VeriEdge: Verifying and Enforcing Service Level Agreements for Pervasive Edge Computing

Xiaojian Wang, Ruozhou Yu, Dejun Yang, Huayue Gu, Zhouyu Li

***Abstract*—Edge computing gained popularity for its promises of low latency and high-quality computing services to users. However, it has also introduced the challenge of mutual untrust between user and edge devices for service level agreement (SLA) compliance. This obstacle hampers wide adoption of edge computing, especially in pervasive edge computing (PEC) where edge devices can freely enter or exit the market, which makes verifying and enforcing SLAs significantly more challenging. In this paper, we propose a framework for verifying and enforcing SLAs in PEC, allowing a user to assess SLA compliance of an edge service and ensure correctness of the service results. Our solution, called VeriEdge, employs a verifiable delayed sampling approach to sample a small number of computation steps, and relies on randomly selected verifiers to verify correctness of the computation results. To make sure the verification process is non-manipulable, we employ verifiable random functions to post-select the verifier(s). A dispute protocol is designed to resolve disputes for potential misbehavior. Rigorous security analysis demonstrates that VeriEdge achieves a high probability of detecting SLA violation with a minimal overhead. Experimental results indicate that VeriEdge is lightweight, practical, and efficient.**

***Index Terms*—Edge computing, service-level agreement, verifiable computing, service verification, lightweight**

I. Introduction

With key benefits in low latency and network traffic reduction over the cloud, edge computing enables many real-time services such as metaverse [1], [2], autonomous driving [3], and cloud gaming [4]. Pervasive Edge Computing (PEC) [5] is an emerging paradigm that utilizes diverse distributed devices as edge devices. Device owners can receive monetary compensation by offering their devices for computing services.

However, due to device heterogeneity and dynamicity, utilizing PEC brings new performance and security concerns. The mutual untrust between users and edge devices necessitates a solution to help users verify compliance with service level agreements (SLAs) [6]–[9] and provide assurance to edge devices against service disputes. Limited user resources hinder independent computation and necessitate reliance on external verifiers to assist in computing verification. Yet, selecting a reliable verifier is also a complex task since the verifiers are commonly driven by financial interests and can be untrusted as well. In addition, fairly opportunities for verifier participation are vital for market stability, long-term viability, and preventing dominance or collusion by resource-intensive devices.

Existing work on SLA monitoring [10], [11] mainly focuses on cloud computing, where only one or two entities (cloud providers) need to be monitored instead of many as in PEC. Cryptographic approaches such as verifiable computation [12] are not suitable for PEC due to their high overhead. Incentive-based solutions make assumptions on verifiers' financial interests to ensure correctness [13]. Lightweight execution correctness checking schemes are only for specific tasks [14]. In general, there lacks a framework that addresses SLA compliance for general stateful applications, specifically in PEC environments characterized by heterogeneity and dynamicity.

Motivating example: An example PEC application is large-scale security surveillance using many distributed sensors (traffic cams or vehicle dash cams), assisted by heterogeneous PEC devices for continuous video processing such as object detection, recognition and tracking [15]. A PEC device may not faithfully process the offloading video data while still trying to claim the service reward. To guarantee accountability and reliability of surveillance, monitoring SLA becomes imperative. Other similar scenarios include demand response during sports events [16], edge-assisted autonomous driving [17], etc., which can benefit from PEC, but need SLA assurance to ensure acceptable application or user performance.

To improve overall quality and reliability of services provided in PEC, we propose VeriEdge, a framework for general SLA verification and enforcement in PEC. VeriEdge enables lightweight execution verification via delayed sampling and replay. Using commitments and verifiable random functions (VRFs), the sampling, verifier selection and verification process can be fully verified by an external party such as a service provider or trusted third-party arbitrator. VeriEdge allows a wide range of devices to become candidates for verification, and ensures honest parties prevail in disputes. By opportunistically verifying computational results via randomly selected verifiers, user can gain confidence in an edge device's service without a high verification overhead, or detect misbehavior such as free riding, malicious output, or collusion.

Our main contributions are summarized as follows:

- We propose the VeriEdge framework for SLA verification and enforcement in dynamic PEC environments with untrusted edge devices, which ensures SLA compliance, non-manipulable verification, and fair dispute resolution.
- We design a commit-then-sample procedure to perform lightweight sampling and verification of intermediate computation results with non-repudiability.
- We propose a verifier selection and computation verification procedure based on VRFs, which ensures verifiable fairness and a high probability for misbehavior detection.
- We perform rigorous security analysis to show that Ver-

Wang, Yu, Gu, Li ({xwang244, ryu5, hgu5, zli85}@ncsu.edu) are with NC State University, Raleigh, NC 27606. Yang (djyang@mines.edu) is with Colorado School of Mines, Golden, CO 80401. This research was supported in part by NSF grants 2045539, 2007391, 2008056 and 2008935. The information reported here does not reflect the position or the policy of the federal government.

iEdge can achieve high assurance of SLA verification.

- We implement a prototype of VeriEdge on a stateful object tracking application on commodity edge devices to demonstrate its superior efficiency and scalability.

Organization. §II reviews related work. §III introduces preliminary building blocks. §IV introduces our system model, threat model, and problem statement. §V gives an overview of VeriEdge. §VI presents the detailed design of VeriEdge. §VII presents security analysis of VeriEdge. §VIII shows the performance of VeriEdge. §IX concludes the paper.

II. Related Work

Many have studied SLA monitoring in cloud computing [18]. Dong *et al.* [11] designed smart contracts to enforce faithful cloud execution by cross-checking the result from two clouds. Khan *et al.* [10] studied scalability of blockchain-based SLA monitoring for cloud services. Badshah *et al.* [19] proposed an SLA monitoring framework to offer online monitoring services by a third party. SLA monitoring for cloud is usually limited to monitoring a few fixed cloud entities, relying on a resource-intensive blockchain or assuming the user maintains a service reference implementation [20], impractical for PEC scaling.

Verifiable computation can be used to ensure the correctness of computation results [21], [22]. However, it may suffer from significant overhead due to reliance on expensive cryptography such as fully homomorphic encryption [23], short probabilistically checkable proof (PCPs) [24], and/or succinct arguments [25]. Some lightweight execution correctness checking schemes target specific services [12] such as polynomial delegation [26] and matrix inverse [14].

Incentive-based solutions can help ensure honest task execution. Zhao *et al.* [27] proposed a framework that employs edge tampering detection and edge honesty incentives to effectively detect the authenticity of edge's results and identify dishonest edge nodes. Küpçü *et al.* [13] proposed a scheme that ensures correct execution of outsourced jobs by combining game theory and cryptography. This scheme assumes that the majority of outsourced parties are rational and checks computation results' correctness through complete re-execution.

Recently, a commit-then-selection method is proposed for verifier selection in an edge computing marketplace [28], but it relies on a static verifier list agreed by user and edge server, and lacks verifiability and fairness among market participants. Other existing work overlooks the verifier selection, resulting in lack of a comprehensive framework for SLA compliance monitoring for general stateful applications.

III. Preliminaries

A. Verifiable Random Function

A verifiable random function (VRF) [29] is a public-key pseudorandom function that can prove correctness of its outputs, and consists of three algorithms as follows:

1) $\mathsf{VRFKeyGen}(1^k)\rightarrow(SK,PK)$: It takes security parameter k as input, and outputs a (SK,PK) key pair.
2) $\mathsf{VRFProve}(SK,x)\rightarrow(y,\pi)$: It takes secret key SK and an input x, and generates an output y and its proof π.

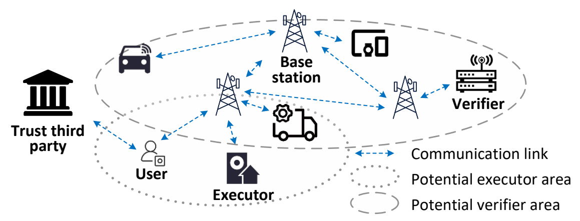


Fig. 1. System architecture and network topology.

