HEMVM: a Heterogeneous Blockchain Framework for Interoperable Virtual Machines

This paper introduces HEMVM, an innovative heterogeneous blockchain framework that seamlessly integrates diverse virtual machines (VMs), including Ethereum Virtual Machine (EVM) and Move Virtual Machine (MoveVM), into a unified system. This integration facilitates interoperability while retaining compatibility with existing Ethereum and Move toolchains by preserving high-level language constructs. HEMVM's unique cross-space operations allow users to interact with contracts across various VMs using any wallet software, effectively resolving the fragmentation in user experience caused by differing VM designs. Our experimental results demonstrate that HEMVM is both fast and efficient, incurring minimal overhead (less than 4.4%) for intra-VM transactions and achieving up to 9300 TPS for cross-VM transactions. Our results also show that the cross-space operations in HEMVM are sufficiently expressive to support complex decentralized finance interactions across multiple VMs. Finally, the parallelized prototype of HEMVM shows performance improvements up to 44.8% compared to the sequential version of HEMVM under workloads with mixed transaction types.

CCS Concepts: • **Software and its engineering** → *Software design engineering*.

Additional Key Words and Phrases: blockchain, virutal machine, Aptos, Move, Ethereum, EVM, Conflux, Solidity

ACM Reference Format:
. 2025. HEMVM: a Heterogeneous Blockchain Framework for Interoperable Virtual Machines. 1, 1 (March 2025), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchain technology provides the powerful abstraction of a decentralized, resilient, and programmable ledgers on a global scale [11, 34]. Blockchains, often referred to as Web 3.0 or Distributed Ledger Technologies (DLTs), vastly expand individual autonomy and enable value transfers without the need for a central authority. They have also fundamentally reshaped the financial landscape through the emergence of Decentralized Finance (DeFi). DeFi has disrupted the costly legacy model of Traditional Finance (TradFi) with its offering of similar services (lending/borrowing, value exchanging, yield earning, insurance, etc.) for cheaper and without relying on costly middlemen such as banks or brokers.

At the heart of blockhain technology are smart contracts, which are computer programs that allow developers to establish intricate transaction rules governing these ledgers. These smart contracts are executed in a decentralized, peer-to-peer manner across the blockchain's network of nodes, with each virtual machine (VM) instance executing the code of the same smart contracts so that consensus can be achieved across the network. For instance, in Ethereum, the second largest blockchain with a market capitalization of 247 billion U.S. dollars and total value locked (TVL) of 53.2 billion U.S. dollars, developers craft smart contracts using high-level languages such

Author's address:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2025/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

as Solidity [19] and Vyper [21]. These contracts are then translated into the Ethereum Virtual Machine's (EVM) bytecode using specific compilers [18, 20]. Once compiled, the Ethereum client ensures the consistent execution of the bytecode across all network nodes, updating the state of the global ledger.

The virtual machine design of a blockchain plays a pivotal role in determining its performance, security, and overall utility. For example, the virtual machine of Bitcoin has a restricted opcode set, limiting it to basic cryptocurrency transactions with minimal programmability. In contrast, Ethereum VM, being Turing complete, revolutionized the blockchain landscape. It transformed Ethereum into a versatile programmable ledger, capable of encoding complex transaction rules.

This adaptability has facilitated the integration of smart contracts across diverse sectors, from finance and supply chain management to entertainment [23, 37, 38]. As a result, blockchains have gained widespread adoption, with the total value of assets across all blockchains reaching 2.85 trillion U.S. dollars. However, this growth has led to a highly fragmented ecosystem, with over 30 major blockchain platforms operating in parallel - each with its own design, often not compatible with its predecessors. As a result of that, most smart contracts cannot access or interact with assets across these platforms. The core issue is the *lack of VM interoperability*: different blockchain systems use incompatible virtual machines, preventing smart contracts written for one chain from running natively on another. This incompatibility severely limits the seamless transfer of assets and data across networks, ultimately hindering the potential of decentralized applications.

However, value transfer inefficiency is not the only consequence of fragmentation. Data and value silos also hinder broader blockchain adoption - particularly in the case of the EVM. As EVM usage grows and smart contracts become more complex, it struggles to meet rising demands for functionality, efficiency, and security. Limitations such as a small stack size, lack of opcodes for deep stack access, and vulnerabilities like reentrancy [41] are well-known developer pain points. These issues stem from early EVM design decisions, many of which are addressed in newer VM architectures. However, these designs often break compatibility with the established EVM API and tooling. Given Ethereum's dominance and the EVM's widespread use in smart contract development, other blockchains have to develop their unique tooling to improve user and developer experience. This fragmentation burdens users with multiple wallets and address formats, while developers must adapt smart contracts to different languages and environments.

A notable case of VM incompatibility is between EVM and MoveVM. MoveVM powers the Move smart contract language, originally developed for Facebook's Libra project in 2018, which aimed to support a new global digital currency. Recently, Move popularity has increased since its adoption by popular new high-performance blockchain systems Aptos [17] and Sui [22], which now have market capitalizations of $3.6 billion and $7.4 billion, respectively. Designed with strong safety guarantees and inspired by Rust, Move supports formal verification and treats assets as first-class citizens. While developers are drawn to Move for its safety and improved experience, the incompatibility between MoveVM and EVM, combined with EVM's large user base, forces them to choose between security and adoption - often at the cost of user familiarity and tooling support.

Ideally, a blockchain would seamlessly support multiple VMs like EVM and MoveVM, concealing their intricacies from both users and developers. Users ought to have the flexibility to choose any wallet application and interact with smart contracts, regardless of the VM in which they were developed, and the resulting system must not suffer from drops in performance and reduced security guarantees. However, current blockchains display a monolithic structure, with VM designs deeply intertwined with other components such as storage layouts and address formats. Constructing a *heterogeneous* blockchain that harmoniously integrates multiple interoperable VMs in one system remains an unresolved challenge.

HEMVM: This paper presents HEMVM, the first framework for the development of heterogeneous blockchain systems capable of supporting multiple, interoperable virtual machines. As evidence of our approach, we have developed a prototype blockchain of HEMVM. This blockchain incorporates an EVM module that aligns with the existing interfaces and tools of the EVM ecosystem, and a MoveVM module compatible with the Aptos toolchains. Notably, these two VM modules within HEMVM function in an interoperable manner. For example, users can utilize MetaMask, a widely-used wallet tailored for EVM-compatible chains, to engage with smart contracts designed and deployed in the MoveVM module of HEMVM.

To ensure compatibility across VM modules in HEMVM with the established tool chains of both EVM and MoveVM, we have integrated RPC interfaces and transaction formats for the EVM as well as the Aptos MoveVM. To ensure consistent and harmonious processing, HEMVM structures its blockchain state into specific *sub-spaces*. Incoming RPC calls and transactions are routed to the relevant processing sub-space. Each of these sub-spaces maintains its distinct address system and standalone key-value state storage, mirroring the design ethos of their corresponding VM.

At its core, each sub-space within HEMVM operates as an independent ledger, steered by its respective VM. A primary challenge for HEMVM is facilitating interoperability between these sub-spaces. To address this, HEMVM introduces the innovative *cross-space handler* mechanism, which has been inspired by a foreign function interface (FFI) technique used in programming language interoperability designs[10, 15]. In short, this mechanism can be thought of as a unique type of smart contract operation, allowing a smart contract to bundle operations from multiple VMs in one atomic transaction. This design not only enables interoperability but also maintains compatibility with existing tool chains. In essence, legacy tools, which may only recognize a single sub-space, can interpret cross-space invocations as distinct transactions.

Another challenge for HEMVM is the actual implementation of these cross-space operations. A direct method would involve modifying compilers for all related high-level languages. However, HEMVM offers a solution with its novel *internal contract* design. Instead of introducing new VM opcodes, which would necessitate new language constructs and compiler modifications, HEMVM clusters its novel features under a special internal contract address. Invoking these features is seamlessly managed within HEMVM's backend. This approach eliminates the need for new high-level language constructs, allowing HEMVM to integrate smoothly with existing Solidity and Move compilers.

Finally, the design decisions of HEMVM bring surprising benefits to the performance potential of the resulting blockchain system. Due to the fact that there are now multiple virtual machines inside of the system, the execution of their transactions excluding cross-space transactions can be parallelized to achieve better system throughput.

Importantly, the proposed techniques in this paper are versatile and can be adapted to construct heterogeneous blockchain frameworks that accommodate various virtual machines. By providing a common abstraction layer that can interface with multiple virtual machines while maintaining execution speed and security properties, these techniques serve as a foundational component in the broader architecture of cross-chain communication. This facilitates the design of interoperability layers that are not only secure and verifiable but also flexible enough to integrate with existing and future blockchain platforms. Consequently, this work lays important groundwork for subsequent scientific exploration into formal verification of multi-VM systems, composable interoperability architectures, and other generalized multi-VM execution environments.

Experimental Results: We implemented HEMVM on both the Aptos [17] and Conflux [31] blockchain platforms. Conflux is a major blockchain with a virtual machine that closely resembles EVM, but isn't fully compatible with it, which makes it a perfect candidate for this expiremental evaluation. Our evaluation concentrated on its cross-space functionality with widely-used smart

contracts and the performance boost that a parallelized implementation of HEMVM is able to achieve. Benchmarks include such scenarios as native tokens, ERC-20 tokens, decentralized trading, and lending. The results confirm that HEMVM facilitates the use of EVM wallets such as MetaMask for interacting with Move-deployed contracts, and similarly in the reverse. In terms of performance, HEMVM introduces less than 4.4% overhead for intra-VM transactions. For cross-VM transactions, it achieves 320-358 TPS in complex smart contract interactions and reaches up to 9300 TPS for simpler token transfer tasks. The parallelized prototype shows performance boost up to 44.8% for complex workloads.

Contributions: This paper makes the following contributions:

- **Heterogeneous Blockchain Design:** This paper presents the first general framework for building heterogeneous blockchains that seamlessly support multiple, interoperable VMs.
- **HEMVM:** This paper presents HEMVM, an innovative blockchain system that harmoniously integrates both EVM and MoveVM. Notably, HEMVM allows users to employ EVM wallets such as MetaMask to interact with contracts housed in MoveVM and vice versa. This design also improves the overall transaction throughput via enabling execution parallelization between VMs.
- **Cross-Space Handler:** This work proposes a unique cross-space handler mechanism. This design facilitates interoperability between VMs with differing semantics and storage structures, all while maintaining compatibility with established tool chains. We also describe an access control mechanism that makes sure that the proposed cross-space handler mechanism cannot be used by an unauthorized party.
- **Internal Contract Approach:** This paper introduces a novel internal contract approach. This approach allows the introduction of new features to blockchain VMs without altering high-level language constructs, ensuring HEMVM's compatibility with current Solidity and Move compilers.

The remaining of this paper is organized as follows. Section 3 presents an overview of our heterogenous blockchain framework and shows a motivating example. Section 4 presents the technical design of HEMVM. Section 5 discusses the implementation of HEMVM. We evaluate HEMVM in Section 6 and discuss related work in Section 7. We finally conclude this paper in Section 8.

2 BACKGROUND

In this section, we will overview several elements of the building blocks of a blockchain platform that will help in understanding the remaining sections. Blockchain can be conceptualized as a *state machine*. State transitions are commonly referred to as *transactions* and are recorded. These transactions are grouped into blocks, with each block cryptographically linked to its predecessor, forming a chain. This structure ensures data integrity and immutability, preventing retroactive alteration of records. The idea of the blockchain was originally devised for the digital currency Bitcoin, however, the blockchain's concepts have since been adopted and expanded into various applications, and different paradigms and techniques have been developed to offer a robust and transparent blockchain platforms.

