The slowdown is almost 2× smaller because each branch is taken with a probability of 0.5.

We conclude that for diamonds unless the costs of branches are the same, there is very little that can be done. One way to improve the metering instrumentation is to place it based on the branch frequencies obtained via profiling. This approach, however, does not always meet the safety requirements. We also conclude that for a sequence of diamonds, safety does not impact the optimality of instrumentation.

7.2.3 Slowdowns for recursive programs. Next, we consider a recursive program which computes a factorial, `factorial-rec`. The CFG of the program is a single diamond, where one branch calls the factorial function recursively. Figure 37 shows the slowdowns for the program when one or the other branch is taken.

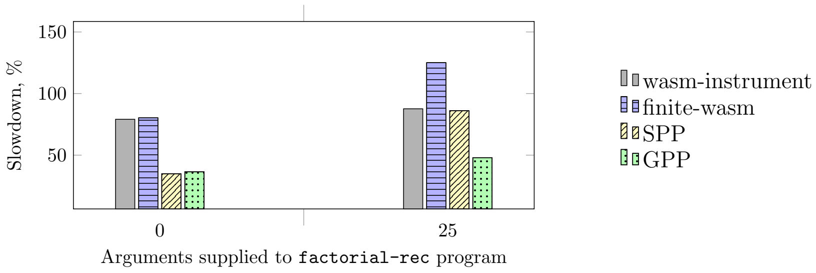


Fig. 37. Slowdowns of the `factorial-rec` microbenchmark when there are 0 or more recursive calls. Lower is better.

We observe that existing algorithms, i.e., *finite-wasm* and *wasm-instrument*, are very conservative and place 4 and 3 metering instrumentations (Figure 38A-B respectively), which results in significant slowdowns of 70-80% on average, with *finite-wasm* performing particularly worse when calling `factorial` function recursively many times.

In contrast, the SPP and the GPP algorithms produce minimal instrumentation, as shown in Figure 38C-D. For the SPP algorithm, the slowdowns are greater when `factorial` is called recursively because it places additional instrumentation in the first basic block. We observe that relaxing the safety constraint can significantly improve the run time (1.8×) of the metered program.

It is worth mentioning that the factorial program is tail-recursive and can be optimized into a *while* loop, which we look at next.

7.2.4 Slowdowns for while and do-while loops. Here, we evaluate the performance of the SPP algorithm when using *while* and *do-while* loops. We use two microbenchmarks: `factorial-iter` which uses a *do-while* loop, and `fibonacci` which uses a *while* loop.

First, we look at `fibonacci` and the slowdowns caused by the instrumentation, shown in Figure 39.

As always, for the experiments, we consider different inputs to obtain profiles over multiple paths in CFG. First, we observe that if the loop is not taken and the program terminates early, the instrumentation produced by the SPP and GPP algorithms has very low overhead, accounting for approximately 40% on our benchmark. This is a drastic improvement compared to 80% overhead when using *wasm-instrument* of *finite-wasm*, which instrument at least three basic blocks along that path, as shown in Figure 40A-B.

A)
```python
def factorial(n):
  meter(4)
  if n == 0:
    meter(1)
    return 1
  else:
    meter(1)  # call
    meter(5)  # remaining
    return n * factorial(n - 1)
```

C)
```python
def factorial(n):
  meter(5)
  if n == 0:

    return 1
  else:

    meter(5)
    return n * factorial(n - 1)
```

B)
```python
def factorial(n):
  meter(4)
  if n == 0:
    meter(1)
    return 1
  else:

    meter(6)
    return n * factorial(n - 1)
```

D)
```python
def factorial(n):

  if n == 0:
    meter(5)
    return 1
  else:

    meter(10)
    return n * factorial(n - 1)
```

Fig. 38. Instrumentation of `factorial-rec` program, shown in Python for easier readability. A) Instrumentation produced by *finite-wasm*, B) Instrumentation produced by *wasm-instrument*, C) Instrumentation produced by the SPP algorithm, D) Instrumentation produced by the GPP algorithm.

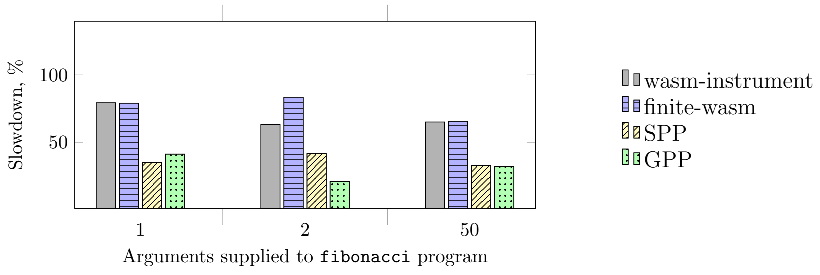


Fig. 39. Slowdowns of the `fibonacci` microbenchmark for different execution paths. Lower is better.

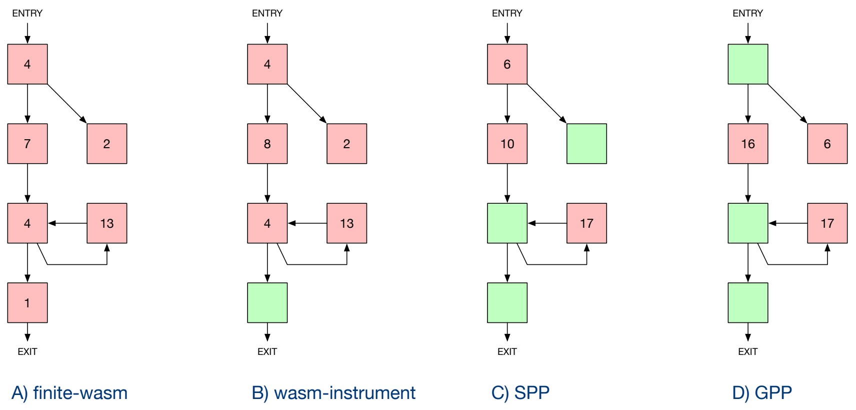


Fig. 40. Metering functions for the CFG of `fibonacci` program. A) Metering function computed by *finite-wasm*, B) Metering function computed by *wasm-instrument*, C) Metering function computed by the SPP algorithm, D) Metering function computed by the GPP algorithm.

Because the SPP algorithm favours the cheapest branches, it leaves the instrumentation both in the block where the control flow diverges and in the more expensive branch. As a result, we observe that when the input is 2, the GPP algorithm can produce smaller instrumentation (Figure 39C-D), which leads to a slowdown of only 20% compared to 40% with the SPP algorithm.

Finally, when the loop is executed many times, we observe that the performance of the SPP and the GPP algorithms balances out: the run time of the loop dominates the run time of the program. Both algorithms are better than the state-of-the-art approaches, halving the slowdown of 30% compared to 60% with *wasm-instrument* of *finite-wasm*.

We consider `factorial-iter` microbenchmark next, which uses a *do-while* loop instead. The slowdowns we observe are presented in Figure 41.

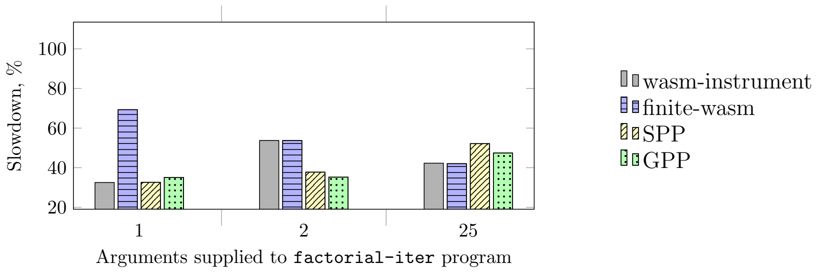


Fig. 41. Slowdowns of the `factorial-iter` microbenchmark for different execution paths. Lower is better.

Because for the SPP algorithm critical edges are split, we had to adapt the code of `factorial-iter` program to split the back-edge of the *do-while* loop and an edge which returns early, as shown in Figure 42C-D. Because Wasm uses structured control flow, we have to introduce an additional branch and `block` instruction inside the loop.

We observe that when the factorial returns early, *finite-wasm* experiences the worst slowdown of approximately 70% compared to 32-35% when using alternative instrumentations. This is because the last basic block is instrumented, as Figure 42A shows. We also note that the program instrumented with the GPP algorithm is slightly slower than the programs obtained with *wasm-instrument* and the SPP algorithm. We believe that this is due to an additional basic block introduced solely for metering. Hence, we conclude that less optimal safe metering instrumentation can have better run time compared to more optimal instrumentation produced by the GPP algorithm.

If the loop body is executed only once, the SPP algorithm outperforms state-of-the-art approaches, with a slowdown being around 36% compared to 53%. Figure 42 clearly explains why this is the case: *wasm-instrument* and *finite-wasm* instrument one more basic block while the algorithm we developed moves the instrumentation to the back-edge.

When the loop is executed many times, the run time is dominated by the loop body. In this scenario, we see that because of the additional basic block for metering added by the SPP algorithm, state-of-the-art instrumentation algorithms have slightly better run time: 42% against 47-50%.

