# Towards a Marketplace for Secure Outsourced Computations

Hung Dang[1(✉)], Dat Le Tien[2], and Ee-Chien Chang[1]

[1] National University of Singapore, Singapore, Singapore
{hungdang,changec}@comp.nus.edu.sg
[2] University of Oslo, Oslo, Norway
dattl@ifi.uio.no

**Abstract.** This paper presents Kosto – a framework that provisions a *marketplace for secure outsourced computations*, wherein the pool of computing resources aggregates that which are offered by a large cohort of independent compute nodes. Kosto protects the *confidentiality* of clients' inputs and the *integrity* of the outsourced computations using trusted hardware's enclave execution (e.g., Intel SGX). Furthermore, Kosto mediates *exchanges* between the clients' payments and the compute nodes' work in servicing the clients' requests without relying on a trusted third party. Empirical evaluation on the prototype implementation of Kosto shows that performance overhead incurred by enclave execution is as small as 3% for computation-intensive operations, and 1.5× for I/O-intensive operations.

## 1 Introduction

Recent years have witnessed an emergence of online marketplaces that offer alternatives to traditional vendor-specific service providers. Examples include Airbnb [1] in lodging, Uber [13] in transportation. In such marketplaces, the shared pool of resources is neither owned, provisioned nor controlled by a single party. Instead, it aggregates that which are offered by a large cohort of independent individuals. Designing a marketplace for secure outsourced computations, however, faces various technical challenges.

The first technical challenge is in protecting the *confidentiality* of the clients' data and the *integrity* of the outsourced computations, for the resource providers (or *compute nodes*) may be untrustworthy. Solutions to protect the confidentiality and integrity of outsourced computations have been studied in the literature [25–27,46]. For examples, homomorphic encryption [26,46] and secure multi-party computation [27] are designed to protect data confidentiality, while verification by replications [2,11] and verifiable computation [25] aim to protect computation integrity. Nevertheless, these approaches either incur high overheads, or support only a limited range of applications. These limitations hinder their adoption in practical systems.

H. Dang and D. Le Tien—Lead authors are alphabetically ordered.

Another technical challenge is in mediating *exchanges* between clients' payments and compute nodes' work in servicing the clients' requests without relying on a trusted third party. One approach is to commits a remuneration for a task into an escrow which shall autonomously release the payment to the compute node upon successful task completion. This approach, however, does not generalize. For micro tasks that yield small remunerations, the transaction fee (i.e., the cost to conduct the payment transaction) becomes an overhead. On the other hand, compute nodes may inadvertently abort macro or complex tasks midway, exerting computational work but could not claim the reward. We believe that an ideal solution to mediate fair exchanges between clients' payments and compute nodes' work would require trusted metering of the compute nodes' work and a self-enforcing, autonomous agent (e.g., smart contract) responsible for settling payments based on the aforementioned metering.

In this paper, we present a framework that enables a marketplace for secure outsourced computations, which we name Kosto. Under our framework, the shared pool of computing resources is contributed to by a cohort of independent *compute nodes*. *Clients* in Kosto can request computational services from the compute nodes, while enjoying confidentiality protection on their data and integrity assurance on their outsourced computations. This is achieved by the use of Trusted Execution Environments (TEEs). In particular, each compute node in Kosto is capable of provisioning a TEE such as Intel SGX *enclave* for outsourced computations. The enclave prevents other processes, the operating system and even the owner of the compute node from tampering with the execution of the code loaded inside the enclave or observing its state. A compute node services a client' request by executing the outsourced computation inside an enclave that is attested to be correctly instantiated. The attestation allows secrets to be provisioned to the enclave only after it is instantiated.

Kosto mediates exchange of the clients' payment and compute nodes' work via a hybrid architecture that combines TEE-based metering with blockchain micro payment channel [10,37]. Our framework incorporates in each enclave an accounting logic that meters the compute node's work. Such metering is then translated to a *payment promise* with which the compute node can settle the payment escrow and claim the corresponding reward. This approach facilitates the exchange between the client and the compute node without incurring excessive transaction fee or involving a trusted third party.

Our experiments reveal that the overhead incurred by enclave execution and the trusted metering is as small as 3% for computation-intensive operations, and 1.5× for I/O-intensive operations. We expect these overheads can be further reduced by incorporating optimizations that enhance the efficiency of enclave execution [36,44,47], thereby allowing Kosto to attain better efficiency.

In summary, this paper makes the following contributions.

- We propose a framework, called Kosto, which facilitates a marketplace for secure outsourced computations. Under our framework, both confidentiality of clients' inputs and integrity of the outsourced computations are protected through the use of TEEs. In addition, Kosto mediates fair exchanges between

clients' payments for the execution of the outsourced computations and compute nodes' work in servicing the clients' requests via a hybrid architecture that combines TEE-based metering with blockchain micro payment channel.
– We implement a prototype of Kosto and evaluate the overhead incurred by enclave execution and the TEE-based metering. The experiments shows that performance overhead incurred by enclave execution and trusted metering is as small as 3% for computation-intensive operations, and 1.5× for I/O-intensive operations.

## 2 Preliminaries

**Intel SGX.** Intel SGX [33] is a set of CPU extensions capable of providing hardware-protected TEE (or *enclave*). Each enclave is associated with a protected address space. The processor blocks any non-enclave code's attempt to access the enclave memory. Memory pages are encrypted using the processor's key prior to leaving the enclave. Intel SGX provides attestation mechanisms allowing an attesting enclave to demonstrate to a validator that it has been correctly instantiated [15], and to establish a secure, authenticated connection via which they can securely communicate sensitive data.

**Ethereum Smart Contract.** Ethereum enables *smart contract* which is an "autonomous agent" associated with a predefined executable code. Incentive and security mechanisms of the Ethereum ecosystem encourage miners to execute the contract's code faithfully [18]. A smart contract could be used to implement an *escrow* that enforces a payment from a payer to a payee once the payee has delivered some service to the payer, while keeping the payment inaccessible to the payee before such condition is met. The transaction fee to settle the escrow does not depend on the monetary value that the escrow holds. Consequently, should the payment value is too small (i.e., micro transaction), the transaction fee becomes a significant overhead.

**Payment Channel.** Payment channel enables two parties to transact a large number of micro payments without incurring high transaction fee or overloading the blockchain with excessive number of transactions [10,37]. A channel is established after a deposit is made on the blockchain (on-chain). A payer makes a micro payment to the payee by issuing a digitally signed and hash-locked transfer, called *payment promise*, and sending it off-chain to the payee. The payee can use such payment promise to close the channel and claim the payment she has been promised so far at any time. The value of the payment promises should not exceed the on-chain deposit, otherwise it cannot be fully collateralized.

## 3 The Problem

### 3.1 System Model

We study a marketplace for secure outsourced general-purposed computations. Unlike vendor-specific cloud services, the pool of computing resources in such a marketplace aggregates that which are offered by a large cohort of independent *compute nodes* (discussed below). More specifically, we consider a system model that comprises the following three main parties: *clients, compute nodes* and *brokers.*

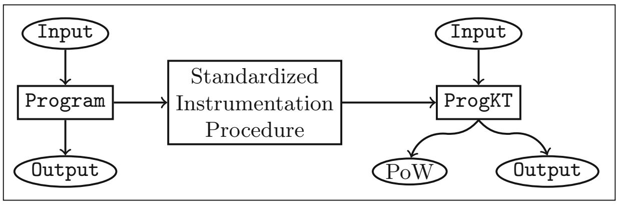


**Fig. 1.** The standardized instrumentation procedure that converts `Program` into `ProgKT`. PoW reflects the compute node's work in executing `Program` on `Input`.