3) $\mathsf{VRFVerify}(PK,x,y,\pi)\rightarrow\{0,1\}$. It takes the public key PK, input x, output y and proof π as input, and outputs 1 if $(y,\pi)=\mathsf{VRFProve}(SK,x)$ and 0 otherwise.

A VRF must satisfy three properties: (i) **Uniqueness:** It is infeasible for a probabilistic polynomial time (PPT) adversary to find a pair of values y_1, y_2 with proofs π_1, π_2 such that $\mathsf{VRFVerify}(PK,x,y_1,\pi_1)=\mathsf{VRFVerify}(PK,x,y_2,\pi_2)$ when $y_1\neq y_2$. (ii) **Provability:** If $(y,\pi)=\mathsf{VRFProve}(SK,x)$, then $\mathsf{VRFVerify}(PK,x,y,\pi)=1$. (iii) **Pseudorandomness:** The output y is indistinguishable from a random string by anyone who does not know the secret key SK.

B. Commitment

A commitment conceals a statement and allows for later revelation [30]. The commitment has two phases, commit and open, which correspond to two algorithms $(\mathsf{Com},\mathsf{Verify})$:

1) $\mathsf{Com}(x,r)\rightarrow c$: It takes a statement x and a random value r as input, and outputs a commitment c.
2) $\mathsf{Verify}(c,x,r)\rightarrow\{0,1\}$: It takes a commitment c, a statement x and a random value r as input, and outputs 1 if $c=\mathsf{Com}(x,r)$, and 0 otherwise.

Briefly, a commitment scheme must satisfy two properties: (i) **Binding:** A committer cannot change the statement after the commit phase. (ii) **Hiding:** Only the committer knows the plaintext of the statement before the verify phase begins.

IV. Models and Problem Statement

In this section, we introduce the system model of VeriEdge, including the parties involved and the interactions among them, the threat model, and the problem formulation.

A. System Model

In PEC, a user outsources computing tasks to an edge device. The service provided to the user must satisfy certain requirements in order for the edge device to be paid. These requirements are collectively defined in the SLA. For example, the user may require continuous service from the edge device (*availability*), each task to be completed within a certain time frame (*responsiveness*), all computation steps to be faithfully executed (*soundness*), and the computation result to be accurate (*correctness*). To ensure that the edge device meets the pre-agreed SLA, user has the right to verify SLA compliance during or after the service. If the user suspects SLA violation, she can initiate a dispute with a Trusted Third Party (TTP).

Fig. 1 shows the involved parties and their interactions.

1) **User.** A user is the client of an edge computing service. User submits tasks to the executor (below) and receives

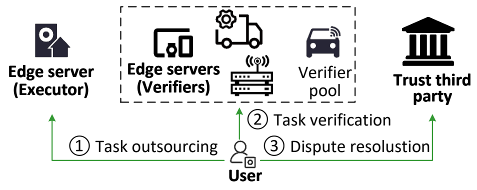


Fig. 2. Overview of VeriEdge.

the results, and wants to make sure that the executor executes the tasks faithfully and returns the correct results.

2) **Executor.** Executor is the edge device that primarily provides a computing service to a user and receives service payment. The executor receives a continuous stream of tasks with user-supplied inputs, executes the tasks based on inputs and internal states, and returns final computation results to the user. In PEC, the executor is typically located close to the user to provide low-latency and high-quality services, such as being connected to the same or a nearby base station as the user. For tasks with stringent real-time requirements, the executor is commonly chosen to be a more powerful edge device to meet the requirements, such as one equipped with advanced CPU, GPU, RAM and/or specialized software.
3) **Verifier.** Many edge devices can voluntarily join the verifier pool to work as verifiers. By participating in this pool, the verifier helps users confirm correct processing of their tasks, and can receive verification payments from users for the verification service. Verifiers can be edge devices that are located farther away from the users, and/or with potentially lower computational capabilities while still being capable of executing verification tasks. For example, verifiers can be cost-efficient but resource-limited devices like smartphones, tablets, etc.
4) **Base station.** The main role of base station is to support communication between user and the edge devices, and provide essential network information to the user such as the lists of available executors and verifiers in the vicinity.
5) **TTP.** TTP is a trusted third party that is only responsible for resolving disputes between users and edge devices, as well as payment settlement. TTP is not involved in the actual task outsourcing and verification process and it can vary in forms, not limited to a physical entity. For example, it can be a smart contract on a blockchain [31].

Fig. 2 shows the interactions involved in the SLA verification and enforcement process, which consists of three stages:

1) **Task outsourcing.** The user submits tasks (inputs) to an executor and receives the computation results (outputs).
2) **Task verification.** After one or many tasks are completed, a user can request intermediate states from executor, and submit the states to one or multiple verifiers for verification. If results from the verifier(s) match the result from the executor, user assumes correctness of executor's execution; otherwise a user may initiate a dispute.
3) **Dispute resolution.** In case of inconsistent results between the verifier(s) and the executor, the user can file a dispute with the TTP. The TTP then resolves the dispute based on submitted evidence, including checking validity of all processes based on the evidence and checking majority vote on the results from executor and verifiers. Payment or punishment can then be settled accordingly.

B. Threat Model and Security Goals

Our threat model primarily considers untrusted or malicious edge devices who try to obtain service or verification payments without faithfully executing the computing tasks, as well as malicious users who want to exploit the dispute process to hamper honest edge devices from receiving proper payments.

Both executors and verifiers are assumed to be untrusted. Executors have the incentive to bypass or guess the results of certain computation steps in the task execution process, aiming to conserve resources, minimize service cost, or avoid violating real-time requirement. For instance, an executor hosting a security surveillance service may only process a small fraction of video frames sent from user device to alleviate its workload, which breaks the user-executor SLA for faithful computation. When user wants to verify the service results, the executor may collude with the verifier to cheat the user. To cheat the user, a verifier colluding with the executor would always return the same (though possibly fake) result for whatever verification task of the user. At any time when a user decides to start or continue outsourcing, we assume there is a sufficiently large set of available verifiers meeting the user's security requirement. We consider a static corruption model following existing work [32], [33], where any collusion between parties is formed before the protocol starts.

A malicious user may manipulate the TTP's judgment during the dispute resolution process to receive services without payment. The user may achieve this by providing different inputs to the verifier(s) and executor, supplying incorrect intermediate states from the executor to the verifier(s), or colluding with certain verifiers to submit fake verification results that are inconsistent with the executor's results.

Base station and TTP are trusted parties performing specific functions. Base station maintains a catalog of local devices and facilitates executor/verifier discovery. TTP resolves dispute based on user-submitted evidences (execution/verification results), but does not perform actual computation for independent verification. We assume they have no ability or incentive to carry out computing verification for users; otherwise bottlenecks and single points of failure may occur, as well as non-trivial overhead to these parties for a large number of users. All parties communicate via authenticated secure channels.

The security goal of VeriEdge is to ensure executor's SLA compliance via dispute-based deterrence, while preventing a malicious user from using dispute as a tool to tamper with an honest executor's service payment. To achieve these goals, VeriEdge needs to ensure the following properties:

1) **SLA compliance.** An edge device cannot forge the proof of services provided to a user without being discovered with a high probability. Continuation of the outsourcing service without termination or dispute by the user indicates satisfaction of all SLA requirements: availability, responsiveness, soundness and correctness.

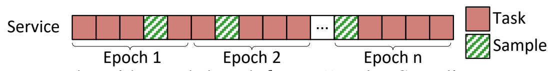


Fig. 3. n epochs with epoch length $l_{\varrho_i} = 5$ tasks. Sampling rate $\eta_s = 0.2$.