We distinguish between two components of a blockchain platform: (1) Storage and Execution where the state machine is being stored and the computation and validation of a new state is performed; (2) Consensus where nodes in the network agree on the latest state to prevent inconsistencies across the network. A blockchain platform must implement a Remote Procedure Call (RPC) interface to allow different nodes in the network to exchange data and to allow applications to interact with the blockchain platform.

2.1 Transaction

Transactions correspond to valid state transitions. If a state transition is deemed invalid - such as a user attempting to transfer more funds than their balance allows - the transition is reverted. In this case, the system restores the state to the most recent valid state, effectively treating the invalid transition as if it never occurred and the corresponding transaction is not recorded in the blockchain. In each blockchain platform, transactions must conform to some standard. For instance, in Ethereum a transaction includes information that is necessary for its execution such as the destination address (*to*), and the function to invoke with the passed parameters (*data*). Also, a transaction includes information to secure it against unauthorized usage such as *nonce* which is an account specific natural number and linearly increments when the account performs transactions and it used to prevent replay attacks.

2.2 Smart Contracts

Pioneered by Ethereum [11, 43], *smart contracts* are programs stored on the blockchain allowing blockchain platforms complex transactions. Commonly, a smart contract is associated with a unique address and a persistent state. A smart contract, like a class in object-oriented programming (OOP), exposes a collection of functions. A transaction can invoke one of those functions by specifying it in the field *data* with the corresponding input parameters. The executions of those functions result in changes to the blockchain's state. Similar to OOP, smart contracts can also interact with other smart contracts through external function calls. Thus, smart contracts significantly enhance blockchain's functionality, enabling complex transaction rules beyond simple financial exchanges.

2.3 Gas

Gas fees are rewards to incentivise maintainers (miners) of a blockchain to process and confirm transactions while preventing free-riders spam attacks. A transaction issuer sets a gas fee that they are willing to pay for their transaction. The amount of gas fees depends on the blockchain platform and the complexity of the corresponding transactions. In particular, smart contracts transactions can differ in their computational complexity - some might consume little resources, while others might run compute-heavy procedures. Also, read and write operations on storage slots requires much greater amounts of gas than read and write operations from local variables. Thus, to charge for each transaction execution fairly, the gas fees must be set appropriately based on the transaction complexity. A transaction will not be included (i.e., will be terminated) if the amount of gas it consumes exceeds the preset gas fees that the author of the transaction willing to pay for it.

2.4 Block

Generally, blockchain platforms perform macro state transitions. A macro state transition consists of a valid execution of a set of transactions that are packed in a *block*. The chain of blocks records all state transitions from the genesis state. Transactions are added into a block until the accumulated gas fees of the added transactions reaches a *block gas fee threshold*. Then, the blockchain platform proposes the new block, it executes its transactions, and incorporates the commitment of the post-execution state into the block header to track the validity of the block when it is inserted into the chain.

Since a distributed blockchain platform cannot directly determine real-world time, the *block height*, which represents the number of blocks in the chain, is used as a timestamp in smart contracts to estimate time periods. In particular, based on the throughput, i.e., how many blocks per time unit the blockchain platform produces, it is then possible to estimate the period of time based on the equivalent of how many blocks were produced between two block heights.

2.5 Ethereum Virtual Machine (EVM)

The *Ethereum Virtual Machine (EVM)* is a stack-based virtual machine that supports Turing-complete smart contracts programs, allowing for a broad range of computational operations [43]. Many blockchains adopt the Ethereum virtual machine (EVM) for execution due to its functionality and popularity among applications developers. In Ethereum, smart contracts are written in the EVM bytecode, stored on the smart contract account, using EVM opcodes such as `SLOAD` and `SSTORE` to access the blockchain state and `CALLER` and `CALLVALUE` to access caller address and the amount of native tokens (e.g., *Ether*) sent. Each EVM opcode has a gas cost associated with it that allows to estimate the gas fees necessary for executing a transaction. In Ethereum, smart contracts are commonly written in high-level programming languages, such as Solidity [19], and are subsequently compiled into the EVM bytecode [18].

In Figure 1a, we show the implementation of a simple fungible token smart contract in Solidity. The implementation leverages Solidity's `mapping` type to store account balances and uses function modifiers such as `public` to enforce access control.

The `mint` function allows the creation of new tokens by increasing the recipient's balance. To ensure valid token creation, it includes a `require` statement preventing minting to the zero address. Additionally, the function emits a `Transfer` event with the sender set to zero address to enable external tracking of transactions. The `transfer` function enables token transfers between accounts. It first checks whether the sender has a sufficient balance using `require`, preventing unauthorized transfers. The function then deducts the `amount` from the sender's balance and adds it to the recipient's balance, ensuring proper accounting of tokens. The `balanceOf` function allows users to query the token balance of a given account by returning the stored value in the `balances` mapping.

2.6 The Move Language

Move is a resource-oriented programming language for writing blockchain applications designed to improve their security and facilitate their formal verification [9, 14]. In Move, smart contracts are called *modules*. Move-based modules are executed by the *Move Virtual Machine (MoveVM)*.

In contrast to EVM and Solidity, Move introduces *resources*, which are first-class data types that cannot be copied or accidentally discarded. Move uses resources to represent digital assets in the state and enforces ownership rules to restrict arbitrary state changes.

Indeed, Move-based modules behave differently from EVM-based and Solidity-based smart contracts. In Figure 1b, we show the implementation of a token module in Move. The *Token* module defines a `Token` resource with a `balance` field, marked with the `key` and `store` abilities, which allow it to be stored in global storage and uniquely identified by an account.

The `mint` function creates a new `Token` resource and assigns it to the recipient's account using `move_to`, ensuring direct ownership transfer without reliance on global state modifications. The `transfer` function retrieves mutable references to the sender's and recipient's token resources using `borrow_global_mut`. It then enforces a balance check with `assert`, preventing transfers that exceed the sender's balance. The `balanceOf` function retrieves the token balance of a given account using `borrow_global`, ensuring safe and efficient state access.

Unlike Solidity, which uses mappings to store balances and function modifiers to enforce access control, Move encapsulates asset logic within the resource itself, preventing unintended state changes. Additionally, Move's type system and borrow semantics naturally enforce security constraints, e.g., `borrow_global`, reducing the need for runtime checks.

```solidity
contract Token {
    // Balances of Tokens
    mapping(address => uint256) public balances;

    // Event logs
    event Transfer(
        address indexed from,
        address indexed to,
        uint256 value
    );

    // Mint amount of tokens
    function mint(
        address to,
        uint256 amount) public returns (bool)  {
        require(
            to != address(0),
            "ERC20: mint to the zero address"
        );
        balances[to] += amount;
        emit Transfer(address(0), to, amount);
        return true;
    }

    // Transfer tokens between accounts
    function transfer(
        address to,
        uint256 amount) public returns (bool) {
        require(
            balances[msg.sender] >= amount,
            "Insufficient balance"
        );
        balances[msg.sender] -= amount;
        balances[to] += amount;
        emit Transfer(msg.sender, to, amount);
        return true;
    }

    // View balance
    function balanceOf(
        address account) public view returns (uint256) {
        return balances[account];
    }
}
```

(a) Solidity-based implementation of token.

```move
module Token {
    use std::signer;

    // Define the Move Token resource
    struct Token has key, store {
        balance: u64,
    }

    // Mint a new Token resource
    public fun mint(
        to: &signer,
        amount: u64
    ) acquires Token {
        let token = Token { balance: amount };

        // Move tokens to address âĂŤoâĂŤ
        move_to(to, token);
    }

    // Transfer tokens between accounts
    public entry fun transfer(
        from: &signer, to: address, amount: u64
    ) acquires Token {
        // Get mutable reference from global storage
        let sender_token = borrow_global_mut<Token>(
                signer::address_of(from)
          );

        let recipient_token =
                borrow_global_mut<Token>(to);

        assert!(sender_token.balance >= amount, 1);
        sender_token.balance -= amount;
        recipient_token.balance += amount;
    }

    // View balance
    public fun balanceOf(
        account: address
    ): u64 acquires Token {
        // Get immutable reference from global storage
        borrow_global<Token>(account).balance
    }
}
```

(b) Move-based implementation of token.

Fig. 1. Implementations of token smart contract.

2.7 Cross-chain Transaction

Cross-chain transactions permit different blockchains to interact, allowing applications on different blockchains to interoperate. Thus, a user with an account in one blockchain can interact with an application that resides in another blockchain without requiring the use to create an account in the second blockchain. Cross-chain transactions include transactions between chains that have the same virtual machine (VM) execution enviroment, e.g., Binance Smart Chain (BSC) and Ethereum, and transactions between chains that have different VM execution enviroments, e.g., Ethereum and Aptos. Cross-chain transactions are typically used for transferring funds from existing blockchain networks to newer ones. A situation where a new blockchain network has only a small amount of funds stored in itself typically results in inefficient markets, high price volatility and poor incentives for potential investors [2]. This makes cross-chain transactions crucial for the healthy development and kickstarting of new blockchain ecosystems.

Cross-chain bridges are the commonly used mechanisms for cross-chain interactions. They consist of a third-party relay protocol that connects the two chains that are involved in the cross-chain transaction. In particular, the off-chain relay protocol listens to updates on the source blockchain and reacts by performing updates on the destination blockchain. Cross-chain bridges support only basic interactions like token transfers cross chains.

In cross-chain bridges, transactions execution and confirmation speeds range from about one minute (centralized systems) to several days (optimistic rollups). Since new high performance blockchains are handling millions of transactions per day, cross-chain bridges are not scalable.

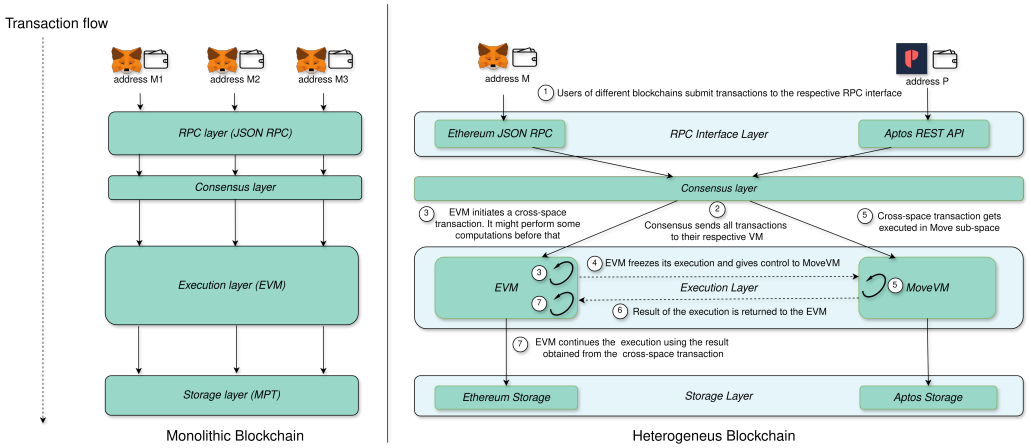

Fig. 2. Overview of different Blockchain Architectures

Furthermore, cross-chain bridges weaken the overall ecosystem security, and it is reported that vulnerabilities in cross-chain bridges have resulted in $3.2 billion in losses since May 2021 [5].

3 OVERVIEW AND EXAMPLE

This section first presents a design overview of our heterogeneous blockchain framework. We then present a motivating example of initiating a cross-VM transaction in HEMVM, our prototype blockchain system.