- **Clients** are the system's end users. A client would like to execute a program `Program` on an input `Input`, obtaining an computation outcome `Output`. The program `Program` can be written by the client, or an open-source software provided by a third party. In either case, the client outsources such computational task to a *compute node* (which we shall define in the following). The clients are not expected to maintain constant connection with the compute node over the course of the outsourced computation. While the clients can discover the compute nodes and initiate the outsourced computation on their own, this approach is unlikely to scale. Instead, we propose to delegate such tasks to a *broker.*
- **Compute nodes** are machines equipped with commodity trusted processors (e.g., Intel SGX processors) capable of provisioning TEEs (or enclaves). A compute node services a client request by running its code in an enclave, and generating an attestation that proves the correctness of the code execution (and thus the result). In return, the compute node receives remuneration $v$ proportional to computational work it has asserted in executing the outsourced task.
- **Brokers** facilitate node discovery and load balancing, and assist the clients in attesting correct instantiation of the enclaves housing the outsourced computations on the compute nodes. Brokers may charge clients and/or compute nodes certain commission fee in return to their services. To eliminate broker monopoly and single-point-of-failure, we allow multiple brokers to co-exist, thus enabling better brokering service for both clients and compute nodes.

Hereafter, we denote by $\mathcal{P}$ a client, by $\mathcal{C}$ a compute node, and by $\mathcal{B}$ a broker. The program to be executed on the compute node incorporates logic that meters the compute node's work in a fine-grained and tamper-proof fashion. For simplicity, let us assume that such logic is defined in the system configuration, and agreed upon by all clients and compute nodes. We further assume that there exists a standardized instrumentation procedure that converts the client's program `Program` into a program `ProgKT` that adheres to the metering logic requirements. Figure 1 illustrates an overview of the instrumentation, and the relation between `ProgKT` and `Program`.

## 3.2 System Goals

We now formalize the security guarantees that a marketplace for outsourced computations should offer. The guarantees motivate and justify our design choices.

- *Correct and attested execution* requires that the output `Output` obtained by the client correctly reflects the faithful execution of `Program` on `Input`.
- *Data confidentiality* requires that `Input`, and secret states of `Program` from a client remain encrypted outside the enclave memory, and thus are not known to any other party, including the compute node (e.g., its OS and its owner). The key to decrypt them resides only inside the enclave.
- *Fair exchange* requires that the work a compute node exhausts in executing the outsourced task is accurately metered and remunerated in fine granularity. At the same time, it dictates that a compute node gets the full reward for the outsourced computation if and only if a client gets a correct result of such computation.

Besides the security guarantees, for practical usability reason, we also wish to *limit the required interaction between client and compute node* and *optimize assignment of clients' request to compute nodes.* The former unburdens clients from constantly maintaining a connection with their assigned compute nodes prior to and during execution of the outsourced computations, while the latter maximizes the resource utilization in the marketplace.

## 3.3 Threat Model

**Trust Assumptions.** We study a threat model in which the parties (namely $\mathcal{C}, \mathcal{B}$ and $\mathcal{P}$) are mutually distrustful. We assume that a standardized instrumentation procedure that converts the client's program `Program` into a program `ProgKT` that meters the compute node's work in a fine-grained and tamper-proof fashion (Fig. 1) can be formally verified and therefore is trusted. We further assume that commodity trusted processors provisioning TEEs on the compute nodes, in particular Intel SGX processors, are implemented correctly and their protection mechanisms are not compromised. Finally, we make an assumption that the Ethereum blockchain is decentralised and trusted (i.e., it is publicly accessible, and its underlying consensus and smart contract execution mechanisms are intact).

**Adversary.** We consider a party who deliberately deviates from a prescribed protocol an *adversary.* The adversarial goal is to violate the system guarantees described earlier in Sect. 3.2, namely confidentiality of client's data, integrity of the outsourced computations, and the fair exchange between $\mathcal{P}$'s payment and $\mathcal{C}$'s work in servicing the former's request.

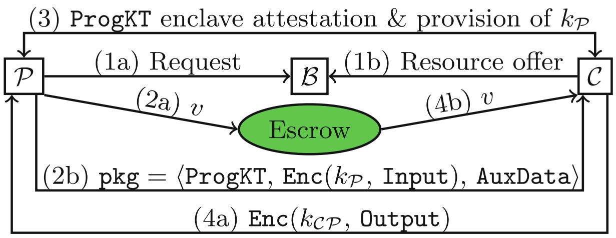

(3) ProgKT enclave attestation & provision of $k_{\mathcal{P}}$
$\mathcal{P}$
(1a) Request
$\mathcal{B}$
(1b) Resource offer
$\mathcal{C}$
(2a) $v$
Escrow
(4b) $v$
(2b) pkg = ⟨ProgKT, Enc($k_{\mathcal{P}}$, Input), AuxData⟩
(4a) Enc($k_{\mathcal{CP}}$, Output)

**Fig. 2.** Kosto overview. $k_{\mathcal{CP}}$ is derived from a secret chosen by $\mathcal{C}$ and $k_{\mathcal{P}}$.

We assume the adversary is computationally bounded, and that the cryptographic primitives employed in the system (e.g., encryption scheme or hash function) are secure. Adversarial clients and brokers can deviate arbitrarily from the prescribed protocol, but they can neither control the compute node's operating system (OS) nor its enclaves' execution. An adversarial compute node can control its operating system, schedule its processes, reorder and tamper with its network messages. Nonetheless, it cannot tamper with the enclaves' execution, nor observe theirs internal state.

We do not consider side-channel attacks against the hardware and the enclave execution [45,48]. Besides, denial of service attack wherein an adversary denies service to honest clients, or blocks honest compute nodes from the system are beyond scope. Consequently, we require some compute nodes to behave correctly so as to guarantee the system's availability. As mentioned earlier, since the clients can serve as their own broker, handling compute node discovery and connecting to the compute nodes directly, there will always be honest self-serving brokers in the system, which eliminates the broker's single-point-of-failure problem.

## 4 Kosto Design

### 4.1 Workflow

$\mathcal{P}$ and $\mathcal{C}$ can post their requests and available resource offers to a broker $\mathcal{B}$ of their choice, perhaps based on $\mathcal{B}$'s reputation or quality of service. $\mathcal{B}$ then evaluates among all requests and offers it has received a suitable assignments of requests to compute nodes. Alternatively, the clients and the compute nodes can directly discover and connect to each other. In such case, they play an additional role of self-serving broker.