2) **Non-manipulable verification.** No entity can manipulate the verifier selection or verification processes, in order to undetectably misbehave or unjustifiably win a dispute.
3) **Dispute resolution.** An honest user can provide valid evidence to win a dispute for detected executor/verifier misbehavior, while a malicious user cannot win a dispute by submitting forged or invalid evidence. The faithful executor and/or verifier(s) can secure the payments eventually, while any malicious executor or verifier faces detection and/or punishment through a dispute or prompt collaboration termination. The user can stop the ongoing work with minimal loss, limited to one round of payment at most, ensuring honest service continuity throughout the long-term incremental service outsourcing process.

C. Service Model and Problem Formulation

We first define the SLA verification and enforcement problem in PEC. Consider a user who wants to use a service s offered by an executor e. A service consists of computation tasks to be executed over time. Each task needs to be completed based on user-supplied inputs and the intermediate state generated from executing the previous tasks, in the meantime updating the intermediate state and/or generating an output to the user. The user's goal is to verify that each task is faithfully executed, based on the correct input and intermediate state, and the received output is correct. If task execution by an edge device does not fulfill availability or responsiveness, user will terminate the service and seek alternative services such as from another edge device. However, the user has no computation power or does not own the proprietary software that the edge hosts, to verify the soundness or correctness by herself.

To verify the execution, user can request some or all tasks to be verified by external verifiers. Specifically, sampling only a subset of tasks for verification reduces the verification overhead and cost incurred in a full replay of all tasks [13]. Verification tasks are sampled in epochs, each epoch consisting of a set of consecutive task executions. The user can specify which tasks to sample in each epoch, and the executor must return the required input/output states of each sampled task for verification. A special case of our model is a stateless service where tasks are independently executed with no intermediate state, which can simplify the verification process. However, we do not restrict a service to be stateless in our model. Fig. 3 shows the relationship among service, task, epoch, and sample.

We use an example object tracking service to provide definitions and give further explanations. In an object tracking service s, a user uploads a series of consecutive frames from a video clip or stream to the executor to detect and track all objects contained in the video. Each frame can be considered as a task. Assuming the user has a total of k frames to process, the task set is $\{t_1, t_2, \dots, t_k\}$. The tasks within a certain interval form an epoch $\varrho_i \triangleq \{t_{i,1}, t_{i,2}, \dots, t_{i,l_{\varrho_i}}\}$. The epoch length l_{ϱ_i} depends on the service and is not necessarily uniform. The user expects to receive an object tracking result for every frame, every few frames, or once per epoch. To track the objects, the service runs an object tracking algorithm Φ_s that maintains an internal state corresponding to the set of objects tracked so far after executing each frame. To verify the service SLA, the user can sample a subset of tasks after each epoch and verify them with external verifiers. The sampling rate of tasks in each epoch is denoted as $\eta_s \in (0, 1]$. At any time, the verifier pool $\mathcal{V}_s$ consists of m verifiers $\mathcal{V}_s \triangleq \{v_1, v_2, \dots, v_m\}$ with $m \geq 3$.

In Definition 1, we define the concept of a q-Algorithm following [13], [28], which forms the strategy of either an honest or a malicious executor or verifier.

Definition 1. ***q-Algorithm:*** *An executor/verifier executes the correct algorithm Φ_s and returns the correct result with probability q, and executes an arbitrary algorithm and returns potentially fake or malicious result with probability $1 - q$. An honest executor/verifier always executes a 1-algorithm.* □

To model collusion between executor and verifiers, we define *pre-collusion ratio* $\delta_s < 1$ as the ratio of verifiers in $\mathcal{V}_s$ which are colluding with e. We similarly define the user-verifier pre-collusion ratio as $\delta_u < 1$.

For a stateful service with executor algorithm Φ_s, we have $\Phi_s(\Theta_{i,j}, I_{i,j}) = (\Theta_{i,j+1}, O_{i,j})$, where $\Theta_{i,j}$ is the state of Φ_s before executing task j of epoch i, $I_{i,j}$ is the user input of task j, and $O_{i,j}$ is the output to user after executing j. Note that "=" indicates equality not assignment. However, it may not always be possible to verify the exact state or output of an execution, such as due to randomness in Φ_s or restriction to run Φ_s on a verifier device. In this case, we also define a pre-negotiated verification algorithm Ψ_s and a correctness checking function C_s. Ψ_s has the same input/output format as Φ_s (and can be Φ_s itself if applicable), and $C_s((\hat{\Theta}, \hat{O}), (\tilde{\Theta}, \tilde{O})) \rightarrow \{0, 1\}$ outputs 1 iff $(\hat{\Theta}, \hat{O})$ matches $(\tilde{\Theta}, \tilde{O})$. Without loss of generality, we assume $C_s(\Phi_s(\Theta_{i,j}, I_{i,j}), \Psi_s(\Theta_{i,j}, I_{i,j})) = 1$ always holds. We define *correctness by verification* as follows:

Definition 2. ***Correctness by verification:*** *Given a stateful or stateless service with executor algorithm Φ_s, a pre-negotiated verification algorithm Ψ_s, and a correctness checking function C_s, along with the verification input $(\Theta_{i,j}, I_{i,j})$, the execution of the executor which output $(\Theta^e_{i,j+1}, O^e_{i,j})$ is correct-by-verification if the following condition is satisfied:*

$$C_s((\Theta^e_{i,j+1}, O^e_{i,j}), \Psi_s(\Theta_{i,j}, I_{i,j})) = 1. \quad \square$$

An honest user would immediately terminate outsourcing to an executor when she finds that the correct-by-verification property does not hold for any task. In other words, continuity of service without interruption or termination indicates that the executor fulfills execution faithfulness by being available, responsive, sound and correct in the execution process. However, as the user cannot even be sure that any verifier she picks will honestly execute the verification algorithm Ψ_s and/or return the faithful result, she would require a probabilistic guarantee for her confidence in the executor's service. We now define *(probabilistic) SLA compliance* as follows:

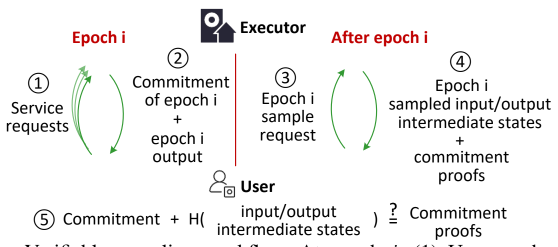

Epoch i
Executor
After epoch i
① Service requests
② Commitment of epoch i + epoch i output
③ Epoch i sample request
④ Epoch i sampled input/output intermediate states + commitment proofs
User
⑤ Commitment + H(input/output intermediate states) ≟ Commitment proofs

Fig. 4. Verifiable sampling workflow. At epoch i, (1) User sends service requests to executor; (2) Executor sends commitment and output of epoch i back to user; After epoch i, (3) User sends sample request for epoch i to executor; (4) Executor returns the input/output and intermediate states along with commitment proofs corresponding to the sample request to user; (5) User checks whether the commitment proof is correct.

Definition 3. ***(Probabilistic) SLA compliance:*** *Given an executor using q-algorithm with $q < 1$, for any pre-collusion ratio δ_s and sampling rate η_s, unterminated outsourcing indicates that user's SLA requirements are met with probability*

$$\Pr[\textit{executor is faithful}] > 1 - \epsilon(n),$$

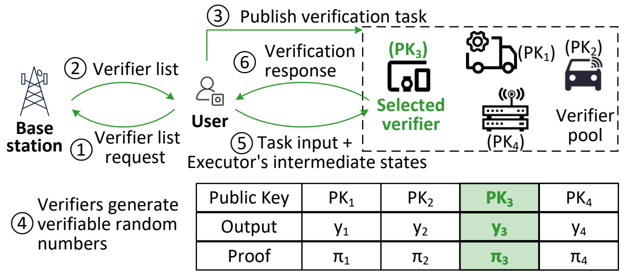
where $\epsilon(n)$ is a negligible function in the number of epochs n, i.e., for all constants c, there exists an integer N such that for all $n > N$, $|\epsilon(n)| < \frac{1}{n^c}$. □

Our goal is to define an edge outsourcing and verification framework which ensures the above property is satisfied for any executor and verifier algorithms and pre-colluding ratios satisfying our system and threat models above.

