HarDTAPE: Hardware Dedicated Trusted transAction Pre-Executor

Sirui He[1], Zhibo Sun[2], Yuan Chen[1], Yajin Zhou[2,3], and Cong Wang[1]
[1]*City University of Hong Kong*, Hong Kong, China
[2]*Zhejiang University*, Hangzhou, China; [3]*BlockSec*, Hangzhou, China
Email: sol.he@my.cityu.edu.hk, {22221030, yajin_zhou}@zju.edu.cn, {yche25, congwang}@cityu.edu.hk

***Abstract*—As the complexity of Blockchain-based Smart Contracts grows, users rely on remote pre-execution services to simulate the behavior of transactions before emitting them on-chain. However, users may be concerned that a dishonest service provider (SP) may leak their execution trace to frontrun them or respond with fake results to mislead them. Benign SPs tried to run their services in trusted execution environments (TEEs) to prove their honesty. However, existing service software and TEEs have attack surfaces for side channel or control flow attacks. Meanwhile, the problem of users' intention leakage through world state access patterns remains unsolved.**

This paper proposes HarDTAPE, a hardware-dedicated trusted transaction pre-executor, to protect the confidentiality and integrity of pre-executed transactions against dishonest SPs. Here, "dedicated" has two meanings: the pre-execution service is implemented as dedicated hardware to guarantee a valid control flow, and each set of hardware is isolated and dedicated to at most one user within each session to eliminate side-channel attacks on shared hardware (e.g., cache evict-and-reload). For access pattern confidentiality, we use Path ORAM to store the world state reassembled into fixed-size pages. We also use pagewise code prefetching to prevent the query type from being recognized. We implemented HarDTAPE on a CPU + FPGA SoC as a proof-of-concept. Using transactions from real-world Ethereum Mainnet blocks as test cases, we show that HarDTAPE has an acceptable run time overhead and throughput.

***Index Terms*—Blockchains, Trusted computing, Accelerator architectures**

I. Introduction

A significant feature of Blockchain 2.0 is allowing arbitrary user-deployed Smart Contracts to run on the network. To take Ethereum as an example, developers can write their decentralized applications (DAPPs) as contracts in high-level languages such as Solidity, compile them to Ethereum virtual machine (EVM) bytecodes, and deploy them on-chain. Users can then easily perform various operations by emitting transactions to interact with these contracts [4]. However, the growing complexity of Smart Contracts causes users difficulties in quantifying the risks in the transaction process. Scam contracts, including Phishing [19], Ponzi scheme [14], and Honeypot [41], may defraud users for profit. The financial losses would be more severe for high-frequency trading (HFT) strategy users if they fail to recognize these malicious behaviors. This leads to the need for transaction pre-execution: to simulate and verify the results of transaction sequences (hereafter referred to as *bundles*) before emitting them on-chain [11].

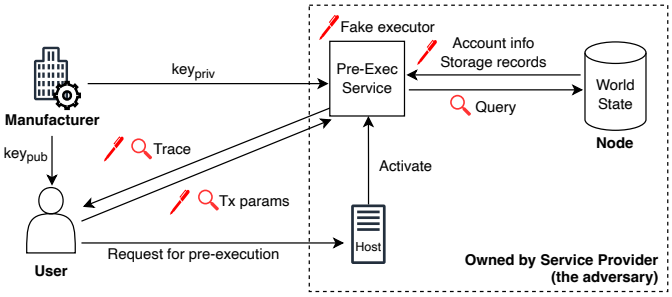

Fig. 1: The pre-execution use case, and the data flow which may be threatened. Magnifiers stand for leakage, and pens stand for manipulation.

Although users may wish to pre-execute their transaction bundles locally, it is technically challenging for non-professional users to deploy an EVM or to maintain the world state (of over 1.1 TB [5]) by themselves. Thus, users currently rely on third-party pre-execution services, such as the Transaction Simulator of Tenderly and BloXroute [3], [11]. However, as with all third-party services, users may be concerned that a malicious pre-execution service provider (SP) can *frontrun* (i.e., perform the profitable transaction on-chain earlier) or *mislead* (i.e., conceal malicious behaviors from the simulation report) them, because the entire simulation process is under the SP's control. Thus, to attract users with higher security demands, benign SPs hope to build trusted pre-execution services to prove that they (or any other malicious entities that attempt to attack their services) cannot perform the above actions.

A. Motivation

A trusted pre-execution service must guarantee the confidentiality and integrity of all data and control flows, even against a powerful adversary, such as a malicious service provider (SP) who controls the entire pre-execution system except for the on-chip circuitry within chip packages. The primary approach is to execute transaction processors, such as Ethereum Virtual Machines (EVMs), inside trusted execution environments (TEEs). These TEEs provide bidirectional isolation, protecting the EVMs from the hosting system and vice versa. However, we argue that this solution is merely a foundation and falls short in security-sensitive financial scenarios.

Although conventional TEEs protect the confidentiality of on-chip states, they often fail to address the privacy of interaction traces with the untrusted external environment, in our case the query and access patterns of contract bytecode and storage, which could inadvertently expose users' intentions for future transactions. Furthermore, both existing TEEs and their associated applications remain vulnerable to exploitable weaknesses, introducing additional risks to this strategy. These concerns are examined in more details below.

Access pattern leakage. Pre-executed transactions may randomly access any Ethereum account and its storage records. These access patterns can reveal the control flow of the transactions, which represents the user's intentions. The adversary may thus gain a Maximal Extractable Value (MEV) opportunity to frontrun the user and get profits [21], [40]. For example, when a user calls the contract of an ERC-20 token during pre-execution, the adversary can learn the target token (and the amount or price) they wish to trade, and submit transactions earlier than the user at a better price. With this exploitation, all other protections become meaningless.

The pre-execution service must support access to arbitrary part of the world state, which is too large to be kept inside the secure world (as adopted by TSC-VEE [26], a TEE designed for a single Confidential Smart Contract). Because the addresses and keys to be accessed are determined only at runtime, we cannot let the user prepare the data required by their transactions as part of the input (as in traditional TEE use cases [35]). Also, simply encrypting the queries is not enough, because when new blocks are broadcasted to the entire network in plaintext, the adversary can map the ciphertext keys to their plaintext using their accumulated frequency of co-occurrence.

Oblivious random access machines (ORAMs), first proposed by Goldreich et al., introduced a query strategy such that the access patterns look random and equivalent from the adversary's perspective [23]. ORAM is a generic solution for access pattern protection tasks. However, each ORAM access brings an $\mathrm{O}(\log n)$ bandwidth overhead, where n is the size of the key space. This overhead has not yet been evaluated for the Ethereum world state which, unlike traditional memory structures, has a sparse key distribution but consistently growing size. Additionally, a transaction may access two types of world state data: byte arrays (i.e., the contract bytecode) and 32-byte integers (i.e., balance and storage records). Improper use of ORAMs may cause the two response types distinguishable, leaving a side channel that helps identify the executing contract and breaks obliviousness.

