DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Information Systems

Design and Implementation of a Secure and Scalable Verification Scheme for Outsourced Computation in an Edge Computing Marketplace

Christopher Harth-Kitzerow

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Information Systems

Design and Implementation of a Secure and Scalable Verification Scheme for Outsourced Computation in an Edge Computing Marketplace

Entwurf und Implementierung eines sicheren und skalierbaren Verifizierungsschemas für ausgelagerte Berechnungen in einem Edge-Computing-Marktplatz

Author:	Christopher Harth-Kitzerow
Supervisor:	Prof. Dr. Jörg Ott
Advisor:	Gonzalo Munilla Garrido
Submission Date:	15.12.2020

Acknowledgments

Abstract

Latency-sensitive Internet of Things (IoT) applications need edge computing to overcome their limited computing capabilities. Idle resources at the edge of a network could be utilized to outsource the computation of an IoT device. An edge computing marketplace could enable IOT devices (Outsourcers) to outsource computation to any participating node (Contractors) in their proximity. In return, these nodes receive a reward for providing computation and storage services.

As IoT devices are often computationally weak, it might be difficult for them to verify if responses returned by a third party Contractor are valid. In fact, Contractors have an incentive to return a computationally less expensive probabilistic result, to save resources while still collecting the reward. Likewise, an Outsourcer has no incentive to rightfully pay an honest Contractor after it has received all computational results, and expensive microtransactions prohibit real-time payment.

In this thesis, we propose a verification scheme that enables verification of arbitrary deterministic functions by letting the Outsourcer send random samples to an additional third party Verifier and compare responses with the ones from the Contractor. Our designed verification scheme also provides publicly verifiable proofs for the Contractor and the Verifier that can be used to enforce payment if all sent responses have been correct.

We consider that all described participants that engage in the verification scheme might be dishonest or lazy. Thus, we compiled a list of potential protocol violations that might threaten the integrity or quality of outsourced computation. Our verification scheme provides techniques to prevent or detect each of the identified protocol violations with high probability.

Object detection is a relevant use case for IoT devices and a computationally expensive task. We tested our verification scheme with state-of-the-art pre-trained Convolutional Neural Network models designed for object detection. On all devices, our verification scheme caused less than 1ms computational overhead, and a neglectable network bandwidth overhead of a maximum of 84 bytes per frame, independent of the frame's size. We also implemented a multi-threaded version of our scripts that performs the task of our verification scheme in parallel to the object detection to eliminate any latency overhead of our scheme. We also utilized a non-blocking message pattern to receive and preprocess new frames in parallel to all other threads to achieve high frame rates. We evaluated the performance of outsourced object detection with our verification scheme with a regular mid-range GPU, and a low-end Edge Accelerator and were able to achieve more than 60 frames per second with either hardware. Our tests showed that a cheap setup of a low-end CPU combined with a low-end Edge Accelerator can be used as a Contractor for outsourced object detection.

Kurzfassung

Contents

1. Introduction

Use with pdfLaTeX and Biber.

1.1. Motivation

In an Edge Computing marketplace, we assume that the Outsourcer is a computationally weak IoT device that outsources real-time data to a Contractor to process. The Contractor can be an edge server or any device in proximity to the Outsourcer that has enough computational resources available to execute the assigned function with sufficient QoS. Resources at the edge might be unavailable at times due to the following reasons:

1. High demand
2. No existing edge hardware near requesting device
3. Software/hardware limitations

Certain techniques such as resource reservation can be used to ensure a certain reliability for Outsourcers to being able to find an available contractor at all times. But even if resources in an edge computing marketplace are unreliable, they can be used to provide temporal QoS improvements compared to a service by a cloud server. Thus, IoT devices that are usually served by the cloud such as smart home assistants may be able to temporarily improve latency when outsourcing to an edge server in proximity. The following temporal quality of service improvements can still add value to an application:

1. An IoT device can temporarily improve latency when being served by an edge server in proximity
2. An IoT can transfer large amounts of data quickly and with low cost
3. An IoT device can offload storage to an Edge Server which might compress the data before sending it over the internet
4. An IoT device can turn off long-range communications technology to save energy

The following figure illustrates different temporal QoS improvements.

An example for an outsourcing task in Edge computing scenarios with high relevance for IoT devices is object detection for the following reasons:

1. Reliable object detection requires high computational complexity

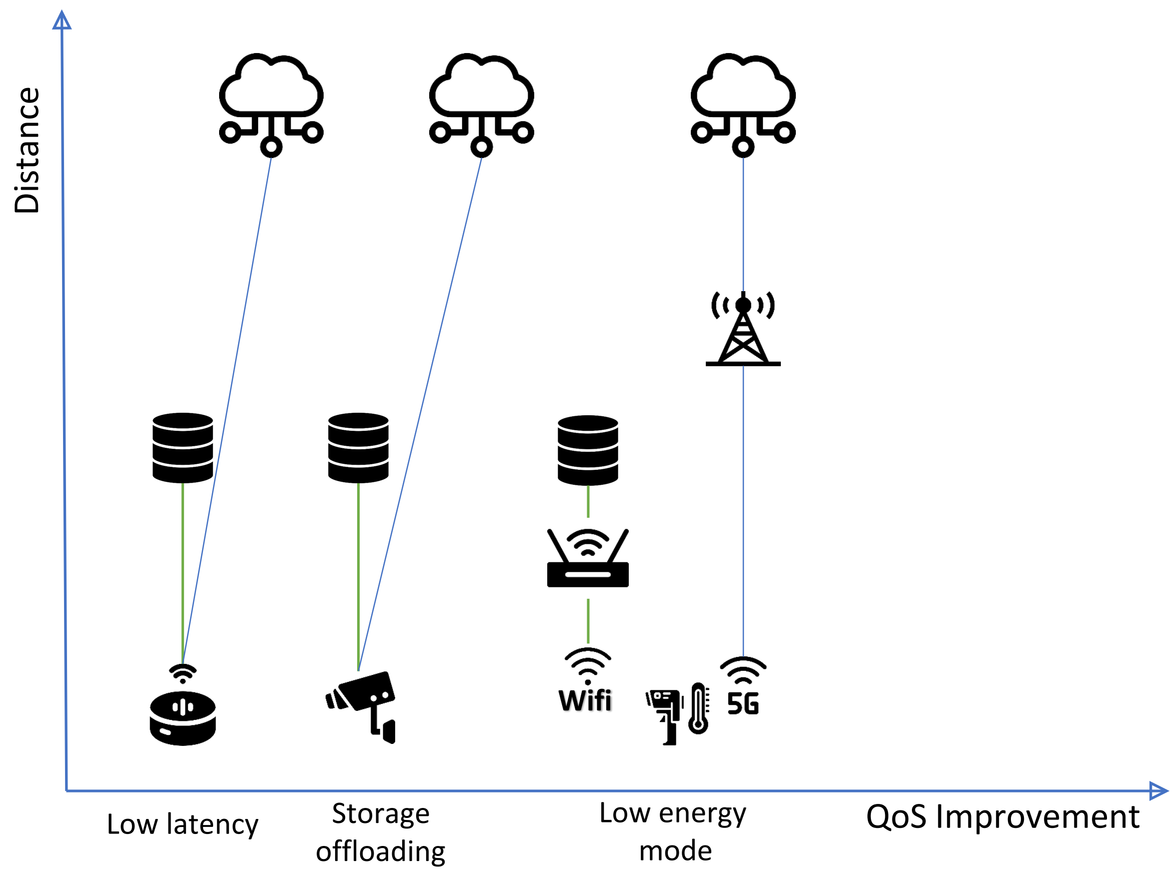


Figure 1.1.: Temporary QoS improvements

2. Object detection with real-time images requires low latency
3. Sending image streams in real-time requires high bandwidth

Also, there are multiple uses cases available for IoT devices that are required to perform object detection.

1. A CCTV tries to identify persons or other objects of interest
2. A connected vehicle tries to identify objects on the road
3. A passenger is displayed real-time information of surrounding objects via augmented reality
4. A robot automatically picks up objects for relocation in a warehouse

The state-of-the-art technique to perform object detection is with a Convolutional Neural Network (CNN). CNNs currently offer the best performance of object detection techniques. Pre-trained models are typically between 100MB and 2GB in size and can go as low as 5MB if designed for specialized APUs. IoT devices might not be able to perform inference themselves due to the following reasons:

1. Lack of storage
2. Lack of computational power
3. Incompatible software

1.2. Assumptions

We assume the following points for outsourced object detection in an Edge Computing marketplace:

1. Contractors (edge servers) are stationary (reappearing actors)
2. Outsourcers (IoT devices) are mobile (reappearing and adhoc actors)
3. Limited set of outsourced task types (reappearing task types)
4. Both parties that agree to an outsourcing contract are rational, expected payoff maximizers
5. The payoff of the players are linear functions of their costs and payments/benefits (and risk-neutral payoff function)
6. Outsourcers pay Contractors for each processed image (this does not mean that micro-transactions or synchronous payment have to be used)
7. Cloud servers distributed pre-trained models to edge server

Additionally, we assume that the other required components of an edge computing marketplace such as resource matching, or a payment scheme is already implemented.

The following figure illustrated a scenario that matches out assumptions and expectations of a typical scenario in an edge computing marketplace:

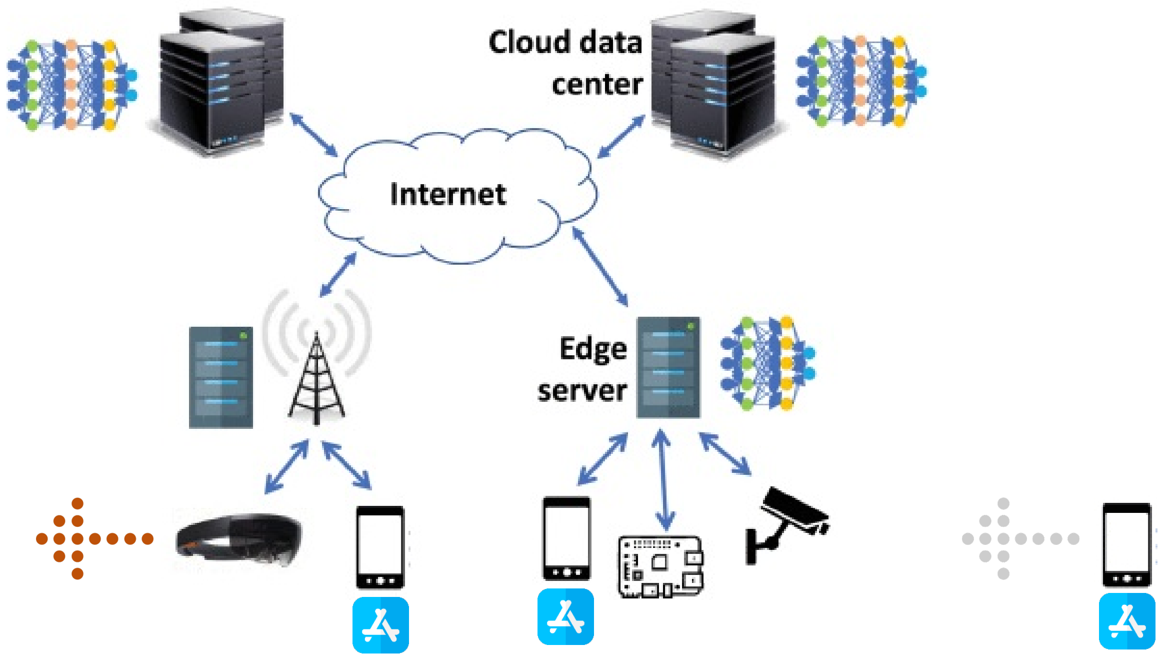


Figure 1.2.: Scenario of an edge computing marketplace with mobile IoT devices outsourcing reappearing object detection task types