We conclude that hoisting the instrumentation to the back-edge might not always be profitable: additional `block` and branch instructions we added lead to the small run-time overhead. We note that, however, the results in native code might differ: because we use *wasmi* interpreter extra bytecode instructions can impact the runtime, while the extra branch in, say x86 assembly, can be still relatively fast. In any case, the results we present here show that optimal instrumentation placement might not always lead to the best running times.

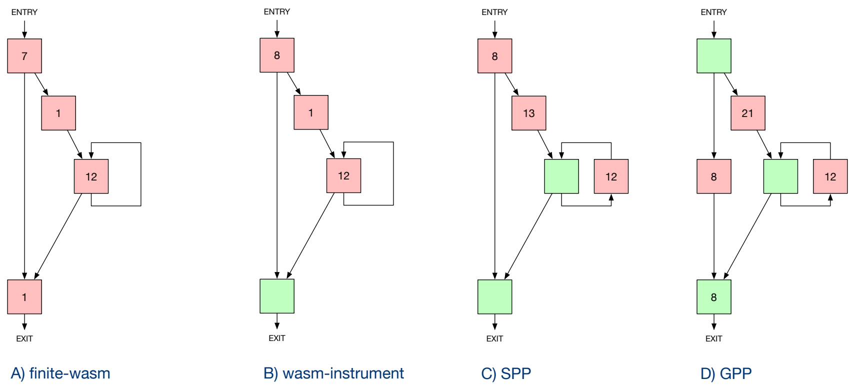


Fig. 42. Metering functions for the CFG of `factorial-iter` program. A) Metering function computed by *finite-wasm*, B) Metering function computed by *wasm-instrument*, C) Metering function computed by the SPP algorithm, D) Metering function computed by the GPP algorithm.

7.2.5 Summary. Based on microbenchmarks we observe that:

- For paths that do not dominate the run time of the program, having multiple instrumentations can be acceptable, as long as other instructions dominate the cost. For example, the instrumentation produced for diamonds by the SPP algorithm.
- In the presence of leaves in the CFG, placing instrumentation in leaves yields significant performance benefits. Then, the SPP algorithm is not optimal, yielding the same run-time overheads as the state-of-the-art approaches.
- Splitting critical edges might not be beneficial, in particular, if we split a back-edge and the loop body is not large. The code needed for extra basic block jumps, in particular in the case of Wasm and interpretation, introduces more overhead than metering instrumentation placed in the loop body.
- We observe that for *while* loops, it is better to shift the metering away from the critical path. For *do-while* loops, it might be better to keep the instrumentation in place. We note that one can always convert the *do-while* into *while*, thus improving the run time.

With these observations, it is possible to adjust the SPP and the GPP algorithms to produce slightly less optimal instrumentation that yields better running times, e.g., based on profiling heuristics.

7.3 Analysis of the impact of Loop Invariant Metering Motion on the run time

Next, we consider how Loop Invariant Metering Motion (LIMM) can decrease the running time of the metered execution and bring it closer to the non-metered one. We consider a `memset-loop` program from the selected microbenchmarks – it has a single loop that sets an array of the given input size to an arbitrary constant.

For the experiments, we consider three variations of the program: 1) unmetered, 2) metered with the SPP algorithm, and 3) metered with the SPP algorithm and with LIMM applied to it. We run experiments for different input sizes: from 0 to 8192 in increments of powers of two. The slowdowns of the metered instances of `memset-loop` for a different number of loop iterations are shown in Figure 43.

We observe that the program instance with the cost of the body hoisted out by LIMM is significantly slower for a small number of iterations.

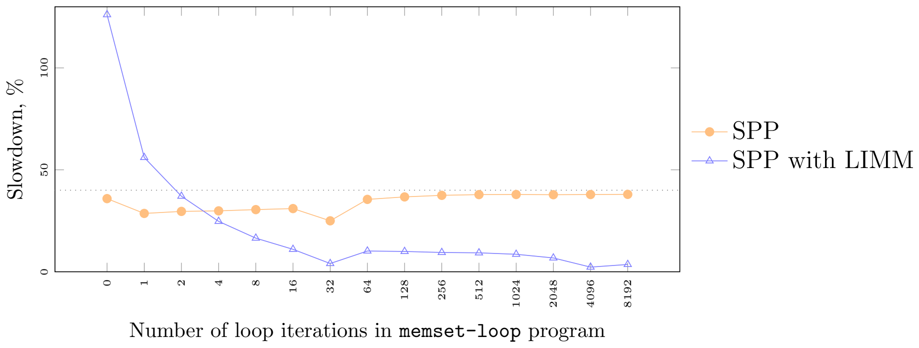


Fig. 43. Slowdown caused to `memset-loop` due to metering computed by the SPP algorithm and by the SPP algorithm with LIMM. Lower is better.

To see why, first recall that for `memset-loop` program, the number of iterations of the loop is non-constant, so one has to be careful with the multiplication of the cost by the number of iterations. There, an overflow must be avoided because it is a security issue: it would make the execution cost of the loop body to be zero.

Hence, the instrumentation placed by the LIMM has to first check for an overflow, in which case the program halts for not having a sufficiently large budget. If the check passes, the multiplication is computed and the cost is metered.

This means that if the loop is not taken, there are extra metering instructions that have to be executed. While in native code that would be just two assembly instructions (multiplication and jump on overflow), in Wasm there are many more which can degrade the performance when using an interpreter. As we see in Figure 43, this is indeed the case, which makes the program more than 2× slower.

At the same time, as the number of iterations increases, the slowdown of the program instrumented with the SPP algorithm converges to around 40%. We suspect that such a high cost of the instrumentation is due to the metering implemented as a host function and the use of an interpreter.

In contrast, the slowdown for the program instrumented by the SPP algorithm with LIMM reduces rapidly and converges towards zero. This happens because the loop body starts to dominate the run time, and the overhead of instrumentation should converge to zero as n grows.

In addition, we verify the hypothesis that the overflow checks for hoisted outside of the loop metering cause regressions on `memset-loop` benchmark. We consider another microbenchmark – `memset-loop-const`. This program is identical to `memset-loop` but instead of a user-provided array size uses a constant-sized array. As a result, the cost of the loop is hoisted out without additional overflow checks. Figure 44 shows the slowdowns we observe.

Indeed, we observe that this time LIMM allows one to reduce the slowdown to approximately 5% for the large number of iterations. Meanwhile, even if the number of iterations is small, the observed slowdowns are never larger than the slowdowns of simple instrumentation computed by the SPP algorithm.

While the focus of this work was not LIMM, we observe that doing this optimization for simple programs can yield significant performance benefits for the instrumented programs. Still, if the loop bounds are not constant, the instrumentation algorithm should take into account the cost of the loop body, the cost of the hoisted instrumentation, and the branch frequency information if available to decide if the LIMM is profitable or not.

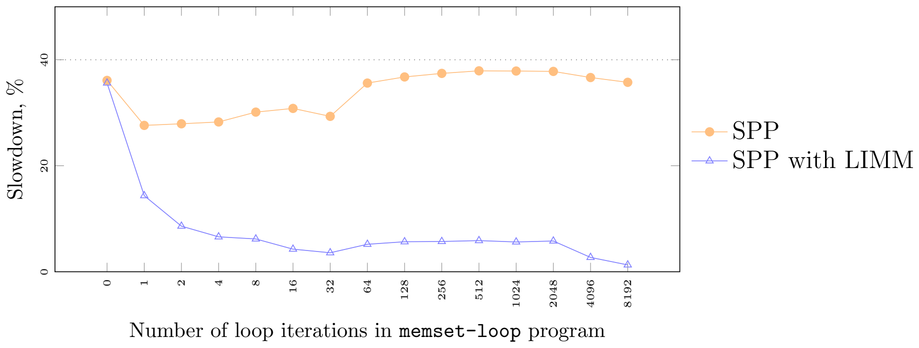


Fig. 44. Slowdown caused to `memset-loop-const` due to metering computed by the SPP algorithm and by the SPP algorithm with LIMM. Lower is better.

7.4 Analysis of the SPP and the GPP algorithms on Solana contracts

We now analyze the metering function computed by the SPP algorithm in more detail, this time using more realistic data – CFGs extracted from Solana contracts.

7.4.1 Convergence of the fixed-point SPP algorithm. First, we look at the run time of the SPP algorithm, analyzing the number of iterations it takes to converge. Figure 45 shows the convergence rate for different CFGs we selected for experiments.

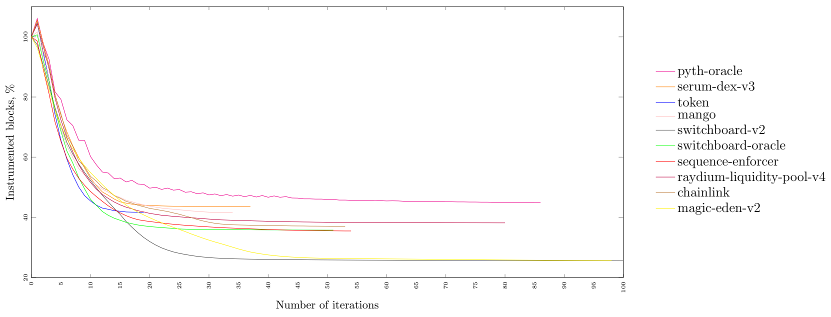


Fig. 45. The rate of change in the number of metered basic blocks over the iterations of the SPP algorithm. For *magic-eden-v2* contract, the number of iterations is capped at 100 for visibility.