TEE vulnerabilities. Some widely adopted TEE products are vulnerable to attacks that exploit *hardware sharing* between different security domains, including Spectre-like cache side channel exploitations [18] and interrupt-based data injections [36]. These attacks are carried out in the untrusted world, but manipulate the state of the registers, caches, or branch predictors shared with the enclave. Heterogeneous TEEs (e.g., [42]) are also affected since their CPU parts are vulnerable.

Software-inherent vulnerabilities. Newer TEEs are more careful when designing isolation mechanisms. For example, the Keystone enclave eliminates hardware sharing with full state flushing and L2-cache partitioning [28]. However, attacks against the protected software itself are still fatal regardless of where it runs. Biondo et al. provided an example of exploiting software vulnerabilities to reuse the TEE runtime libraries to load fake environments into TEEs [15]. In fact, if the goal is only to compromise the protected software itself, the TEE runtime is not essential in these attacks because most software has enough "gadgets" to construct arbitrary logic, as pointed out by existing works [25], [37]. The adversary can apply these attacks to construct fake execution traces with valid signatures.

Summary. As the goal of pre-execution is to mitigate users' risk, we can expect the users to prioritize security over performance. Therefore, we want to provide users with a more security-focused option at an acceptable performance cost. We aim to answer:

- How to protect the control flow (access pattern) confidentiality, and how much is the performance overhead?
- How to build better isolation between security domains (e.g., multiple enclaves, the TEE runtime, and the untrusted world) to reduce the attack surface of hardware sharing vulnerabilities?
- How to eliminate the inherent vulnerabilities in the (software) pre-execution service or its runtime dependencies?

Remark: Although fully homomorphic encryption (FHE) is another commonly used technique to provide data confidentiality, its performance overhead is too high (about 10000 times slower than plaintext [38]) to meet both the response time and throughput requirements. Moreover, its threat model assumes that the confidentiality of the control flow need not be protected, which does not suit our purpose. Therefore, we will not discuss FHE in this paper.

B. Our Solution

As our answer to these questions, we propose HarDTAPE, a trusted transaction pre-executor with dedicated hardware. It has the following features.

Path ORAM for paged world state. HarDTAPE integrates Path ORAMs [39] to the TEE as the backbone of world state access pattern protection. In the Path ORAM, the contract bytecodes are partitioned and the storage records are grouped into pages of the same size. Queries of both types are mixed together with the same response format and consistent time interval to mitigate query type or bytecode length leakage. In our experiment settings, the average execution time overhead of each real world transaction is about 80 ms, which is acceptable for the user (see details in Section VI-C). Superior to TSC-VEE, this approach can support Smart Contracts with large storage or call other contracts.

Dedicated hardware. Unlike conventional TEEs that share hardware between users for throughput, HarDTAPE exclusively assigns an isolated set of hardware (EVM, tracer, and local memory) for each transaction bundle. No context switches are performed during the entire lifecycle of a bundle. Although straightforward, this approach eliminates shared hardware side channels or controlled side effects from the root cause.

Hardware EVM. HarDTAPE runs hardware transaction pre-executors and tracers that directly implement the EVM instruction set architecture (ISA). Compared to software EVM implementations (such as the commonly used Geth [7]), these so-called Hardware EVMs (HEVMs) have fixed functionalities and are immune to control flow attacks. The EVM ISA acts as an abstraction layer that omits the details of the underlying system structure and limits the actions the running transactions (even if malicious) can take.

Contributions. The major contributions of this paper are:

- We pointed out the need for secure transaction pre-execution service for Blockchain (Ethereum), and formalized its essential security features.
- We adapted Path ORAM to the complex data structures of Ethereum to achieve confidential access to the entire world state. We assume the adversary knows the plaintext of on-chain transactions and is active (i.e., may execute transactions to learn or change the distribution of queries).
- We proposed HarDTAPE as a more secure transaction pre-executor to resist side channels or control flow attacks. We designed a proof-of-concept prototype and evaluated its performance in a real-world scenario.

II. Background

A. Ethereum

Ethereum is a distributed ledger widely used for decentralized finance (DeFi). The Ethereum Blockchain maintains a publicly visible world state stored by every node in the network, providing immutability and verifiability but sacrificing confidentiality. As a typical 2nd-gen blockchain, Ethereum allows the deployment of customized smart contracts to be invoked via transactions [4].

World state. Ethereum's world state is a mapping between accounts' addresses and their current states. Currently, it has a size of over 1.1 TB [5]. The account state consists of four fields: balance, nonce, storage (hash), and code (hash). The address is a 20-byte integer. The storage of an account is a key-value-pair (K-V) database, in which both the key and the value of a *storage record* are 32-byte integers. For contract accounts, the code is a byte array that stores the contract's instructions. All these data are stored and authenticated by Merkle Patricia Trees (MPTrees) [32]. Merkle proofs can be generated for any chosen piece of data from any user-specified block, and verified with the block hash.

Transactions. Transactions (Txs) are the atomic units of world state updates. By performing a transaction, a sender transfers balance to a receiver account, and may call a method of the receiver contract. Reverting a transaction discards all modifications to the world state except the gas fee consumed.

Recently, roll-up transactions have been proposed to reduce the congestion of the Ethereum network [12]. While the computation workloads are moved off-chain to regular applications, the results are submitted to the Blockchain by roll-up transactions only for verifiability. These transactions submits thousands of storage record updates with very few other operations.

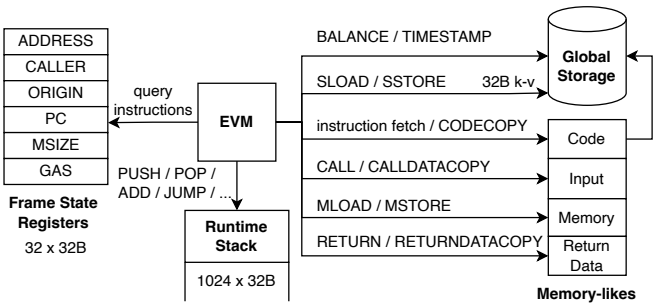

Fig. 2: The EVM programming model within one execution frame. The instructions marked on the arrows are for accessing the pointed structures [8].

On-chain transactions are batched as blocks, which are processed and chained serially. Currently, a block is generated about every 12 seconds, containing about 200 transactions [6]. So, the throughput of the Ethereum Blockchain is about 17 Tx/s. New blocks are broadcasted to and executed by every node in the network, including those run by the pre-execution services, so they end up with the same world state version.

The pre-execution service simulates bundles of transactions. Unlike blocks, world state updates made by simulation bundles are temporary, so multiple bundles can run in parallel.

Programming model. Smart contracts are executed in EVMs. EVM is a 256-bit stack architecture (see Figure 2). Instruction operands are fetched from, and results are written back to a runtime stack with 1024 slots. Also, EVM provides a set of arbitrary length, byte-addressed, unaligned accesses allowed, volatile memories, which we refer to as the *memory-likes*. This includes Code for the contract bytecode, Input for call parameters, Memory for runtime temporary variables, and ReturnData for data returned to the caller. EVM defines `ARITHMETIC` and `JUMP` instructions to build the basic control flow, together with frame state query, `STACK`, and `MEMORY` instructions to access the corresponding data structures. EVM also provides `STORAGE` instructions to access the K-V database, and `CALL-RETURN` instructions to call, deploy, or return from other contracts and transfer values. Details of these instructions and data structures can be found at [8].