Let $v$ be the remuneration that $\mathcal{P}$ pays to $\mathcal{C}$ in exchange for executing `Program` on `Input` and delivering the result `Output`. Kosto requires `Program` to be instrumented into `ProgKT` which incorporates trustworthy metering of the compute node's work. To guarantee payment to $\mathcal{C}$ upon its completion of the computational task, Kosto requires $\mathcal{P}$ to maintain a deposit worth at least $v$ on an on-chain escrow. $\mathcal{P}$ sends $\texttt{pkg} = \langle \texttt{ProgKT}, \texttt{Enc}(k_{\mathcal{P}}, \texttt{Input}), \texttt{AuxData} \rangle$ to $\mathcal{C}$, wherein `Enc()` is a symantically secure symmetric-key encryption scheme [29], and AuxData contains auxiliary data needed for the execution. $\mathcal{C}$ instantiates the ProgKT enclave, and attests to $\mathcal{P}$ that the enclave has been instantiated correctly. Upon successful attestation, a secret key $k_{\mathcal{P}}$ is provisioned to the ProgKT enclave, allowing it to process and compute on Input. Finally, the output Output of the computation is sent to $\mathcal{P}$. Output is encrypted in such a way that its decryption by $\mathcal{P}$ ensures full payment of $v$ to $\mathcal{C}$. Figure 2 depicts the workflow in Kosto.

## 4.2 Enclave Execution

Kosto relies on Intel SGX [33] to offer attested execution and data confidentiality to the outsourced computations. The outsourced program Program should be SGX-compliant (i.e., it inherently supports SGX enclave execution). Techniques that enable enclave executions for unmodified legacy applications, such as Haven [16] and Panoply [41] are orthogonal to Kosto.

A compute node services the client's request by first instantiating the ProgKT enclave, and generating an attestation proving that the enclave has been instantiated correctly. The attestation mechanism allows $\mathcal{P}$ to establish a secure, authenticated connection to the enclave, via which the secret key $k_{\mathcal{P}}$ is communicated. The compute node then invokes the enclave execution on Input to collect the output Output. In addition to Output, the enclave also returns a "proof of work" indicating a computational effort that $\mathcal{C}$ has asserted thus far, which $\mathcal{C}$ can use to claim the remuneration. We elaborate on this in Sect. 4.3.

## 4.3 Fair Exchange

Kosto splits the reward $v$ of the outsourced computation into two portions, namely $v_c = \alpha v$ and $v_d = (1 - \alpha)v$, where $\alpha$ is a parameter set by the client $\mathcal{P}$, and agreed upon by $\mathcal{C}$. The first portion (i.e., $v_c$) remunerates $\mathcal{C}$ for its work on a fine-grained basis, while the second portion (i.e., $v_d$) rewards the delivery of the result. The configuration of the parameter $\alpha$, and by its extension, the remuneration policy, is beyond Kosto's scope.

$\mathcal{C}$ is entitled to $v_c$ upon the completion of the outsourced computation. In case the computation is inadvertently aborted midway, $\mathcal{C}$ is still remunerated with a fraction of $v_c$ according to its progress prior to the suspension. The remaining portion of $v$, namely $v_d$, is only payable to $\mathcal{C}$ when the computation output is delivered to $\mathcal{P}$. This discourages $\mathcal{C}$ from denying $\mathcal{P}$ of the result. Additional mechanism that disincentivises result withholding (e.g., requiring $\mathcal{C}$ to make a security deposit which is forfeited should they repeatedly abort the computation [17,30]) can also be incorporated into Kosto.

**TEE-Based Metering.** To enable an fair exchange described above, Kosto has to meter the compute node's work in a fine-grained and tamper-proof fashion. We follow Zhang et al. [49] in implementing a reliable metering logic inside the enclave. More specifically, Kosto requires the client's program Program to be instrumented into a wrapper program ProgKT (see Fig. 1). The wrapper program reserves the logic of the original program (i.e., it executes Program's logic on

`Input`), while keeping a counter of the number of instructions that has been executed. This is then used as a measurement of the compute node's work.

`ProgKT` maintains the instruction counter in a reserved register which is inaccessiable to any other process. To prevent a malicious `Program` from manipulating the instruction counter, Kosto does not support `Program` that is multi-threaded or contains writeable code pages [5,49]. When the `ProgKT` enclave halts or exits, it returns a "proof of work" (i.e., the number of instruction executed) based on which Kosto settles the payment of $v_c$ (or a fraction of it). We note that if the compute node (i.e., its OS) intentionally kills the enclave process, `ProgKT` does not return such proof of work, which eliminates a remuneration-draining attack where a malicious compute node deliberately interrupts the enclave execution before it finishes, so as to drain $v_c$ without an intention of completing the outsourced computation.

We remark that the restriction of single-threaded `Program` is not necessary a severe limitation, for threading in SGX enclave is much different compared to that of legacy software [3]. In particular, one cannot create or destroy an SGX thread on the fly, and an SGX thread is mapped directly to a logical processor. Consequently, a typical SGX-compliant program (i.e., a program that inherently supports SGX-enclave execution) is often single-threaded.

**On the Choice of Instruction Counting.** One may argue that instructions are not the most accurate metric for CPU effort. Alternative metrics include CPU time and CPU cycles. Nevertheless, these metrics are subject to manipulation by the malicious OS. Even if they were not manipulated, they are incremented even when an enclave is swapped out [49]. Consequently, we believe that instruction counting is the most appropriate method for securely measuring the compute node's effort using available tools in SGX.

**Micro Payments with Off-Chain Payment Channel.** One naive approach to settle the proof of work is for $\mathcal{C}$ to send it to $\mathcal{P}$, who then responds with a transaction paying a corresponding amount of reward to $\mathcal{C}$. This approach, however, does not payment for $\mathcal{C}$ in case $\mathcal{P}$ neglects her outsourced computation. Another approach is to have $\mathcal{P}$ commit a number of equally-valued micro transactions, each of which contains a fraction of $v_c$, to a payment escrow on the blockchain, and to structure the proof such that it can be used to autonomously claim a subset or all of those micro transactions. Nonetheless, settling a large number of micro transactions on the blockchain incurs high overhead.

Kosto sidesteps this challenge by leveraging payment channel [10], allowing two parties to transact a large number of micro payments without incurring high transaction fee or overloading the blockchain with transactions. It is assumed that a payer and a payee maintain a payment channel (discussed in Sect. 2), and each micro payment is represented by a payment promise to be communicated off-chain (i.e., off the blockchain) between the payer and the payee. To settle the payments, the payee posted the latest payment promise (accompanied by settling-data such promise requires, if any) to the blockchain, thereby closing the channel. However, establishing a new channel for each pair of client and