We observe that for all CFGs we consider, on average 30 iterations are sufficient to compute nearly optimal valid and safe metering function. Certain contracts take many iterations to converge: `pyth-oracle`, `switchboard-v2`, `raydium-liquidity-pool-v4`, and `magic-eden-v2`. However, for all these contracts the changes to the metering function are marginal.

We conclude that it can be sufficient to run the SPP algorithm for a fixed number of iterations, to ensure predictable running times and ensure that adversarial programs do not take a very long time to converge.

7.4.2 The number of metered vertices in a valid metering function. Next, we evaluate the number of metered vertices in the metering function computed by the SPP algorithm. Because the problem of the minimum metering instrumentation has not been studied previously, there are no general state-of-the-art algorithms that place the metering instrumentation (i.e., compute the metering function) for arbitrary CFGs.

To add more variety to our evaluation, we additionally consider three algorithms (not necessarily optimal, safe, and practical for the placement of the metering instrumentation), shown in Table 28.

Table 28. Additional algorithms which can also be used to compute a valid metering function.

Name	Description
DomTree	Algorithm which instruments the leaves of the dominator tree. As shown in Section 6, it is sufficient to meter the leaves. We note that the instrumentation based on this algorithm does not yield a safe and path-minimal metering function.
PostTree	Same as DomTree, but instead the leaves of the post-dominator tree are used. For this algorithm, we add a unique fake exit vertex to ensure that the post-dominator tree exists even if the contract has multiple exits.
MST	The metering function is computed based on the algorithm developed by Knuth to solve the optimal profiling problem [52, 53]. The goal of optimal profiling is to instrument a minimal set of vertices in a CFG with integer counters so that from the frequency counts of instrumented vertices all remaining counts can be inferred. The metering function can be obtained by multiplying the frequency count of a vertex by its cost.

One can observe that the MST algorithm from Table 28 cannot be used in practice to place the metering instrumentation. For metering, costs have to be checked at run time, to ensure that the execution stays within the budget. The MST algorithm, in contrast, first computes the frequency counts and only after the metering function. If there are no cycles in the control-flow graph, the MST algorithm is only $|\pi|$-safe where π is the longest entry-exit path in the program. However, if there are cycles in the program, there are no guarantees of termination. While the MST algorithm is impractical, it is still interesting to compare how the number of metered vertices (instrumented basic blocks) compares with the instrumentation placed for optimal profiling.

We now compare how many vertices are metered for different CFGs of real Solana contracts, when using the algorithms listed in Table 28 and the SPP algorithm we developed in this work. First, we split all critical edges beforehand prior to running the algorithms. We present the experimental results for such a scenario in Figure 46, showing the percentage of metered vertices.

We observe that both DomTree and PostTree yield the highest percentages of metered vertices on all benchmarks: 38.7% and 43.6% on average. Interestingly, we observe that using the leaves of the dominator tree compared to those of the post-dominator tree, on all but one contract, results in a more optimal metering function, with fewer vertices being metered.

We suspect that this pattern can be explained by branching patterns in CFGs. For example, inspecting the `mango` contract, we observe that its vertices have many predecessors, which means there are more leaves in the post-dominator tree than in the dominator tree. Because using the leaves of dominator trees reduces the number of metered vertices, we conclude that in Solana contracts, it is more common for control flow to diverge than to converge.

We observe that the MST algorithm is more optimal than the DomTree and the PostTree algorithms, collecting the frequency counts only for 31.9% of vertices, on average. Similarly, the SPP algorithm meters on average 30.4% of all vertices, performing slightly better than the MST algorithm.

Our experiments empirically prove that the placement of the metering instrumentation requires less instrumentation (-1.5%) than the optimal profiling problem if critical edges are split. Indeed,

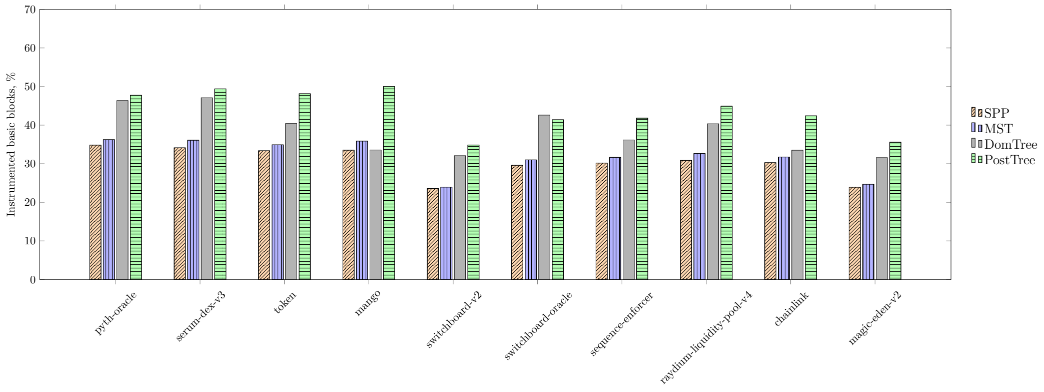


Fig. 46. Percentage of metered vertices using different algorithms: SPP, MST, DomTree, and PostTree. The total number of vertices used to calculate the percentage is the sum of the number of vertices in the CFG of a contract, and the number of critical edges. Lower is better.

when computing frequency counts, the values of the counters cannot be combined together. For metering, it is possible to merge multiple instrumentations if they meter the same cost.

It is important to mention that the SPP, the DomTree, and the PostTree algorithms rely on splitting critical edges. As a result, when evaluating the number of instrumented vertices previously, the split edges were taken into account, thereby decreasing the percentage of instrumented vertices. Hence, in Figure 47 we show the percentages of instrumented basic blocks calculated based on the size of the original vertex set. We therefore additionally consider another algorithm: *MST-no-split* which is identical to MST, but calculates the frequency counts without splitting critical edges.

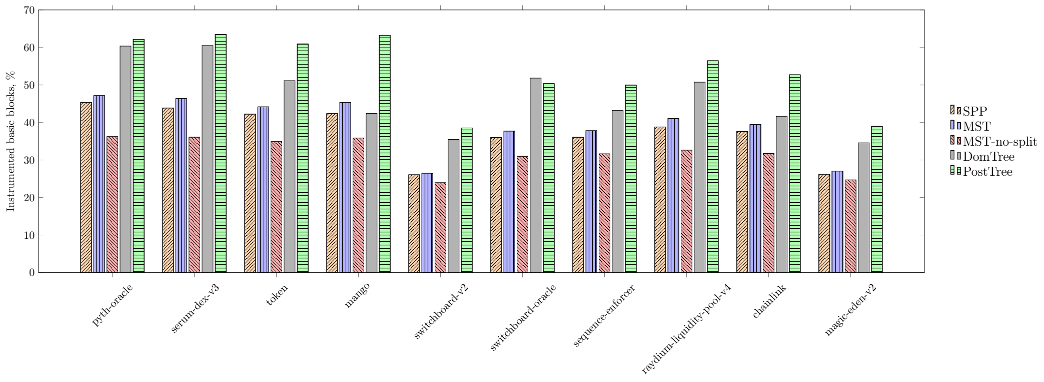


Fig. 47. Percentage of metered vertices using different algorithms: SPP, MST, DomTree, and PostTree. The total number of vertices used to calculate the percentage is the number of vertices in the original CFGs before critical edges are split. Lower is better.

We observe that because contracts have a large proportion of critical edges as observed in Section 5, the percentage of instrumented vertices grows significantly for some of the contracts.

In particular, we see that using dominator and post-dominator trees leads to 8.5% and 10.1% additional instrumentation, producing in total 47.2% and 53.7% instrumented basic blocks. Similarly, the instrumentation produced by the SPP algorithm increases to 37.5% compared to the previous value of 30.4%. We observe that because `switchboard-v2` and `magic-eden-v2` have the smallest fraction of critical edges compared to other contracts, the percentage of instrumented blocks does not change significantly for these programs.

We also observe that the MST algorithm, without splitting edges, produces more optimal instrumentation, instrumenting only 31.8% of blocks, on average. Hence, we note that while our technique to split critical edges minimizes the number of instrumented basic blocks, it has its overheads – more vertices are added to the program solely for metering. Still, it is not clear if newly added vertices worsen the run time significantly.