The procedure between a pair of `CALL` and `RETURN` is called an *execution frame*, and we name the logical structure of these execution frames as the *call stack*. Each execution frame has a context including runtime stack, memory-likes, frame state (e.g., contract address and remaining gas), and the current version of the world state (e.g., balance, nonce, and storage records). At the end of each execution frame, world state modifications are discarded or committed (merged to its caller) depending on whether the transaction is reverted. The remaining parts are volatile, i.e., they are set (or reset to empty) on entry, dumped on call, reloaded on return, and cleared on exit of the execution frame.

Ethereum assigns a gas cost for each operation based on its computing resource consumption, resulting in an almost stable

gas usage over time [13]. This does not only profit the miners who can earn gas fees from validating transactions, but also prevents potential DoS attacks against the network.

B. Trusted Execution Environments

Trusted execution environments (TEEs) are isolated regions in a computing system for processing sensitive code and data. They are designed against an adversary who wishes to break the confidentiality or the integrity of protected applications. Usually, the adversary is assumed to be powerful enough to control all the software and hardware of the hosting system, except the circuits inside the chip package where the TEE is running (this implies that the chip manufacturer is trusted). Thus, only the *on-chip* components can be used as the trusted computing base (TCB), while the *off-chip* components should be accessed with caution. There is usually a privileged module called a Hypervisor or a Monitor that creates, destructs, and handles the IO of TEE instances.

CPU manufacturers have integrated hardware support for TEEs in their CPU products, such as ARM Trustzone, Intel SGX, AMD SEV, or RISC-V Keystone. Although claimed to be secure, published control flow attacks [15], [29] and side channel attacks [27], [34], [43] against these TEEs can achieve their goal without violating the TEEs' threat model. Other literature [17], [20] also summarized the attack surfaces of common shared-hardware TEEs, indicating they are vulnerable to attacks from the insecure world or adversary-controlled TEE instances. In conclusion, commonly applied shared-hardware TEEs are not secure enough for our use case.

C. ORAMs

The term "obliviousness" in ORAM refers to the feature of "a client concealing its access pattern to the (untrusted) remote storage" [39]. To achieve this, ORAMs hide each real query in a sequence of carefully chosen dummies. By shuffling the data *blocks* (i.e., the atomic piece of data) after each access, each possible sequence appears with an equal probability independent of the real query. Data *blocks* are stored in the untrusted memory after randomized encryption to look equivalent from the adversary's perspective.

Obliviousness can be achieved in different ways, and Path ORAM [39] is a simple one. In a Path ORAM (and other tree-based ORAMs), data blocks are organized as a tree with $\mathrm{O}(\log n)$ depth, and both query and eviction cover a root-to-leaf path on that tree. Path ORAM applies a client-server model. The server stores the tree, and the client maintains a stash of $\mathrm{O}(\log n)$ temporary blocks and a $\mathrm{O}(n)$ position map indicating the path of each block in the tree. The position map can be stored in *higher-level ORAMs* recursively if it is too big. Path ORAM incurs a $\mathrm{O}(\log n)$ bandwidth overhead when the block size is approximately $\mathrm{O}(\log^2 n)$. As a stateless design, a Path ORAM server can serve multiple clients concurrently.

III. HarDTAPE Use Case and Threat Model

A. Use Case

Before diving into the details of our design, we again take a closer look at the pre-execution use case depicted in Figure 1. We define the four involved parties in the secure transaction pre-execution scenario.

- **User**. A user of the pre-execution service, e.g., an HFT designer who wishes to test some transaction bundles. Users request for confidential, correct, and quick-responding pre-execution services. It takes about 600 ms for a user to obtain the details of a transaction online (e.g., using the JSON RPC API provided by quicknode.com [10]), so we hope that the pre-execute time for each transaction does not exceed this level.
- **Manufacturer**. The trusted creator of the computing devices used by the pre-execution service (i.e., HarDTAPE). The Manufacturer assigns cryptographic features and a secure source of randomness which we use to build the chain of trust.
- **Service Provider** (SP). The computing power holder, who purchases devices from the Manufacturer and processes the users' pre-execution requests. We assume an untrusted SP (the *adversary*) who wishes to disclose the user's behaviors or provide them with misleading results.
- **Node**. An Ethereum full node that provides fresh on-chain data, which is also under the SP's control.

When a user calls the pre-execution service, the SP looks for an idling pre-executor, activates it, and assigns it to the user. The user verifies the pre-executor with the Manufacturer's credentials and sends the parameters of the transactions to the assigned pre-executor. The pre-executor starts running upon receipt. During the process, the pre-executor may query for the balance of accounts, bytecodes of called contracts, or records in the K-V storage, from the Node if not found locally. The behaviors of the transactions are recorded by the tracer. After all transactions in the bundle have been simulated, the traces of each transaction, including the ReturnData, gas cost, balance transferred, and storage modifications, are sent back to the user. At last, the pre-executor is released and returns to the idle state.

B. Threat Model

The threat model of HarDTAPE aligns with existing TEE products such as SGX or Trustzone. Intuitively, we trust the on-chip components authenticated by the Manufacturer, and distrust all the off-chip components controlled by the SP.

The adversary's goal is to break the confidentiality or integrity of the user's data or control flow. They are assumed to have full control over (and physical access to) all the off-chip components. However, although they can launch pre-execution runs of any Ethereum contracts in parallel with the user, they cannot directly observe or modify any on-chip state. In particular, we consider an adversary who wishes to launch the following attacks: (A1) Provide the user with a fake pre-executor. (A2) Access or probe (e.g., via side channels) the on-chip state from the untrusted hosting system or other pre-executors on the same chip. (A3) Seize the control flow of the pre-executor, the running transaction, or the Hypervisor if exists. (A4) Leak or tamper with the swapped-out runtime data when the secure memory is insufficient. (A5) Monitor

the data swap pattern to learn the memory usage; this may expose the program counter trace and the bytecode size, which are distinguishable identities of Smart Contracts. (A6) Provide the user with fake on-chain data. (A7) Monitor the user's on-chain data query and access pattern. Defenses against physical or fault injection attacks are discussed by other works such as [33], and are orthogonal to our work.

IV. Design

Now we introduce HarDTAPE where the secure transaction pre-execution service runs on. For better isolation, it is designed as a coprocessor running separately from the host machine, and protects all the data flow to and from the chip package. It can be connected to the host system with PCIe, Ethernet, or other interface peripherals. We recommend Ethernet for its scalability, in which case the chip must be installed on a motherboard with network peripherals.