V. VeriEdge Overview

Considering the unique characteristics of edge computing such as heterogeneity, untrustworthiness, limited resources, and dynamicity, we propose VeriEdge, a verifiable SLA verification and enforcement framework for general edge computing services. As shown in Fig. 2, VeriEdge consists of three components: **verifiable sampling** for sampling the outsourced tasks to be verified; **VRF-based verifier selection** for selecting verifiers to verify the sampled tasks; and **dispute resolution** for resolving disputes between user and executor/verifier(s).

Verifiable sampling. Since users are unable to verify all intermediate results due to limited resources, they need to employ a sampling-based approach that is verifiable for dispute. The goal of verifiable sampling is two-fold: (i) the executor cannot know which tasks will be sampled before executing the tasks and returning the results, to avoid the executor cheating by only faithfully serving the tasks to be sampled; (ii) the executor cannot return wrong intermediate states of a sampled task to mislead verification and evade being detected of its misbehavior. To achieve these goals, the verifiable sampling process has two stages: *commitment* and *verification*. By first letting the executor commit on all intermediate states and task outputs during execution, our process allows the user to post-select random tasks to be checked, and verify that the intermediate states and outputs returned by the executor are the same as being executed. Meanwhile, to counteract the possibility of a malicious user attempting to dispute for refund after receiving the service, we require the user to collect and provide valid evidence that cannot be manipulated by her to initiate a dispute. This ensures faithful dispute resolution by TTP. The verifiable sampling process is illustrated in Fig. 4.



Public Key	PK_1	PK_2	PK_3	PK_4
Output	y_1	y_2	y_3	y_4
Proof	π_1	π_2	π_3	π_4

Fig. 5. VRF-based verifier selection workflow. (1) User sends verifier list request to base station; (2) Base station sends available verifier list to user; (3) User publishes the verification task to the verifier pool; (4) Verifiers generate verifiable random numbers to compete for the verification task; (5) User finds the winner in the verifier pool as the selected verifier and sends the task input and executor's intermediate states to the selected verifier; (6) User gets the verification response from the selected verifier.

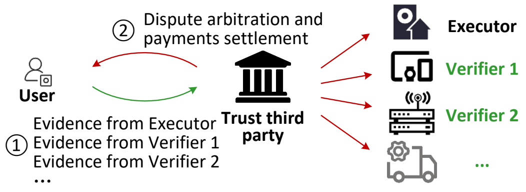


Fig. 6. Dispute resolution workflow. (1) User sends the dispute evidence to TTP; (2) TTP resolves the dispute and settles payment.

VRF-based verifier selection. To verify the sampled tasks are faithfully executed, the user selects from the set of available verifiers to carry out verification. The main goal of verifier selection is to ensure that the selection process is faithful and verifiably random, which serves to guarantee that: (i) the executor cannot be colluding with the same verifier(s) and continuously cheat the user; (ii) similarly the user cannot be colluding with the same verifier(s) to cheat the executor; (iii) all verifiers have an equal chance to ensure fairness of their competition for verification payments. To achieve this, each verifier generates an unpredictable random number as its competition token for each task using a VRF, which cannot be manipulated but can be verified. Fig. 5 depicts the process.

Dispute resolution. When results from the executor and (some of) the verifiers do not match, the user can initiate a dispute to the TTP for arbitration. The user provides the evidence from executor and all selected verifiers, containing the inputs, outputs, and intermediate states of samples, along with the supplementary information for verifiable verifier selection, to initiate a dispute. TTP assesses the evidence, and determines, regulates and/or punishes the responsible party with financial or regulation means. See Fig. 6 for the dispute process.

VI. VeriEdge Design

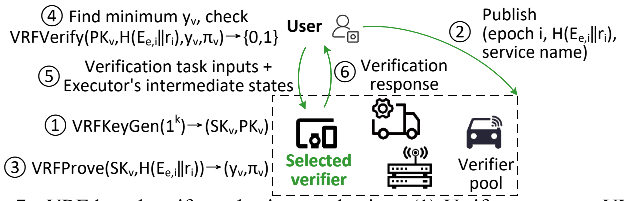
In this section, we present the detailed design of VeriEdge.

A. Task Execution and Verifiable Sampling

In PEC, the lack of mutual trust between the executor and the user necessitates the need to not only ensure that a malicious executor cannot evade detection of misbehavior, but also prevent malicious users from attacking an honest executor by manipulating dispute results. This differs significantly from a similar problem in middlebox verification [20], where the "user" is assumed to be fully trusted. To address this, we first devise a novel sampling, verification and aborting process

Algorithm 1: Task execution and verifiable sampling

```
/*                    During epoch $i$                    */
1  Executor initiates $\gamma_i \leftarrow \emptyset$ and $\mathcal{I}_i \leftarrow \emptyset$;
2  for task $t_{i,j}$ in epoch $\varrho_i$ do
3      User sends $(I_{i,j})_u$ to executor;
4      Executor saves $I_{i,j}$ in $\mathcal{I}_i$;
5      Executor runs algorithm $\Phi_s$ on $I_{i,j}$, and stores
         $S_{i,j} \triangleq (H(I_{i,j}), \Theta_{i,j}, \Theta_{i,j+1}, O_{i,j})$ in $\gamma_i$;
6      Executor sends $(H(I_{i,j}), O_{i,j})_e$ to user;
7  Executor constructs Merkle tree $M_i$ with all $S_{i,j} \in \gamma_i$;
8  Executor sends the root of $M_i$ to user;
/*                    After epoch $i$                    */
9  User sends executor a sample request $\nu_i \triangleq (epoch\ i,$
     $\{H(I_{i,k}) \mid k \in \mathcal{K}_i\}, H(r_i))_u$, where $\mathcal{K}_i$ contains
     sample indices for epoch $i$ and $r_i$ is a random number;
10 Executor generates evidence $E_{e,i} \triangleq$
     $(\{(H(I_{i,k}), \Theta_{i,k}, \Theta_{i,k+1}, O_{i,k}) \mid k \in \mathcal{K}_i\}, H(r_i))_e$
     based on the request $\nu_i$ and proof of evidence $\chi_i$
     from $M_i$, and sends $\chi_i$ and $E_{e,i}$ to user;
11 (Optionally) Executor can delete local $\mathcal{I}_i$, $\gamma_i$ and $M_i$;
12 for sample $H(I_{i,k})$ in $\nu_i$ do
13     if $E_{e,i}.H(I_{i,k}) = H(I_{i,k})$ and proof of $H(I_{i,k})$ in
         $\chi_i$ is valid for $(H(I_{i,k}), \Theta_{i,k}, \Theta_{i,k+1}, O_{i,k})$ of
         $E_{e,i}$ then User can continue using the service;
14     else User aborts the service.
```


④ Find minimum y_v, check
VRFVerify(PK_v,$H(E_{e,i}||r_i)$,y_v,π_v)→{0,1}
User
② Publish
(epoch i, $H(E_{e,i}||r_i)$,
service name)
⑤ Verification task inputs +
Executor's intermediate states
⑥ Verification
response
① VRFKeyGen(1^k)→(SK_v,PK_v)
③ VRFProve(SK_v,$H(E_{e,i}||r_i)$)→(y_v,π_v)
Selected
verifier
Verifier
pool
Fig. 7. VRF-based verifier selection mechanism. (1) Verifiers generate VRF public-secret key pair; (2) User publishes verification tasks to the verifiers in verifier pool; (3) Verifiers generate VRF output and its corresponding proof based on the verification task; (4) User finds the minimum output among all the outputs and checks the correctness of the proof, then chooses the corresponding verifier as the selected verifier; (5) User sends the task input and the intermediate states from executor to the selected verifier; (6) User gets the verification response from the selected verifier.

based on proofs and digital signatures to ensure truthful evidence collection for later dispute resolution, as in Fig. 4.