7.4.3 Summary. Based on our experiments, we observe that:

- On real contracts, the convergence rate of the SPP algorithm can vary greatly. Still, on average a small number of iterations is needed to obtain a nearly optimal placement of the metering instrumentation which is also safe. Having small cycles with low costs leads to a large number of iterations.
- Sub-optimal instrumentation using dominator and post-dominator leaves has a significantly larger percentage of instrumented basic blocks compared to the SPP, while also not satisfying the safety requirement. In particular, up to 10.1% more basic blocks can be instrumented.
- The SPP algorithm instruments 30.4-37.5% basic blocks, on average.
- Splitting critical edges, while yielding easier implementation and construction of the SPP algorithm has its overhead. For example, the state-of-the-art approach developed for optimal profiling problem instruments 5.7% fewer blocks if critical edges are not split compared to the SPP algorithm.

7.5 Analysis of the cost of consistency recovery and the impact of the cost model

Finally, we analyze how expensive the recovery mechanism proposed in Section 6 is. Here, we have to consider two scenarios: 1) a malicious contract runs out of budget and 2) a malicious contract throws an exception.

7.5.1 The cost of consistency recovery when running out of budget. We recall that to recover consistency, the execution continues on running out of budget until an exception is thrown or another metering instrumentation is found. Since we consider the case where the execution runs out of budget, the additional work a validator has to do is equivalent to $c(\pi)$, where c is the cost function and π is the path in the CFG between the metering instrumentation where the out-of-budget is detected for the first time and the metering instrumentation where it is detected for the second time. Therefore, in the worst case, the cost of recovering consistency is equal to the cost of the largest interval between a pair of instrumented points in the program.

7.5.2 The cost of consistency recovery when throwing an exception. If the execution fails due to an exception, the validator has to recalculate the actual cost. One can observe that the complexity of recalculation depends if refund tables have been pre-computed or not.

Here, we consider a scenario when the refund tables are computed online. In this case, the additional work the validator has to do is bounded by

$$\sum_{c \in S_{call}} |V_c|$$

where S_{call} is the call stack and V_c is the number of vertices in the control-flow graph of a function $c \in S_{call}$.

7.5.3 Summary. A well-designed cost model must ensure that even if the contract runs out of budget, or throws an exception, the user who submitted the contract pays for the additional work performed by the validator.

This can be enforced by ensuring that the user has a sufficient balance to cover for recovery costs at all times. This can be trivially upper bounded by taking the maximum of

- the sum of the number of vertices in functions that lead to the deepest call stack, and
- the cost of the longest interval between any two instrumentation points in the program and its callees.

We observe that such a bound is not straightforward to compute, and therefore even larger bounds can be used by validators as long as they are efficiently computable. For example, the size of the contract and all other contracts it depends on can be used.

8 REVISITING COST MODELS FOR BLOCKCHAINS

Having studied existing cost models in Section 3 and their implementations (metering) in Section 4, we observe that the models have been designed with the only goal in mind – to approximate the cost of real execution of a contract. This design leads to multiple problems, including but not limited to inefficient metering or poorly written low-cost code which can impact the earnings of validators and the liveness of the blockchain itself.

We believe that the source of the problem is the approach taken when a cost model is designed. Typically, a cost model is created first. Then, based on the cost model developers or compilers optimize contracts to reduce execution costs. When a contract is executed, validators use the existing cost model to meter execution as efficiently as possible. In short, the cost model does not incentivize smart contract developers or compilers to produce good and efficient programs that can be easily metered.

In this section, we propose a new approach to designing a cost model for blockchains (or similar domains), which we call the Cost Model Standard (CMS). The goal of the CMS is to define a set of rules and guidelines which incorporate the economic interests of all parties – smart contract developers, compilers, and validators, as well as to penalize abnormal and malicious behaviour. Essentially, this enforces certain semantics on the cost model, which can be used to reason about the smart contract language and the runtime. Such a standard enables multiple implementations of compilers, VMs, and cost models, to boost the diversity in the blockchain space. While there does not exist an ideal cost model, we believe that the CMS is a first step forward in this direction.

We start by considering the stakeholders involved in the development and execution of contracts, listed in Table 29. For each stakeholder, we consider what they supply to the system, and what it needs in return.

Table 29. Stakeholders in the blockchain system. Each stakeholder can bring value to the system (supply), as well as needs something in return (demand).

Stakeholder	Supply	Demand
Smart contract developers	Popular, useful, and low-cost contracts. If the contract is cheap, there might be a higher demand for it.	A reliable compiler that produces the cheapest possible code. Just like in regular software, developers do not want to manually optimize contracts.
Compilers	Produce low-cost code.	A cost model to know what to optimize for.
Validators	Provide their computational resources and execute contracts for a fee.	Since validators want to maximize their profits, they prefer fast and expensive contracts, which also admit efficient metering. The last condition is crucial, as we have seen that metering can significantly reduce the profits of validators.
Blockchain	The overall infrastructure, including distributed consensus and long-term storage.	Reliable validators, contract developers to write applications that attract users, many users to saturate the network capacity, and a cheap cost model to be competitive with other platforms.
Regular users	Transactions to be executed.	A reliable and cheap system.
Malicious users	Halt the blockchain network, affect its performance or bankrupt validators.	Minimize the fees paid when attacking.

With that, we can list the requirements CMS must enforce for any cost model. Ideally, it should ensure that the involved stakeholders can (or cannot in the case of malicious users) reach their goals but also satisfy their demands.

(1) The cost model should be a deterministic approximation of the execution cost, e.g., the hardware and the software a validator uses to run contracts. This requirement has been used by existing cost models already. Moreover, the cost model should avoid leaking implementation details when assigning costs.
(2) The cost model should prevent malicious users from attacking the network. We note that cost models usually rapidly scale the cost of execution as the resource consumption grows when a contract is executed. For example, if we recall the equation for the cost of memory expansion on Ethereum we observe that the cost grows linearly for the first 724 bytes, and then becomes quadratic for the number of bytes expanded. However, this is a reactive measure, i.e., when a malicious user attempts to consume a lot of memory, the cost is calculated. Some cost models, e.g., those by Aptos or Solana instead opt for enforcing hard limits on the resource consumption, which can be limiting for applications. Therefore, the cost model, where possible, should enforce a set of *preventive measures* to ensure that the network cannot be attacked. We discuss such measures later in this section.
(3) The cost model should be efficiently implementable. Where possible, the cost model should incentivize programs that can be metered efficiently.

Based on the aforementioned requirements, and our findings from the previous sections, we now present the CMS standard. In the rest of the section we present a set of rules any cost model under the CMS must satisfy, as well as guidelines on how certain rules can be enforced in practice.

8.1 The Cost Model Standard 1.0

8.1.1 Base cost. A cost model must always account for the fixed amount of work used to execute a transaction. The cost should account for at least the following: the start-up cost of the VM, transaction validation, long-term storage of the transaction details, and the consensus protocol. The base cost must be fixed because the implementation details cannot be leaked outside of a particular protocol implementation.

The base cost has to be calculated based on the system and the load of the network (e.g., transaction history). It has to account for the cost of the common case (e.g., taking an average over a sliding window of the most recent transactions, or a median).

8.1.2 Code size cost. It is crucial to have a fee associated with the size of the contract. The cost should be proportional to the code size, e.g., $O(n)$ for a contract with n instructions.

There are multiple factors why the code size has to be taken into account by the cost model.

(1) Code is stored on-chain, and there should be a fee for long-term storage.
(2) Code validation (or running any analysis at run time) is done for the whole contract. Accounting only for executed instructions is not sufficient because then the analysis of the not executed code is not metered.
(3) JIT compilation, if available, can also compile the whole contract. Again, this suffers from the same problem as code validation.

Such a design also mitigates an attack vector where some dead code can be injected into the contract which is very computationally-intensive to analyze but is never executed.

8.1.3 Code analysis cost. On-chain code can be analyzed and processed at run time, e.g., for bytecode verification or for placing the metering instrumentation. In this case, the cost for code analysis should also be included depending on the run-time of the algorithms.

In practice, algorithms have to be linear in program size (or worst-case sub-linear, e.g., $O(n \log n)$ for code with n instructions or n basic blocks) as otherwise, the cost would grow astronomically fast. In this case, the cost of analysis is already taken into account by the cost of the code size.

While this approach incentivizes the design of better linear-time algorithms, it is not always possible to achieve that. For example, algorithms that use fixed-point iteration (which are popular in bytecode verification analyses) can be a vulnerability. While in practice, these algorithms have a small number of iterations, each with an effective complexity of $O(n)$, there exist program instances for which a fixed-point iteration can have non-linear polynomial complexity.

In these cases, there are different ways the problem can be mitigated. We suggest:

- to associate a per-iteration cost (or cost for each meet operation of a fixed-point), or
- to limit the number of iterations.

For example, in Section 7 we saw that the iterative SPP algorithm achieves nearly optimal placement of the metering instrumentation under 30 iterations, whereas on certain contracts it takes more than 100 iterations. In this case, using a bounded number of iterations is reasonable.

We also note that it is better to avoid performing code analysis at run time, as suggested by the following guidelines:

- If possible, the analyses are performed at the contract deployment time, either when a contract is created or upgraded. This is particularly applicable to bytecode verification, which we would use as an example of how the proposed framework would work.
- Once the code is deployed to a blockchain, it cannot be tampered with, and therefore if it passed verification before it would so still. [31] This means that extra run-time overhead and additional costs are only added at deployment time.