1.3. Problem Statement

1.4. Research Objective

We identified the following issues when evaluating existing verification schemes proposed by current academic literature:

1. Some existing verification schemes use approaches such as Fully Homomorphic encryption that are not practical for complex functions in real-time yet.
2. We found no comprehensive list of possible protocol violations in computing marketplaces that threaten the security of a designed verification scheme.
3. Existing verification schemes often rely on trusted third parties (TTPs) that might be unavailable or become a bottleneck in real-world use cases. Other verification schemes utilize Blockchains instead of trusted third parties to record communication between participants. However, frequent Blockchain transactions cause high transaction costs and high computational overhead from a system perspective.
4. Existing verification schemes usually assume that the Outsourcer or a third-party Verifier is a trusted entity, or only design the security of their scheme from the perspective of one party rather than from an ecosystem perspective.

The objective of our research is to solve these issues by first identifying a verification approach that is practicable today and only causes low computational overhead for each participant. Afterward, we aim to compile a comprehensive list of potential protocol violations that this approach might be vulnerable to. We will then analyze which techniques proposed by current academic literature can be added to our preferred approach to prevent identified protocol violations. Using this knowledge, we intend to combine these techniques and develop new ones to design a verification scheme that is ideally resistant to all of our identified protocol violations. We expect our verification scheme to rely on a minimal amount of trusted third parties or Blockchains, and consider that the Outsourcer, Contractor, and potential Verifiers may act dishonestly or lazy.

1.5. Thesis Approach

2. Literature Review

2.1. Components of an Edge Computing Marketplace

This section gives an overview of components and their requirements that make an edge computing marketplace viable. An Edge computing marketplace is a special realization of a computing marketplace, that is specialized on letting IoT devices outsource computation to third-party computing resources at the edge for a fee. Thus, an edge computing marketplace is a small computing marketplace that only build an ecosystem of nodes in proxmity to each other. In general, there are the following main components that an edge computing marketplace should feature. First, it should provide a matching and price-finding algorithm that lets Oursourcers and Contractors identify each other and negotiate on a price. After they agreed on the Contract, the Outsourcer sends inputs to the Contractor to process with an assigned function.

Second, it should propose a verification scheme that participants in the ecosystem can use to verify results of the computation and if a party is entitled to a payment. As IoT devices are often computationally weak, it might be difficult for them to verify if responses returned by a third party Contractor are valid. In fact, Contractors have an incentive to return a computationally less expensive probabilistic result, to save resources while still collecting the reward. Likewise, an Outsourcer has no incentive to rightfully pay an honest Contractor after it has received all computational results, and expensive microtransactions prohibit real-time payment.

As edge computing marketplaces are small networks that require less routing than traditional internet application, some authors propose a network architecture that is computation-centric and focuses on maximizing performance for transmission of outsourced computation at the edge.

Additionally, some applications may require privacy preservation of the inputs, functions, or sender information if one of those is confidential.

Lastly, a payment scheme should ensure that payment is handled securely, and with low transaction fees.

From these identified components, we will focus on designing a secure verification scheme for outsourced computation that is resistant to cheating attempts, scales with many participants and causes low computational overhead for each participant.

Table 2.1.: Components of an Edge Computing Marketplace

Component	Requirements
Verification of Outsourced Computation [1]	Secure
	Scalable
	Efficient
Payment [2]	Low fees
	Secure
	Scalable
Matching and price-finding [3]	Fair prices
	High Matching rate
Privacy Preservation [4] [5] [6]	Preserving privacy of sent data
	Preserving privacy of sender information
	Efficient
Network Architecture [7]	Low overhead
	Computation centric

2.2. Verification of Computation

This section gives an overview of different types of functions that can be verified by schemes that can be found in current academic literature.

There are a few verification schemes that can be used for arbitrary functions that rely for example re-execution, trusted hardware, or encryption. Other verification schemes instead only work on certain functions, allowing them to exploit mathematical properties that can be applied to certain function types. These tailor-made verification schemes may restrict the usability of a verification scheme to work on other functions than the ones that it was designed for but may improve efficiency compared to re-executing the function that needs to be verified.

Verification Schemes for Arbitrary Functions

Verification schemes for arbitrary functions generally work on either of the following three principles that the Outsourcer can utilize:

1. Restricting software or hardware access of the party conducting the computation thus preventing any modification.
2. Encrypting all values the function is executed on thus making it infeasible for the party conducting the computation to generate fake outputs[8] [9].

3. Re-executing the function on all inputs or random ones and comparing if the outputs returned by the party conducting the computation is identical.

The following paragraphs show a more detailed overview of the different techniques used for verification schemes for arbitrary functions.

Software Virtualization Lightweight containers have become increasingly popular since the introduction of cloud computing. Containers can be integrated into the host operating system thus reducing the software overhead compared to virtual machines. When running containers on third party hardware, code integrity becomes critical. Currently, one of the most widespread types of containers are Docker containers that promise a lightweight and secure option for running code on different platforms and machines. However, Docker containers were found to be vulnerable to multiples attacks such as backdoors, phishing attacks, account hijacking, and unprotected API calls[10].

These security vulnerabilities leave multiple options for a dishonest third party to return fake results even when only having access to a secured docker container.