At the start of the service, user selects an edge device as the executor. The executor can be chosen using various existing computation offloading schemes, *e.g.*, resource-allocation based [34]–[36], game theory based [37], crowdsourcing based [38], learning based approaches [39]. User then submits the task inputs to executor as a sequence in each epoch. To ensure that executor cannot manipulate the outsourcing and verification process, the executor is required to **commit** to each and every task's before and after states and input/output values. All commitments in each epoch are then organized in a Merkle tree, with each leaf node representing the execution of a task. The Merkle tree root is sent to the user upon completion of each epoch. After an epoch, the user will **sample** random tasks from the completed epoch, ask the executor to provide input/output and intermediate states of the tasks, along with Merkle tree proofs for verification. The proofs ensure that the returned input/output and intermediate states correspond to the sampled tasks' execution as committed in the Merkle tree.

An implicit step in our protocol design (including in subsequent subsections) is that all messages are signed and immediately checked upon reception to make sure the signatures are valid. This ensures authenticity of the messages, especially when they are presented as evidence for dispute resolution. We use $(\cdot)_u$, $(\cdot)_e$, $(\cdot)_v$, and $(\cdot)_{bs}$ to represent a message signed by the user, executor, verifier, and base station, respectively.

Algorithm 1 describes the verifiable sampling process. When epoch i starts, user submits signed tasks to executor. Executor runs Φ_s on each task and stores the hash of the input $H(I_{i,j})$, intermediate states $\Theta_{i,j}, \Theta_{i,j+1}$ and output $O_{i,j}$ in its local memory γ_i (lines 2–5). It then sends the hashed task input plus the output to user with its signature (line 6). At the end of epoch i, the executor constructs a Merkle tree M_i using the saved data and obtains the root of M_i (line 7). The executor then returns this root as a commitment to the user (line 8). After epoch i, the user forms a set $\mathcal{K}_i$ by specifying desired sample indices for epoch i, and sends a signed sample request ν_i to the executor (line 9). The request also contains hash of a user selected random number r_i, which will be combined with the sample response from the executor for verifiable verifier selection (Algorithm 2). The executor then constructs proofs by traversing the Merkle tree M_i and extracting the sibling nodes of the requested samples. The executor also retrieves the corresponding input/output and the intermediate states both before and after each sample from its local memory γ_i. Then the executor returns proof χ_i and signed evidence $E_{e,i}$ to the user for verification (line 10). This completes the executor's duty for epoch i (except in possible dispute), and the executor can claim its service payment by providing a user's signed sampling request and its signed response as a billing invoice. Optionally, the executor can delete the locally stored data of epoch i to conserve storage space (line 11). Upon receiving the χ_i and $E_{e,i}$ from the executor, user verifies proofs and $E_{e,i}$ contents for each sample, and continues the service if they match, or terminates the service if not (lines 12–14).

B. VRF-based Verifier Selection

Considering the high dynamicity of edge devices in the PEC scenario, where new edge devices can join at any time and existing ones can leave, maintaining a static verifier pool is impractical. Furthermore, the willingness of edge devices to participate in specific types of tasks can vary over time. For example, their inclination to participate may change with workload or they may be more interested in certain types of tasks. Hence, a dynamic mechanism is needed to select verifiers while ensuring that every willing verifier has a fair chance of being selected and earning rewards from performing verification. This motivates them to remain engaged and contributes to sustaining the entire ecosystem. Meanwhile, the verifier selection process should be verifiable, meaning that the TTP can verify that the selection process is fair and not manipulated. This prevents users from intentionally selecting

Algorithm 2: VRF-based verifier selection

```
1  Base station maintains an active verifier list $\mathcal{V}_s$;
2  User sends a verifier list request $\varpi_{u,i} \triangleq (epoch\ i, H(E_{e,i}||r_i), s_{name})_u$ to base station and gets the signed verifier list $B_{u,i} \triangleq (\varpi_{u,i}, \mathcal{V}_s)_{bs}$;
3  User sends $\varpi_{u,i}$ to all the verifiers in $\mathcal{V}_s$;
4  Each verifier $v$ runs $\mathsf{VRFProve}(SK_v, H(E_{e,i}||r_i)) \rightarrow (y_v, \pi_v)$ and sends $P_{v,i} \triangleq (PK_v, y_v, \pi_v)_v$ to user;
5  Upon receiving all the $P_{v,i}$ from verifiers, user constructs a proof list $\mathcal{P}_i \triangleq \{P_{v,i} \mid v \in \mathcal{V}_s\}$;
6  User finds the verifier $v^* \leftarrow \arg\min\{y_v \mid v \in \mathcal{P}_i\}$;
7  User verifies selected verifier's legitimacy by invoking $\mathsf{VRFVerify}(PK_{v^*}, H(E_{e,i}||r_i), y_{v^*}, \pi_{v^*}) \rightarrow \{0,1\}$;
8  User sends $(\{(\Theta_{i,k}, I_{i,k}) \mid k \in \mathcal{K}_i\})_u$ to $v^*$;
9  Selected verifier $v^*$ processes the verification task and sends the verifier evidence $E_{v,i} \triangleq (\{(H(I_{i,k}), \Theta_{i,k}, \Theta_{i,k+1}, O_{i,k}) \mid k \in \mathcal{K}_i\})_v$ back to user.
```

colluding verifiers to cheat the executor via malicious disputes. To achieve these, we propose a VRF-based verifier selection mechanism, as shown in Fig. 7. See Algorithm 2 for details.

Verifiers interested in verification tasks of s can join the verifier pool via the base station. The base station maintains an active verifier list $\mathcal{V}_s$, which includes the identities of all verifiers willing to participate (line 1). To get the verifier list from base station, user sends a signed verifier list request $\varpi_{u,i}$ to base station and gets the signed verifier list in return (line 2). According to this verifier list, the user then publishes $\varpi_{u,i}$ to the verifiers in the verifier pool and waits for their responses (line 3). Note that $\varpi_{u,i}$ contains random number r_i which is not manipulable by the executor (since the executor only knows the hash of it), and $E_{e,i}$ constructed by the executor which is not manipulable by the user. The hash $H(E_{e,i}\|r_i)$ thus serves as a publicly verifiable coin guaranteeing randomness of the verifier selection result for both the user and the executor. Each verifier v that receives $\varpi_{u,i}$ calculates $\mathsf{VRFProve}(SK_v, H(E_{e,i}||r_i)) \rightarrow (y_v, \pi_v)$ and sends $P_{v,i} \triangleq (PK_v, y_v, \pi_v)_v$ back to user (line 4), where PK_v represents verifier v's public key, y_v denotes its VRF output, and π_v represents the proof of y_v. Upon receiving responses from all verifiers in $\mathcal{V}_s$, user constructs a proof list $\mathcal{P}_i \triangleq \{P_{v,i} \mid v \in \mathcal{V}_s\}$, and finds the verifier v^* with the smallest y_v in $\mathcal{P}_i$ (lines 5–6) as the selected verifier. Note that this step can be extended to randomly select any $V > 1$ verifiers with V smallest values of y_v, as used in Algorithm 3. The user checks the correctness of the output and proof from the selected verifier and sends the inputs and intermediate states of the verification task to the selected verifier (lines 7–8). Then the selected verifier processes the tasks and generates the signed verifier evidence and sends it back to user (line 9).