3.1 Design Overview

Figure 2 presents the overview of our framework. The conventional architecture of blockchain systems, as depicted on the left side of Figure 2, is structured into four critical layers. The first layer, the Remote Procedural Call (RPC) layer provides a set of API interfaces, enabling external applications to interact with the blockchain. Following this is the consensus layer which is responsible for generating a consistent transaction order for execution by each node. The third layer, the virtual machine layer, is where each transaction is executed. Lastly, there is the storage layer, which maintains the state of the blockchain ledger.

Standard blockchain systems are inherently monolithic and restricted to hosting only a single virtual machine due to the interdependencies among the different layers. This limitation arises from the significant impact that the design of a virtual machine has on both the storage and the RPC layers. For instance, EVM and MoveVM feature distinct transaction formats, necessitating unique APIs for interaction. Additionally, they conceptualize the blockchain ledger state differently: EVM utilizes a two-level key-value store where each contract is linked to a key-value map for storing its state values, while MoveVM organizes the blockchain state as a hierarchical tree structure, based on its specific resource types. This monolithic nature restricts blockchains to operate exclusively with applications designed for their respective VMs. For example, it is impossible to use MetaMask wallet to operate with Aptos blockchain because MetaMask is designed for EVM.

The right side of Figure 2 presents the architecture of our heterogeneous blockchain framework. This framework is partitioned into multiple sub-spaces, each tailored to support a specific VM. Independent storage, VM execution, and RPC layers within each sub-space ensure complete and compatible support for the corresponding VM. Unified under a shared consensus layer, these

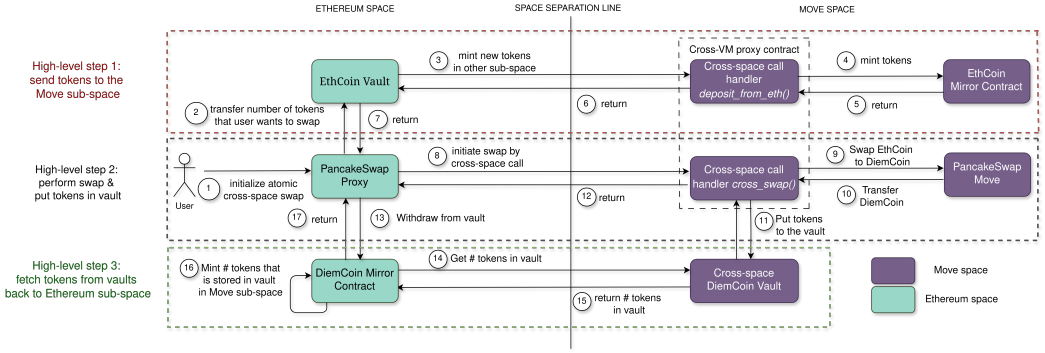

Fig. 3. Cross-space swapping using PancakeSwap protocol.

```
1  public entry fun swap_exact_input<X, Y>(
2    sender: &signer, x_in: u64, y_min_out: u64
3  ) {...}
```

Fig. 4. Interface of PancakeSwap required for the example.

sub-spaces collectively order transactions before dispatching them for execution in the relevant sub-space. The design of our framework is adaptable to any consensus algorithm, given the typical orthogonality of consensus mechanisms to VM designs.

A key innovation in our framework is the incorporation of cross-space operations as new opcodes within each VM. These opcodes enable a contract in the source VM to invoke a handler function in another contract in the target VM. Upon completion of the call in the target VM, the transaction resumes in the original contract within the source VM, as illustrated in Figure 2. Combined with the atomic nature of blockchain transaction execution, our cross-space operation facilitates the orchestration of complex interactions across different VMs. It allows for sophisticated functionalities such as cross-space token transfers and trades, which we will demonstrate in an upcoming example.

3.2 Motivating Example

HEMVM enables developers to leverage both the EVM and MoveVM stacks for contract development and deployment. Users can choose their preferred interface, such as MetaMask or the Petra wallet, to interact with contracts within their respective sub-spaces - whether on EVM or MoveVM. When both the user and the developer operate within the same VM sub-space, interactions are straightforward. However, complications arise when users need to engage with contracts from different sub-spaces.

We next present an motivating example to illustrate how a user can use Metamask wallet, which only supports EVM, to initiate a cross-space transaction on HEMVM, our prototype framework, to trade with PancakeSwap, a dencentralized finance contract deployed in MoveVM. PancakeSwap is an automatic market making (AMM) contract that maintains pools of digital asset pairs and enables users to exchange digital assets with these pools. In our example, we suppose PancakeSwap maintains a pool of two assets: one following the ERC-20 standard from EVM and another following the Move Coin standard. The user wants to trade ERC-20 tokens for Move Coins. Throughout this section and until the rest of this work, we will refer to the sub-space where EVM and MoveVM reside as EVM sub-space and Move sub-space, respectively.

Figure 4 presents the simplified code snippet of PancakeSwap interface. `swap_exact_input()` at line 3 is the function that implements the coin exchange functionality. The function has two type

```
1  contract ProxyRouter {
2    bytes32 constant handler =
3     "<swap_wrapper_addr>"
4    function swap(
5      address tokenIn, address tokenOut,
6      uint amountIn, uint amountOut) {
7      IXProxy portalIn = getProxyAddr(tokenIn);
8      IXProxy portalOut = getProxyAddr(tokenOut);
9      ...
10     tokenIn.transferFrom(
11       msg.sender, address(this), amountIn);
12     // Step 1: cross-chain deposit
13     portalIn.deposit(handler, amountIn);
14     // Step 2: cross-chain call
15     bytes[] args = new bytes[](2);
16     args[0] = encodeU64(amountIn);
17     args[1] = encodeU64(amountOut);
18     bytes[] coins = new bytes[](2);
19     coins[0] = toMoveType(tokenIn);
20     coins[1] = toMoveType(tokenOut);
21     string aptos_module = "swap";
22     string func = "swap_exact_input_handle";
23     crossVM.callMove(
24       handler, aptos_module, func,
25       args, coins);
26     // Step 3: withdraw swapped coins
27     portalOut.withdraw(msg.sender);
28   } ... }
```

(a) Cross-chain proxy contract swap() implementation.

```
1  contract VaultErc20 is IXProxy {
2    bytes32 constant handler =
3      "<coin_wrapper_addr>"
4    function deposit(
5      bytes32 receiver, uint256 amount) {
6      token.transferFrom(
7        msg.sender, address(this), amount);
8      ...
9      bytes[] args = new bytes[](2);
10     args[0] = crossVM.encodeBytes32(receiver);
11     args[1] = crossVM.encodeU64(amount);
12     bytes[] coinType = new bytes[](1);
13     coinType[0] = toMoveType(getUnderlyingToken());
14     crossVM.callMove(
15       handler, "cross_vm_coin", "deposit",
16       args, coinType);
17   } ... }
18 module coin_wrapper::cross_vm_coin {
19   fun ihe_deposit<CoinType>(
20     caller: vector<u8>,
21     message: vector<vector<u8>>
22   ): vector<u8> ... {
23     let amount = decode_u64(&mut message);
24     let receiver = decode_address(&mut message);
25     let coin = make_coin(amount);
26     coin::deposit<CoinType>(receiver, coin);
27     b""
28   } ... }
```

(b) Simplified code for deposit proxies.

Fig. 5. Implementation snippets for some of the proxy contracts.

parameters and three value parameters. The first type parameter `X` denotes the coin type which the user sells to PancakeSwap and the second type parameter `Y` denotes the coin type which the user buys from PancakeSwap. The first value parameter is `sender`, which acts as an identifier for the call initiator. Second parameter `x_in` is the amount of tokens that the caller wants to swap, and `y_min_out` acts as a minimum number of tokens that the caller would like to receive after the swap, i.e., the swap will abort if the user does not provide enough coin `X`.

Note that this is a function in a Move contract and users normally cannot interact with Move contract using Metamask wallet which only supports EVM interfaces. We next show how cross-space operations in HEMVM make such interactions possible.

Cross-space Transaction Steps: Figure 3 presents the high-level diagram of our example cross-space swap transaction. The cross-space swap consists of three main parts. The first part is to move ERC20 assets from the EVM space into the Move space, shown as steps 1 to 7 in Figure 3. The second part is to invoke PancakeSwap to perform the swap, shown as steps 8 to 12 in Figure 3. The last part is to retrieve the obtained Move Coin from the Move space to the EVM space, shown as steps 13 to 17.

Because Metamask wallet cannot interact with non-EVM contracts, developers will deploy three proxy contracts in each space to enable the above transaction interactions. Figures 5a, 5b, and 6 show snippets of these proxy contracts. Proxy contracts in the EVM space provide interfaces for Metamask to interact with, they are paired with proxy contracts in the Move space to handle cross-space interactions. Note that the development efforts of these proxy contracts are negligible because they can be automatically generated based on the source code of PancakeSwap.

Figure 5a presents the main proxy contract in the EVM space that orchestrates this cross-space transaction. The user will use Metamask to call `swap()` at line 4 in Figure 5a to initiate the transaction. `swap()` first calls `transferFrom()` function from the ERC20 standard to transfer user's ERC20 tokens that they want to exchange to the proxy contract. Then it calls `deposit()` function (shown at line 4 in Figure 5b), which handles cross-space transfer to the Move sub-space. In the second part, `swap()` initiates a cross-space call at line 23 to the Move contract shown in

```
1  module swap_wrapper::swap {
2    use pancake::router;
3    ...
4    fun ihe_swap_exact_input_handler<X,Y>(
5      caller: vector<u8>,
6      message: vector<vector<u8>>
7    ): vector<u8> {
8      ...
9      let out = decode_u64(&mut message);
10     let in = decode_u64(&mut message);
11     router::swap_exact_input<X,Y>(
12       &cashier_signer(), in, out);
13     // transfer money to the vault contract
14     sweep_out<Y>();
15     b"" } }
```

Fig. 6. Move swap proxy.

Figure 6, which performs the swap and manages the received tokens. Lastly, after the swap has been performed, the function withdraws the tokens back to the EVM space by calling `withdraw()` function at line 27. We next discuss the deposit and swap steps in details. We omit the withdraw step, because it is symmetric to the deposit step.

Deposit from EVM Space to Move Space: Figure 5b presents the code snippet of two proxy contracts, one deployed in the EVM space and another deployed in the Move space. The two contracts together implement the functionality of moving the ERC20 token from the EVM space to the Move space. Call to `deposit()` at lines 6-17 in Figure 5b implements a lock-and-mint mechanism. The function first transfers tokens from the caller to the address of the contract, at line 6. It then invokes a cross-space call using `crossVM.callMove()`, a built-in function of HEMVM. The call will invoke `ihe_deposit()` at the Move proxy contract `coin_wrapper::cross_vm_coin` (line 19 in Figure 5b). `ihe_deposit()` eventually mints the corresponding amount of new coins at line 25. Note that the new coins are minted to the address stored in the variable of `handler` shown at line 3 in Figure 5a, which corresponds to another proxy Move contract for the follow up swap operation. If the user wants to move the asset back to the EVM space, the user needs to burn the coin in the Move space and then `VaultERC20` will unlock its tokens from the EVM space. We omit this part for brevity.

Invoke Swap: To perform the swap, the cross-space call at line 23 in Figure 5a invokes the function `ihe_swap_exact_input_handler()` in the Move proxy contract in Figure 6 using the `crossVM.callMove()` built-in function. Function `ihe_swap_exact_input_handler()` decodes the arguments at lines 8-11 in Figure 6 and eventually calls `swap_exact_input()` in PancakeSwap to actually swap the tokens. If the swap is successful, the obtained token will be also in the address of the Move proxy contract. The function therefore transfers the tokens out to prepare for the withdraw steps at line 16.