The chip package comprises multiple HEVMs, a CPU for the Hypervisor firmware, and a configuration-security unit (CSU), together with memory and DMA controllers. The order of steps for simulating user transaction bundles is labeled in Figure 3. (1) When the chip is powered on, the CSU verifies and boots the secure bootloader (SBL), which resets the HEVMs and boots the Hypervisor. The Hypervisor establishes a connection to the ORAM server run by the SP. (2) When a user tries to connect, the Hypervisor responds to the user's remote attestation request, generates keys for authentication and encryption, and establishes a secure channel for the user. (3) When receiving a user's transaction bundle, it queues until an HEVM is idle. The Hypervisor exclusively assigns the idling HEVM to the user and activates it. (4) The HEVM executes the transactions in the bundle. Most of the pre-execution functionalities can be completed by the HEVM alone. (5) When the HEVM needs to swap data with the untrusted external memory, query for on-chain data, or finish execution, it emits an exception to the Hypervisor. (6) To handle the exceptions, the Hypervisor may interact with the off-chip components with protected *messages*. The data field of the messages is automatically authenticated and encrypted if needed. (7) When handling contract calls, the call stack must allocate space for a new execution frame. Pages from lower execution layers are dumped to the untrusted memory if the on-chip call stack has insufficient free space. (8) On-chain data, including contract bytecodes, are queried from the ORAM server. (9) The trace of the transactions is stored temporarily by the on-chip tracer until the bundle finishes execution. Then, the Hypervisor sends the trace to the user via the secure channel. (10) The HEVM is reset to the idle state and all its on-chip memories are cleared. World state modifications made by the pre-executed transactions are not written into any persistent storage such as the ORAM. (11) When new blocks are created on-chain, HarDTAPE synchronizes the world state after executing these blocks to the ORAM for later access. The synchronized blocks and their access patterns need not be confidential because they are public. However, the integrity of the results must be guaranteed.

	code	input	memory	return
$<$1k	9.5%	95.0%	92.7%	100.0%
1-4k	25.3%	4.0%	5.7%	0.0%
4-12k	39.6%	0.2%	0.6%	0.0%
12-64k	25.6%	0.0%	0.0%	0.0%
$>$64k	0.0%	0.1%	0.1%	0.0%

(a) Memory-like size by type in bytes per frame.

	keys
$\leqslant 4$	79.9%
5-16	19.0%
17-64	0.01%
> 64	0.00%

(b) Storage keys per frame.

	depth
1	40.8%
2-5	52.6%
6-10	6.3%
> 10	0.2%

(c) Call depth per Tx.

TABLE I: The distribution of memory size per frame, storage records per frame, and call depth per transaction from recent real-world blocks (Ethereum Mainnet #19145194 - #19145293).

In the remaining part of this section, we will discuss the protection schemes designed for each step.

A. Booting and Secure Channel Initiation

First, the user must ensure their transactions are processed by a trusted device. Most existing TEE works have provided their implementations of secure boot and remote attestation protocols [28], [44]. The generic idea is to use a hardware-visible secret (e.g., a physically unclonable function (PUF)) as the root of trust to seed or decrypt a pair of asymmetric device keys, which are used to sign the booted image (software binaries, FPGA bitstreams, etc.). The PUF and the device keys are generated by the trusted Manufacturer. In our proof-of-concept implementation, we followed the design of [44]. After the user receives a valid attestation report from the Hypervisor, the user and the Hypervisor each generate a pair of elliptic curve digital signature algorithm (ECDSA) keys, and cooperatively create a session AES key using Diffie-Hellman key exchange (DHKE). This establishes the secure channel between them, which is later used to transmit user inputs and pre-execution results.

B. Transaction Simulation

Second, transactions in the user's input bundle are sequentially simulated in the HEVM. The HEVM should be functionally equivalent to the interpreter module of Geth, a popular implementation of Ethereum Node. The only difference is that the world state modifications are temporary, and the traces are visible only to the owner of the pre-execution bundle.

Contract instruction interpretation. The HEVM directly executes EVM bytecodes. It should be able to decode EVM instructions with variable length, perform 256-bit arithmetic, and maintain the data structures of the current execution frame. Because the HEVMs are not shared between users, cache prefetching and speculative branching can be enabled securely without introducing cache-probing side channels.

Lu et al. have proposed the Smart Contract Unit (SCU), a novel HEVM design with high performance [30]. Theoretically, by applying our security mechanisms to manage its Input

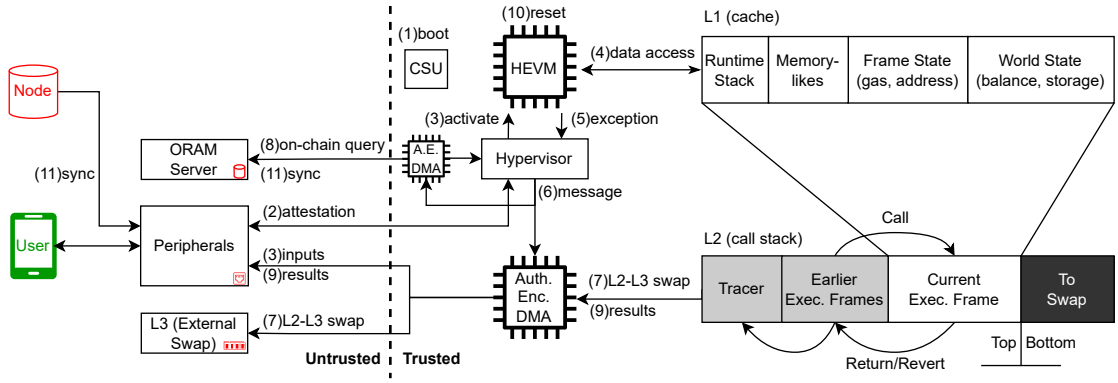

Fig. 3: The workflow of HarDTAPE. The number indices represent the order in a pre-execution lifecycle.

Buffer (corresponding to our layer 2 memory in Figure 3), SCU cores can be ported as a good HEVM implementation. Unfortunately, SCU is not open-sourced, so we designed a four-stage pipelined HEVM in our proof-of-concept system as the performance baseline. To support the rapid upgrades of the EVM ISA, this simple design can be replaced by any future hardware EVM accelerators. However, for security reasons, we require the replacing HEVM not to add non-standard instructions that violate the isolation, e.g., access the memory space of other HEVM cores.

Gas maintenance. The gas cost of the instructions is automatically accumulated simultaneously as the instructions are interpreted. Some instructions have additional dynamic gas costs according to the increment of Memory size or whether an address or a storage key has been accessed before. The HEVM can obtain this information by checking the `MSIZE` environment register or recording whether a call stack miss is triggered. The SP can prevent DoS attacks (occupying an HEVM too long) by charging gas fees or setting low gas limits because the gas cost approximately represents the computing resource consumption.

Data organization. As described in Section II-A, the execution of a transaction consists of multiple execution frames, each maintains a runtime stack of 1024 slots, permanent storage of up to 2^{256} slots, and runtime memory-likes (i.e., Code, Input, Memory, and ReturnData) of theoretically infinite size. It is impossible to put all these data structures on the limited trusted hardware, so we designed a 3-layer memory structure. We highlight this structure as the most significant difference from existing works which only support one contract, and our security features are mainly implemented here.

Layer 1, or the cache, stores the recently accessed data of the current execution frame. Each HEVM has its isolated layer 1 memory, so users cannot see transactions pre-executed by each other. It is in the same clock domain as the HEVM circuit and is kept small to achieve a higher clock frequency. The layer 1 memory is partitioned for the runtime stack, each of the memory-likes, the frame state, and the world state respectively. Because almost every EVM instruction fetches operands from and writes results to the runtime stack, we store the entire runtime stack (up to 32 KB) in the cache for efficiency. The frame state consists of 32 32-byte slots, corresponding to the EVM instructions with opcode 0x30 - 0x4A (e.g., `ADDRESS`, `CODESIZE`, refer to the official documentation [8] for details).