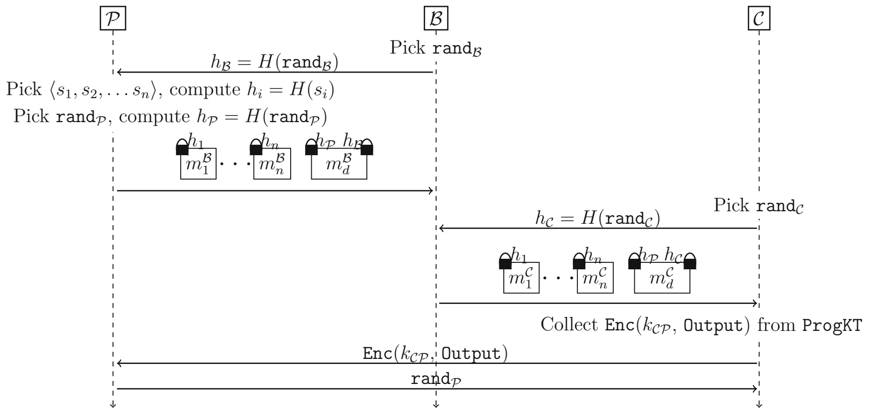

$\mathcal{P}$
$\mathcal{B}$
$\mathcal{C}$
Pick $\texttt{rand}_\mathcal{B}$
$h_\mathcal{B} = H(\texttt{rand}_\mathcal{B})$
Pick $\langle s_1, s_2, \ldots s_n \rangle$, compute $h_i = H(s_i)$
Pick $\texttt{rand}_\mathcal{P}$, compute $h_\mathcal{P} = H(\texttt{rand}_\mathcal{P})$
$h_1$ $m_1^\mathcal{B}$ $\ldots$ $h_n$ $m_n^\mathcal{B}$ $h_\mathcal{P}$ $h_\mathcal{B}$ $m_d^\mathcal{B}$
Pick $\texttt{rand}_\mathcal{C}$
$h_\mathcal{C} = H(\texttt{rand}_\mathcal{C})$
$h_1$ $m_1^\mathcal{C}$ $\ldots$ $h_n$ $m_n^\mathcal{C}$ $h_\mathcal{P}$ $h_\mathcal{C}$ $m_d^\mathcal{C}$
Collect $\texttt{Enc}(k_{\mathcal{CP}}, \texttt{Output})$ from $\texttt{ProgKT}$
$\texttt{Enc}(k_{\mathcal{CP}}, \texttt{Output})$
$\texttt{rand}_\mathcal{P}$

**Fig. 3.** An overview of the fair exchange in Kosto. $m_i^\mathcal{B}$ and $m_i^\mathcal{C}$ are hash-locked by $h_i$, $m_d^\mathcal{B}$ by $h_\mathcal{P}$ and $h_\mathcal{B}$, $m_d^\mathcal{C}$ by $h_\mathcal{P}$ and $h_\mathcal{C}$, and $k_{\mathcal{CP}} = k_\mathcal{P} \oplus \texttt{rand}_\mathcal{C}$.

compute node is inefficient. Kosto, instead, makes use of multi-hop channels[1] to better utilize the channel capacity, requiring fewer channels to be established.

To this end, Kosto assumes that each client $\mathcal{P}$ maintains a payment channel with the broker $\mathcal{B}$ that, in turn, maintains a channel with each compute node $\mathcal{C}$. The payment from $\mathcal{P}$ to $\mathcal{C}$ does not require a direct channel; rather, it could be securely routed via $\mathcal{B}$, in a sense that once $\mathcal{C}$ collects a payment from $\mathcal{B}$, the latter is guaranteed of a corresponding payment from $\mathcal{P}$[2]. We assume that each payment channel has sufficiently large capacity (i.e., its on-chain deposit) to accommodate the payment of various outsourced computations during its lifetime.

Figure 3 summarizes the fair exchange of the reward $v$ and the outsourced computation of ProgKT. $v$ is split over $n+1$ micro payments, $n$ of which summing up to $v_c$, while the last one is worth $v_d$. The protocol does not require any communication between $\mathcal{P}$ and $\mathcal{C}$ *prior to* or *during* the computation, nor an on-chain channel between them. It, however, requires an off-chain communication between $\mathcal{P}$ and $\mathcal{C}$ in the final step to decrypt the output.

**Payment of $v_c$.** Without loss of generality, let us assume that the payment of $v_c$ is divided into $n$ equally-valued payment promises, which are routed via $\mathcal{B}$. That is, $\mathcal{P}$ generates $n$ payment promises to $\mathcal{B}$, and $\mathcal{B}$ generates the corresponding $n$ payments promises to $\mathcal{C}$ with the same value and claiming condition.

To generate the $n$ payment promises $\langle m_1^\mathcal{B}, m_2^\mathcal{B}, \ldots m_n^\mathcal{B} \rangle$ to $\mathcal{B}$, $\mathcal{P}$ first picks $n$ random strings $\langle s_1, s_2, \ldots s_n \rangle$, and computes their hashes $\langle h_1, h_2, \ldots h_n \rangle$ (i.e., $h_i = H(s_i)$). A digest $h_i$ is used to lock a promise $m_i^\mathcal{B}$, such that $\mathcal{B}$ can only use $m_i^\mathcal{B}$ to close the channel if it is aware of $\mathtt{s}_i$ such that $H(\mathtt{s}_i) = h_i$. The payment

[1] While we discuss unidirectional channels, Kosto supports bidirectional channels.
[2] While $\mathcal{B}$ could charge a service fee for the routing, for simplicity, we assume $\mathcal{B}$ offers such routing free of charge. Extending Kosto to support such service fee is trivial.

promise $m_i^{\mathcal{B}}$ is worth $[\texttt{debt}_{\mathcal{P}} + (i \times v_c)/n]$ wherein $\texttt{debt}_{\mathcal{P}}$ is the accumulated amount of unsettled payment for $\mathcal{P}$'s previous requests. Finally, $\mathcal{P}$ encrypts the random strings $\langle s_1, s_2, \ldots s_n\rangle$ with $k_{\mathcal{P}}$, and attaches them as well as the payment promises to `AuxData`.

Similarly, $\mathcal{B}$ generates the corresponding promises $\langle m_1^{\mathcal{C}}, m_2^{\mathcal{C}}, \ldots m_n^{\mathcal{C}}\rangle$ to $\mathcal{C}$. Each promise $m_i^{\mathcal{C}}$ is locked by $h_i$ (i.e., the same hash-lock as $m_i^{\mathcal{B}}$), and worth $[\texttt{cred}_{\mathcal{C}} + (i \times v_c)/n]$ wherein $\texttt{cred}_{\mathcal{C}}$ is the accumulated unsettled credit that $\mathcal{C}$ is entitled to claim for its previous services. $\mathcal{B}$ includes these promises into the `AuxData` before forwarding `pkg` to $\mathcal{C}$.

**Payment of $v_d$ Upon Output Delivery.** To ensure that the remaining portion of $v$, namely $v_d$, can only be collected upon the delivery of the output to $\mathcal{P}$, `ProgKT` encrypts the output using a key $k_{\mathcal{CP}}$ derived from $k_{\mathcal{P}}$ and a secret $\texttt{rand}_{\mathcal{C}}$ chosen and committed to by $\mathcal{C}$. At the same time, the full payment of $v$ is encumbered until the disclosure of $\texttt{rand}_{\mathcal{C}}$.

As shown in Fig. 3, besides the $n$ payment promises above, $\mathcal{P}$ generates another payment promise $m_d^{\mathcal{B}}$ to $\mathcal{B}$ that is worth $[\texttt{debt}_{\mathcal{P}} + v]$ and is hash-locked by two digests $h_{\mathcal{B}}$ and $h_{\mathcal{P}}$. Similarly, $\mathcal{B}$ also generate one more payment promise $m_d^{\mathcal{C}}$ to $\mathcal{C}$ that is worth $[\texttt{cred}_{\mathcal{C}} + v]$, and hash-locked by $h_{\mathcal{P}}$ and $h_{\mathcal{C}}$. The three hash-locks $h_{\mathcal{P}}$, $h_{\mathcal{B}}$ and $h_{\mathcal{C}}$ can be settled by three independent settling-data $\texttt{rand}_{\mathcal{P}}$, $\texttt{rand}_{\mathcal{B}}$ and $\texttt{rand}_{\mathcal{C}}$ chosen independently at random by the three parties $\mathcal{P}, \mathcal{B}$ and $\mathcal{C}$, respectively.