Cross-Space Operations: This example highlights the capability of the cross-space opeations of HEMVM to implement sophisticated interactions. In our example code snippets there are two cross-space operations. The first operation can be found at line 14 in Figure 5b. It invokes the Move function to mint coins during the deposit part. Another cross-space operation is at line 23 in Figure 5a and it invokes the Move proxy to eventually call PancakeSwap. HEMVM supports cross-space calls to the Move space with its built-in fuction `crossVM.callMove()`. The first argument of the call specifies the address of the target contract in the Move space. The second and the third arguments are the contract name and the function name to call. The fourth and the fifth arguments are the value arguments and the type arguments supplied for the invoked Move function. In the deposit case, the cross-space call invokes the function `ihe_deposit()` in the Move contract `coin_wrapper::cross_vm_coin` at line 19 in Figure 5b. The type argument is the coin type of the ERC20 token and the value argument is the amount being transferred.

$\langle prog \rangle$ ::= `program` $\langle inst \rangle^*$

$\langle inst \rangle$ ::= $\langle EVMinst \rangle$ | $\langle MoveVMmod \rangle$ | $\langle Crossinst \rangle$

$\langle Crossinst \rangle$ ::= `VMtoVMCreateDispatcher` $(\text{ad}_s, \text{val}, \text{c}, \text{gp})$ | `VMtoVMCreateHandler` $(\text{ad}_{sys}, \text{ad}_s, \text{val}, \text{c}, \text{gp})$
| `VMtoVMCallDispatcher` $(\text{ad}_s, \text{ad}_t, m, \text{val}, \text{gp})$ | `VMtoVMCallHandler` $(\text{ad}_{sys}, \text{ad}_s, \text{ad}_t, m, \text{val}, \text{gp})$
| `Decode data` | `Encode data`

Fig. 7. Core language of HEMVM cross-space instructions. a^* indicates zero or more occurrences of a.

Our example shows how a heterogenous blockchain system can potentially provide solutions to the fragmentation problem introduced by new VMs, e.g., users can use their favorite wallets to interact with contracts deployed in multiple VMs. Note that one advantage of HEMVM is that the developers do not need to modify the source code of existing contracts. The required proxy contracts can also be automatically generated based on the original contract interface.

4 HEMVM DESIGN

This section presents the design of HEMVM and its formalization. We introduce the instructions that HEMVM adds to the core list of EVM and MoveVM opcodes in order to perform cross-ledger (cross-space) operations. We describe the semantics of the new operations, and also describe the methodology for packing cross-space transactions into blocks.

4.1 Core HEMVM Language

HEMVM defines an environment that allows accounts and smart contracts in one ledger (sub-space) to interact with accounts and smart contracts in the other ledger (sub-space). Note that the interactions between the two sub-spaces are bidirectional: each of the two sub-spaces can initiate a cross-space call to the other sub-space.

In Figure 7, we present the syntax of a simple programming language that we use to formalize HEMVM cross-space design approach. The language extends the standard EVM operations `EVMinst` (e.g., `load` and `push`) and the default MoveVM modules with their built-in functions `MoveVMmod` (e.g., `Account` and `Block`) with a new set of operations `Crossinst`, e.g., `VMtoVMCreate` and `VMtoVMCall`, to enable a heterogeneous blockchain with two ledgers (sub-spaces) under HEMVM. For brevity, in Figure 7 we omit the full description of the EVM standard instructions `EVMinst` and the MoveVM default modules `MoveVMmod` since they are not necessary in understanding the design of HEMVM.

In `Crossinst`, the four operations are categorized into two groups: (1) the first two operations, i.e., `VMtoVMCreateDispatcher` and `VMtoVMCreateHandler`, enable a caller in one sub-space to create a smart contract (resp., module) in a another sub-space; and (2) the last two operations, i.e., `VMtoVMCallDispatcher` and `VMtoVMCallHandler`, enable a caller in one sub-space cross-space to invoke a function within an existing smart contract (resp., module) that is located in a another sub-space. For simplicity, in the rest of the paper, we will refer to the first group as 'cross-space smart contract creation' and the second group as 'cross-space function call'. Such definition of `Crossinst` implies that the deployment of new bytecode and function calls can now be initiated through two mechanisms: either through a native operation that is defined in `EVMinst`/`MoveVMmod` if the transaction sender is located in the same sub-space as the target sub-space or through the smart contract/module creation operations defined in `Crossinst` if the transaction sender is located in a different sub-space compared to the target sub-space.

We use $\mathbb{A}$ to represent the set of addresses that is partitioned between the two spaces. `VMtoVMCreateDispatcher` $(\text{ad}_s, \text{val}, \text{c}, \text{gp})$ and `VMtoVMCreateHandler` $(\text{ad}_{sys}, \text{ad}_s, \text{val}, \text{c}, \text{gp})$ operations allow an address $\text{ad}_s \in \mathbb{A}$ in one space to create a new contract or module in the other space. In these operations, parameter c ∈ <prog> stands for the bytecode of the whole contract or module that needs to be deployed, parameter $\text{val} \in \mathbb{N}$ stands for the amount of native tokens that

is sent along the transaction creation function, and parameter $gp \in \mathbb{N}$ stands for the number of native tokens that the transaction is allowed to spend for gas. Note that parameters val and gp are natural numbers, i.e., $\mathbb{N}$, and the native tokens for sub-spaces do not necessarily have to be identical. The choice to maintain uniform or distinct native tokens across sub-spaces is orthogonal of the handling of cross-vm operations in HEMVM framework.

Similar to the previous two operations, the operations `VMtoVMCallDispatcher` (ad_s, ad_t, m, val, gp) and `VMtoVMCallHandler` $(ad_{sys}, ad_s, ad_t, m, val, gp)$ allow an address $ad_s \in \mathbb{A}$ in one sub-space to call a function m in a deployed contract or module $ad_t \in \mathbb{A}$ in another sub-space and transfer val native tokens. Note that for uniformity, `VMtoVMCallDispatcher` and `VMtoVMCallHandler` can also perform simple cross-space native tokens transfer where in this case the parameter m will be empty. In this case, the account ad_t does not need to contain a deployed code. Lastly, the operations `Encode` data and `Decode` data are used to encode and decode data between the two popular serialization formats Recursive-Length Prefix (RLP) and Binary Canonical Serialization (BSC) that are used in the EVM and MoveVM ledgers, respectively.

Note that the design of the HEMVM can be generalized to any pair of virtual machines since the intra-space instructions are not involved in the definition of the cross-space operations. For instance, we have also adopted the framework of HEMVM in Conflux [31], a production blockchain network, to allow accounts and contracts between two EVM based ledgers to interact.

4.2 Semantics of Cross-space Operations

Global state: The global state in HEMVM is a tuple $\sigma = (h, acts)$ where h is the current block height, and acts stores the states of all accounts. Variable acts maps each account address $ad \in \mathbb{A}$ to $acts(ad) = (non, b, vars, c)$ where non is the account nonce, b is the account balance, vars is the persistent valuation of contract or module variables, i.e., a mapping from the contract/module variables to their values in the ledger, and c is the whole bytecode of the smart contract or module deployed at the address ad.

Storage layout: In the design of HEMVM we use a key-value map to represent the storage where the keys are the accounts addresses, and we abstract away the differences between EVM contract-centric storage (i.e., smart contract stores all of its data under its account address) and Move resource-centric storage (i.e., resources related to a module can be stored across different account addresses). However, note that for both storage types the HEMVM storage abstraction is valid since both types can be represented as key-value map where the keys are accounts addresses. Thus, the domain of addreses for the global state component acts is partitioned between the two sub-spaces.

Execution of high-level cross-space operations: In HEMVM, a cross-space smart contract deployment transaction (resp., cross-space regular function call transaction) consists of the two operations, (`VMtoVMCreateDispatcher`, `VMtoVMCreateHandler`) (resp., (`VMtoVMCallDispatcher`, `VMtoVMCallHandler`)), that occur atomically together one after the other: first operation dispatches the transaction from the originating sub-space and the second operation handles the transaction in the target sub-space. For example, for a cross-VM call `crossVM.callMove(...)` from the example in Figure 5a, the pair (`VMtoVMCallDispatcher`, `VMtoVMCallHandler`) would be executed. The four operations in `Crossinst` are implemented as functions in an *internal contract or module*, i.e., a precompiled contract or module, within each of the two sub-spaces and it acts as an interface for facilitating cross-space interactions between programs on both. Internal contracts allow HEMVM to extend both execution engines without introducing new low-level opcodes which will require introducing high-level language features for using them, e.g., Solidity language primitives, and extending the compilers, e.g., Solc compiler, to translate those features to the low-level opcodes. We use the variable $ad_{sys} \in \mathbb{A}$ to represent the reserved address of the internal contract or module.

$$\frac{(\mathsf{non}_s, \mathsf{b}_s, \_, \_) = \mathsf{acts}(\mathsf{ad}_s) \quad \mathsf{crossVmDispatcherPerm}(\mathsf{ad}_s) \quad (\_, \mathsf{b}_{sys}, \_, \_) = \mathsf{acts}(\mathsf{ad}_{sys}) \\ \mathsf{b}_s \geq \mathsf{gp} + \mathsf{val} \quad \mathsf{acts}' := \mathsf{acts}[\mathsf{ad}_s \mapsto (\mathsf{non}_s + 1, \mathsf{b}_s - \mathsf{gp} - \mathsf{val}, \_, \_);\ \mathsf{ad}_{sys} \mapsto (\_, \mathsf{b}_{sys} + \mathsf{val}, \_, \_)] \\ \texttt{VMtoVMCreateHandler}\ (\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{val}, \mathsf{c}, \mathsf{gp})}{(\mathsf{h}, \mathsf{acts}) = [\![\ \texttt{VMtoVMCreateDispatcher}\ (\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{val}, \mathsf{c}, \mathsf{gp}_s)\ ]\!] \Rightarrow (\mathsf{h}, \mathsf{acts}')}$$

$$\frac{(\mathsf{non}_{sys}, \mathsf{b}'_{sys}, \_, \_) = \mathsf{acts}(\mathsf{ad}_{sys}) \quad \mathsf{crossVmHandlerPerm}(\mathsf{ad}_{sys}) \quad \mathsf{isFresh}(\mathsf{ad}_n) \\ \mathsf{vars}_0 := \mathsf{init}(\mathsf{ad}_n) \quad \mathsf{acts}' := \mathsf{acts}[\mathsf{ad}_n \mapsto (\mathsf{non}_{sys}, \mathsf{val}, \mathsf{vars}_0, \mathsf{c});\ \mathsf{ad}_{sys} \mapsto (\mathsf{non}_{sys} + 1, \mathsf{b}'_{sys} - \mathsf{val}, \_, \_)]}{(\mathsf{h}, \mathsf{acts}) = [\![\ \texttt{VMtoVMCreateHandler}\ (\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{val}, \mathsf{c}, \mathsf{gp})\ ]\!] \Rightarrow (\mathsf{h}, \mathsf{acts}')}$$