The sizes of other partitions are chosen based on the statistics of recent transactions. We evaluated transactions from 100 recent blocks (namely block #19145194 - #19145293 from Ethereum Mainnet) as the *evaluation set*, and the results are listed in Table I. Among them, in over 99% of the execution frames, the Code size is less than 64 KB, the Memory and Input size are less than 4 KB, the ReturnData size is less than 1 KB, and no more than 64 storage records are accessed. According to these insights, we assign 64 KB for the Code cache, 4 KB for other memory-like caches, and set the page size to 1 KB. We also assign 4 KB for the world state cache, which is suitable for 64 account balances or storage records.

Layer 2, or the on-chip call stack, manages the execution frames. The layer 2 memory is also isolated for each HEVM, but has a larger size and lower frequency. The layer 1 caches only swap data with the corresponding fields in the current (topmost) layer 2 execution frame. The size of an execution frame may grow as it accesses higher Memory addresses.

The `CALL-RETURN` instructions are implemented by managing the top of the call stack. Specifically, when the HEVM performs a `CALL` instruction, all the contents of layer 1 are evicted to layer 2, and a new execution frame is created for the callee. Its frame state and Input fields are set according to the call parameters, and its world state fields are initialized with a copy of the caller's version containing all the modifications made before this call. When the HEVM performs a `RETURN` or `STOP` instruction, the callee's world state is merged into that of the caller, overwriting existing values. When the HEVM performs a `REVERT` instruction, the callee's world state is discarded. In both cases, the callee execution frame is popped from the call stack, and the ReturnData is exported as part of the execution trace. The tracer is implemented as a virtual frame added below all execution frames, to preserve the execution results across transactions in a bundle.

Layer 3, or the untrusted memory, contains the swapped-out contents from layer 2, in case the on-chip memory is

insufficient. It can be implemented either as a memory region allocated from the hosting system memory or use extra DRAMs installed on the HarDTAPE motherboard. The base offsets of the execution frames are kept on-chip to prevent leakage or manipulation of the call stack structure. However, we must also consider that the swapping behavior may leak information about the control flow to the adversary. The generic solution is to implement the layer 3 memory as an ORAM, which however might be too expensive. Fortunately, if the current execution frame is small enough to be stored entirely on-chip, cache misses from layer 1 can always be handled by layer 2 and are therefore secure. This assumption is practical according to our measurements in Table I. Thus, we only need to protect the creation, expansion, and deletion pattern of the execution frames in layer 2.

Consider the following strategy: We treat the layer 2 address space as a ring, and conduct size expansion and data swapping in the granularity of 1 KB pages. When expanding the current execution frame, we dump the current bottom pages from layer 2 to make room for the new pages. When returning to a lower execution frame, we reload all its pages to maintain the assumption that the current execution frame is entirely on-chip. This makes the page-swapping order only related to the total size of the call stack. At this point, the adversary can still infer the page number of each execution frame by counting consecutive reloads. Here we propose a simple solution: we randomly pre-evict and pre-load more pages than required. The random numbers are sampled from a secure source of randomness proposed by the Manufacturer. This always allows the HEVM to continue execution, while the swap sizes observed by the adversary are added with random noises following a distribution unrelated to the actual size. Although we do not achieve strict obliviousness, the adversary can only obtain the noisy sizes and depths of the call stacks, which are too rough to identify the pre-executed contract.

We recommend a layer 2 memory capacity larger than four execution frames, such that there are always enough pages to pre-evict/load to generate large noises. Observing that each execution frame normally occupies less than 256 KB of memory, our proof-of-concept system provides 1 MB layer 2 memory for each HEVM. We keep in mind that the adversary may deploy malicious contracts that intentionally exceeds this limit. When the size of an execution frame reaches half of the layer 2 memory size, the bundle will be regarded as an attack and stopped with a Memory Overflow Error.

Exceptions. Interactions between the HEVM and the untrusted world, including layer 3 data swaps and non-local world state queries, are handled by the Hypervisor. The HEVM emits exceptions to the Hypervisor CPU to call these functionalities, and writes the exception metadata (e.g., types, memory offsets, storage keys) into local registers accessible only to the Hypervisor. The HEVM stalls until it receives the response from the Hypervisor, and does not switch context to process transactions from other users. Although sacrificing throughput, this prevents the potential shared hardware side channels (described in Section I) from the root cause.

C. Messages

Third, bidirectional data flow that passes through the trusted-untrusted border must be protected. We let the Hypervisor handle both EVM exceptions and external inputs (from the user, the ORAM server, or the Node). The hosting system cannot access the on-chip memory space directly. Thus, to transmit data to the HEVMs, it must store the message in an Hypervisor-accessible buffer (e.g., send via Ethernet), and then inform the Hypervisor with a non-preemptive interrupt. The Hypervisor loads the message header, checks the validity of its type, length, and target offset, and then invokes the Authenticated Encryption DMA hardware (the A.E.DMAs in Figure 3) to handle the copy.

The confidentiality and integrity of user interactions and layer 3 memory contents are protected by the AES-GCM cipher using the session key generated during remote attestation. The contents of the ORAM are protected with another AES ORAM key, which we will introduce in the next section. When synchronizing new blocks from the Node, although confidentiality is not required (because the blocks are already public), we require Merkle proofs to authenticate the fetched world state data.

Remark: The Merkle proofs are only checked during block synchronization. Once written into the ORAM, the data integrity is protected by AES-GCM, so we do not need Merkle proofs during pre-execution. This reduces both the performance overhead and the risk of leaking access patterns.

D. World State Queries

Finally, HarDTAPE must keep the access pattern of the world state queries confidential. This time, we cannot adopt a strategy akin to layer 3 memory swapping. While page swap offsets only reveal the call stack size, the queried addresses and keys directly expose the user's behavior.

ORAM why and how. We choose Path ORAM as the backbone of our protection scheme for its proven obliviousness and simplicity [39]. Although Snoopy [22] provides better scalability and throughput by using hash table ORAMs and batched queries, the number of queries in the pre-execution scenario is too small to create a query batch quickly, and the latency of each query is too long (over 300 ms). Other approaches such as PANCAKE [24] or Waffle [31] achieve *sub*-obliviousness (i.e., smoothing the distribution of queries) with lower performance overhead on a Zipfian-distributed workload. However, they are not designed against an *active* adversary who can send requests to interfere with the distribution, which is in our threat model.

The SP should run a Path ORAM server instead of an ordinary Ethereum Node to store the entire 1.1 TB world state. Extra spaces for dummy blocks and metadata may be required. Meanwhile, the Hypervisor is integrated with a Path ORAM client, whose local stash and highest-level position map are kept on-chip. In the following, to distinguish from Ethereum blocks, we use italicized *block* to denote the basic unit of ORAM data.