Trusted Execution Envoirements Trusted execution environments (TEEs) like Intel's Software Guard Extensions (SGX) provide an isolated environment to execute code, protected from all other software running on the same machine. TEEs are architectural extensions integrated into processors that provide security guarantees against. It promises secure execution of code even when it runs on third party hardware. Apart from high computational overhead, there are multiple unsolved security challenges associated with SGX, Even with the latest update of SGX, attackers could get access to the attestation key which is used to attest computation as genuine results originated from the secure enclave[11]. Thus, when a response is returned to the outsourcing machine, Intel SGX fails to provide secure proof about the validity of the result in practice.

Fully Homomorphic Encryption Fully Homomorphic Encryption (FHE) is a form of encryption that enables functions to be executed on encrypted data. The result of the computation is encrypted. By decrypting the result, the output is identical to the operations performed on the unencrypted data. [8] [9] The main problem with FHE is that encrypting and decrypting create high computational overhead which makes its application not practical for most use cases. Also, weak outsourcing machines will most likely not be able to encrypt all inputs in a reasonable time.

Re-execution Arbitrary deterministic functions can be verified by re-executing the outsourced function on all or certain inputs. Re-execution can be performed by the outsourcing machine itself or can be also outsourced to another machine. When comparing an output gained by performing re-execution and an output received, it can be verified whether both outputs are identical[**eisle**].

It will be shown in chapter 4 that even with a fraction of samples that are re-executed a dishonest party can be detected with high confidence. Re-execution is a practical approach of verifying computation of arbitrary function with the possible disadvantage that if the outsourcing machine is too weak to perform re-execution of the outsourced samples, a third party is needed for re-executing the function on its behalf.

Each of the presented techniques has certain advantages and disadvantages. The advantage of software virtualization is that the function and inputs itself do not have to be modified and there is almost no computational overhead introduced to the machine outsourcing the computation. The disadvantage is, however, that currently there is no entirely secure method available that prevents interference of any kind.

Likewise, the advantage of trusted execution environments is that inputs and functions do not have to be transformed, While in theory security guarantees of trusted execution environments can be higher than the one's of software virtualization, both currently fail to prevent certain attacks. Additionally trusted execution environments cause significant computational overhead to the executed function.

Encrypting all values with homomorphic encryption has the advantage of being secure against any type of interference with the function. Any modification can be easily detected as the machine conductin the computation is not able to generate valid encrypted outputs based on arbitrary values. Therefore, homomorphic encryption is usually the most computationally expensive verification scheme of all available methods.

Re-executing the outsourced function on all or certain values to verify if results gained from re-executing match received results from the party conducting the computation. The re-execution can be either computed by the machine that receives results from the outsourced computation or a third party.

The advantage of this method is that it is very cost-efficient and can lead to less than 1% overall overhead of the verification scheme while still maintaining a high detection rate of false results. The disadvantages of this method are that it requires the outsourcing party or a third party to conduct computation.

Due to its cost-efficiency and high security, we conclude that re-executing computation is currently the most practical way of verifying computation for arbitrary function. A summary of verification schemes for arbitrary function can be found below in

Approach	Advantages	Disadvantages
Software Virtualization [10]	low computational overhead, inputs and function do not have to be transformed	not secure
Trusted Execution Environments [11]	inputs and function do not have to be transformed	not secure
Fully Homomorphic Encryption [8] [9]	highly secure	Extreme computational overhead, often unpractical
Re-execution [12]	low computational overhead when sampling is used, secure	need for a third party if outsourcing machine is not able to re-execute the function

Verification Schemes for Specific Functions

While the approaches listed in the last section can be used for arbitrary function, there are also verification schemes designed for specific functions that exploit mathematical properties specific to that function. For example, for certain function types there might be a function $v(x, y)$ that given an input x and an output $y = f(x)$ efficiently verifies the correctness of computing y. We evaluated the following verification schemes for specific function.

Pre-Image Computation Pre-Image computation describes function types that given an input Domain D and a value y, the computations should return all values $x \in D$ for which $f(x) = y$ [13] [14]. While verifying $f(x) = y$ for a given y and input domain is trivial for most functions, the challenge when designing a verification scheme for Pre-Image computation lays in ensuring that all values x are processed and not only a subsection.

Ringers An efficient solution to this problem is the use of true ringers and bogus ringers. The main idea of this approach is to pre-compute the function for a set of values $x \in D$ (true ringers) and $x \notin D$ (bogus ringers). Only if returned values x by the party conducting the whole computation match results of these pre-computed values, the result is accepted[13] [14].

Magic numbers Another approach is the use of 'magic numbers' which rewards the party conducting the computation when finding a value y that is contained in the set of magic numbers. The more pre-images x a party computes, the more likely it is to find an x satisfying $f(x) = y$ and gain a reward. Thus, magic numbers can serve as milestones to ensure that only a party that runs the computation on all inputs is guaranteed to be rewarded fully[15].

While both presented approaches to verifying pre-image computations, it is questionable if there are many use cases for pre-image computations in edge marketplaces.

Matrix Multiplication Matrix multiplication is a common operation for image processing as well as a computationally complex component of convolutional neural network training and inference. Currently, the most efficient matrix multiplication algorithm runs in $O(n^(2.3729))$ time.

Freivald's algorithm In general, when calculating matrix multiplication $AXB = C$, Freivald's algorithm can be used to verify $AXB = C$ given A, B, C in $O(n^2)$ with high probability [16].

While this property might be useful, using a verification algorithm that needs the result as an input instead of performing re-execution always bears the problem of lazy behavior where a machine might return that $AXB = C$ without ever running the verification algorithm. For this reason, most verification schemes available for matrix multiplication do not make use of Freivald's algorithm but instead use techniques that are not relying on a verification algorithm that needs the result matrix C as an input.

Permutation There are verification schemes for matrix multiplication based on Homomorphic encryption, secret sharing [17], and permutation [18] [19]. Verification schemes based on permutation promise to feature input and output privacy like Homomorphic encryption but with higher efficiency. As the matrix multiplication problem has to be encrypted to a secure matrix multiplication problem, permutation is still not practical with current hardware to perform real-time computations. Also, there is criticism that a solution using permutation can be even less practical than Homomorphic encryption due to high communication costs.