$$\frac{(\mathsf{non}_s, \mathsf{b}_s, \_, \_) = \mathsf{acts}(\mathsf{ad}_s) \quad \mathsf{crossVmDispatcherPerm}(\mathsf{ad}_s) \quad (\_, \mathsf{b}_{sys}, \_, \_) = \mathsf{acts}(\mathsf{ad}_{sys}) \\ \mathsf{b}_s \geq \mathsf{gp} + \mathsf{val} \quad \mathsf{acts}' := \mathsf{acts}[\mathsf{ad}_s \mapsto (\mathsf{non}_s + 1, \mathsf{b}_s - \mathsf{gp} - \mathsf{val}, \_, \_);\ \mathsf{ad}_{sys} \mapsto (\_, \mathsf{b}_{sys} + \mathsf{val}, \_, \_)] \\ \texttt{VMtoVMCallHandler}\ (\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{ad}_t, m, \mathsf{val}, \mathsf{gp})}{(\mathsf{h}, \mathsf{acts}) = [\![\ \texttt{VMtoVMCallDispatcher}\ (\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{ad}_t, m, \mathsf{val}, \mathsf{gp})\ ]\!] \Rightarrow (\mathsf{h}, \mathsf{acts}')}$$

$$\frac{(\_, \mathsf{b}_t, \mathsf{vars}_t, \mathsf{c}_t) = \mathsf{acts}(\mathsf{ad}_t) \quad \mathsf{crossVmHandlerPerm}(\mathsf{ad}_{sys}) \quad (\mathsf{non}_{sys}, \mathsf{b}'_{sys}, \_, \_) = \mathsf{acts}(\mathsf{ad}_{sys}) \\ \mathsf{acts}' := \mathsf{acts}[\mathsf{ad}_n \mapsto (\_, \mathsf{b}_t + \mathsf{val}, \mathsf{vars}, \mathsf{c}_t);\ \mathsf{ad}_{sys} \mapsto (\mathsf{non}_{sys} + 1, \mathsf{b}'_{sys} - \mathsf{val}, \_, \_)] \\ \mathsf{acts}'' := \rho_{(m, \mathsf{c}_t, \mathsf{ad}_s, \mathsf{ad}_{sys})}(\mathsf{acts}')}{(\mathsf{h}, \mathsf{acts}) = [\![\ \texttt{VMtoVMCallHandler}\ (\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{ad}_t, m, \mathsf{val}, \mathsf{gp})\ ]\!] \Rightarrow (\mathsf{h}, \mathsf{acts}'')}$$

$\mathsf{isFresh}(a) \triangleq$ creates a fresh address a $\in \mathbb{A}$

Fig. 8. The operational semantics of cross-space transactions. init returns the initial state of a contract persistent variables. $\rho_{(m, \mathsf{c}_t, \mathsf{ad}_s, \mathsf{ad}_{sys})}(\mathsf{acts}_t)$ takes the target sub-space state acts_t and executes the method m implemented in the code c_t over the given state. crossVmDispatcherPerm and crossVmHandlerPerm check whether the invoker of the cross-space operations has the correct privileges to do so. isFresh(a) checks whether an address a already contains state data in the sub-space within which it is executing.

The `VMtoVMCreateDispatcher` ($\mathsf{ad}_s, \ldots$) (resp., `VMtoVMCallDispatcher` ($\mathsf{ad}_s, \ldots$)) is the operation, i.e., function, in the internal contract or module through which a user with an address ad_s can initiate a cross-space contract deployment transaction (resp., cross-space regular function call transaction) in the originating sub-space. Consequently, `VMtoVMCreateDispatcher` (resp., `VMtoVMCallDispatcher`) from the internal contract or module with the address ad_{sys} calls `VMtoVMCreateHandler` ($\mathsf{ad}_{sys}, \ldots$) (resp., `VMtoVMCallHandler` ($\mathsf{ad}_{sys}, \ldots$)) in the internal contract or module to complete the cross-space contract deployment transaction (resp., cross-space regular function call transaction) in the target sub-space. Note that the address space is partitioned between two sub-spaces, meaning that a user address ad_s uniquely defines the sub-space where the transaction originates.

Opcode semantics: Figure 8 presents the semantic rules of the cross-space operations execution in HEMVM. We use $\sigma = [\![\ \texttt{VMtoVMInstr}\ ]\!] \Rightarrow \sigma'$ to interchangeably denote the global state transition from σ to σ' after executing the corresponding cross-space operation.

Cross-space contract/module creation: The transitions labeled by `VMtoVMCreateDispatcher` (ad_s, val, c, gp) corresponds to the initiation of a cross-space contract creation transaction initiated by an account with an address ad_s in the first sub-space. First, it ensures that the balance of the sender is sufficient to pay for the cross-space account creation transaction[1]. Then, ad_s transfers the value amount val (that can be zero) to ad_{sys} in the first sub-space. Finally, it invokes the handler opcode `VMtoVMCreateHandler` in the second sub-space[2] where the new contract will be created.

[1]If the balance is not sufficient, the operation is cancelled, and since all transactions in blockchain systems are atomic, the state gets reverted back to the last correct state known to both of the sub-space ledgers.

[2]For simplicity we assume the internal contract and module in the two sub-spaces have the same address ad_{sys}.

Algorithm 1 A procedure to pack cross-space operations. On first execution, both blk and BLK are empty.

```
1:  procedure BlockPacking(VMtoVMInstr*, σ)
2:      σ' ← ExInst(VMtoVMInstr, σ);
3:      (h, acts) ← σ;
4:      (h, acts') ← σ';
5:      tx ← (acts, acts');
6:      blk ← blk ∪ {tx};
7:      if blk is full
8:        BLK ← BLK ∪ {blk};
9:        blk ← ϵ;
10:       h ← h + 1;
11:     σ ← (h, acts');
12:     return (σ, blk, BLK);
13: end procedure
```

This triggers the transition `VMtoVMCreateHandler` $(\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{val}, \mathsf{c}, \mathsf{gp})$, which creates a fresh address ad_n in the destination sub-space acts for the new account. Then, it executes an ordinary deployment transaction on the freshly created address ad_n with the code c and transfers to it the specified value amount val from ad_{sys}.

Cross-space function calls: The transition labeled by `VMtoVMCallDispatcher`$(\mathsf{ad}_s, \mathsf{ad}_t, m, \mathsf{val}, \mathsf{gp})$ corresponds to the initiation of a cross-space function invocation transaction from the address ad_s. First, the transition ensures that the balance of the sender is sufficient to pay for the cross-space account creation transaction. Then, ad_s transfers the value amount val to ad_{sys} in the first sub-space. Finally, it invokes the handler opcode `VMtoVMCallHandler` in the second sub-space that contains the target address ad_t. This triggers the transition `VMtoVMCallHandler` $(\mathsf{ad}_{sys}, \mathsf{ad}_s, \mathsf{ad}_t, m, \mathsf{val}, \mathsf{gp})$ in the second sub-space, which calls the function m in the contract/module associated with the address ad_t in its sub-space and transfers the balance val to ad_t from ad_{sys}.

In the above opcodes, the predicates `crossVmDispatcherPerm` and `crossVmHandlerPerm` have to be satisfied in order for the transitions to happen. Those predicates enfroce access control for cross-VM operations, we will present them in details in Section 4.4. In a nutshell, they check whether a given address has permission to execute a cross-VM operation.

4.3 Cross-space Transactions Packing

In Algorithm 1, we give the procedure for packing cross-space operations as one transaction that involves updating the states of the two sub-spaces. It is executed every time a cross-space transaction needs to be packed into a block. Each cross-space transaction consists of invoking either `VMtoVMCreateDispatcher` and `VMtoVMCallDispatcher` which, in turn, invokes `VMtoVMCreateHandler` or `VMtoVMCallHandler`, respectively. Thus, Algorithm 1 takes a cross-space operation `VMtoVMInstr`$^* \in$ {`VMtoVMCreateDispatcher`, `VMtoVMCallDispatcher`} that is initiating a cross-space transaction and a global state σ. It returns:

(1) Current block `blk` which is not yet finalized and to which the network will keep adding new transactions until it is full,

(2) Chain of blocks BLK which contains blocks that have already been finalized and organized in a cryptographically secure chain.
(3) Updated global state σ after the execution of the cross-space transaction.

Algorithm 1 first executes the cross-space operation as detailed in the transition rules in Figure 8 denoted ExInst (line 2). Then, it creates a transaction to represent the states changes of each sub-space (line 5). The created transaction is then added in the current block being packed (line 6). If the block is full, it is then added to the chain of blocks and the block being packed is set to empty and the block height is incremented (lines 7-10).

Native token transfer in a MoveVM-based sub-space: In contrast to an EVM-based sub-space, in a MoveVM-based sub-space it is not possible to transfer native tokens during a contract deployment (i.e., constructor function call) or a regular function call. Thus, a cross-space operation from an EVM-based sub-space to a MoveVM-based sub-space actually involves three intra-space operations. In particular, before the internal module performs the operation to deploy a contract or call a function in the MoveVM-based sub-space, it first performs an additional operation to deposit the native tokens amount into the specified address.

4.4 Cross-space Access Control

HEMVM has an access control mechanism to protect both the dispatcher and handler functions of cross-space operations from malicious interactions. In particular, the dispatcher function should authenticate that the initiator of cross-space operation is eligible to do so, i.e., crossVmDispatcherPerm(ad) in Figure 8. Also, the handler function should verify that it is invoked within a cross-space operation, i.e., crossVmHandlerPerm(ad) in Figure 8. In an EVM-based sub-space, HEMVM uses the msg.sender, an EVM built-in variable that refers to the address of the caller, to verify the initiator of a transaction. In particular, the dispatcher function uses msg.sender to implement an access control mechanism to limit who can perform cross-space operations. Therefore, the implementation of the handler function can check the value of the passed msg.sender to verify whether the invocation is legitimate or not. If the developer wants the handler function only being called during the cross-space operations, it should limit the caller of the handler function to only the internal contract address ad_{sys}. Thus, in an EVM-based sub-space, we will have:

$$\mathsf{crossVmDispatcherPerm(ad)} \triangleq \mathsf{ad} \in \mathbb{A}_{allowed} \qquad \mathsf{crossVmHandlerPerm(ad)} \triangleq \mathsf{ad} = \mathsf{ad}_{sys}$$

where $\mathbb{A}_{allowed}$ is the set of addresses allowed to initiate cross-space transactions from EVM-based sub-space.

On the other hand, in an MoveVM-based sub-space the msg.sender feature is not available. In this case, HEMVM uses an approach similar to the *MintCapability in Aptos Coin* in the Move language [8]. In particular, the dispatcher function uses a cross-space permission capability, i.e., a permission to perform certain operations [13, 33], to implement an access control mechanism to only allow the holder of the *CrossSpaceCapability* to call it.

```
struct CrossSpaceCapability<phantom CallType> has copy, store, drop {}
```

In the Move-based sub-space, we will have then:

crossVmDispatcherPerm(ad) ≜ borrow_global<CrossSpaceCapability>(signer :: address_of(ad))

where we ensure that the address ad has the *CrossSpaceCapability*.

For the handler function, the function must be declared as a private function (the default visibility in Move language [8]), thus disallowing other modules within its sub-space from calling it. The internal module ad_{sys} will bypass the function visibility and invoke the handler function since it is

integrated within the MoveVM execution engine. Thus, we do not need to do check for the handler function in the Move-based sub-space: crossVmDispatcherPerm(ad) = True

Note that the handler function may also need to validate the address of the cross-space operation initiator. To address this, in HEMVM the internal contract (or module) embeds the address of the cross-space transaction sender ad_s as an additional parameter in the function call transaction.