8.1.4 Instruction costs. Instructions costs have been a good approximation of the actual execution costs. Therefore, under the CMS all instructions have to be assigned a constant cost (there can be extra dynamic costs for certain instructions still, but should be discouraged). The existing cost models have already been doing that. However, one has to also scale the cost of instructions with implicit control flow.

8.1.5 Transaction failure cost. The cost model should, where possible, use a deposit-based system to incentivize successful execution of transactions and penalize for abnormal termination.

We recall that there are multiple ways when executing a smart contract can result in an unexpected failure. For example, if the VM enforces a stack limit when executing a program, the program that violates the limit has to be halted. Similarly, if the program runs out of the user-defined budget, the execution abnormally terminates.

To date, when submitting a transaction to the blockchain, users also submit a maximum fee they are willing to pay for the execution. The fee can be typically estimated by locally simulating the execution of a transaction based on the most recent state of the blockchain.

In addition to the fee, we propose to use a failure deposit. The role of the deposit is to freeze a fixed amount of the sender's balance before a transaction is executed. If the transaction successfully terminates within the specified budget or terminates with an expected failure (e.g., failed assertion), the deposit is returned. Otherwise, the deposit is consumed.

The deposit system achieves multiple goals:

- First, more optimal metering instrumentation placement algorithms can be used by validators. We have observed in Section 6 that it is sufficient to enforce the validity of metering

[31] An argument against such an approach would be that the bytecode verifier itself can have a bug, which can be fixed after some contracts are deployed. However, we do not believe such logic should guide the decision-making, as any software can have bugs.

placement and use a recovery mechanism to achieve consistency when the program abnormally halts. Setting the right deposit, proportional to the cost of recovery, allows one to optimally place the instrumentation, increase the revenue of validators, and ensure faster execution of transactions. In case the program runs out of budget, thereby violating consistency, the deposit is large enough to account for the work a validator has to do to recover the correct state.
- Second, such a deposit incentivizes developers to write robust and well-tested code. For example, contract developers have to ensure that the code does not terminate with unexpected exceptions, such as divide-by-zero, by inserting more checks where necessary. The compiler or the runtime can then determine if the check is redundant and optimize it away.
- Lastly, the deposit system only affects malicious users which aim at slowing down or halting the network by using code that abnormally terminates.

We note that the deposit fee system does not open a way to arbitrage and does not influence transaction selection for execution. Firstly, it can simply be not included in the fees distributed to validators. At the same time, even if it is, validators are unlikely to prioritize transactions with large failure deposits. We believe that this is due to the following reasons.

- A higher failure deposit means a higher cost of recovery, and therefore for the validator, it becomes a high-risk gain.
- The fact whether the program will fail unexpectedly is not deterministic, and in general cannot be predicted by a validator.
- Contract developers and users are aware of such an arbitrage and are incentivized to write code that is robust to these cases and is guaranteed not to fail.

We note that failure deposits impact the predictability of the latency and make it harder to pack transactions into the block with a fixed cap on the maximum amount of computation. However, deposits are designed so that they are not paid normally, and therefore we believe that they can be disregarded when estimating transaction latency.

8.1.6 Bucketed billing. If possible, a bucket-based billing mechanism should be preferred. This approach has been already used by Sui network [76], and based on their documentation [32] because of the two reasons:

- it incentivizes developers not to optimize contracts for marginal gains, and instead prioritize optimizations that lead to significant cost reductions, and
- the developers of the Sui protocol can adapt the cost model without introducing additional costs for users or disrupting their services, thereby allowing gas costs to stay stable over time.

While such an approach and reasoning are favourable, we note that there is an additional important benefit. Having bucket-based billing allows to avoid high failure deposits and computationally expensive recovery mechanisms.

Consider an example shown in Figure 48. If the SPP algorithm for placing the metering instrumentation from Section 6 is used, then when the program aborts on implicit exception, the actual cost has to be calculated. However, with the bucket-based approach, if the last metered cost fits within the bucket, its cost does not have to be recalculated. We note that recovery is still needed to differentiate between out-of-budget and program exceptions.

8.1.7 Control-flow costs. We have seen that control flow is a problem for smart contracts. First, the execution time can vary unpredictably in the presence of branches, as we have observed with

[32]https://docs.sui.io/learn/tokenomics/gas-in-sui

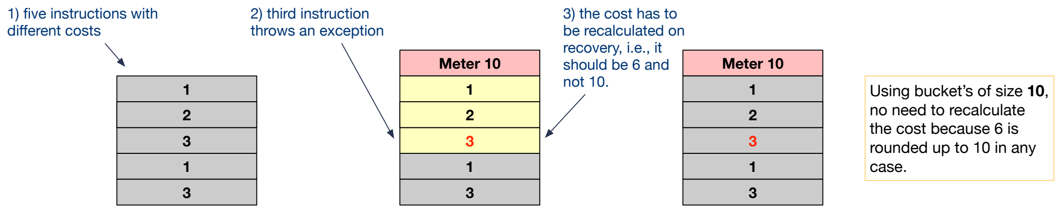

1) five instructions with different costs
2) third instruction throws an exception
3) the cost has to be recalculated on recovery, i.e., it should be 6 and not 10.
Meter 10
Meter 10
1
2
3
1
3
Using bucket's of size 10, no need to recalculate the cost because 6 is rounded up to 10 in any case.

Fig. 48. Example, where using the bucket-based instruction cost billing allows one to avoid the recovery mechanism if instruction throws an exception. Here, if buckets have a cost of 10, then the real cost to reach instruction that throws is not needed.

SBF programs for instance. Moreover, the control flow also complicates the implementation of the cost model – the placement of the metering instrumentation. For example, the SPP algorithm presented in Section 6 splits critical edges, and similarly, state-of-the-art approaches place the instrumentation when the control flow diverges.

Therefore, the cost model has to take control flow into account. We note that the higher costs for branching instructions can already be used to implicitly price the divergence in control flow. However, we propose to have additional costs for critical edges.

Splitting critical edges is a well-known technique in compilers, used to enable some optimizations or better register allocation, and to be able to place instrumentation for profiling, computing coverage, or as we have seen – metering. Moreover, when translating from Static Single Assignment (SSA) form critical edge splitting is also used [41].

Additionally, the cost model must have a cost assigned to loops. While loops are not common in smart contracts as we have seen in Section 5, if they are used, they are typically located on the hot path throughout the execution and can account for the majority of the run time.

Loops (or even worse, arbitrary cycles in the control-flow graph) in particular cause problems with placing the metering instrumentation. Therefore, we propose to have an additional cost associated with cycles in the control-flow graph, based on the following taxonomy:

(1) natural loops, with known iteration space,
(2) natural loops, with unknown iteration space,
(3) cycles due to unstructured control flow.

The rationale is the following. If the loop has known iteration space, e.g., an affine loop, then it enables more optimizations, allowing for faster execution, such as LIMM to hoist the metering instrumentation outside of the loop.

Otherwise, the metering instrumentation has to be kept inside the loop body. Moreover, loops can be implemented as a random-access pattern, and so such loops are computationally more expensive.

We in particular distinguish between structured and unstructured control flow. Natural loops satisfy certain properties which can be used to place metering instrumentation even faster and more optimally. With unstructured control flow, it is no longer possible. Moreover, a malicious user can use the unstructured control flow to slow down the execution.

8.1.8 Memory costs. The cost model, if possible, should prefer static memory allocations performed prior to execution. Dynamic allocations can still be possible but have to be more costly. Ideally, the cost per byte of allocation should scale down with the number of bytes allocated to incentivize bulk allocations.

It is well-known that memory allocations and deallocations can worsen the performance of a program significantly. Moreover, the impact might not be predictable because the underlying system used to execute the contract can move and remap the memory, or re-organize pages. Also, dynamic allocations can fragment memory and introduce metering instrumentation with variable costs, which is significantly harder to optimize efficiently.

In case the memory is allocated using an instruction such as `grow_memory` in Wasm, it is important to enforce that given two such instructions, it should be less costly to compute the total allocation first and allocate once, instead of using two allocations.

The cost model ideally should also take access patterns into account. From the performance optimization standpoint, it is well-known that consecutive or repeated accesses improve cache locality (spatial or temporal). Hence, we propose for the cost model to adjust the cost of the memory accesses in the following way:

- The cost of any memory access is estimated based on the latency of a RAM access on the standard validator hardware, scaled by a constant of choice. Note that the disk accesses are very unlikely, and therefore setting a slightly higher cost per access should cover occasional long-latency accesses that go to the disk.
- Certain access patterns have a discount. The rationale is that this approach makes the costs cheaper, but only for a fixed set of cases that are almost guaranteed to have small latencies (e.g., cache hits). This allows one to keep the system secure and protect it from malicious users. In addition, it is easier for compilers to optimize towards a certain pattern, e.g., tiled or vectorizable loop.