C. Dispute Resolution

Let V be the number of verifier results per sample required to initiate a dispute, pre-negotiated for the service. To obtain enough evidence for dispute, user may sequentially or batch select a set of verifiers $\mathcal{V}_{s,i}$ to independently verify samples

Algorithm 3: Dispute resolution

```
   /*        User initiates dispute        */
1  User initiates $\mathcal{V}_{s,i} \leftarrow$ {first selected verifier};
2  while $|\mathcal{V}_{s,i}| < V$ do
3      User selects verifier $v$ with the next smallest $y_v$ and gets new evidence from $v$ according to lines 6–9 in Algorithm 2, and inserts $v$ to $\mathcal{V}_{s,i}$;
4  User sends evidences $E_{e,i}$, $\{E_{v,i} \mid v \in \mathcal{V}_{s,i}\}$ and $E_{u,i} \triangleq (epoch\ i, B_{u,i}, \mathcal{P}_i, \{H(I_{i,k}) \mid k \in \mathcal{K}_i\}, r_i)_u$ to TTP;
   /*  TTP starts dispute resolution  */
5  (1) TTP checks that all signatures in $E_{e,i}$, $E_{u,i}$, and $\{E_{v,i} \mid v \in \mathcal{V}_{s,i}\}$ (including signatures of messages included in each evidence) are valid;
6  (2) TTP verifies VRF-based verifier selection process for all selected verifiers in $\mathcal{V}_{s,i}$ according to $B_{u,i}$;
7  (3) For any sample $H(I_{i,k})$ in $E_{u,i}$, TTP checks if executor's and verifiers' evidences also contain $H(I_{i,k})$;
8  if any of (1)–(3) fails then return "Malicious user";
9  for sample $H(I_{i,k})$ in $E_{u,i}$ do
10     TTP performs majority vote on all executor/verifiers' results $\mathcal{X}_{i,k}$, and gets $X^*_{i,k} \triangleq \arg\max_{X \in \mathcal{X}_{i,k}} \{\sum_{X' \in \mathcal{X}_{i,k}} C_s(X, X')\}$;
11     if $\sum_{X' \in \mathcal{X}_{i,k}} C_s(X^*_{i,k}, X') > \frac{V+1}{2}$ and executor result $X^e_{i,k}$ gives $C_s(X^e_{i,k}, X^*_{i,k}) = 0$ then
12         return "Malicious executor";
13 return "No malicious behavior detected".
```

in epoch i using Algorithm 2. In the sequential method, user selects the rest $V-1$ verifiers only when the first verifier's result does not match with the executor's, while in the batch method user directly selects V verifiers in first round. Since the sequential method incurs lower cost when both parties are honest, we assume the sequential method is used to save cost.

Algorithm 3 describes the dispute resolution process. If there is a discrepancy between results provided by the first verifier and the executor, user repeatedly selects $V-1$ verifiers from the verifier pool to perform independent verifications (line 3). Next, the user constructs evidence $E_{u,i}$, which includes the epoch i for the dispute, the verifier list signed by base station $B_{u,i}$, the proof list $\mathcal{P}_i$ containing outputs and proofs from all verifiers in epoch i, the hash values of all samples' inputs, and the random number r_i. The user signs $E_{u,i}$ and sends it along with the evidence from the executor and evidence from verifiers to the TTP for resolution (line 4).

In dispute resolution, TTP first verifies validity of all submitted evidence, including: (1) all signatures of evidences are valid (line 5), (2) verifier selection is faithful based on the valid $H(E_{e,i}\|r_i)$ and all available verifiers' random numbers that match the base station-signed verifier list (line 6), and (3) all execution and verification results are claimed to be generated from the same user input and input state (line 7). If any of these checks fail, TTP considers the user as malicious (line 8). TTP checks result consistency from executor and verifiers using majority voting (lines 9–12). Let $\mathcal{X}_{i,k} = \{(\Theta_{i,k+1}, O_{i,k})\}$ denote the set of all outputs for task k in epoch i from the executor and all selected verifiers. Specifically, for each sample in the user's evidence, TTP first calculates the "most common" result $X^*_{i,k} \in \mathcal{X}_{i,k}$ for the sample utilizing the C_s function. Then TTP checks if the result $X^*_{i,k}$ constitutes a majority vote. If so, and yet the executor's result does not match with $X^*_{i,k}$, the executor is flagged as malicious. If no majority can be formed, additional verification is needed by utilizing the cloud for a full replay, and/or involving the service provider. Payment or punishment settlement then depends on the dispute result.

VII. Security Analysis

Sound and correct execution. The soundness of execution and correctness of the results are ensured through two key mechanisms. Firstly, the commitment ensures that the intermediate states have not been tampered with by the executor after execution. Secondly, matching the task inputs and intermediate states from both the executor and verifiers serves to verify the correctness of the verification inputs. This ensures that the verification inputs are consistent with the execution.

Given the commitment, a user can immediately detect an executor who modifies the execution trace to try to evade detection of misbehavior such as lazy execution.

For correctness by verification, first consider a single epoch. Based on our model, the probability that all samples in epoch i are processed faithfully is $q^{l_{\varrho_i}\eta_s}$. In this case, the executor will remain undetected regardless of the verifier's behavior.

If at least one sample is not faithfully processed by executor, consider two cases. First, if the executor colludes with at least one of the selected verifiers, they will consistently return a matched result. Since verifier selection is verifiably random, the probability of non-detection of executor's behavior is related to the fraction of verifiers who participate in the collusion, given as $(1 - q^{l_{\varrho_i}\cdot\eta_s}) \cdot \delta_s$. Second, verifiers not colluding with the executor will return unmatched results, leading to a detection probability of 1. Overall, the probability of the executor not being caught in one epoch i is $q^{l_{\varrho_i}\cdot\eta_s} + (1 - q^{l_{\varrho_i}\cdot\eta_s}) \cdot \delta_s$. The following theorem shows the probabilistic SLA compliance against a malicious executor:

Theorem 1. *The probability of a malicious executor remaining undetected throughout n epochs, referred to as the* ***escape probability****, is $\prod_{i=1}^{n} \left(q^{l_{\varrho_i}\cdot\eta_s} + (1 - q^{l_{\varrho_i}\cdot\eta_s}) \cdot \delta_s\right)$.* □

Non-manipulable dispute. Under our threat model, a user can only try to attack the executor by maliciously issuing and winning disputes to request refund and/or cause financial loss to the executor. Note that this requires valid signatures of all messages submitted as evidence; otherwise the dispute will always fail. Evidences sent by honest executor/verifiers cannot be forged by the user due to the need for valid signatures.

To falsely accuse an executor of misbehavior, the user must initiate a dispute and win a majority vote from the verifiers against the executor. However, since the verifier selection is verifiably random with the use of VRFs, the user's only winning chance is to randomly select colluding verifiers as the majority in $\mathcal{V}_{s,i}$, which reduces with increasing V. The following theorem shows the upper bound probability that a malicious user can cheat the executor per epoch:

TABLE I
Evaluation platforms

Platform	CPU	OS	Memory
HWI-AL00 Phone	Hisilicon Kirin 960 2.36GHz, 8 cores	Android 8.0.0 (ARM)	6GB
Raspberry Pi 4 Model B	Broadcom BCM2835 700MHz, 4 cores	Ubuntu 22.10 (ARM)	3.7GB
Laptop	Apple M1 Pro	Ventura 13.3.1 (ARM)	16GB
Desktop	AMD Ryzen 3945WX 4.0GHz, 12 cores	Ubuntu 20.04.5 LTS (x86)	256GB

Theorem 2. *Assume the SLA contract requires V verifiers for a majority vote. The probability that a malicious user can win the dispute for an arbitrary epoch is at most $\delta_u^{\lfloor \frac{V}{2} \rfloor + 1}$.* □

VIII. Performance Evaluation

A. Implementation and Experiment Settings

We evaluated the performance of VeriEdge by implementing an object tracking service on edge devices, which utilized state-of-the-art real-time multi-object, segmentation and pose tracking with Yolov8 [40]. The intermediate states of the service consisted of the bounding box coordinates and the index of each object in each frame. The pre-negotiated verification algorithm tracks objects using executor's intermediate states and user's input, while the correctness checking algorithm verifies consistency between verifiers and executor's object tracking results. We tested the performance of VeriEdge on the KITTI dataset [41]. We utilized the gRPC framework (v1.56.0) [42] to implement the communication between parties. All protocols were implemented in Python. The VRF functionality was implemented based on the RSA Full Domain Hash VRF [43]. We implemented the Merkle tree using the merkletools library (v1.0.3) [44]. SHA-256 was used for hashing, and RSA digital signature [45] for message signing.