5 IMPLEMENTATION

We now present a prototype implementation of the HEMVM framework that supports cross-space operations between an EVM-based space and a MoveVM-based space, called HEMVM as well. HEMVM is implemented on top of the Aptos client [16] in Rust programming language. The implementation of HEMVM consists of two main components: a shared database storage to hold the states of both EVM-based and MoveVM-based spaces; and an execution engine that support both intra-space and cross-space transactions.

5.1 HEMVM Storage

HEMVM uses one shared key-value database storage to store the states of the two spaces, while keeping the two states logically separated within the database. The database keys correspond to account addresses in both Ethereum and Aptos. HEMVM key-value database storage extends AptosDB to support mapping accounts addresses in Ethereum. Note, however, that addresses in Ethereum and Aptos have different sizes, i.e., 20 and 32 bytes, respectively. To address this, we extend the Ethereum addresses with 12 bytes of zeros. Thus, all the keys of the shared key-value databases are of size 32 bytes. To store or fetch the state values of the EVM-based space, we convert them from RLP to BSC or BSC to RLP using the Encode and Decode serialization and deserialization procedures, respectively.

Note that the probability of a collision between an Aptos address and Ethereum address is very small. Consider the case where the EVM space contains one billion unique addresses (Ethereum has less than 260 millions unique addresses at the end of 2023), the chance of a randomly generated Move address colliding with any existing EVM address is $10^9/2^{256} \approx 8.64 \times 10^{-69}$. An attacker would have to generate on average 1.16×10^{77} different addresses to deliberately cause a collision.

5.2 HEMVM Execution Engine

HEMVM extends the Aptos execution engine to include an EVM execution engine to process Ethereum transactions. Our implementaiton of EVM in HEMVM is based on the EVM modules in OpenEthereum [1]. HEMVM then extends both Apots and EVM execution engines to support cross-space transactions.

On the MoveVM side, the internal module is implemented as a native *aptos_framework* Move module called *cross_vm* providing first-class features for executing cross-operations to EVM-based space. It provides the following Move interface for MoveVM to EVM cross-space function call:

```
public native fun call_evm(coin, addr, function, params, cap): vector<u8>;
```

where `coin` parameter specifies the cross-space transferred native tokens. The handler function in the EVM-based space is uniquely identified by the tuple `(addr, function)`, and its arguments are specified in `params`. The last parameter `cap` refers to the capability of the caller to perform cross-space function call. The function returns an array of bytes encoding the result of the computation and returned data.

On the EVM side, HEMVM provides a functional interface for cross-space calls and the functions are implemented within its execution engine. The reserved address 0x08880...06 is assigned to the internal contract account. Thus, when a call is performed to an arbitrary address, the execution

engine checks whether this address corresponds to the reserved address of the internal contract account. If it is, it then redirects the control flow from low-level bytecode execution to a custom Rust module that defines the required logic. In particular, this module decodes the function call that needs to be performed and redirects the execution to the corresponding function implementation within the module.

The internal contract provides the following Solidity interface for EVM to MoveVM cross-space function call:

```
function callMove(addr, module, func, data, types) external payable returns
  (bytes memory);
```

where `callMove` takes a triplet `(addr, module, func)` that uniquely identifies the handler to be invoked in the MoveVM-based space. It also accepts two additional parameters `data` and `types`, which represent encoded function arguments and type arguments respectively. `callMove` is `payable` function, allowing the transfer of native tokens cross-spaces. It returns an array of bytes in order to encode arbitrary return data.

5.3 2EVM-based Implementation of HEMVM

We have also implemented the HEMVM design in Conflux [31], a high-performance blockchain network. Conflux has a VM that is similar to Ethereum VM, i.e., ConfluxVM can execute all EVM instructions. However, Conflux has a different transaction format and a different rule for generating account addresses from public keys than Ethereum VM. For instance, Conflux uses *base32* format address, e.g., cfx : aa... while the EVM space uses *hexadecimal checksum* format address, e.g., 0xaAD8... The new HEMVM-based Conflux system consists of two spaces; one new space that is fully EVM-compatible, and the other space is based on the ConfluxVM and is backward compatible with previous Conflux accounts and transactions formats. It allows a smooth porting of Ethereum compatible decentralized applications (DApps) to Conflux network.

Similar to before, the two main implementation components of HEMVM-based Conflux system are the storage and the execution engine. The storage component is implemented on top of the Conflux key-value authenticated data structure storage. The keys from the two spaces (EVM and ConfluxVM) are mapped. In particular, the storage key for an account in the EVM space corresponds to the storage key for the account with the same address in the ConfluxVM (native) space where we insert the byte 0x81 at position 20 (indexing starts at 0).

The new execution engine uses an internal contract account associated with a reserved address 0x08880...06 that provides the following Solidity interface for applications to perform cross-space operations:

```
function createEVM(calldata init) external payable returns (bytes20);
function transferEVM(bytes20 to) external payable returns (memory output);
function callEVM(bytes20 to, calldata data) external payable returns (memory output);
```

Since Solidity is the most popular programming language for writting DApps, the above Solidity interface makes it easy for DApps developers to write smart contracts in Solidity that perform cross-space operations using the set of functions declared in the interface. The new execution engine also supports the precompiled contracts of Ethereum from address 0x00..01 to address 0x00..08.

5.4 Parallel Execution

In HEMVM, the storage for different virtual machines (VMs) is completely independent, ensuring that intra-space transactions - those confined to a single VM - do not create race conditions when executed in parallel. This separation allows HEMVM to significantly boost performance

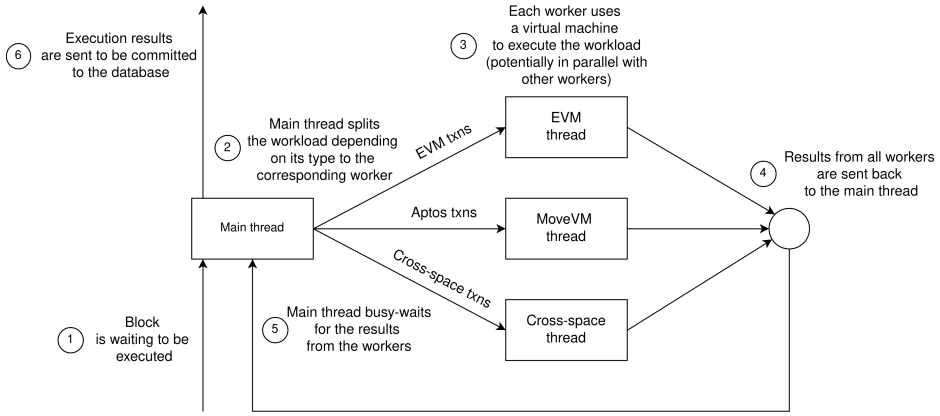

Fig. 9. Implementation of the parallelized execution in HEMVM.

by parallelizing intra-space transactions that do not involve cross-space operations. However, transactions that span multiple VMs (i.e., cross-space transactions) can introduce race conditions, as they involve interactions between different VMs, requiring special handling to ensure data consistency.

To enable HEMVM to handle parallel execution intra-space transactions while ensuring safe and correct execution of cross-space transactions, we implement a parallel version of HEMVM using thread worker pattern. In particular, we split transactions in three diffrent categories: EVM intra-space transactions, MoveVM intra-space transactions, and cross-space transactions. Each transaction type will have its own worker thread, illustrated in Figure 9, that will be executing only transactions of this type. The main thread sends transactions to the corresponding worker thread, batching similar transactions in order to reduce the communication latency. MoveVM and EVM intra-space transactions can be executed in parallel, however, cross-space transaction worker blocks the execution for any other worker in order to avoid any race conditions. Once the execution of a transaction has been finished in any of the workers, the results are sent back to the main thread.

5.5 Gas Model

The HEMVM framework is designed orthogonal from any particular gas incentive mechanism. This is because the gas fees of transactions are blockchain specific, and they do not necessary depend only on the cost of the operations in the transactions, but also on the nature of the blockchain and its ecosystem. Thus, in the prototype implementation of HEMVM we did not adopt any gas model.

However, note that when designing an incentive mechanism for a HEMVM-based system of two ledgers, one needs to carefully consider the transactions throughput of the two ledgers when fixing the gas cap per block. For instance, fixing the gas cap in each ledger space alone without considering the other ledger may result in undesirable behaviors. If each block of a ledger $\mathbb{B}_1$ can pack 1000 transactions, and each block of another ledger $\mathbb{B}_2$ can only pack 500 transactions, then a cap of 500 cross-chain transactions in $\mathbb{B}_1$ will be enough to fill the capacity of a single block in $\mathbb{B}_2$ with cross-chain transactions alone, preventing $\mathbb{B}_2$ from packing its own transactions. Thus, to prevent this denial-of-service scenario either the block size of $\mathbb{B}_2$ needs to increase or the cap on the cross-chain transactions in $\mathbb{B}_1$ needs to decrease.

In the HEMVM-based Conflux implementation multiple gas incentive mechanisms are adopted. Specifically, the total gas limit of cross-space transactions within a block cannot exceed half of the block gas limit. In a cross-space function call transaction, only $\frac{1}{10}$ of the transaction gas can be used

to cover the fees of executing operations in the destination space. This is aimed to limit gas usage and discourage users from frequently launching complex cross-space transactions.

6 EVALUATION

We next evaluate the proposed heterogenous blockchain framework and our prototype system, HEMVM. The goal of our evaluation is to answer the following questions:

(1) Is our proposed framework robust enough to support multiple VMs running end-to-end efficiently?
(2) Is our cross-space operation design expressive enough to implement sophisticated multi-VM smart contract interactions?
(3) Is our proposed framework general enough to support different kinds of VMs?

Benchmark Scenarios: To evaluate our prototype system, HEMVM, we consider two different groups of workloads. The first group tests the performance of single-threaded prototype for homogeneous workloads, and the second one considers multi-threaded prototype under mixed workloads.

For the first group of workloads, we consider seven different benchmark scenarios: native coin transfer in EVM, ERC20 coin transfer in EVM, native coin transfer in MoveVM, custom coin transfer in MoveVM, interaction with PancakeSwap in MoveVM (i.e., our example in Section 3), interaction with UniswapV2 [23] in EVM, and interaction with a simplified version of CompoundV2 [29] in EVM. UniswapV2 and PancakeSwap are two popular decentralized trading protocols based on automatic market making. CompoundV2 is a popular decentralized lending protocol that allows users to borrow digital assets using other assets as collateral.

To estimate the overhead of using the HEMVM on an existing blockchain, we also run native coin, custom coin and AMM swap benchmarks on a forked Aptos client that has no HEMVM modifications. To evaluate the capability of our proposed framework to support different kinds of VMs, we also implemented HEMVM in Conflux [31], a production blockchain system with two EVM-like VMs. One of the VM is fully compatible to EVM and another is based on EVM but modified the transaction format to support a different incentive mechanism. We evaluate the native and ERC20 coin transfer scenarios between these two EVM-like VMs.

Second workload group evaluates the performance increase of parallelized HEMVM vesrion compared to the sequential one under the mixed transaction workloads. We focus on native coin transfers, custom coin transfers, and on decentralized trading protocols, however, we create different variations of the workloads with and without cross-space transactions to test the limits of the prototype.

Experimental Methodology: For each benchmark scenario that involves cross-space transactions, we deploy corresponding proxy contracts. We generate 100K transactions with fixed sender and fixed receiver for the sequential group of workloads (Table 1), and generate 500K transactions with random senders and receivers for parallelized prototype (Table 2) with a pool of 12.5K accounts for each of the spaces (25K accounts in total). We then send the generated transactions into HEMVM and measure the end-to-end transaction processing throughput. For Table 1, we run each scenario twice, one from the same VM space and one from the other VM space via cross-space operations. For Table 2, we run each scenario in a parallelized prototype and in a sequential prototype to compare their performance. To avoid potential performance bottlenecks introduced by the consensus layer or the transaction pool, we run our experiments with a single node blockchain that has the transaction pool removed, which allows to immediately pack and execute received transactions. All experiments have been run on a machine with AMD Ryzen Threadripper PRO 5945WX 12-Cores CPU and 32GB RAM.