8.1.9 Historical behaviour and the ranking system. We propose to use transaction history for profiling smart contracts directly on-chain. The idea is to inject counters into smart contract code, which, as code is executed many times, will collect online statistics about branch frequencies, success rates, etc. Based on these values, the system can adjust its behaviour:

- Adapt the strategy for placing the metering instrumentation – if the contract fails many times, it might be better to instrument it using a straightforward metering scheme.
- If there is a contract with high demand, it might be profitable to JIT compile it.

The benefit of this approach over local profiling is that 1) all validators will make the same decisions because they observe the same global profiling information, and 2) it makes it possible to penalize malicious behaviour, e.g., redundant JIT compilation, again, because the profiling happens globally.

Lastly, we propose that the cost model has to take the on-chain profiling information into account to rank contracts. For example, the contracts can be ranked on how likely they are to fail.

One can think of this as a credit score used by the banks: if the credit score is high, it means that a person is likely to pay the bills on time, and this results in more favourable terms when taking a mortgage, credit, etc. At the same time, if the credit score is low, the conditions are worse and more fees have to be paid.

In the blockchain system, this means that validators should prefer to execute contracts with low failure rates and potentially predictable execution paths.

8.2 The future and the takeaways of the Cost Model Standard

We highlight that the proposed standard is only the first small step to defining the semantics of cost models for blockchains, e.g., we have not even considered costs for storage here. Most importantly, the goal of the standard is to build incentives between multiple stakeholders involved in the blockchain execution: contract developers, optimizing compilers, validators, and their runtimes. In this way, developers and compilers can be incentivized to produce code that validators can execute securely, quickly, and also meter efficiently.

There are many missing pieces in the proposed standard, including

- how to price long-term persistent storage,
- how to price upgrades of smart contracts,
- how to price cross-contract calls and code caching,
- how to price protocol-related costs, such as execution of a block of transactions or sharding.

However, we leave this as a future work.

9 RELATED WORK

9.1 Blockchains and cost models

While this work covers the cost models of Ethereum, Solana, NEAR, and Aptos, other existing blockchains do not have significantly different cost models.

The networks that support smart contracts primarily use gas to determine the execution cost of a transaction, similar to Ethereum. Examples of such networks are Binance Smart Chain [14], Polkadot [80], FileCoin [54], or Avalanche [71].

Due to the low transaction throughput of Ethereum, so-called L2 networks such as Polygon [50] emerge, offering more efficient execution. While the transaction fees are lower on these networks, contracts are still executed using EVM or EVM-compatible virtual machines that meter the computation in gas.

Sui [76] is another blockchain that, like Aptos, originated from Diem and uses gas to price the computation. In contrast to other gas-based chains, it uses bucket-based gas metering, where final gas costs are rounded up towards a fixed value. This approach allows Sui to have more stable gas costs. Tron [11] uses the concept of energy very similar to gas. Its difference from the other chains is that the native token is locked before transaction execution to ensure there are sufficient funds to pay for it at all times.

While widely used, the concept of gas is not the only way of assigning costs to computations. The most prominent and well-known blockchain, Bitcoin [67], uses a flat fee based on the transaction size in bytes. Because Bitcoin is designed as a distributed ledger, such a simple model suffices. Cardano [5] also uses a simple fee structure, accounting for the transaction size and applying a linear function to it (defined by the protocol).

9.2 Gas price estimation

While this work explores the metering of instruction costs (or gas) at run time, there is an alternative approach of determining the execution cost statically prior to execution.

For example, Grech et. al. [47] proposed a static analysis tool called MadMax to detect vulnerabilities in smart contracts due to out-of-gas behaviours. The tool is based on symbolic execution and finds cases when an out-of-gas exception leaves the contract in an invalid state.

Another application of static gas-cost analysis is to infer the gas usage for a particular contract. In particular, it helps users set the gas limit when submitting a transaction, and developers make the contracts cheaper. GASPER [39] is a tool that uses symbolic execution to identify expensive code patterns automatically. Another tool, Gasol [24], allows one to infer an upper bound on gas usage for Solidity contracts or EVM bytecode. It allows one to select from multiple cost models, e.g., measure only the cost of storage instructions. GASTAP [25] is similar to Gasol but can also infer parametric upper bounds on the gas cost. The parameters used in the inferred bounds are the inputs or contract sizes. Marescotti et. al. [61] determine worst-case gas consumption of Ethereum contracts using symbolic model checking. They unroll the loops to a limit to calculate the gas consumption. Recently, Le et. al. [57] proposed to use machine learning and historical data to estimate gas costs.

While the majority of related work aims at estimating the gas costs on the client side, the work by Das and Qadeer [40] presents a static analysis technique to find the exact gas costs and verify them in linear time, thus making it useful for validators and blockchain runtimes. The authors claim their tool can eliminate the need for dynamic gas metering. However, the proposed approach does not work with unbounded loops or unstructured control flow, limiting its applicability in practice.

9.3 Optimal instrumentation placement

The problem of optimally placing the instrumentation is well-known, with applications ranging from profile-guided optimization [56] to software testing.

A classical instance of the problem is computing the frequency counts of the basic blocks, which are used to obtain the profile of a program. The research was pioneered by Nahapetian [66] who studied how many basic blocks are sufficient and necessary to instrument to obtain the frequency counts for all basic blocks. He proposed an approach based on Kirchhoff circuit laws and network flows. An alternative formulation was proposed by Knuth and Stevenson [52, 53]. They presented an algorithm based on computing the minimum spanning tree of a control-flow graph, which is now used in numerous modern instrumentation-based profiling tools.

Later, Ball and Larus [28] proposed a taxonomy of profiling problems based on what to profile (i.e., vertices or edges of the control-flow graph) and where to place the instrumentation (i.e., in vertices or on edges). In particular, it was shown that while some instances of the problem admit an optimal solution, some do not [69].

Ball and Larus [29, 30] also studied the path profiling problem. The goal of the problem is to determine how many times each acyclic path in a control-flow graph executes.

Another application where optimal instrumentation placement is important is computing the basic block coverage. The goal is to use the boolean flags to identify which basic blocks are executed. Using boolean flags significantly reduces the code size compared to frequency counts, which are 64-bit integers. Agrawal [22, 23] studied how to find a small subset of vertices in a control-flow graph that can be used to infer the coverage of remaining blocks in a program. The techniques he developed decreased the cost of coverage testing. Tikir and Hollingsworth [78] proposed an approach to dynamically insert and remove instrumentation code to reduce the impact of computing the code coverage at runtime. Recently, Chen et. al. [38] proposed a taxonomy of coverage problems, similar to the profiling problems introduced by Ball and Larus. Additionally, they proposed a linear-time algorithm to compute the minimum block coverage.

At the same time, there is no academic work on placing the metering instrumentation. Still, placing the metering instrumentation is a widely used technique employed by many blockchains such as Solana, NEAR, or Polkadot. Additionally, existing WebAssembly runtimes such as Wasmer [13] and *wasmtime* [16] have a built-in notion of metering. However, they do not optimize the placement of the metering instrumentation.

9.4 Serverless computing

Serverless computing, which is widely popular today, shares a lot in common with existing blockchain systems. In contrast to traditional cloud computing, serverless vendors charge users based on computation rather than a fixed amount of CPU hours or bandwidth. Typical serverless services include Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS). FaaS is extremely similar to the blockchain setting and smart contracts: the code is deployed to a serverless platform and can be invoked by a request from a client.

Nowadays, many companies provide serverless services, including but not limited to AWS Lambda [1], Cloudflare Workers [6], and Google BigQuery [42]. Serverless applications can be built using different languages, e.g., JavaScript or Rust, and are similar to smart contracts written in general-purpose languages.

Serverless platforms, akin to blockchains, also meter the usage of resources. Metering is crucial to make auto-scaling decisions, e.g., detecting over-provisioned resources and re-assigning them to under-provisioned ones [82]. In addition, similarly to blockchains, serverless providers enforce limits on resource consumption to protect from DoS attacks by adversarial requests that can overload the system. Usually, the limits are applied to CPU resources, memory, and disk [60]. In addition, there are limits on the maximum amount of time for a function to complete and the number of concurrent function executions.

However, a fundamental difference from blockchains is that in the serverless setting, there is no need for determinism (both for metering and enforcing the limits). Even though the execution time of functions can vary significantly [45], there is no need for reproducibility. In contrast, reproducibility is crucial for blockchains: it should be possible to re-execute the whole transaction history and obtain the same results as any other node in the system, irrespective of its geographical location or hardware.

To meter and limit resources at run time, serverless platforms use significantly less expensive approaches than blockchain systems. Usually, serverless functions run inside a container, such as Docker [63], which allows measuring resource consumption per container instance. In addition, light VMs such as Firecracker [21] implement resource limiters that cap the bandwidth and throughput. Usually, they leverage OS-level primitives to get the resource consumption of a process, e.g., `getrusage` on Linux.