Matrix transformation Another approach is to instead of encrypting the input matrix A and B to generate a random matrix A' and B' and then sending $A' - A$ and $B' - B$ to on of two remote servers while the other remote server receives different transformed matrices. [20]. As only the outsourcing machine knows the original matrices and the two remote servers receive a different transformed matrix, collusion should be not possible. The advantage of this scheme is that it additionally provides input privacy and creates less than $O(n^2)$ computational complexity, We can conclude that this scheme uses complete re-execution but provides input privacy through matrix transformation. Also, we analyzed that this scheme also works with sampling-based re-execution.

While presented schemes feature input privacy in addition to secure verification, these features come with high complexity with the exception to [20]. Thus, we can conclude that general-use sampling-based re-execution is the best technique for verifying matrix multiplication. If the Outsourcer is powerful enough to conduct verification itself, Freivald's algorithm can improve the performance of this verification. Transforming matrices before sending them to remote machines according to [20] might be a useful extension for providing input privacy also for verification schemes based on re-execution.

Matrix Inversion Matrix inversion problems can be found in scientific and engineering problems. Matrix inversion comes with the same computational complexity as matrix multiplication. The correctness of a result can also be checked similarly: Given an input matrix A resulting matrix B it simply has to be checked whether AXB is equal to the identity matrix. Freivald's algorithm can be used to do this verification more efficient [16].

Secret sharing and Monte-Carlo verification One proposed verification scheme for verifying matrix inversion uses secret sharing and two remote servers that perform matrix inversion [21]. This has the advantage that input privacy is preserved. This scheme utilizes a Monte-Carlo verification algorithm to only check if random samples are correct and not all outputs.

In conclusion, verifying matrix inversion is a similar problem as verifying matrix multiplication and therefore comes with the same complexity implication. Overall, matrix multiplication offers more use cases than matrix inversion.

Linear Equations Linear equations are a popular type of computational tools used in analyzing and optimizing real-world systems. Large scale linear equations are computationally complex to solve.

One proposed verification scheme for verifying large scale linear equations is to use Homomorphic encryption and outsource only certain complex operations to a remote server [22]. This scheme features the general advantages and disadvantages of using Homomorphic encryption.

Polynomials Polynomial functions are fundamental in engineering and scientific problems.

One proposed verification scheme utilizes arithmetic circuits to calculate the polynomial and ensure input and output privacy. The Contractor generates a proof that is checked by a third party Verifier whether computation was executed correctly, without revealing inputs [23]. Signatures are used for recording messages. In their test setup, the verification scheme takes 22ms per delegated polynomial function. Security concerns with the initial scheme have been addressed in a later publication [24]

Regression Analysis Linear Regression Analysis is a data analysis technique applied across multiple domains. Performing linear regression over a large dataset is a computationally expensive task.

Input transformation One proposed verification scheme for verifying linear regression analysis features input and output privacy and an efficient verification algorithm that works on encrypted results. Given an input problem $\varphi(X, Y)$ the Outsourcer uses a secret key k to transform the problem into $\varphi_k(X', Y')$. The advantage of using a transformed problem for linear regression analysis is that outputs can be verified on the transformed problem without re-transforming it first [25]. To verify if computation has been carried out correctly the Outsourcer has to check whether the regression parameter β' can be used to satisfy $Y' = X' * \beta'$. Thus verifying linear regression analysis is computationally inexpensive due to its trivial verification algorithm.

Convolutional Neural Networks In deep learning, a Convolutional Neural Network(CNN) is a class of deep neural networks that is usually applied to image analyzing digital images. CNNs achieve state of the art performance in classifications tasks such as object recognition, optical character recognition, face recognition, and natural language processing. Compared to other types of neural networks, CNNs are easily trained with fewer parameters and connections while providing a better recognition rate. However, both training a CNN and using a CNN for predictions on new data (inference) is still extremely computationally expensive and often requires the use of powerful GPUs or APUs. For weak machines, outsourcing the inference of a CNN is particularly interesting to utilize recognition results for its application.

Partial verification One proposed verification scheme utilizes embedded proofs to generate a proof over a part of the used matrix multiplication during inference instead of the whole process [26]. Essentially, this scheme further optimizes re-execution by only checking parts of one iteration instead of re-executing the whole computation. It should be noted, however, that only verifying the matrix multiplication part is not sufficient to verify the integrity of the whole inference process.

Zero-knowledge succinct non-interactive argument Another approach is to use Zero-knowledge succinct non-interactive arguments (zk-SNARK). zk-SNARK have the advantage that they provide input and output privacy and privacy of the used model. However, zk-Snarks require a proof time between 8 hours [27] to 10 years [28] and 80GB to 1400TB storage overhead. This means that while offering valuable privacy-preserving properties on top of verification and outsourcing, zk-SNARK are far from practical.

Matrix and weights transformation SecureNets introduces a verification scheme where both data X and weight matrix W get transformed to X' and W' before sent to the Contractor to preserve input and mode privacy. When receiving a result Y' based on X' and Y' the Contractor simply randomly checks if $y'_{i,j}$ equals $w'_{i,:} \cdot x'_{:,j}$ for a random row i and a random column j [29]. Thus, this scheme uses an optimized re-execution protocol and preserves input and model privacy.

Arithmetic circuits and random challenges SafetyNets introduces a verification scheme for deep neural networks that can be presented as arithmetic circuits. This has the advantage that model privacy is preserved. The Outsourcer sends random challenges to the Contractor and aborts when a random challenge is not successfully met [30]. This is a similar concept to using sampling-based re-execution. The disadvantage of this scheme is that it only works with some types of neural networks.