Mixing query types. There are two types of queries to be protected: the K-V style query for 32-byte integers (e.g.,

account balance, code length, or storage records), and the Code query for the contract bytecode with variable size of tens of KBs (as shown in Table I). This leads to three problems: (1) The size of each storage record is too small to reach the $\mathrm{O}(\log^2 n)$ size lower bound, as demanded by Path ORAM to achieve the $\mathrm{O}(\log n)$ overhead [39]. If we set the *block* size to b bytes, the 1.1 TB full sync size [5] should be partitioned into $n = 1.1 \times 10^{12}/b$ *blocks*. If we have $b = 32$, the *blocks* size in bits $8b = 256$ is far less than $\lceil \log_2 n \rceil^2 \approx 1225$. (2) If the K-Vs and Codes are stored separately, the difference in response size may leak the type of queries. The adversary may record the number and time interval of K-V queries between two Code queries (i.e. within an execution frame). This pattern can possibly be used to identify the running contract. (3) The Code of an execution frame cannot be retrieved all at once, as this would reveal a burst query pattern, also distinguishing Code queries from (sporadic) storage record accesses.

To solve problems (1) and (2), we divide the Codes into 1 KB *blocks*, and group the values of 32 storage records with consecutive keys also into 1 KB *blocks*. We use this grouping strategy because contracts compiled by Solidity assign variables and array elements to consecutive keys [9]. This 1 KB *block* size eliminates the difference between the two types of responses and also satisfies the $\mathrm{O}(\log^2 n)$-bit *block* size requirement (in this case $n \approx 10^9$). The local stash size should be $\mathrm{O}(\log n) \approx 30$ pages, which fits in an on-chip memory of about 1 MB.

To solve problem (3), we use pagewise Code prefetching to spread the Code queries among storage queries. After each ORAM access, an interval timer is set to a random value of approximately half of the global average gap between queries. When this timer goes to zero, we prefetch the next Code page. Intuitively, we insert a prefetch query in the middle of every two original queries. Therefore, the time intervals observed by the adversary become approximately consistent.

ORAM key protection. As well known, the ORAM should perform a randomized re-encryption after each access [39]. We keep the encryption key on-chip by the Hypervisor. To reduce storage cost, the SP can run one ORAM server for multiple HarDTAPE instances, leveraging the statelessness of Path ORAM. Because the ORAM clients are managed by the trusted Hypervisor, they can trust each other and share the same ORAM key. The key is chosen randomly by the first HarDTAPE Hypervisor when deployed. When adding a new HarDTAPE device, it queries the ORAM key from a previous device through a DHKE secure channel.

V. Security Analysis

In this section, we theoretically show how our design can defend against each possible attack described in Section III-B.

Environment authenticity (A1). The remote attestation protocol cryptographically proves the authenticity of the device and the correctness of the deployed Hypervisor to the user, as long as the Manufacturer is honest and does not provide the device private key to the adversary. We used the method described in [44] (i.e., sign the generated session key and a user-generated nonce) to defend against man-in-the-middle and replay attacks.

The Hypervisor and the HEVMs are not reconfigurable once booted, so there is no time-of-check to time-of-use (TOCTTOU) vulnerabilities.

Isolation of the HEVMs and the Hypervisor (A2). Each HEVM has its dedicated hardware set, which is exclusively assigned to only one user. Also, guaranteed by our boot process, the Hypervisor is the only piece of software running on-chip. This dedicated hardware design prevents the adversary from launching any attack (e.g., cache Evict-and-Reload) directly from the resources occupied by the users or the Hypervisor.

An adversary-controlled HEVM can only access its own layer 1 and 2 memory because the fixed circuit limits the data or control path. Trying to overflow the layer 2 memory with one execution frame will result in a Memory Overflow Error. Trying to overflow the layer 2 memory with multiple execution frames will only cause the lowest frame to be swapped out. In all cases above, the on-chip memory spaces of other HEVMs or the Hypervisor will not be affected.

An adversary from the untrusted world has no access to any on-chip components. Although they can send interrupts to the Hypervisor, these interrupts are non-preemptive, and the Hypervisor only responds to them when in a secure state (when it is idle). Therefore, the handling process of other inputs will not be corrupted. Assuming that the on-chip components cannot be physically manipulated, the off-chip adversary cannot threaten the HEVMs or the Hypervisor.

Control flow integrity (A3). Control flow attacks do not apply to HEVMs because their circuits are fixed. This is an important reason why we choose hardware EVMs. Also, the call stack frame pointers cannot be tampered with because they are stored in the trusted memory.

The runtime memory required by the Hypervisor is about 250 KB (see Section VI-A) and is small enough to be kept always on-chip. Due to the isolation we have demonstrated, directly probing or manipulating the Hypervisor is impossible. The adversary can only turn to input-based vulnerabilities (e.g., input buffer overflow) to perform code injection or code reuse attacks [16], [25]. However, this is also impossible because the Hypervisor never stores the inputs in its runtime memory. It only checks the message header (which has a fixed size of only 32 bytes) and controls the DMA hardware to load the proper amount of data into the HEVMs. Thus, to the best of our knowledge, we significantly raise the bar for exploitable vulnerabilities for control flow attacks against our system.

Off-chip swap data protection (A4). Assume the AES-GCM and ECDSA algorithms can be trusted. The AES session key and the two ECSDA private keys (for the user and the Hypervisor respectively) are different for each session. Thus, although the adversary can physically access the peripherals and the layer 3 memory, they cannot decrypt or fake the contents without the correct keys.

Off-chip swap pattern protection (A5). Page swaps will only occur when the call stack overflows the 1 MB layer 2 memory, which seldom happens according to the statistics of

recent transactions (Table I). Even when overflows do occur, the adversary can only learn the size of the execution frames in page (1 KB) granularity, added with random pre-evict and pre-load noises described in Section IV-B. We argue that this information would be too imprecise to help the adversary identify the running contract.

Query data integrity (A6). The integrity of ORAM-managed world states is properly ensured by the ORAM protocol itself [39]. The protection of the ORAM key is guaranteed by the Hypervisor.

Contents of new blocks synchronized from the Node are authenticated by block hashes and Merkle proofs. The user can fetch the block hash that identifies their desired version from multiple Nodes and trust the most acknowledged one.

Query obliviousness (A7). We trust the obliviousness of the Path ORAM and all related security claims therein [39]. The ORAM client which manages the local stash and position map is trusted because it is implemented fully on-chip as part of the Hypervisor. The fixed-sized responses and the code prefetching strategy introduced in Section IV-D prevent an adversary from distinguishing between access to bytecodes and K-V pairs. Also, Merkle proofs are only fetched when synchronizing on-chain blocks, during which obliviousness is not required.

Conclusion. HarDTAPE is more secure than traditional solutions against (A2) because the strictly isolated hardware sets remove the attack surfaces of side channels. It is more secure against (A3) because the hardware-based EVM and DMA modules shrink the size of the software TCBs. Also, HarDTAPE is a successful application of ORAM to defend against (A7) in the Ethereum scenario. In other respects, it is equally secure to traditional solutions because the same set of techniques are applied.

VI. Evaluation

In this section, we evaluate our design from four perspectives: (1) Evaluate the resource consumption of the design. (2) Verify the correctness by comparing the execution trace with on-chain ground truth.(3) Evaluate the performance on real-world transactions. (4) Estimate the scalability based on the throughput of the HEVM and the ORAM server.