Alternative approaches to metering were proposed recently. F. Alder et al. [26] studied how metering can be done reliably in an adversarial setting and proposed to use a pair of threads for metering the compute time: one for the worker running the computation, and one for the timer. Another interesting metering strategy was proposed by Shaffer et al. [72]. They implement function-level resource monitoring by polling resource consumption of a process at regular intervals.

While serverless providers offer fine-grained billing as low as one millisecond, it is not clear how such a metering is implemented and what is the overhead. To the best of our knowledge, there are no publicly available studies on the overhead of metering execution in serverless providers.

10 CONCLUSION

In this work, we studied multiple popular blockchains, namely Ethereum, Solana, NEAR, and Aptos, focusing on how they execute smart contracts and calculate transaction fees for the execution. We learned that while existing blockchain systems have different VM and bytecode specifications and semantics, they use similar cost models.

In particular, all existing cost models are chain-oriented, i.e., they are designed 1) to protect the network from Denial-of-Service (DoS) attacks, 2) to ensure users pay their share for the load on the infrastructure, and 3) to cap the consumption of certain computational resources. Because the cost model is an approximation, we found that generally, blockchains have simple models where the total cost of execution is a weighted sum of

- a base cost to cover for the protocol and VM implementations,
- costs for executed bytecode instructions,
- a cost for the amount of memory consumed by the VM,
- and a cost for the used long-term storage.

We identified that the instruction costs used by existing cost models do not always reflect the real execution cost. In particular, by conducting some experiments we discovered that instructions that can trap or control-flow instructions are under-priced. While typically they have costs similar to other instructions, the variance in latency can lead to $10 - 30\times$ slower execution. This, in turn, reduces the blockchain's throughput and decreases the profits validators receive for processing transactions.

We also observed that existing cost model implementations (metering) are not always efficient, and fail to incentivize the right set of behaviour from the users or smart contract developers. We found that the state-of-the-art approaches fail to scale and can be attacked by malicious users when the metering is implemented by instrumenting the contract's code. In particular, metering schemes used by blockchains often place an excessive amount of instrumentation code, leading to slowdowns of more than $2\times$ on some programs or increasing their code size by at least 30%. We showed that this can be exploited by malicious users in order to slow down the network and reduce validators' earnings.

Such a defensive strategy to instrument smart contracts is used because the goal of the cost model is to protect the network from DoS attacks and to ensure that even malicious users pay their fair share of the costs. However, by analyzing the execution behaviour of smart contracts on multiple popular blockchains we observed that such scenarios are unlikely. We found that, in general, transactions have a very low failure rate of around 5-10%, and usually because of user-defined exceptions. For example, on the Aptos blockchain, failures due to running out of gas correspond to less than 1% of all traffic. Even though out-of-gas exceptions are rare, all validators are paying the cost for the defensive metering.

Motivated by this, and the lack of formalism and soundness in existing approaches to metering, we formally defined the requirements any metering scheme should satisfy. Based on the observation about the unlikeliness of failures, the algorithms used for metering should be tailored towards successful execution, and, in case of failure, fallback to a recovery mechanism to calculate the exact execution cost. As a result, one can develop algorithms that can simultaneously meter multiple basic blocks in the contract, even in the presence of implicit control flow, exceptions, or function calls.

To develop even more efficient algorithms for instruction cost metering, we formulated a new problem of the minimum metering instrumentation, which aims to find a minimal set of instrumentation points in the control-flow graph of a program so that when the program is executed, the total cost of the executed instructions is known and that there is almost always enough budget to cover for execution cost at any point during the run time. While being similar to the problems of frequency counts solved by Knuth, or minimal coverage instrumentation, the minimal metering instrumentation problem is different because it requires online checks for a sufficient budget. Moreover, the metering instrumentation that covers the paths with identical costs can be combined. To the best of our knowledge, the problem had not been studied before.

In this work, we proposed an iterative algorithm to calculate the minimum metering instrumentation under the condition that the budget has to be large enough at any point during the execution of a program. The algorithm, evaluated on multiple microbenchmarks and real-world contracts allows one to reduce the run-time overhead of metering instrumentation by a factor of 2. Moreover, we showed that it is sufficient to instrument only 30.4-37.5% of the basic blocks in real contracts.

We also observed that if the condition of having a sufficient budget at any point at run time is relaxed, the problem becomes significantly harder. While we have not presented an algorithm to solve this instance of the minimum metering instrumentation problem, the experimental results suggest that the run-time overhead of metering can be reduced further.

Since the problem of minimum metering instrumentation had not been yet studied, our work paves the way for future research and further improvements. First, one can develop algorithms that solve the relaxed version of the minimum metering instrumentation, or show its hardness. Also, it would be interesting to understand the trade-offs between the different requirements of the metering schemes, to find the right balance between security and optimality of the placed instrumentation. It is also worth exploring whether the metering instrumentation can be optimized, e.g., by hoisting it outside loops. Another interesting future direction is to improve the running time of computing the placement of the metering instrumentation. For example, the instrumented code can be stored on-chain and verified when loaded by the VM. Alternatively, a tiered-compilation approach can be used – interleaving computations of the metering instrumentation with execution.

Finally, after conducting a detailed study on existing cost models and metering schemes, we realized that the current design of the cost models needs re-thinking and should not be chain-oriented only. Instead, we propose to include multiple stakeholders such as users, smart contract developers, compilers, validators, and the blockchain itself. We believe that by identifying and matching the right set of behaviour amongst the interested parties, it is possible to ensure that the cost model:

- penalizes malicious behaviour, e.g., running out of budget during execution or DoS attacks,
- incentivizes fast, cheap, and analyzable code, and
- can be efficiently metered.

We presented the Cost Model Standard – a first attempt to describe the relationship between different stakeholders and their needs and bring this over to the cost model. While not yet complete, we believe the standard we propose is the first step towards defining safe, robust, and simple cost models for blockchains, making the networks more secure, performant, and cheaper.

REFERENCES

[1] 2014. AWS Lambda. https://aws.amazon.com/lambda, visited on 10.08.2023.

[2] 2015. Go Ethereum: Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum, visited on 31.07.2023.

[3] 2015. Solidity: A statically-typed programming language designed for developing smart contracts that run on Ethereum. https://soliditylang.org, visited on 08.08.2023.

[4] 2016. The Ethereum EVM JIT. https://github.com/ethereum/evmjit, visited on 31.07.2023.

[5] 2017. Cardano (ADA). https://cardano.org

[6] 2017. Cloudflare Workers. https://workers.cloudflare.com, visited on 10.08.2023.

[7] 2017. NearVM implementation. https://github.com/near/nearcore/tree/master/runtime/near-vm, visited on 14.08.2023.

[8] 2017. Vyper. https://docs.vyperlang.org/en/stable, visited on 08.08.2023.

[9] 2018. A Library for WebAssembly Module Instrumentations. https://github.com/paritytech/wasm-instrument, visited on 31.07.2023.

[10] 2018. Rust virtual machine and JIT compiler for eBPF programs, adapted by Solana blockchain. https://github.com/solana-labs/rbpf, visited on 31.07.2023.

[11] 2018. TRON: Advanced Decentralized Blockchain Platform. https://tron.network/static/doc/white_paper_v_2_0.pdf, visited on 12.10.2023.

[12] 2018. wasmi – WebAssembly (Wasm) Interpreter. https://github.com/paritytech/wasmi, visited on 27.09.2023.

[13] 2018. WebAssembly Runtime. https://github.com/wasmerio/wasmer, visited on 31.07.2023.

[14] 2019. BNB Smart Chain: A Parallel Blockchain to Beacon Chain to Enable Smart Contracts. https://github.com/bnb-chain/whitepaper/blob/master/WHITEPAPER.md, visited on 10.10.2023.

[15] 2019. Evmone: fast Ethereum Virtual Machine implementation. https://github.com/ethereum/evmone, visited on 31.07.2023.

[16] 2019. wasmtime: A standalone runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime, visited on 26.09.2023.

[17] 2020. The Libra Blockchain. https://diem-developers-components.netlify.app/papers/the-diem-blockchain/2020-05-26.pdf, visited on 08.08.2023.

[18] 2020. Proposal for exception handling in WebAssembly. https://github.com/WebAssembly/exception-handling/blob/master/proposals/exception-handling/Exceptions.md, visited on 08.08.2023.

[19] 2022. Move Virtual Machine implementation adapted for Aptos blockchain. https://github.com/aptos-labs/aptos-core/tree/main/third_party/move, visited on 31.07.2023.

[20] 2023. fiite-wasm: Cheating a little to solve the halting problem at scale. https://github.com/near/finite-wasm, visited on 26.09.2023.