**Dynamic Runtime Checks.** The fair exchange requires the wrapper enclave `ProgKT` to perform some dynamic checks at runtime prior to executing `Program`'s logic. More specifically, besides `Input` and `AuxData`, `ProgKT` also consumes the hash-lock $h_{\mathcal{C}}$ and $\texttt{rand}_{\mathcal{C}}$. It first verifies the validity of the settling-data $\langle s_1, s_2, \ldots s_n\rangle$ (i.e., $h_i = H(s_i) \forall \langle h_i, s_i\rangle \in \texttt{AuxData}$). Next, it checks if $h_{\mathcal{C}} = H(\texttt{rand}_{\mathcal{C}})$. Only when the verification passes does it execute `Program` on `Input`, obtaining `Output`. It then encrypts `Output` with $k_{\mathcal{CP}} = k_{\mathcal{P}} \oplus \texttt{rand}_{\mathcal{C}}$, producing an encrypted output $\texttt{Enc}(k_{\mathcal{CP}}, \texttt{Output})$. Finally, the enclave returns the appropriate settling-data $s_i$ based on the instruction counter and the encrypted output (if it successfully completes the computation) to $\mathcal{C}$.

**Payment Settlement.** The settling-data $s_i$ renders the promise $m_i^{\mathcal{C}}$ claimable, enabling $\mathcal{C}$ to collect (a portion of) $v_c$ according to its work. In order to collect a payment from a promise, one posts the corresponding settling-data to the blockchain, thereby making it publicly available.

To obtain the settling-data necessary to claim $m_d^{\mathcal{C}}$ (i.e., the full reward $v$), $\mathcal{C}$ has to send the encrypted output to $\mathcal{P}$, who then responds with $\texttt{rand}_{\mathcal{P}}$. If $\mathcal{C}$ chooses to settle the payment thereby closes the channel between $\mathcal{C}$ and $\mathcal{B}$, it has to post both $\texttt{rand}_{\mathcal{P}}$ and $\texttt{rand}_{\mathcal{C}}$ on the blockchain. Since all data posted to the blockchain are publicly available, $\mathcal{P}$ can now collect $\texttt{rand}_{\mathcal{C}}$ to compute $k_{\mathcal{CP}}$ and obtain `Output`, while $\mathcal{B}$ can collect $\texttt{rand}_{\mathcal{P}}$ to claim $m_d^{\mathcal{B}}$. Alternatively, should $\mathcal{C}$ wish to maintain the channel, it back-propagates the settling-data to $\mathcal{B}$ and $\mathcal{P}$ so that they can update $\texttt{cred}_{\mathcal{C}}$, $\texttt{debt}_{\mathcal{P}}$, and $\mathcal{P}$ can decrypt the encrypted output. In a situation where $\mathcal{P}$'s response is invalid (i.e., its digest produced by the standard hash function $H(\cdot)$ does not match $h_{\mathcal{P}}$), $\mathcal{C}$ can check this invalidity locally and use it as a evidence to accuse $\mathcal{P}$ of conducting mischief. In such situation, fair exchange requirement is still guaranteed (i.e., $\mathcal{C}$ does not claim $v_d$ from $\mathcal{B}$, who in turn does not claim $v_d$ from $\mathcal{P}$ and $\mathcal{P}$ cannot decrypt $\texttt{Enc}(k_{\mathcal{CP}}, \texttt{Output})$ to obtain $\texttt{Output}$).

## 4.4 Delegated Attestation

Kosto relieves $\mathcal{P}$ from conducting a remote attestation with $\mathcal{C}$ at the beginning of every request execution by implementing a *delegated attestation* scheme. The scheme requires each broker $\mathcal{B}$ to run an attestation manager enclave $\texttt{AM}$, and each compute node $\mathcal{C}$ to run a key handler enclave $\texttt{KH}$. The execution of $\texttt{AM}$ and $\texttt{KH}$ are protected by Intel SGX.

Without loss of generality, the delegated attestation builds a chain of trust that comprises three links. The first and second links are established via remote attestations between $\mathcal{P}$ as a validator and $\texttt{AM}$ as an attesting enclave, and $\texttt{AM}$ as a validator and $\texttt{KH}$ as an attesting enclave. The final link entails $\texttt{ProgKT}$ enclave to prove its correctness to $\texttt{KH}$ via local attestation. Chaining all three links together, $\mathcal{P}$ gains confidence that the $\texttt{ProgKT}$ enclave has been properly instantiated on the compute node $\mathcal{C}$ using the correct code, without contacting $\mathcal{C}$ or the IAS.

Each attestation manager enclave has its own (unique) public-private key pair $(\texttt{pk}_{\texttt{AM}}, \texttt{sk}_{\texttt{AM}})$ that are generated uniformly at random during the enclave instantiation. Upon successfully instantiating $\texttt{AM}$, $\mathcal{B}$ requests the trusted processor for its remote attestation $\pi_{\texttt{AM}} = \langle M_{\texttt{AM}}, \texttt{pk}_{\texttt{AM}} \rangle_{\sigma_{TEE}}$, where $M_{\texttt{AM}}$ is the enclave's measurement, and $\sigma_{TEE}$ is a group signature signed by the processor's private key. The certificate $\pi_{\texttt{AM}}$ attests for the correctness of the $\texttt{AM}$ enclave and its public key. Nonetheless, the only party that can verify $\pi_{\texttt{AM}}$ is the IAS acting as group manager [15]. Kosto converts $\pi_{\texttt{AM}}$ into a *publicly verifiable* certificate by having $\mathcal{B}$ obtain and store the IAS response $\texttt{Cert}_{\texttt{AM}} = \langle \pi_{\texttt{AM}}, \texttt{validity} \rangle_{\sigma_{IAS}}$ where $\sigma_{IAS}$ is the IAS's publicly verifiable signature on $\pi_{\texttt{AM}}$ and the $\texttt{validity}$ flag. By examining $\texttt{Cert}_{\texttt{AM}}$, any party can verify the correctness of and establish a secure connection to the $\texttt{AM}$ enclave.

Likewise, every compute node $\mathcal{C}$ runs a key handler enclave $\texttt{KH}$. $\mathcal{C}$ obtains (from the IAS) and stores a publicly verifiable certificate $\texttt{Cert}_{\texttt{KH}} = \langle \pi_{KH}, \texttt{valid} \rangle_{\sigma_{IAS}}$, where $\pi_{KH}$ is $\texttt{KH}$'s remote attestation containing its measurement $M_{KH}$ and its unique public key $\texttt{pk}_{\texttt{KH}}$. By examining $\texttt{Cert}_{\texttt{KH}}$, any party can be assured of the correctness of $\texttt{KH}$ and communicate securely with it.