Implementation and experiment setup. The prototype of HarDTAPE is synthesized using Xilinx Vivado 2021.2 [1] and implemented on an XCZU15EV MPSoC chip [2]. This MPSoC comprises a quad-core ARM Cortex-A53 applications processor to run the Hypervisor, and FPGA resources to run the HEVMs. The chip package provides the isolation and on-chip secure resources (PUF and secure RNG) we demand, so it can be directly used in production. An SP can simply write the bitstream of our design to the off-the-shelf MPSoC to run their service.

The clock frequency of the HEVMs is set to 0.1 GHz, and the ARM core runs at 1.4 GHz. The FPGA BlockRAMs act as the layer 1 and 2 memory of the HEVMs and the ORAM local stash. A 256 KB on-chip memory is used as the runtime memory of the Hypervisor. The chip is installed on a motherboard with 2GB DDR4 (untrusted) memory and Ethernet peripherals. The ORAM server runs on a Windows server with 32 GB DDR4 memory and an Intel Core i7-12700 CPU running at 4.35 GHz. HarDTAPE can access the server via Ethernet with a 2 ms latency. As a baseline, we run Geth [7] on the same server. The ORAM Server and Geth are never activated simultaneously and never compete for memory.

We use blocks #19145194 - #19145293 from the Ethereum Mainnet as the *evaluation set* in all experiments in this paper. For Geth, all referred data are prefetched to the server's main memory. For ORAM-disabled configurations of HarDTAPE, these data are prefetched to the untrusted memory. For the ORAM-enabled configuration, these data are not prefetched to HarDTAPE but only synchronized to the ORAM server.

A. Resource Utility

Hardware. According to the utilization report generated by Vivado, when deploying one HEVM instance, the total resource consumption includes 103388 LUTs, 37104 FFs, and 509 KB BlockRAM. Limited by the LUT utility bottleneck, each XCZU15EV chip can deploy up to three HEVMs.

Software (memory). The size of the Hypervisor binary is 156 KB. This includes a network protocol stack, which can be run in the untrusted world. For security purposes, the Hypervisor does not require any heap memory. However, it used up to 92 KB stack space during the experiment. The total memory usage is 248 KB and fits in the 256 KB on-chip memory.

B. Pre-execution Correctness

We verify that the behavior of HarDTAPE is identical to a standard Node in the network. Specifically, quicknode.com provides the `debug_traceTransaction` RPC method to fetch ground truth traces of real-world transactions, including step-by-step `PC`, stack contents, remaining Gas, storage record accessed, and contract calls [10]. The Memory Overflow Error may occur when executing roll-up transactions (see Section II-A), which may exceed the layer 2 frame size limit. Support for these contracts is left as future work. By comparing our traces with the ground truth, we examined that HarDTAPE can run the remaining transactions correctly.

C. Performance and Overheads

We compare the performance of HarDTAPE and Geth on the evaluation set, using each transaction as a separate bundle. This estimates the lower bound of our performance, because more transactions in a bundle lead to less time-consuming ECDSA verifications and signatures.

Overall response time. We measured the end-to-end response time of HarDTAPE with different configurations. The end-to-end time refers to the period between the SP receives the user's requests and sends out the traces. The results are shown in Figure 4. With all security features turned on, the -full configuration uses 164.4 ms for each transaction on average. As pointed out in Section III-A, a latency of less than 600 ms is reasonable for the user.

Security feature overheads breakdown. To show the performance overhead of each of our security features respectively,

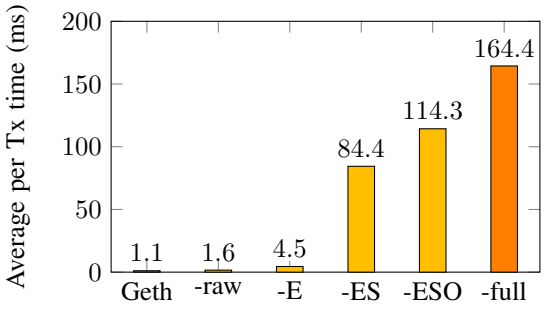

Fig. 4: End-to-end per-transaction time cost of HarDTAPE and Geth on real-world evaluation set. -raw stands for HarDTAPE disabling all security features, and -E, -ES, -ESO, and -full means additionally enabling (E)ncryption, (S)ignature, (O)RAM for storage and (full)y enabling all protections respectively. The -full configuration is to be applied by the SP.

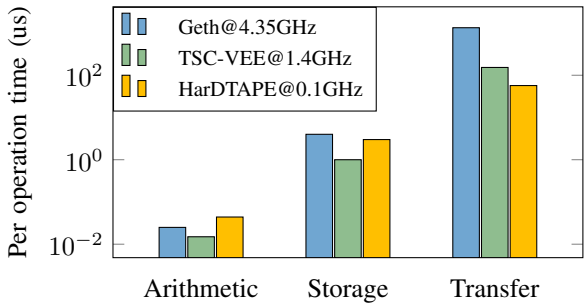

Fig. 5: Execution time per operation (log scale) of Geth, TSC-VEE, and HarDTAPE s.t. all used data are found locally.

we test HarDTAPE with the following configurations to compare with Geth. The -raw configuration runs the HEVM while disabling all off-chip data protections. Then, we step-by-step toggle on encryption (-E), user data signature and verification (-ES), ORAM for storage (-ESO), and ORAM for all world state data (-full) based on the last configuration. The results are shown correspondingly in Figure 4.

The unprotected configuration (-raw) brings about 0.5 ms overhead per transaction compared to Geth. Using an HEVM with higher performance can remove this overhead, but as we will point out later, the performance bottleneck lies in the security features. Thus, optimization of the HEVMs is out of the scope of this paper.

The encrypted configuration (-E) introduces 2.9 ms overhead. The overhead is small because the layer 2 memory of each HEVM is large enough in most cases, and encryption is only applied to user inputs and returning traces.

In contrast, the ECDSA signature (-ES), although only used for inputs and traces as well, adds 80 ms overhead to the overall runtime. This is expected because the ECDSA procedure is much more complex than AES-GCM. Fortunately, only one ECDSA signature is needed for each bundle independent of its size, so this overhead can be amortized to all its transactions.

The Path ORAM introduces another 80 ms for all world state queries (-full), in which about 30 ms are for key-value style queries (-ESO) and the rest are for bytecodes. We admit this is a large performance overhead compared to Geth and the -raw version. However, we recall that pre-execution is designed for risk-avoiding users who always have higher confidentiality requirements, so this sacrifice is worth it.

Local execution performance. It is hard to compare the common-case average performance of HarDTAPE with other works about EVMs in TEEs due to their differences in targeted use cases. For instance, TSC-VEE [26] is designed for single Confidential Smart Contracts and does not support cross-account contract calls. It prefetches all the bytecode and storage records to the secure memory. In contrast, HarDTAPE cannot do so because we must support access to the entire 1 TB world state. However, we may consider a practical case in the pre-execution scenario: users may frequently call the same contract and operate on the same set of storage records when testing their FHT strategies. In this case, all data can be found locally after first access, thus there is no security overhead.