[21] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation (NSDI'20)*. https://www.usenix.org/system/files/nsdi20-paper-agache.pdf

[22] Hiralal Agrawal. [n. d.]. Dominators, Super Blocks, and Program Coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. https://doi.org/10.1145/174675.175935

[23] Hira Agrawal. 1999. Efficient Coverage Testing Using Global Dominator Graphs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99)*. https://doi.org/10.1145/316158.316166

[24] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In *Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-030-45237-7_7

[25] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2021. Don't run on fumes—Parametric gas bounds for smart contracts. *Journal of Systems and Software* 176 (2021), 110923. https://doi.org/10.1016/j.jss.2021.110923

[26] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-FaaS: Trustworthy and Accountable Function-as-a-Service Using Intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW'19)*. https://doi.org/10.1145/3338466.3358916

[27] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. 1999. Dominators in Linear Time. *SIAM J. Comput.* 28, 6 (1999), 2117–2132. https://doi.org/10.1137/S0097539797317263

[28] Thomas Ball and James R. Larus. 1994. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (jul 1994), 1319–1360. https://doi.org/10.1145/183432.183527

[29] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. https://doi.org/10.1109/MICRO.1996.566449

[30] Thomas Ball, Peter Mataga, and Shmuel Sagiv. 1998. Edge profiling versus path profiling: the showdown. In *ACM-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/268946.268958

[31] Martin Becze. 2015. EVM 2.0. https://github.com/ethereum/EIPs/issues/48

[32] Alex Beregszaszi, Andrei Maiboroda, and Pawel Bylica. 2021. EIP-3670: EOF - Code Validation. https://eips.ethereum.org/EIPS/eip-3860

[33] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, Davide Sestili, and Francesco Tiezzi. 2020. Ethereum smart contracts: Analysis and statistics of their source code and opcodes. *Internet of Things* 11 (2020), 100198. https://doi.org/10.1016/j.iot.2020.100198

[34] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2019. Move: A Language With Programmable Resources. https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf

[35] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert E. Tarjan, and Jeffery R. Westbrook. 2008. Linear-Time Algorithms for Dominators and Other Path-Evaluation Problems. *SIAM J. Comput.* 38, 4 (2008), 1533–1573. https://doi.org/10.1137/070693217

[36] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. 1998. A New, Simpler Linear-Time Dominators Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 6 (nov 1998), 1265–1296. https://doi.org/10.1145/295656.295663

[37] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. 2019. EIP-1559: Fee market change for ETH 1.0 chain. https://eips.ethereum.org/EIPS/eip-1559

[38] Li Chen, Ellis Hoag, Kyungwoo Lee, Julian Mestre, and Sergey Pupyrev. 2022. Minimum Coverage Instrumentation. arXiv:2208.13907

[39] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. https://doi.org/10.1109/SANER.2017.7884650

[40] Ankush Das and Shaz Qadeer. 2020. Exact and Linear-Time Gas-Cost Analysis. In *Static Analysis: 27th International Symposium, SAS 2020, Virtual Event, November 18–20, 2020, Proceedings*. https://doi.org/10.1007/978-3-030-65474-0_15

[41] Maarten Faddegon. 2011. *SSA Back-Translation: Faster Results with Edge Splitting and Post Optimization*. Master's thesis. TU Delft.

[42] Sérgio Fernandes and Jorge Bernardino. 2015. What is BigQuery?. In *Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS '15)*. https://doi.org/10.1145/2790755.2790797

[43] Agner Fog. 2022. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University* (2022). https://www.agner.org/optimize/instruction_tables.pdf

[44] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. 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 (PPoPP '23)*. https://doi.org/10.1145/3572848.3577524

[45] Samuel Ginzburg and Michael J. Freedman. 2021. Serverless Isn't Server-Less: Measuring and Exploiting Resource Variability on Cloud FaaS Platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing (WoSC'20)*. https://doi.org/10.1145/3429880.3430099

[46] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

[47] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2020. MadMax: Analyzing the out-of-Gas World of Smart Contracts. *Commun. ACM* 63, 10 (sep 2020), 87–95. https://doi.org/10.1145/3416262

[48] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. https://doi.org/10.1145/3062341.3062363

[49] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. DFINITY Technology Overview Series, Consensus System. arXiv:1805.04548

[50] Jaynti Kanani, Sandeep Nailwal, and Anurag Arjun. 2017. Polygon (previously Matic) Whitepaper. https://github.com/maticnetwork/whitepaper, visited on 10.10.2023.

[51] Sital Kedia. 2023. AIP-27: Sender Aware Transaction Shuffling. https://github.com/aptos-foundation/AIPs/blob/main/aips/aip-27.md, visited on 09.10.2023.

[52] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume 1 (2nd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc.

[53] Donald E. Knuth and Francis R. Stevenson. 1973. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics* 13 (1973), 313–322. https://doi.org/10.1007/BF01951942

[54] Protocol Labs. 2017. Filecoin: A Decentralized Storage Network. https://filecoin.io/filecoin.pdf, visited on 31.07.2023.

[55] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. https://doi.org/10.1109/CGO.2004.1281665

[56] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-Guided Size Optimization for Native Mobile Applications. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC 2022)*. https://doi.org/10.1145/3497776.3517764

[57] Chunmiao Li, Shijie Nie, Yang Cao, Yijun Yu, and Zhenjiang Hu. 2020. Dynamic Gas Estimation of Loops Using Machine Learning. In *Blockchain and Trustworthy Systems*. https://doi.org/10.1007/978-981-15-9213-3_34

[58] Chao Liu, Jianbo Gao, Yue Li, and Zhong Chen. 2020. Understanding Out of Gas Exceptions on Ethereum. In *Blockchain and Trustworthy Systems*. https://doi.org/10.1007/978-981-15-2777-7_41

[59] Yulin Liu, Yuxuan Lu, Kartik Nayak, Fan Zhang, Luyao Zhang, and Yinhong Zhao. 2022. Empirical Analysis of EIP-1559: Transaction Fees, Waiting Times, and Consensus Security. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. https://doi.org/10.1145/3548606.3559341

[60] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. 2022. A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. *ACM Comput. Surv.* 54, 11s, Article 222 (sep 2022). https://doi.org/10.1145/3510412

[61] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*. https://doi.org/10.1007/978-3-030-03427-6_33

[62] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. https://www.usenix.org/legacy/publications/library/proceedings/sd93/mccanne.pdf

[63] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (mar 2014). https://dl.acm.org/doi/10.5555/2600239.2600241

[64] Matthias Mettler. 2016. Blockchain technology in healthcare: The revolution starts here. In *2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom)*. https://doi.org/10.1109/HealthCom.2016.7749510

[65] E. Morel and C. Renvoise. 1979. Global Optimization by Suppression of Partial Redundancies. *Commun. ACM* 22, 2 (feb 1979), 96–103. https://doi.org/10.1145/359060.359069

[66] A. Nahapetian. 1973. Node Flows in Graphs with Conservative Flow. *Acta Inf.* 3, 1 (mar 1973), 37–41. https://doi.org/10.1007/BF00288650

[67] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com* (2009).

[68] Daniel Perez and Benjamin Livshits. 2019. Broken Metre: Attacking Resource Metering in EVM. arXiv:1909.07220

[69] R. L. Probert. 1982. Optimal Insertion of Software Probes in Well-Delimited Programs. 8, 1 (jan 1982), 34–42. https://doi.org/10.1109/TSE.1982.234772

[70] Nathan Reiff. 2023. The Collapse of FTX: What Went Wrong With the Crypto Exchange? https://blog.chain.link/reentrancy-attacks-and-the-dao-hack, visited on 12.10.2023.

[71] Kevin Sekniqi, Daniel Laine, Stephen Buttolph, and Emin Gün Sirer. 2020. Avalanche Platform. https://www.avalabs.org/whitepapers, visited on 09.10.2023.

[72] Tim Shaffer, Zhuozhao Li, Ben Tovar, Yadu Babuji, TJ Dasso, Zoe Surma, Kyle Chard, Ian Foster, and Douglas Thain. 2021. Lightweight Function Monitors for Fine-Grained Management in Large Scale Python Applications. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS49936.2021.00088

[73] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. 2008. Virtual Machine Showdown: Stack versus Registers. *ACM Trans. Archit. Code Optim.* 4, 4, Article 2 (jan 2008), 36 pages. https://doi.org/10.1145/1328195.1328197

[74] Avi Spielman. 2016. *Digitally rebuilding the real estate industry.* Master's thesis. MIT. https://dspace.mit.edu/handle/1721.1/106753

[75] The Aptos Team. 2022. The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure. https://aptos.dev/aptos-white-paper

[76] The MystenLabs Team. 2023. The Sui Smart Contracts Platform. https://docs.sui.io/paper/sui.pdf

[77] The NEAR Team. 2017. The NEAR White Paper. https://near.org/papers/the-official-near-white-paper.

[78] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2002. Efficient Instrumentation for Code Coverage Testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. https://doi.org/10.1145/566172.566186

[79] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper* 2014 (2014). https://gavwood.com/paper.pdf

[80] Gavin Wood. 2016. Polkadot: Vision for a Heterogeneous Multi-Chain Framework. https://assets.polkadot.network/Polkadot-whitepaper.pdf

[81] Anatoly Yakovenko. 2017. Solana: A new architecture for a high performance blockchain v0.8.13. https://solana.com/solana-whitepaper.pdf.

[82] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating Serverless Computing by Harvesting Idle Resources. In *Proceedings of the ACM Web Conference 2022 (WWW '22)*. https://doi.org/10.1145/3485447.3511979