**Delegated Attestation Protocol.** Fig. 4 depicts the workflow of Kosto's delegated attestation. After instrumenting $\texttt{Program}$ into $\texttt{ProgKT}$ and verifying the correctness of the instrumentation, $\mathcal{P}$ initiates the delegated attestation by obtaining $\texttt{Cert}_{\texttt{AM}}$ from $\mathcal{B}$ and verifies its validity. It then establishes a secure and authenticated channel with $\texttt{AM}$ using $\texttt{pk}_{\texttt{AM}}$. $\mathcal{P}$ then sends $\texttt{pkg} = \langle \texttt{ProgKT}, \texttt{Enc}(k_{\mathcal{P}}, \texttt{Input}), \texttt{AuxData} \rangle$ to $\mathcal{B}$, and $k_{\mathcal{P}}$ to $\texttt{AM}$ via the secure channel. Once $\mathcal{B}$ finds a compute node $\mathcal{C}$ that is willing to match $\mathcal{P}$'s request, $\texttt{AM}$ obtains $\texttt{Cert}_{\texttt{KH}}$ from $\mathcal{C}$, verifies its validity, and establishes a secure and authenticated connection with $\mathcal{C}$'s $\texttt{KH}$ to communicate $k_{\mathcal{P}}$. $\mathcal{B}$ then sends $\texttt{pkg}$ to $\mathcal{C}$.

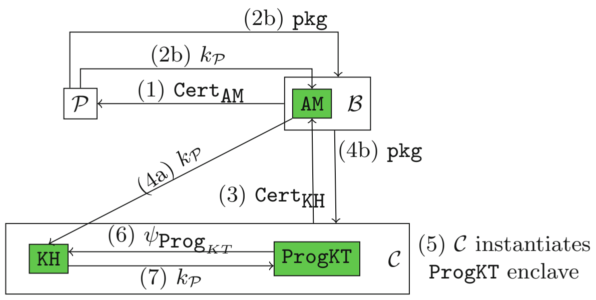

(2b) pkg
(2b) $k_{\mathcal{P}}$
(1) $\texttt{Cert}_{\texttt{AM}}$
$\mathcal{P}$
AM
$\mathcal{B}$
(4a) $k_{\mathcal{P}}$
(4b) pkg
(3) $\texttt{Cert}_{\texttt{KH}}$
(6) $\psi_{\texttt{Prog}_{KT}}$
KH
ProgKT
$\mathcal{C}$
(7) $k_{\mathcal{P}}$
(5) $\mathcal{C}$ instantiates ProgKT enclave

**Fig. 4.** An overview of the delegated attestation scheme.

The compute node instantiates an enclave to execute ProgKT, and performs a *local attestation* with KH to prove its correctness. Upon successfully attestation, KH sends the key $k_{\mathcal{P}}$ to the ProgKT enclave. Once the ProgKT enclave completes the computation, it returns the encrypted output, which is then sent to $\mathcal{P}$ (perhaps being routed through $\mathcal{B}$).

This mechanism only invokes IAS to obtain attestation certificates for AM and KH, instead of constantly involving IAS in every task execution. Further, it allows $\mathcal{P}$ to post a request (along with the payment) and then go offline until the time she wishes to collect the output, as opposed to remaining online till her request is picked up by some computation node.

## 5 Security Arguments

### 5.1 Attested Execution and Data Confidentiality

Kosto's relies on Intel SGX [33] to offer attested execution and data confidentiality to outsourced computations. In particular, SGX enables isolated execution [43] ensuring that code loaded and running inside the enclaves cannot be tampered with by any other processes including the operating system or hypervisor. This, in combination with attestation capabilities, allows Kosto to offer attested execution in which the computation correctness is guaranteed. Moreover, data (i.e., input, output) and secret states of the enclave execution always remain encrypted outside of the enclave memory, thus their confidentiality are guaranteed. Furthermore, SGX memory encryption engine is capable of protecting data integrity and preventing memory replay attacks [28,32].

Nonetheless, SGX's attested execution does not inherently offer protections against side-channel leakages [24,40,48]. The access pattern incurred by data (or code page) moving between the enclave and the non-enclave environment (e.g., page fault) could leak sensitive information about the code or data being processed within the enclave. Such side-channel leakage could be mitigated by ensuring that the enclave execution is *data oblivious*; i.e., the access pattern no longer depends on the input data [21]. While Kosto does not explicitly eliminate side-channel leakage, it could benefit from a vast amount of research on defenses against side-channel leakages [20,21,31,40,42], which we shall incorporate into Kosto in future work.

## 5.2 Fair Exchange

**TEE-Based Metering.** To enable an fair exchange between client's payment and compute node's computation, Kosto necessitates dynamic runtime checks incorporated within the enclave that houses the outsourced computation. We implement this by providing a compiler that instruments any SGX-compliant program `Program` into a wrapper program `ProgKT`. We believe that these additional steps and the overall instrumentation are simple enough to lend themselves to formal verification and vetting by `Program` writer, or by the client.

As we mentioned earlier, the original `Program` should not contain writable code pages, for they would allow the program to rewrite itself at runtime and thus evade the instrumentation. This could be enforced by requiring the code page to have either *write* or *executable* permission exclusively (i.e., it cannot have both permission at the same time). This practice has also been recommended by Intel to the enclave writers [5].

In addition, Kosto requires `Program` to be single-threaded. While the instruction counter is maintained in a reserved register which is inaccessible to any other processes (Sect. 4.3), it remains accessible by different threads of `Program`, should it be multi-threaded. Thus, a malicious program that has multiple threads could manipulate the instruction counter value by carefully crafting the interactions of its threads.

**Payment of $v_c$.** Kosto builds on payment channel [10] to enable efficient micro payments and relies on the security of the Ethereum blockchain to ensure payment escrow is faithfully executed. To optimize for efficiency and avoid overloading the blockchain, Kosto securely routes payment from $\mathcal{P}$ to $\mathcal{C}$ via the broker $\mathcal{B}$. A careful design of hash-lock payment promises, wherein promise from $\mathcal{P}$ to $\mathcal{B}$, and that of $\mathcal{B}$ to $\mathcal{C}$ could be settled using the same settling-data, guarantees that $\mathcal{B}$ can always claim from $\mathcal{P}$ which he pays to $\mathcal{C}$ on behalf of $\mathcal{P}$.

**Ensuring Output Delivery.** At the end of the computation, `ProgKT` enclave encrypts the `Output` using key $k_{\mathcal{CP}} = k_{\mathcal{P}} \oplus \texttt{rand}_{\mathcal{C}}$. Since $m_d^{\mathcal{C}}$ is partially locked by $\texttt{rand}_{\mathcal{C}}$, the decryption of the output and the settling of $m_d^{\mathcal{C}}$ are bound together. In particular, in order to claim $m_d^{\mathcal{C}}$, $\mathcal{C}$ has to post $\texttt{rand}_{\mathcal{C}}$ to the blockchain, making it publicly available. This enables $\mathcal{P}$ to compute $k_{\mathcal{CP}}$ and obtain `Output`. Should $\mathcal{P}$ deny $\mathcal{C}$ of $\texttt{rand}_{\mathcal{P}}$ after receiving the encrypted output, the latter does not reveal $\texttt{rand}_{\mathcal{C}}$, causing the output to remain encrypted. On the other hand, should $\mathcal{C}$ wish to deny $\mathcal{P}$ of the output, it would have to forfeit $v_d$. In sum, it is either the case that $\mathcal{P}$ obtains the output *and* $\mathcal{C}$ is entitled to claim $m_d^{\mathcal{C}}$, or both of them are denied of the exchange's outcome (i.e., `Output` for $\mathcal{P}$ and $v_d$ for $\mathcal{C}$).