We show that HarDTAPE is almost as efficient as TSC-VEE in this case, by design benchmarks to evaluate the time cost of arithmetic instructions, local storage accesses, and contract calls (to ERC20-transfer). We run these benchmarks on Geth, HarDTAPE, and TSC-VEE, such that bytecodes and storage keys are warmed up to their lowest-level cache. As shown in Figure 5, except Geth runs slower on the Transfer benchmark, there is no significant difference between the three platforms. This implies that, if we adapt TSC-VEE to pre-execution, it will have no performance advantage when appended with the same signature and ORAM overheads.

D. Scalability

Transaction bundles can run in parallel because they are not designed to be aware of each other in pre-execution. Thus, a HarDTAPE chip equipped with three HEVM cores can approximately process $3 \times (1/0.164) = 18$ transactions per second. The throughput of one HarDTAPE instance can already support the current throughput of Ethereum Mainnet of about 17 transactions per second. So, at least two HarDTAPE instances (one for pre-execution and one for block synchronization) are enough to run the pre-execution service.

By increasing the number of HarDTAPE instances, this throughput can continue to grow until the ORAM server becomes the bottleneck. Because an ORAM server occupies even more storage than a full-synchronized Node, we assume the SP can only afford to run one ORAM server. We set up a timer on the ORAM server to estimate the maximum supported client number which does not stack its query queue. We observed that the server averagely requires 25 us to process a query, while the average gap between queries from each HEVM is 630 us. Therefore, each ORAM server can approximately support $\lfloor 630/25 \rfloor = 25$ full-load HEVMs.

VII. Conclusion

Users have security concerns over existing transaction pre-execution services because they did not consider access pattern

confidentiality and may have inherent security issues. As a solution, we propose HarDTAPE, a security-focused transaction pre-execution coprocessor. It has a smaller software TCB, better resists side channel and control flow attacks with dedicated hardware EVM executors, and protects the access pattern confidentiality with a Path ORAM. We show by experiments that HarDTAPE can achieve acceptable performance and throughput on SoCs with limited on-chip resources.

ACKNOWLEDGMENT

The authors sincerely thank the reviewers for their invaluable feedback. This work was supported in part by the National Key R&D Program of China (No. 2022YFE0113200) and the National Natural Science Foundation of China (NSFC) under Grant U21A20464, by the Research Grants Council of Hong Kong under Grants CityU 11218322, 11219524, R6021-20F, R1012-21, RFS2122-1S04, C2004-21G, C1029-22G, C6015-23G, and N_CityU139/21, and by the Innovation and Technology Commission of Hong Kong (ITC) under Mainland-Hong Kong Joint Funding Scheme (MHKJFS) under Grant MHP/135/23. This work was also supported by the InnoHK initiative, The Government of the HKSAR, and the Laboratory for AI-Powered Financial Technologies (AIFT).

REFERENCES

[1] Amd vivado™ design suite. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html.
[2] Amd zynq™ ultrascale+™ mpsocs. https://www.amd.com/en/products/\adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html.
[3] blockxroute bundle simulation. https://docs.bloxroute.com/apis/mev-solution/bundle-simulation.
[4] Ethereum. https://ethereum.org/.
[5] Ethereum full sync chart. https://etherscan.io/chartsync/chaindefault.
[6] Etherscan, the ethereum blockchain explorer. https://etherscan.io/.
[7] go-ethereum. https://geth.ethereum.org/.
[8] Intro to ethereum. https://ethereum.org/en/developers/docs/intro-to-ethereum/.
[9] Layout in storage, solidity docs. https://docs.soliditylang.org/en/v0.8.26/internals/layout_in_storage.html.
[10] Quicknode transaction trace records. https://www.quicknode.com/docs/ethereum/debug_traceTransaction.
[11] Tenderly simulator. https://tenderly.co/transaction-simulator.
[12] What are "rollups" in crypto? https://www.coinbase.com/zh-cn/learn/wallet/what-are-rollups-in-crypto.
[13] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. The opbench ethereum opcode benchmark framework: Design, implementation, validation and experiments. *Perform. Evaluation*, 146:102168, 2021.
[14] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: Identification, analysis, and impact. *Future Gener. Comput. Syst.*, 102:259–277, 2020.
[15] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *USENIX Security*, 2018.
[16] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM ASIACCS*, 2011.
[17] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems. In *IEEE S&P*, 2020.
[18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. In *IEEE EuroS&P*, 2019.
[19] Weili Chen, Xiongfeng Guo, Zhiguang Chen, Zibin Zheng, and Yutong Lu. Phishing scam detection on ethereum: Towards financial security for blockchain ecosystem. In *IJCAI*, 2020.
[20] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.
[21] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *IEEE S&P*, 2020.
[22] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.
[23] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43:431–473, 1996.
[24] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, 2020.
[25] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE S&P*, 2016.
[26] Zhaolong Jian, Ye Lu, Youyang Qiao, Yaozheng Fang, Xueshuo Xie, Dayi Yang, Zhiyuan Zhou, and Tao Li. TSC-VEE: A trustzone-based smart contract virtual execution environment. *IEEE Trans. Parallel Distributed Syst.*, 34:1773–1788, 2023.
[27] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+evict. In *ACM/IEEE DAC*, 2021.
[28] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. Keystone: A framework for architecting tees. *CoRR*, abs/1907.10119, 2019.
[29] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, 2017.
[30] Tao Lu and Lu Peng. SCU: A hardware accelerator for smart contract execution. In *IEEE Blockchain*, 2023.
[31] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. *IACR Cryptol. ePrint Arch.*, page 1285, 2023.
[32] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.
[33] Makoto Nagata, Takuji Miki, and Noriyuki Miura. Physical attack protection techniques for IC chip level hardware security. *IEEE Trans. Very Large Scale Integr. Syst.*, 30:5–14, 2022.
[34] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel SGX. *CoRR*, abs/2006.13598, 2020.
[35] Arttu Paju, Muhammad Owais Javed, Juha Nurmi, Juha Savimäki, Brian McGillion, and Billy Bob Brumley. Sok: A systematic review of TEE usage for developing trusted applications. *CoRR*, abs/2306.15025, 2023.
[36] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. HECKLER: breaking confidential vms with malicious interrupts. In *USENIX Security*, 2024.
[37] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.
[38] Vasily Sidorov, Ethan Yi Fan Wei, and Wee Keong Ng. Comprehensive performance analysis of homomorphic cryptosystems for practical data processing. *CoRR*, abs/2202.02960, 2022.
[39] Emil Stefanov and Elaine Shi. Path O-RAM: an extremely simple oblivious RAM protocol. *CoRR*, abs/1202.5150, 2012.
[40] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In *USENIX Security*, 2021.
[41] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *USENIX Security*, 2019.
[42] Ke Xia, Yukui Luo, Xiaolin Xu, and Sheng Wei. SGX-FPGA: trusted execution environment for CPU-FPGA heterogeneous architecture. In *ACM/IEEE DAC*, 2021.
[43] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. Trusense: Information leakage from trustzone. In *IEEE INFOCOM*, 2018.
[44] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. Shef: shielded enclaves for cloud fpgas. In *ACM ASPLOS*, 2022.