	Base VM	Intra-space	Cross-space
Aptos Native	Move	4952	N/A
Aptos Custom	Move	4784	N/A
Aptos PancakeSwap	Move	1220	N/A
HEMVM Native	Move	4784	4770
HEMVM Custom	Move	4581	1100
HEMVM Native	EVM	17552	2152
HEMVM ERC20	EVM	7036	1368
HEMVM PancakeSwap	Move	1219	355
HEMVM UniswapV2	EVM	1025	320
HEMVM CompoundV2	EVM	1019	358
Conflux Native	EVM	9363	9337
Conflux ERC20	EVM	9285	5491

Table 1. Performance results of homogeneous benchmark scenarios for sequential prototoype (100K transactions, fixed sender and receiver). We count each cross-space transaction as one transaction.

In the end, in order to understand how cross-space operations enable implementation of sophisticated smart contract interactions, we also present case studies for the UniswapV2 and CompoundV2 scenarios.

6.1 Experimental results

Sequential Prototype: Table 1 presents our experimental resutls for the sequential prototypes. Each row of the table corresponds to the result of one benchmark scenario. The first column shows the name of each scenario. The second column shows the base VM type of the benchmark contract or the token. The third column shows the resulting transaction per second (TPS) measurement when we run a corresponding scenario inside one VM space. The fourth column shows the result TPS when we run the corresponding scenario with our cross-space opeations, e.g., the PancakeSwap contract is deployed in the Move space and we send EVM transactions via proxy contracts to interact with it.

Our results show that the HEMVM client demonstrates high performance. When running only intra-space transactions, HEMVM achieved 4784 TPS for native Move coin transfers, 4581 TPS for custom coin transfers and 1219 TPS for PancakeSwap transactions. For comparison, the native Aptos blockchain can achieve 4925, 4784 and 1220 TPS for native coin transfers, Move coin transfers and PancakeSwap transactions, respectively. Therefore, the overhead of integrating additional VM is less than 4.4 %. Furthermore, our proposed framework achieves 17552 TPS for native EVM transfers, 7036 for ERC20 token transfers, and 1019-1219 TPS for complicated DeFi transactions. HEMVM client demonstrates fast intra-space performance due to the fact that the proposed framework only adds new opcodes without changing existing virtual machine designs.

Our results also show that the performance of cross-chain operations is acceptable. Because each cross-space transaction involves multiple calls via proxy contracts, it will be significantly more complicated than the original intra-space transaction. Nevertheless, HEMVM achieves more than 1100 TPS for cross-space transfers and more than 300 TPS for cross-space DeFi smart contract transactions.

Our results further show the proposed framework can be adopted for different kinds of VMs. In fact, the cross-space transactions between two EVM-like spaces in the production blockchain system are faster than the cross-space transactions in HEMVM. This is because the two virtual

Experiment payload	Par. TPS	Seq. TPS	Diff. (%)
Native (80% ETH, 20% Aptos Native)	8560	7460	14.7%
Native (70% ETH, 20% Aptos Native, 5+5% ETH + Aptos Native Cross)	5912	6051	-2.3%
Coin (60% ERC20, 40% MoveCoin)	4979	4127	19.4%
Coin (55% ERC20, 35% MoveCoin, 5+5% ERC20 + MoveCoin Cross)	3347	3143	6.4%
PancakeSwap (15% Native AMM, 15% MoveCoin, 70% ERC20)	3547	2986	18.7%
PancakeSwap (15+10% Native+Cross AMM, 15% MoveCoin, 60% ERC20)	1559	1531	1.8%
Uniswap (20% Native AMM, 30% ERC20, 50% MoveCoin)	2958	2343	26.2%
Uniswap (20+10% Native+Cross AMM, 30% ERC20, 40% MoveCoin)	1510	1421	6.2%
DeFi(45% Native Uni, 55% Native Pancake)	1457	1006	44.8%
DeFi(45+5% Native+Cross Uni, 45+5% Native+Cross Pancake)	968	830	16.6%

Table 2. Performance results of non-homogenous benchmark scenarios for parallelized prototype (500K transactions, random accounts). We count each cross-space transaction as one transaction.

```
function swapExactTokensForTokens(
  uint amountIn, uint amountOutMin,
  address[] calldata path, address to,
  uint deadline
) returns (uint[] amounts) { ... }
```

Fig. 10. Interface of UniswapV2 swapTokensForTokens().

machines are similar and, therefore, the cost of switching the execution environment is smaller comparing to crossing between the EVM and Move spaces.

Parallelized Prototype: We now present the results for the parallelized version of HEMVM. Table 2 summarizes the transaction per second (TPS) performance across different workloads. The first column lists the workload composition, while the second and third columns provide the TPS measurements for the parallelized and sequential prototypes, respectively. The fourth column shows the percentage change in performance when comparing the two versions.

Our results show that parallelization significantly enhances transaction throughput in HEMVM, especially for workloads involving complex transactions. In particular, the decentralized finance (DeFi) trading workload, which includes Uniswap and PancakeSwap transactions, saw a 44.8% improvement in performance. This improvement is largely due to the fact that complex transactions are more constrained by the VM execution layer, where parallelization has the most impact. The degree of performance improvement, however, varies depending on the ratio of cross-space transactions within the workload. While cross-space transactions cannot be parallelized alongside other transactions in the current prototype, HEMVM still achieved a 6.2%-16.6% throughput improvement in three out of five scenarios, even when 10% of the transactions involved cross-space operations.

6.2 Case Study

We next present case studies on the UniswapV2 and CompoundV2 scenarios to illustrate how HEMVM enables sophisticated cross-VM smart contract interactions. Note that we already presented how HEMVM operates with PancakeSwap in Section 3.2 in detail.

UniswapV2: UniswapV2 [23] is a popular decentralized trading protocol using automated market making mechanisms similar to PancakeSwap [36]. The smart contract of UniswapV2 is based on EVM. In our experiments, HEMVM successfully enabled users to use Aptos wallets, which only operates with the Move interface, to trade with UniswapV2 via cross-space operations in HEMVM.

Figure 10 shows the code snippet of the UniswapV2 interface for the swap operation. The first parameter `amountIn` of `swapExactTokenForTokens()` denotes the amount of input tokens the user plans to trade. The second parameter `amountOutMin` denotes the minimum amount of output tokens the user must obtain or the trade will abort. The third parameter `path` is an array of token addresses denoting the trading path. For simple trade between two tokens, the array will contain two token addresses, the first being the input token address and the second being the output token address. The fourth parameter `to` denotes the address that will receive the obtained token and the last parameter `deadline` denotes the time limit of this trade.

Similar to the example in Section 3, HEMVM enables this cross-space transaction in three steps. HEMVM first moves neccessary assets from the Move space to the EVM space via a pair of proxy contracts. HEMVM then invokes the UniswapV2 swap function via a cross-space call. In the last step, HEMVM withdraws the obtained tokens back to the Move space via another pair of proxy contracts. The high level control flow diagram is similar to Figure 3 but in the reverse direction (i.e., crossing from the Move space to the EVM space).

Figure 11 presents the code snippet of the main proxy contract in the Move space which the Aptos wallet interacts with. Function `swap_exact_tokens_for_token()` encodes cross-space call parameters at lines 8-21 and eventually initiates a cross-space call at line 22. This cross-space call execution will start at the handler defined in a contract in the EVM side shown as line 26 in Figure 11, which in turn calls the UniswapV2 swap function at line 29.

HEMVM provides the cross-space call functionality via `cross_vm::call_evm()` built-in functions, where `cross_vm` is the internal contract address alias in HEMVM that clusters all cross-space builtin functions. The first parameter denotes the amount of native coin transfers of this cross-call, which is none in this case. The second and the third parameters denote the contract and the handler function to invoke. The fourth parameter contains the encoded parameters of this cross-call. The last parameter enables the access control for the handler functions.

CompoundV2: CompoundV2 is a popular decentralized digital asset lending protocol for EVM [29]. Users can deposit digital assets into the protocol vault and then lend out other kinds of digital assets using the deposited assets as collateral. To maintain the solvency of the protocol, CompoundV2

```
1  public entry fun swap_exact_tokens_for_tokens<X, Y>(
2    account: &signer, amount_in: u64,
3    amount_out_min: u64, deadline: u64,
4    cross_uniswap_wrapper_eth_address: vector<u8>) ... {
5    ...
6    let encoded_amount_in = bcs::to_bytes(&amount_in);
7    let encoded_amount_out_min = bcs::to_bytes(
8      &amount_out_min);
9    let encoded_deadline = bcs::to_bytes(&deadline);
10   let handler = cross_uniswap_wrapper_eth_address;
11   let funcName = b"handleSwapExactTokensForTokens";
12   let call_cap = &borrow_global<CapStore>(
13     @coin_wrapper).call_cap;
14   let params: vector<vector<u8>> = vector[
15     raw_type<X>(), raw_type<Y>(),
16     encoded_amount_in, encoded_amount_out_min,
17     encoded_deadline];
18   cross_vm::call_evm(
19     option::none(), handler, string::utf8(funcName),
20     params, call_cap
21   ); ... }
22 function handleSwapExactTokensForTokens(
23   string caller, bytes[] data) public ... {
24   ...; router.swapExactTokensForTokens(...); ...
25 }
```

Fig. 11. Cross-space call from Move sub-space to EVM sub-space

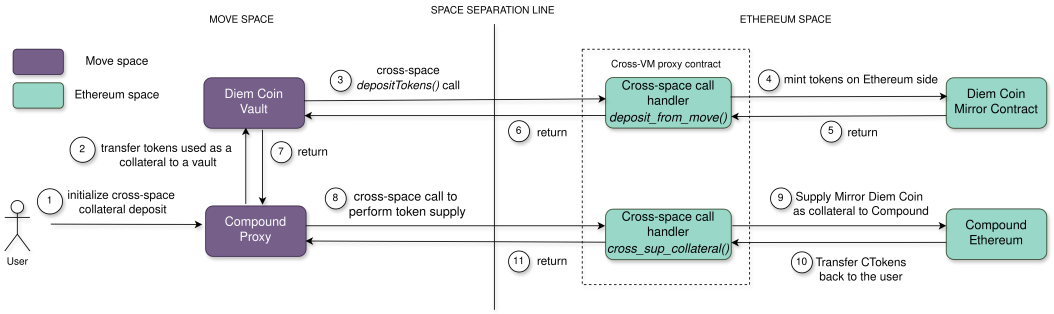

Fig. 12. Cross-space collateral deposit using simplified CompoundV2 protocol.

makes sure that the total value of the collateral assets is greater than the total value of the borrowed assets by a margin for each user. If a user violates this constraint, their position will be closed by CompoundV2 and his collateral will be sold at other markets such as Uniswap to pay back his loans.

With the cross-space operations, HEMVM successfully enables users to use the Aptos wallet to interact with a simplified version of CompoundV2 deployed in the EVM space. There are two cases for this simplified CompoundV2 protocol: depositing assets and borrowing assets. Figure 12 presents the high level control flow diagram for the deposit case. In the deposit case, HEMVM first uses a pair of proxy contracts to enable the movement of assets from the Move space to the EVM space, shown as steps 2-7 in Figure 12. HEMVM then calls the CompoundV2 via the main EVM proxy contract to supply the deposit, shown as steps 8-11 in Figure 12. The borrow case is symmetric to the lending case. HEMVM first calls the CompoundV2 via a handler in the main EVM proxy contract to borrow assets. HEMVM then moves the borrowed assets from the EVM space back to the Move space.