We evaluated the performance of VeriEdge on four platforms as shown in Table I. By default, the user client was run on the Raspberry Pi, while other parties were run on the desktop. The object tracking service ran for 100 epochs, with each epoch consisting of 100 frames. The default setting of VeriEdge had a verifier pool with 30 verifiers, with a predefined verifier number of 2 and a sampling rate of 0.01. For comparison, we also implemented a full replay approach (the **Baseline**) without sampling, where the result of every single task was independently verified by the verifiers.

B. Evaluation Results

1) Communication and Computation Overhead: Table II shows the communication overhead and execution time of VeriEdge and Baseline. Communication overhead was measured in terms of the message size exchanged between parties. Execution time was evaluated by assessing the delay in each step. Compared to raw application without verification, VeriEdge introduced extra communication costs including Merkle tree root, user sample request, and proofs from the executor, which increased communication cost by 0.0028%. Regarding execution time, compared to only returning the outputs of each epoch, VeriEdge increased the execution time by 1.14%. Compared with Baseline, VeriEdge achieved significant *savings*

TABLE II
COMMUNICATION COST AND EXECUTION TIME OF VERIEDGE AND BASELINE FOR ONE EPOCH

Description	Communication Cost (bytes)[*]			Execution Time (ms)		
	Message	**VeriEdge**	**Baseline**	**Step**	**VeriEdge**	**Baseline**
Obtaining inputs for verification (verifiable sampling in VeriEdge)	User → executor task request	86759200	86759200	Executor Merkle tree construction	0.33	0.34
	Executor → user Merkle tree root	113	113	User got results from executor	2778.76	3719.26
	Executor → user results	72	72	Executor generated proof	**28.45**	30.81
	User → executor sample request	515	-	User got proof response from executor	**30.29**	34.48
	User got proof from executor	**1789**	70124	User validated proofs from executor	**1.07**	79.56
VRF-based verifier selection	User → BS verifier list request	330	330	User got BS verifier list	61.82	90.27
	BS → user verifier list response	832	832			
				Verifier key generation	124.27	96.21
	User → first verifier task request	**2612023**	86703281	User found first verifier	30.56	30.56
				User got results from first verifier	**643.78**	19762.93
	First verifier → user response	**552**	14160	User checked correctness	**0.07**	221.44
Dispute resolution	User → second verifier request	**2612023**	86703281	User got results from second verifier	**642.77**	22205.78
	Second verifier → user response	**552**	14160			
	User → TTP dispute request	**56700**	104865	User got the dispute result from TTP	20.04	108.86
	TTP → user dispute response	16	17			

[*] Message sizes of trivial text messages such as "success" or "fail" are omitted. The **bold** text highlights the steps where VeriEdge saves overhead and time compared to Baseline. The underlined text represents the inherent overhead of the object tracking application. BS denotes base station.

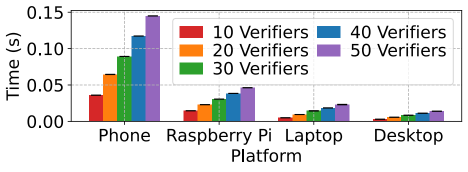


Fig. 8. VRF-based verifier selection time.

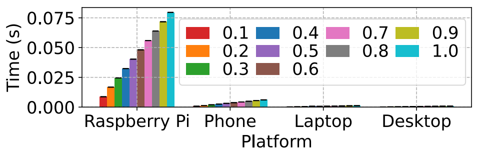


Fig. 9. Merkle tree commitment checking time with different sampling rates.

in execution time and communication cost across verifiable sampling, VRF-based verifier selection, and dispute resolution.

To evaluate the computation overhead on the user side, we collected the execution time of VRF-based verifier selection and Merkle tree commitment checking process on four different platforms. The results are presented in Figs. 8 and 9, with error bars representing the 95% confidence interval obtained from running each experiment 1000 times. Fig. 8 shows the delay of VeriEdge for selecting a verifier and verifying the proof of the verifier for one epoch. With more verifiers in the pool, the time taken by the user to find the final verifier with the minimum output also increased. Additionally, platforms with higher configurations exhibited shorter processing times. The longest time taken was less than 150ms among the four platforms. Fig. 9 illustrates the computation overhead of verifying the proofs returned from executor at different sampling rates.[1] As the sampling rate increased, the number of proofs needed to be verified increased, leading to longer checking time. Among the four platforms, the longest time was less than 80ms even at a sampling rate of $\eta_s = 1$.

2) Security Analysis: We numerically analyzed the impact of different honest rates, pre-collusion ratios, sampling rates, and the number of epochs on the executor escape probability and illustrated the results on a logarithmic scale in Fig. 10. The

[1]We note that the opposite comparison of VRF and Merkle tree operations on Raspberry Pi versus Phone was caused by different performance optimization of the same library on different operating systems.

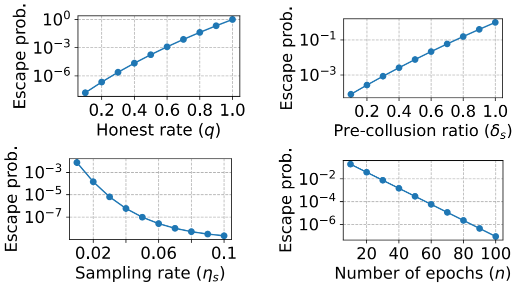


Fig. 10. Executor escape probability vs. different security parameters.

default settings are: the honest rate $q = 0.7$ (in Definition 1), the sampling rate $\eta_s = 0.01$, the pre-collusion ratio $\delta_s = 0.5$, the number of epochs $n = 30$, and the epoch length $l_{\varrho_i} = 100$. With a higher honest rate, the escape probability increased as more faithfully handled tasks were likely to be sampled. With a higher pre-collusion ratio, the escape probability increased due to a greater chance of the user randomly selecting a colluding verifier. Conversely, with an increase in the sampling rate, the escape probability decreased since it became more likely to sample tasks that were not faithfully handled. Even with a small sampling rate, a relatively low escape probability can be achieved. Finally, an increase in the number of epochs contributed to a higher probability of at least one epoch being caught, consequently reducing the executor escape probability.

IX. CONCLUSION

This paper proposed VeriEdge, a general SLA verification and enforcement framework for PEC. We designed a commit-then-sample scheme to enable lightweight execution verification with non-repudiability. Considering the heterogeneity and dynamicity of PEC, we proposed a VRF-based verifier selection protocol to ensure fair and random selection of verifiers and defend against potential collusion. A dispute resolution protocol was also proposed to ensure validity of the verification process and deter SLA violation. Through security analysis, we demonstrated that VeriEdge can achieve high assurance of SLA verification. Experiments on commodity devices showed the superior communication and computation efficiency of VeriEdge compared to full replay-based verification.

References

[1] H. Lee, J. Lee, D. Kim, S. Jana, I. Shin, and S. Son, "Adcube: Webvr ad fraud and practical confinement of third-party ads." in *USENIX Security Symposium*, 2021, pp. 2543–2560.