Homomorphic encryption AlexNet claims to be the first Homomorphic CNN on encrypted data with GPUs. The authors use GPUs to accelerate CNN performance over encrypted data. While their Homomorphic CNN can classify images with state-of-the-art accuracy, inference takes between 5-300 seconds per image [31]. This shows that Homomorphic encryption is not practical yet for verifying the inference of a CNN.

In summary, the practical approaches for verifying CNN inference rely on re-execution or similar concepts. With application-specific adjustments used by schemes such as SecureNets re-execution can be further simplified and input/output, and model privacy can be added at additional computational cost.

Other functions

Input transformation Input transformation can also be applied to other functions. The general approach is to transform an input x with a random value r to create x' that can be outsourced to a remote machine. This approach works whenever the resulting y' of applying outsourced function f on x' can be re-transformed to y by the Outsourcer. If this approach can be combined with re-execution to enable secure re-execution that is resistant to collusion and preserves input and output privacy. One proposed scheme utilizes input transformation for computing the characteristic polynomial and eigenvalues of a matrix [32].

Arithmetic circuits Arithmetic circuits can be used for all problems that can be represented as an arithmetic circuit. Using arithmetic circuits has the same benefit of fully homomorphic encryption but is less computationally expensive. When a problem can be presented as an arithmetic circuit [33] proposes a verification scheme with a third-party Verifier. The disadvantage of this scheme is that computational cost is still high and the Outsourcer is assumed to be honest.

In summary, we conclude that approaches for verification schemes for specific functions still show several similarities. Most of these schemes use either a modified form of re-execution or encryption with their approaches to enable secure verification. Encryption can sometimes replace the need for re-execution as the Contractor is not able to generate a false encrypted response that meets simple integrity checks when decrypted. Therefore, these schemes are also not practical for specific functions. specially schemes that rely on Homomorphic encryption are also not practical in real-time when optimized for specific functions.

Schemes that use re-execution are mostly practical and use either a more efficient verification algorithm than re-executing the whole functions such as only re-executing parts of a function, random challenges, or efficient algorithms such as Freivald's algorithm. Additionally, input transformation seems a promising technique that works for a range of different functions and enables input and output privacy at a much lower computational cost than Homomorphic encryption, secret sharing, or schemes that rely on arithmetic circuits.

Schemes that try to optimize re-execution by only re-performing parts of the function over one input are problematic as incorrect results that were caused by other parts of the function can be overlooked. We conclude that it is more secure to decrease the input size of values that are re-executed rather than focusing on parts of the function.

After assessing multiple schemes with different approaches we conclude that sampling-based re-execution is practical for arbitrary functions and features low computational overhead. If the use case also requires privacy-preservation then input transformation is a technique that works for a range of specific functions and preserves input and output privacy. Also, Input transformation in combination with re-execution can easily prevent lazy Verifier behavior and collusion attempts. If the Outsourcer uses two different random numbers to transform the inputs sent to Verifier and Contractor, collusion between Verifier and Contractor is infeasible as they cannot produce two transformed outputs that are equal after re-transforming without knowing the random numbers used by the Outsourcer.

It should be noted that nearly all evaluated verification schemes in this section are not secure if the threat model includes a dishonest Outsourcer or colluding parties. Most verification schemes are designed from an Outsourcer perspective and not from an ecosystem perspective where all parties are protected against dishonest or lazy behavior. The following sections will focus on how different verification schemes target threats that can result in a scenario with re-execution and Verifiers.

The tables 2.2 and 2.3 below summarizes verification scheme approaches for specific

functions, their use cases, and their trade-offs.
’

Table 2.2.: Verification scheme approaches for different function types

Function type	Use Case	Approach	Advantage	Disadvantage
Pre-Image Computation	Scientific function evaluations	Ringers [13] [14]	Efficient verification	Computational overhead for Outsourcer to calculate bogus ringers
		Magic Numbers [15]	Efficient verification	Appraoch assumes honest Outsourcer
Matrix Multiplication	Fundamental operation for image processing and Neural Networks	Freivald's Algorithm [16]	More efficient than re-execution	Requires result matrix as input - promotes lazy behavior
		Secret sharing [17]	More efficient than Homomorphic Encryption	High computational overhead
		Permutation [18] [19]	More efficient than Homomorphic Encryption	Not practical
		Matrix Transformation [20]	Input and output privacy, low computational overhead	-
Matrix Inversion	Scientific and Engineering problems	Freivald's Algorithm [16]	More efficient than re-execution	Requires result matrix as input - promotes lazy behavior
		Secret sharing and monte-carlo verificaiton [21]	More efficient than Homomorphic Encryption	High computational overhead
Linear Equations	Analyzing and Optimizing real-world systems	Homomorphic Encryption [22]	Input and output privacy	Not practical

Table 2.3.: Verification scheme approaches for different function types

Function type	Use Case	Approach	Advantage	Disadvantage
Polynomials	Fundamental functions in engineering and scientific problems	Arithmetic Circuits [23] [24]	More efficient than Homomorphic Encryption	Moderate computational overhead
Regression Analysis	Data analysis	Input transformation [25]	More efficient than Homomorphic Encryption	-
Convolutional Neural Networks	Classification tasks, Natural language processing	Partial Verification [26]	More efficient than re-execution	Part of the computation remains unverified
		zk-SNARK [27] [28]	Input, output, and model privacy	Not practical
		Matrix and weights transformation [29]	More efficient than zk-SNARKs	Transforming complex models can be difficult
		Arithmetic Circuits and random challenges [30]	Efficient verification	Only works with certain CNNs
		Homomorphic Encryption [31]	Input, output, and model privacy	Not practical in real-time

2.3. Verification of Convolutional Neural Network Inference

As we test our designed verification scheme with object detection using Convolutional Neural Networks, we explain in this section why sampling-based re-execution is also currently the most practical technique to verify neural network inference.