Note that the above pattern can work with unmodified CompoundV2 contract code. The main proxy contract in the EVM space will maintain the position in CompoundV2 instead of the user. From the point of view of CompoundV2, it is the proxy contract in EVM which deposits and borrows assets. It does not need to be aware of any cross-space interactions. To avoid mixing positions of different users, we will need to deploy one proxy contract in EVM for each cross-space user from Move. This may incur additional gas cost, but we can make this cost very small by reducing the size of the proxy contract using delegate calls.

7 RELATED WORK

Cross-chain Interoperability: The emergence of numerous blockchains has highlighted the critical need for cross-chain interoperability techniques. Belchior et al. provide a comprehensive survey of existing solutions in this domain [7]. Atomic swap techniques, which utilize hashed time locks, facilitate the exchange of digital assets between different blockchains, ensuring transaction atomicity [6, 24, 39, 47]. Additionally, lock-and-mint bridges offer another method for transferring assets, where a bridge locks assets in a smart contract on one blockchain and then mints equivalent tokens on another [28, 30]. These bridges can be either centrally managed or fully decentralized, with the latter often incurring high gas costs due to the need to verify blockchain consensus protocols. Innovative solutions such as ZKBridge [45] use zero-knowledge proofs to batch verifications and reduce these costs.

While cross-chain interoperability and its security are a key concern in the blockchain space [4, 12, 25, 32, 35, 42, 46], HEMVM addresses a distinct challenge. Its aim is not to facilitate interaction

between different blockchains, but rather to create a single blockchain that hosts multiple virtual machines. Traditional cross-chain methods primarily focus on asset transfer across chains [40]. However, they fall short in solving user experience fragmentation due to the introduction of new virtual machines. For example, a cross-chain bridge connecting EVM and Move blockchains would still require users to have separate wallets for each which may create vulnerabilities [4], a complexity that HEMVM seeks to eliminate.

Blockchain with Subchains: Polkadot [44], Cosmos [26], and Avalanche [27] have proposed creating a hierarchical multi-blockchain ecosystem consisting of a central hub blockchain and multiple subchains. This structure allows the hub to verify subchain states and facilitate their interaction. Although each subchain could theoretically support different VMs, this hierarchical approach has inherent limitations. For instance, even if the hub enables cross-chain transactions across subchains, these transactions are less secure and slower compared to HEMVM's cross-space transactions, as they depend on the integrity of all involved subchains and incur the latency of multiple blockchains.

8 CONCLUSION AND FUTURE DIRECTIONS

The emergence of diverse virtual machine designs has led to an increasingly fragmented blockchain ecosystem. This paper introduces a novel heterogenous blockchain framework designed to counteract this fragmentation by integrating multiple virtual machines within a single blockchain. This integration is pivotal, as it allows users and developers to engage with the blockchain using their preferred interfaces and virtual machines, bridging gaps between different virtual machine technologies. Our comprehensive evaluation not only underscores the efficiency and practicality of our framework but also demonstrates the expressiveness of our cross-space operation design in enabling the implementation of sophisticated functionalities found in popular smart contracts. Our proposed heterogeneous blockchain framework streamlines user interaction and lays the groundwork for continued innovation in blockchain technology.

Although the blockchain ecosystem remains fragmented, and this work only begins to explore the space of cross-VM interoperability, we hope that it also motivates future research at the intersection of blockchain interoperability and programming language design. In particular, we identify two promising directions for further exploration.

First, this work could be extended by developing cross-VM debugging and developer tooling to streamline the development of multi-VM DeFi protocols. Given that these protocols span heterogeneous execution environments, developers currently lack unified tools for inspecting, tracing, or debugging interactions that cross VM boundaries. A practical extension would involve building language-agnostic debuggers, trace visualizers, or state diffing tools that can operate across VMs while abstracting away low-level incompatibilities. Such tools would further lower the barrier to multi-VM development and reduce the likelihood of subtle cross-context bugs. Next, one could envision generalizing the current approach to support interoperability across more than two virtual machines. While this presents new challenges - including increased complexity and the need to balance performance, security, and expressiveness - it also holds the potential to unify currently fragmented virtual machine ecosystems. Such unification could dramatically improve the developer experience by enabling seamless application development across multiple blockchains.

9 DATA AVAILABILITY

We plan to publicly release the source code of HEMVM and the experimental data as part of the paper's artifact. Upon acceptance of the paper, we will submit the artifact for evaluation. A preliminary anonymized version of the artifact is currently available at [3].

REFERENCES

[1] Openethereum - wiki. https://openethereum.github.io.

[2] Khamis Hamed Al-Yahyaee, Walid Mensi, Hee-Un Ko, Seong-Min Yoon, and Sang Hoon Kang. Why cryptocurrency markets are inefficient: The impact of liquidity and volatility. *The North American Journal of Economics and Finance*, 52:101168, 2020.

[3] Anonymized. https://github.com/OOPSLA2025Submission/HEMVM, 2024. Accessed: 2024-10-15.

[4] André Augusto, Rafael Belchior, Miguel Correia, André Vasconcelos, Luyao Zhang, and Thomas Hardjono. Sok: Security and privacy of blockchain interoperability. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 3840–3865. IEEE, 2024.

[5] André Augusto, Rafael Belchior, Jonas Pfannschmidt, André Vasconcelos, and Miguel Correia. Xchainwatcher: Monitoring and identifying attacks in cross-chain bridges, 2024.

[6] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Blitz: Secure Multi-Hop payments without Two-Phase commits. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4043–4060. USENIX Association, August 2021.

[7] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *ACM Computing Surveys (CSUR)*, 54(8):1–41, 2021.

[8] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezerand Tim Zakian, and Runtian Zhou. Move: A language with programmable resources. https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf, 2020. Accessed: 2024-10-03.

[9] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, 1, 2019.

[10] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In Nick Benton and Andrew Kennedy, editors, *First International Workshop on Multi-Language Infrastructure and Interoperability, BABEL 2001, Satellite Event of PLI 2001, Firenze, Italy, September 8, 2001*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 36–52. Elsevier, 2001.

[11] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform., 2013.

[12] E. Chan, M. Chrobak, and M. Lesani. Cross-chain swaps with preferences. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 261–275, Los Alamitos, CA, USA, jul 2023. IEEE Computer Society.

[13] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143âĂŞ155, March 1966.

[14] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover, 2022.

[15] Kathleen Fisher, Riccardo Pucella, and John H. Reppy. A framework for interoperability. In Nick Benton and Andrew Kennedy, editors, *First International Workshop on Multi-Language Infrastructure and Interoperability, BABEL 2001, Satellite Event of PLI 2001, Firenze, Italy, September 8, 2001*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 3–19. Elsevier, 2001.

[16] Aptos Foundation. Aptos-core. https://github.com/aptos-labs/aptos-core. Accessed: 2024-10-07.

[17] Aptos Foundation. The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf, 2022. Accessed: 2024-10-07.

[18] Ethereum Foundation. Solidity compiler. https://docs.soliditylang.org/en/latest/installing-solidity.html, 2024. Accessed: 2024-10-07.

[19] Ethereum Foundation. Solidity programming language. https://docs.soliditylang.org/en/v0.8.14/, 2024. Accessed: 2024-10-07.

[20] Ethereum Foundation. Vyper compiler. https://docs.vyperlang.org/en/stable/installing-vyper.html, 2024. Accessed: 2024-10-07.

[21] Ethereum Foundation. Vyper programming language. https://docs.vyperlang.org/en/stable/, 2024. Accessed: 2024-10-07.

[22] Sui Foundation. The sui smart contracts platform. https://docs.sui.io/paper/sui.pdf. Accessed: 2024-10-07.

[23] Uniswap Foundation. Uniswap v2 core. https://uniswap.org/whitepaper.pdf, 2021. Accessed: 2024-10-07.

[24] Maurice Herlihy. Atomic cross-chain swaps. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 245–254. ACM, 2018.

[25] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. Cross-chain deals and adversarial commerce. *VLDB J.*, 31(6):1291–1309, 2022.

[26] Jae Kwon and Ethan Buchman. Cosmos whitepaper. *A Netw. Distrib. Ledgers*, 27, 2019.

[27] Ava Labs. https://assets-global.website-files.com/5d80307810123f5ffbb34d6e/6008d7bbf8b10d1eb01e7e16_Avalanche%20Platform%20Whitepaper.pdf, 2020. Accessed: 2024-10-07.
[28] Chainlink Labs. https://docs.chain.link/ccip, 2024. Accessed: 2024-10-07.
[29] Compound Labs. https://compound.finance/documents/Compound.Whitepaper.pdf, 2019. Accessed: 2024-10-07.
[30] Rongjian Lan, Ganesha Upadhyaya, Stephen Tse, and Mahdi Zamani. Horizon: A gas-efficient, trustless bridge for cross-chain transactions. *arXiv preprint arXiv:2101.06000*, 2021.
[31] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. *A Decentralized Blockchain with High Throughput and Fast Confirmation*. USENIX Association, USA, 2020.
[32] Huaixi Lu, Akshay Jajoo, and Kedar S. Namjoshi. Atomicity and abstraction for cross-blockchain interactions, 2024.
[33] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
[34] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
[35] Zeinab Nehaï, François Bobot, Sara Tucci-Piergiovanni, Carole Delporte-Gallet, and Hugues Fauconnier. A tla+ formal proof of a cross-chain swap. In *Proceedings of the 23rd International Conference on Distributed Computing and Networking*, ICDCN '22, pages 148–159, New York, NY, USA, 2022. Association for Computing Machinery.
[36] PancakeSwap. https://pancakeswap.finance/, 2024. Accessed: 2024-10-07.
[37] Maciel M Queiroz, Renato Telles, and Silvia H Bonilla. Blockchain and supply chain management integration: a systematic review of the literature. *Supply chain management: An international journal*, 25(2):241–254, 2020.
[38] Alesja Serada, Tanja Sihvonen, and J Tuomas Harviainen. Cryptokitties and the new ludic economy: how blockchain introduces value, ownership, and scarcity in digital gaming. *Games and Culture*, 16(4):457–480, 2021.
[39] Stefan Thomas and Evan Schwartz. A protocol for interledger payments. *URL https://interledger. org/interledger. pdf*, 2015.
[40] Rob van Glabbeek, Vincent Gramoli, and Pierre Tholoniat. Cross-chain payment protocols with success guarantees. *Distributed Comput.*, 36(2):137–157, 2023.
[41] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. Efficiently detecting reentrancy vulnerabilities in complex smart contracts, 2024.
[42] Qiuyang Wei, Xufeng Zhao, Xue-Yang Zhu, and Wenhui Zhang. Formal analysis of ibc protocol. In *2023 IEEE 31st International Conference on Network Protocols (ICNP)*, pages 1–11, 2023.
[43] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.
[44] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White paper*, 21(2327):4662, 2016.
[45] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3003–3017, 2022.
[46] Yingjie Xue and Maurice Herlihy. Hedging against sore loser attacks in cross-chain transactions. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 155–164, New York, NY, USA, 2021. Association for Computing Machinery.
[47] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William Knottenbelt. Xclaim: Trustless, interoperable, cryptocurrency-backed assets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 193–210. IEEE, 2019.