### 5.3 Delegated Attestation

Kosto's delegated attestation relies on $\texttt{AM}$ and $\texttt{KH}$ enclaves to attest correct instantiation of $\texttt{ProgKT}$ enclave. Therefore, their correct instantiations are of utter importance. Fortunately, these enclave are fixed (as opposed to the $\texttt{ProgKT}$ enclave that houses client-defined program), and thus are easy to vet and verify.

Kosto's delegated attestation requires minimal involvement of $\mathcal{P}$ (i.e., examine the publicly verifiable certificates $\texttt{Cert}_{\texttt{AM}} = \langle \pi_{\texttt{AM}}, \texttt{validity} \rangle_{\sigma_{IAS}}$). By checking that $\pi_{\texttt{AM}}$ indeed contains the expected measurement $\mathcal{M}_{\texttt{AM}}$, that its validity flag indicates $\texttt{valid}$, and that the certificate has been properly certified (using Intel's published public key [9]), $\mathcal{P}$ can ascertain the correct instantiation of $\texttt{AM}$. Moreover, using the public key $\texttt{pk}_{\texttt{AM}}$ included in $\pi_{\texttt{AM}}$, $\mathcal{P}$ can establish a secure and authenticated channel to $\texttt{AM}$ via which the secret key $k_{\mathcal{P}}$ is communicated. Likewise, $\texttt{AM}$ can verify the correct instantiation of $\texttt{KH}$ and securely communicate $k_{\mathcal{P}}$ to the latter in the exact same manner. The security of the local attestation and communication between $\texttt{KH}$ and $\texttt{ProgKT}$ enclave follows directly from Intel SGX's specifications [15]. Therefore, provided that cryptographic primitives in use are secure, and SGX hardware protection mechanisms are not subverted, Kosto's delegated attestation is secure.

## 6 Evaluation

### 6.1 Experimental Setup

All experiments are conducted on a system that is equipped with Intel i7-6820HQ 2.70 GHz CPU, 16 GB RAM, 2TB hard drive, and running Ubuntu 16.04 Xenial Xerus. We evaluate the overhead of Kosto's enclave execution using a number of computational tasks including five benchmarks (i.e., mcf, deepsjeng, leela, exchang2, and xz) selected from SPEC CPU2017 [12], and two standard cryptographic operations (i.e., SHA256 and AES Encryption). The enclave trusted codebases are implemented using Intel SGX SDK [4]. To quantify the cost of task matching in Kosto, we measure the runtime of the Mucha-Sankowski algorithm [34] that we implemented in C. All experiments are repeated over 10 runs, and the average results are reported.

### 6.2 Cost of Enclave Execution

**Overhead in Execution Time.** We evaluate the five SPEC CPU2017 benchmarks in three different execution modes, namely *baseline*, *SGX-compliant* and *Kosto-compliant*. The baseline mode compiles the benchmarks as-is and runs them in untrusted execution environment. SGX-compliant mode requires porting the benchmarks to support SGX-enclave execution. This entails replacing standard system calls and libraries in the original code with SGX-compliant ones supported in the SGX SDK [4]. Finally, the Kosto-compliant mode further instruments SGX-compliant code with dynamic runtime checks and TEE-based metering discussed in previous section.

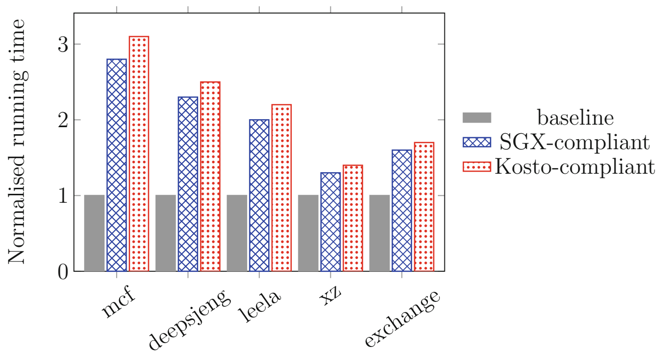
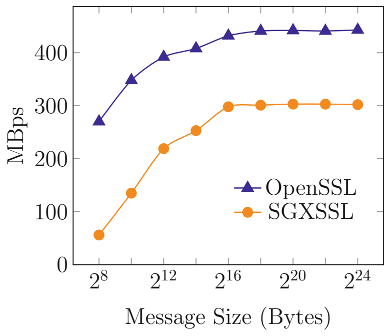
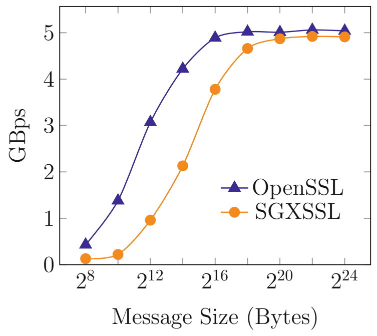


**Fig. 5.** Kosto's enclave execution overhead. The running time of each benchmark is normalized against its own baseline mode's.



(a) SHA256 throughput



(b) AES-GCM throughput

**Fig. 6.** Throughput of enclave and non-enclave based cryptographic operations.

Figure 5 compares the running time of the five benchmarks in three modes, with the running time of each benchmark normalized against its own baseline. We observe that the SGX-compliant mode incurs from 1.5× to 3.7× overhead over the baseline. This overhead is mostly due to enclave's control switching. The instrumentations introduced in Kosto-compliant mode incur an extra 8%–14% overhead relative to the SGX-compliant mode.

Various techniques have been proposed for minimizing the overhead of enclave execution, typically by reducing the control switching between the enclave code and the untrusted application that services OS-provided functions [36,44,47]. We leave the incorporation of such optimization into Kosto for future work.

**Overhead in Throughput.** Next, we measure the overhead in throughput incurred by enclave execution on computation-intensive works. This set of experiments measure performances of SHA256 and AES-GCM encryption operations under *OpenSSL* [7] and Intel *SGXSSL* [6] implementations against exponentially increasing input size (ranging from 256 B to 4 MB). OpenSSL implementation runs in an untrusted non-enclave memory, whereas SGXSSL ports OpenSSL to support SGX enclave execution.

Figure 6a shows a significant gap between the throughput of SGXSSL and OpenSSL implementations of SHA256 when a message size is small (e.g., OpenSSL's throughput is upto 5× for 1 KB message). Nonetheless, such a gap reduces as the message size increases (e.g., as small as 1.5× for 4 MB message). A similar trend is observed in throughput of AES-GCM encryption (the decryption throughput is similar), with the throughput overhead incurred by enclave execution reduces from 6.3× for 1 KB message to 3% for 4 MB message. We attribute this throughput gap to the I/O cost and context switching that enclave execution incurs. Fortunately, this overhead is amortized as the input size increases.

## 7 Related Works

**Decentralised Outsourced Computation**. Golem [2] explores a marketplace for outsourced computation. Unlike Kosto, it does not feature the attested execution environment. Consequently, Golem needs to redundantly execute the same task on multiple compute nodes in order to verify the execution correctness. Concurrent to our work, AirTNT [14] proposes the use of enclave execution for outsourced computations, and devises a protocol that allows fair exchange between the client and the compute nodes. Such protocol necessitates a separate payment channel for every pair of client and compute node, and requires constant communication between the two parties over the course of the outsourced computation (i.e., highly interactive). Kosto, in contrast, alleviates the client and the compute nodes from these inconveniences.