As discussed in the previous sections we evaluated 5 verification approaches for verifying CNN inference and 4 approaches for verifying arbitrary functions. As software virtualization and trusted execution environments have still too many security problems, we do not consider them as a secure choice for verifying neural network inference. Fully Homomorphic encryption, zk-SNARKs, and also AlexNet which is based on Homomorphic encryption are still far off from allowing inference on real-time video/image streams. Arithmetic circuits are more efficient than Homomorphic encryption but transforming an existing CNN to an arithmetic circuit is not only difficult from a methodological point of view but also comes with additional computational overhead. Transforming input matrix and weights before sending them to a remote server is a promising approach for preserving input, output, and model privacy but also has the problem that transforming CNN models with a complex architecture is difficult.

This leaves us with re-execution as an approach that can verify arbitrary functions and does not require any transformation of the CNN model or the input values. This also makes this technique particularly efficient for the Outsourcer. Since CNN inference is a complex task and computationally weak machines may not even be able to perform sampling-based re-execution in real-time, a third-party Verifier may be included in verification. The possible optimization of only verifying parts of the matrix multiplications of the CNN inference is not advised due to the risk that other parts of the functions are not executed properly.

In summary, sampling-based re-execution is out preferred technique to verify CNN inference due to its following properties:

1. Low computational overhead for Outsourcer
2. No methodologically difficult modification of the CNN model required
3. Provides high security if all potential threats (collusion, lazy, and dishonest behavior) are addressed
4. Little to no computational overhead for Contractor and Verifier

2.4. Verification of outsourced Computation using Re-execution

Lots of verification schemes use re-execution performed by the Outsourcer or a third party entity to verify the integrity of outsourced computation results. As discussed in previous chapters, re-execution is a practical and secure way to verify arbitrary functions. However, a verification scheme that is resistant to different threats has to feature additional measurements on top of re-execution to ensure that no party can cheat in the process. Using re-execution without additional measurements does not protect for example from collusion where Verifier and Contractor agree on a false response that when generated saves computational resources in comparison with performing the actual computation. Likewise, there have to be measurements so that the Outsourcer is not able to reject payment or falsely accuse Verifier or Contractor of cheating. In this section, we analyze different verification schemes that use re-execution or similar approaches and give an overview of assumptions and measurements these schemes introduce to deal with dishonest behavior or unintended protocol violations such as timeouts.

Outsourcing computation is a variation of the principal-agent problem. The Outsourcer can be viewed as the principle that outsources a job to a Contractor that can be viewed as the agent. Multiple attributes make this problem sensitive to dishonest behavior.

1. The Outsourcer is not able or does not intend to verify the correctness of the whole outsourced computation.
2. The Contractor wants to utilize minimal resources for outsourced computation.
3. Contractor might be "lazy but honest", lazy and dishonest or malicious to save resources or harm the Outsourcer.
4. The Outsourcer might be dishonest to avoid payment after correct computation.

Verification schemes

Uncheatable grid computing [34] This verification scheme introduces commitment-based sampling. The Contractor commits to a Merkle tree hash every few intervals and sends it to the Outsourcer. The Outsourcer then randomly selects a few samples to recompute them and send a proof of membership challenge to the Contractor. Only if the verified output matches the initial response and only if the proof of membership challenges is successful, the contract is continued otherwise, the Contractor is detected cheating.

The major advantage of this scheme is that an Outsourcer can detect a cheating Contractor with high probability and low computational overhead. Merkle trees are used to require only one signature per specified interval instead of signing all responses. Sampling-based re-execution is used to verify results with low computational overhead and high probability.

The disadvantage of this scheme is that it does not feature an ecosystem perspective and therefore does not deal with dishonest Outsourcers. This additionally leads to the problem that the only action the Outsourcer can take is to abort a contract. It cannot enforce a fine towards the Contractor as there are no digital signatures used in this scheme to record communication securely. Also, it does not consider that a third party Verifier might be needed to perform the re-execution. Other possible contract violations than dishonest behavior such as timeouts or low response rates are not considered. Assuming no third party Verifier is needed to perform sampling-based re-execution, the following security problems remain.

1. The Contractor can claim to not have sent a false response when detected due to the lack of signatures.

2. The Outsourcer can reject payments due to the lack of signatures.

These disadvantages lead to the conclusion that this scheme can only be used from the Outsourcer's perspective to detect dishonest behavior of a Contractor and abort the contract if dishonest behavior is detected. This makes cheating a risk-free strategy for the Contractor apart from losing an active contract. This scheme has to be modified if the Outsourcer is not powerful enough to perform re-execution and relies on a third-party Verifier.

Security and privacy for storage and computation in cloud computing [35] This verification scheme uses sampling-based re-execution to verify results of outsourced computations from an Outsourcer to a cloud server. Within specified intervals, the cloud server commits to signed Merkle roots over its results when returning them to the Outsourcer. The cloud user can instruct a Verifier to verify these results by checking if the signature of the signed Merkle root is correct and if random samples were computed and returned correctly. The Verifier sends back the result to the cloud user which can then decide on further actions.

The main advantages of this scheme are that no third party is required during the verification process except the Verifier, Outsourcer, and the Contractor. Also, digital signatures are used to provide a record of all outputs that the Contractor sent. Merkle trees are used to require only one signature per specified interval instead of signing all responses. Sampling-based re-execution is used to verify results with low computational overhead and high probability.

The major disadvantage of this scheme is that Verifier and Outsourcer are assumed to be honest. This can lead to the following security problems:

1. The Outsourcer and the Verifier collude to falsely accuse the Contractor of cheating
2. The Contractor and the Verifier collude to send back false results by investing minimal computational resources
3. A lazy Verifier can always respond that responses to be verified are correct to not invest any computational resources
4. It is not specified how a Contractor can enforce payment from a dishonest Outsourcer that tries to reject payment

These security problems lead to the conclusion that this verification scheme only works if the Verifier and the Outsourcer can be assumed as honest. In this case, it successfully detects all cheating attempts by a Contractor with high probability, low computational overhead, and no requirement of a third party.