[2] H. Duan, J. Li, S. Fan, Z. Lin, X. Wu, and W. Cai, "Metaverse for social good: A university campus prototype," in *ACM MM*, 2021, pp. 153–161.

[3] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, and I. Stoica, "Ekya: Continuous learning of video analytics models on edge compute servers," in *USENIX NSDI*, 2022, pp. 119–135.

[4] Z. Meng, T. Wang, Y. Shen, B. Wang, M. Xu, R. Han, H. Liu, V. Arun, H. Hu, and X. Wei, "Enabling high quality real-time communications with adaptive frame-rate," in *USENIX NSDI*, 2023, pp. 1429–1450.

[5] Z. Ning, P. Dong, X. Wang, S. Wang, X. Hu, S. Guo, T. Qiu, B. Hu, and R. Y. Kwok, "Distributed and dynamic service placement in pervasive edge computing networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1277–1292, 2020.

[6] V. Narasayya and S. Chaudhuri, "Multi-tenant cloud data services: State-of-the-art, challenges and opportunities," in *ACM SIGMOD*, 2022, pp. 2465–2473.

[7] A. Habib, S. Fahmy, S. R. Avasarala, V. Prabhakar, and B. Bhargava, "On detecting service violations and bandwidth theft in qos network domains," *Computer Communications*, vol. 26, no. 8, pp. 861–871, 2003.

[8] L. Yang, J. Cao, H. Cheng, and Y. Ji, "Multi-user computation partitioning for latency sensitive mobile cloud applications," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2253–2266, 2014.

[9] A. Banchs, M. Fiore, A. Garcia-Saavedra, and M. Gramaglia, "Network intelligence in 6g: Challenges and opportunities," in *ACM MobiArch*, 2021, pp. 7–12.

[10] K. M. Khan, J. Arshad, W. Iqbal, S. Abdullah, and H. Zaib, "Blockchain-enabled real-time sla monitoring for cloud-hosted services," *Cluster Computing*, pp. 1–23, 2022.

[11] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. Van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *ACM CCS*, 2017, pp. 211–227.

[12] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *IEEE SP*, 2015, pp. 253–270.

[13] A. Küpçü, "Incentivized outsourced computation resistant to malicious contractors," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 6, pp. 633–649, 2015.

[14] C. Hu, A. Alhothaily, A. Alrawais, X. Cheng, C. Sturtivant, and H. Liu, "A secure and verifiable outsourcing scheme for matrix inverse computation," in *IEEE INFOCOM*, 2017, pp. 1–9.

[15] T. Zhang, A. Chowdhery, P. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *ACM MobiCom*, 2015, pp. 426–438.

[16] J. Xing, H. Ai, L. Liu, and S. Lao, "Multiple player tracking in sports video: A dual-mode two-way bayesian inference approach with progressive observation modeling," *IEEE Transactions on Image Processing*, vol. 20, no. 6, pp. 1652–1667, 2010.

[17] X. Zhang, A. Zhang, J. Sun, X. Zhu, Y. E. Guo, F. Qian, and Z. M. Mao, "Emp: Edge-assisted multi-vehicle perception," in *ACM MobiCom*, 2021, pp. 545–558.

[18] B. Yin, Y. Cheng, L. X. Cai, and X. Cao, "Online sla-aware multi-resource allocation for deadline sensitive jobs in edge-clouds," in *IEEE GLOBECOM*, 2017, pp. 1–6.

[19] A. Badshah, A. Jalal, U. Farooq, G.-U. Rehman, S. S. Band, and C. Iwendi, "Service level agreement monitoring as a service: an independent monitoring service for service level agreements in clouds," *IEEE Big Data*, 2022.

[20] X. Zhang, H. Duan, C. Wang, Q. Li, and J. Wu, "Towards verifiable performance measurement over in-the-cloud middleboxes," in *IEEE INFOCOM*, 2019, pp. 1162–1170.

[21] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish, "A hybrid architecture for interactive verifiable computation," in *IEEE SP*, 2013, pp. 223–237.

[22] J. Kilian, "Improved efficient arguments: Preliminary version," in *Springer CRYPT0*, 1995, pp. 311–324.

[23] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Springer CRYPTO*, 2010, pp. 465–482.

[24] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer, "On the concrete efficiency of probabilistically-checkable proofs," in *ACM STOC*, 2013, pp. 585–594.

[25] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *Springer ASIACRYPT*, 2010, pp. 321–340.

[26] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *IEEE SP*, 2020, pp. 859–876.

[27] Z. Zhao, Y. Zeng, J. Wang, H. Li, H. Zhu, and L. Sun, "Detection and incentive: A tampering detection mechanism for object detection in edge computing," in *IEEE SRDS*, 2022, pp. 166–177.

[28] C. Harth-Kitzerow and G. M. Garrido, "Verifying outsourced computation in an edge computing marketplace," *arXiv preprint arXiv:2203.12347*, 2022.

[29] C. Peikert and J. Xu, "Classical and quantum security of elliptic curve vrf, via relative indifferentiability," in *Springer CT-RSA*, 2023, pp. 84–112.

[30] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[31] Y. Du, Z. Wang, J. Li, L. Shi, D. N. K. Jayakody, Q. Chen, W. Chen, and Z. Han, "Blockchain-aided edge computing market: Smart contract and consensus mechanisms," *IEEE Transactions on Mobile Computing*, 2022.

[32] J. Camenisch, A. Lehmann, G. Neven, and A. Rial, "Privacy-preserving auditing for attribute-based credentials," in *Springer ESORICS*, 2014, pp. 109–127.

[33] V. Shoup, "Practical threshold signatures," in *Springer EUROCRYPT*, 2000, pp. 207–220.

[34] C. Wang, C. Liang, F. R. Yu, Q. Chen, and L. Tang, "Computation offloading and resource allocation in wireless cellular networks with mobile edge computing," *IEEE Transactions on Wireless Communications*, vol. 16, no. 8, pp. 4924–4938, 2017.

[35] M. F. Pervej, R. Jin, and H. Dai, "Resource constrained vehicular edge federated learning with highly mobile connected vehicles," *IEEE Journal on Selected Areas in Communications*, 2023.

[36] S. Misra, R. Tourani, F. Natividad, T. Mick, N. E. Majd, and H. Huang, "Accconf: An access control framework for leveraging in-network cached data in the icn-enabled wireless edge," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 1, pp. 5–17, 2017.

[37] F. Zafari, K. K. Leung, D. Towsley, P. Basu, and A. Swami, "A game-theoretic framework for resource sharing in clouds," in *IFIP WMNC*, 2019, pp. 8–15.

[38] T. Shi, Z. Cai, J. Li, and H. Gao, "Cross: A crowdsourcing based sub-servers selection framework in d2d enhanced mec architecture," in *IEEE ICDCS*, 2020, pp. 1134–1144.

[39] F. Tütüncüoğlu, S. Jošilo, and G. Dán, "Online learning for rate-adaptive task offloading under latency constraints in serverless edge computing," *IEEE/ACM Transactions on Networking*, 2022.

[40] "yolo_tracking," accessed 2023-07-31. [Online]. Available: https://github.com/mikel-brostrom/yolo_tracking

[41] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *IEEE CVPR*, 2012, pp. 3354–3361.

[42] "gRPC: A high performance, open-source universal RPC framework," accessed 2023-07-31. [Online]. Available: https://grpc.io/

[43] "RSA-VRF," accessed 2023-07-31. [Online]. Available: https://github.com/DreamWuGit/RSA-VRF

[44] "pymerkletools: Python tools for creating Merkle trees, generating Merkle proofs, and verification of Merkle proofs," accessed 2023-07-31. [Online]. Available: https://github.com/Tierion/pymerkletools

[45] "PKCS v1.5 (RSA)," accessed 2023-07-31. [Online]. Available: https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html