**Reliable Resource Accounting**. Early approaches to resource accounting in the context of outsourced computations rely on nested virtualization and TPMs, or place a trusted resource observer underneath the service provider's software [19,39]. Alternatively, REM [49] instruments the client's program with dynamic runtime checks that maintain an instruction counter to self account its computational effort. The correctness and integrity of these runtime checks are enforced by the trusted hardware. Kosto adopts REM's approach in metering the compute nodes' work.

**SGX-Based Systems**. Trusted hardware, in particular Intel SGX processors, have been used to enhance security in various application domains, including data analytics [21,24,38], machine learning [35] and outsourced storage [20,23]. In addition, SGX has also been utilized to scale the blockchain [8,22]. To our knowledge, Kosto is the first solution to provision a full-fledged marketplace for secure outsourced computations using Intel SGX.

## 8 Conclusion

We have presented Kosto – a framework enabling a marketplace for secure outsourced computations. Kosto protects confidentiality of clients' input, integrity of the computations, and ensures fair exchange between the clients and the compute nodes. Our experiments show that Kosto is suitable for computation-intensive operations, incurring an overhead as low as 3% over untrustworthy non-enclave execution. I/O-intensive operations are also supported, albeit as a higher overhead (e.g., 1.5×). We leave an incorporation of enclave execution optimizations and defenses against side-channel leakages to future work.

**Acknowledgement.** This research has been supported by the National Research Foundation, Prime Minister's Office, Singapore under its Strategic Capability Research Centres Funding Initiative. We thank the anonymous reviewers their helpful feedback and insightful suggestions. Opinions and findings expressed in this work are those of the authors and do not necessarily reflect the views of any of the sponsors.

## References

1. Airbnb. https://www.airbnb.com
2. Golem. https://golem.network/
3. Intel SGX notes. https://intelsgx.blogspot.com/2016/06/great-notice-about-basics-of-sgx.html
4. Intel SGX SDK for Linux. https://github.com/01org/linux-sgx
5. Intel Software Guard Extensions Enclave Writer's Guide. https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf
6. Intel Software Guard Extensions SSL. https://github.com/intel/intel-sgx-ssl
7. OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/
8. Proof of elapsted time. https://sawtooth.hyperledger.org
9. Public key for Intel attestation service. https://software.intel.com/en-us/sgx/resource-library
10. Raiden network. http://raiden.network
11. SETI@home. https://setiathome.berkeley.edu/
12. SPEC CPU2017 Benchmarks. https://www.spec.org/cpu2017/Docs/overview.html
13. Uber. https://www.uber.com
14. Al-Bassam, M., Sonnino, A., Król, M., Psaras, I.: Airtnt: fair exchange payment for outsourced secure enclave computations. arXiv preprint arXiv:1805.06411 (2018)
15. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: HASP (2013)
16. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. In: OSDI (2014)
17. Bentov, I., Kumaresan, R., Miller, A.: Instantaneous decentralized poker. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10625, pp. 410–440. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70697-9_15
18. Buterin, V.: Ethereum: a next-generation smart contract and decentralized application platform (2014). https://github.com/ethereum/wiki/wiki/White-Paper
19. Chen, C., Maniatis, P., Perrig, A., Vasudevan, A., Sekar, V.: Towards verifiable resource accounting for outsourced computation. In: ACM SIGPLAN Notices (2013)
20. Dang, H., Chang, E.C.: Privacy-preserving data deduplication on trusted processors. In: IEEE CLOUD (2017)

21. Dang, H., Dinh, T.T.A., Chang, E.C., Ooi, B.C.: Privacy-preserving computation with trusted computing via scramble-then-compute. In: PETs (2017)
22. Dang, H., Dinh, T.T.A., Loghin, D., Chang, E.C., Lin, Q., Ooi, B.C.: Towards scaling blockchain systems via sharding. In: SIGMOD (2019)
23. Dang, H., Purwanto, E., Chang, E.C.: Proofs of data residency: checking whether your cloud files have been relocated. In: AsiaCCS (2017)
24. Dinh, T.T.A., Saxena, P., Chang, E.C., Ooi, B.C., Zhang, C.: M2R: enabling stronger privacy in MapReduce computation. In: USENIX Security (2015)
25. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 465–482. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_25
26. Gentry, C., et al.: Fully homomorphic encryption using ideal lattices. In: STOC (2009)
27. Goldreich, O.: Secure multi-party computation. Manuscript, Preliminary version (1998)
28. Gueron, S.: A memory encryption engine suitable for general purpose processors. IACR Cryptology ePrint Archive (2016)
29. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press, Boca Raton (2014)
30. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: CCS (2016)
31. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: ObliVM: a programming framework for secure computation. In: IEEE S&P (2015)
32. Matetic, S., et al.: ROTE: rollback protection for trusted execution. In: USENIX Security (2017)
33. McKeen, F., et al.: Innovative instructions and software model for isolated execution. In: HASP, Article no. 10 (2013)
34. Mucha, M., Sankowski, P.: Maximum matchings via Gaussian elimination. In: FOCS (2004)
35. Ohrimenko, O., et al.: Oblivious multi-party machine learning on trusted processors. In: USENIX Security (2016)
36. Orenbach, M., Lifshits, P., Minkin, M., Silberstein, M.: Eleos: ExitLess OS services for SGX enclaves. In: EuroSys (2017)
37. Poon, J., Dryja, T.: The Bitcoin lightning network: scalable off-chain instant payments (2016)
38. Schuster, F., et al.: VC3: trustworthy data analytics in the cloud using SGX. In: IEEE S&P (2015)
39. Sekar, V., Maniatis, P.: Verifiable resource accounting for cloud computing services. In: WSCC (2011)
40. Shinde, S., Chua, Z.L., Narayanan, V., Saxena, P.: Preventing page faults from telling your secrets. In: AsiaCCS (2016)
41. Shinde, S., Le Tien, D., Tople, S., Saxena, P.: Panoply: low-TCB Linux applications with SGX enclaves. In: NDSS (2017)
42. Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: CCS (2013)
43. Subramanyan, P., Sinha, R., Lebedev, I., Devadas, S., Seshia, S.A.: A formal foundation for secure remote execution of enclaves. In: CCS (2017)
44. Taassori, M., Shafiee, A., Balasubramonian, R.: VAULT: reducing paging overheads in SGX with efficient integrity verification structures. In: ASPLOS (2018)

45. Van Bulck, J., et al.: Foreshadow: extracting the keys to the Intel SGX Kingdom with transient out-of-order execution. In: USENIX Security (2018)
46. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13190-5_2
47. Weisse, O., Bertacco, V., Austin, T.: Regaining lost cycles with HotCalls: a fast interface for SGX secure enclaves. In: ISCA (2017)
48. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: deterministic side channels for untrusted operating systems. In: IEEE S&P (2015)
49. Zhang, F., Eyal, I., Escriva, R., Juels, A., Van Renesse, R.: REM: resource-efficient mining for blockchains. In: USENIX Security (2017)