Bitcoin-based Fair Payments for Outsourcing Computations of Fog Devices [1] This scheme uses commitment based sampling and assumes that outsourcing jobs are exclusively pre-image computations. Commitment-based sampling refers to the use of Merkle trees that are first use to commit to a range of output values via a Merkle root hash. The Outsourcer then sends random challenges picked by sampling techniques to verify results with high probability. At the start of the scheme, the Contractor and the Outsourcer create a Bitcoin contract to create a tamperproof, publicly verifiable agreement. The contract includes the task and a deposit transaction by the Outsourcer that can be redeemed by the Contractor if its condition is met. Once the Contractor sends results to the Outsourcer that satisfies the challenge, the Contractor gets automatically paid by the Bitcoin transaction. A trusted third party is used to resolve conflicts.

The proposed verification scheme has a few advantages. Using Merkle trees for committing to outputs is an efficient way to verify a large number of outputs with only a few or even one sample and with low communicational overhead. Using a Blockchain is a very secure method of creating records of key communication and automatically enforce payments.

There are however also major disadvantages, why we do not recommend this verification scheme for most use cases. One major disadvantage of this scheme is that it exploits trivial verification properties of pre-image computation which makes it unusable for other more frequent use cases. Also, the scheme assumes to have a trusted third party available at all times for conflict settling. Besides, there is a security problem as responses from the Contractor are not digitally signed. Therefore, no proof is available for the Outsourcer to reject payment properly. For this reason, the scheme does not prevent a dishonest Contractor from sending false or no responses and does not prevent a dishonest Outsourcer to claim that no valid computation result was received.

Integrity verification of cloud-hosted data analytics computations [36] This verification scheme uses artificial data values to enable verification even for computationally weak Outsourcer. This scheme is mainly targeted at use cases with a large amount of data such as machine learning or deep learning tasks. The key idea of this verification scheme is that the Outsourcer adds artificial data values to its inputs and checks whether the Contractor reacts accordingly. For example, the Outsourcer could insert an object at a certain position in an image and verify if an object detection response sent by the contractor detects the artificially added object successfully.

The main advantage of this method is that it can be utilized by a weak Outsourcer without the need for a third-party Verifier. Also, the computational overhead of modifying input data is low.

There are some disadvantages however to the technique itself. If we stick to the example with an image, then artificially added objects can block other relevant parts of the images. Thus, we advise to only use this technique rarely for data that is not used for further analysis but rather discarded. Also, a Contractor might detect an unregular change of two consecutive images when an object is added artificially and react with honest computation accordingly. This issue is addressed by the paper but comes with additional computational overhead of transforming input data. It should finally be noted that even if an object is added artificially, there might be the chance that the CNN performing object detection is not able to detect it even if it behaves honestly. Thus, this technique has to be used probabilistically where missing detections are tolerated to a certain extent. The alternative of not placing the artificial data randomly but falling back on a set precomputed to ensure detection with 100% probability is not advised as the Contractor can learn this limited set of precomputed images over time.

If these concerns are addressed, adding artificial data values can be a promising technique to replace re-execution in certain use cases due to higher efficiency. It should be noted, however, that this technique still has to be supported by other measures that prevent dishonest behavior by Outsourcer and Contractor. As this verification scheme was not designed from an ecosystem perspective, the usual drawbacks that result from not using signatures or other forms of records come along with this scheme if no measures are added.

Game-Theoretic Analysis of an Incentivized Verifiable Computation System [37] This paper modifies an existing verification scheme with re-execution and Verifiers by adjusting game-theoretic incentives and prevention of Sybil attacks. These modifications are also relevant from previously discussed verification schemes in this chapter that suffer from possible collusion of Contractor and Verifier. At first, this scheme proposes to randomly select a Contractor and a Verifier and to assign fines and bounties when a party is detected cheating or successfully accuses another party of cheating. Sybil attacks refer to the problem that a Contractor or Verifier can generate multiple identities to be selected both as Contractor and as Verifier. This scheme proposes two third parties - an arbiter and a judge to prevent this behavior. The verification scheme uses Ethereum as a Blockchain solution to securely enforce payment, fines, and bounties. Merkle Trees are used to commit to multiple outputs at once.

The advantage of this scheme is that it successfully prevents collusion of Verifier and Contractor if both parties are rational, expected payoff maximizing entities. Also, it is the only scheme that we analyzed that addresses Sybil attacks.

The disadvantage of this scheme is that preventing Sybil attacks comes with the high overhead of a third party and Ethereum transactions. Also the Outsourcer, judge, and arbiter are assumed to be honest which may not be a valid assumption.

Anticheetah: Trustworthy computing in an outsourced (cheating) environment [38] The authors of this paper propose a system called Anticheetah which outsources a computation to multiple nodes in a multi-round approach. Each round different inputs are sent to each node and compared by a trusted master node in case the same input got sent to more than one node. Each round all nodes are evaluated by the master node for the following criteria in order of importance: Reliability, cost-efficiency, and timeliness. This leads to an accurate reputation balance over time for each node. In each round, nods are selected first that have the best reputation balance. If a node gets its reputation decreased in reliability even though it was honest, due to a multi-round approach this scenario does not have a big impact on overall node balance. The scheme assumes guaranteed message delivery, no lost messages, and zero communication overhead.

The advantage of this scheme is that due to its multi-round reputation system, collusion between multiple nodes is disincentivized. Also, through a reputation system that also includes cost-efficiency and timeliness, it can be ensured that over time only nodes that can provide a high quality of service remain as Contractors.

The disadvantage of this scheme is that multiple nodes are needed for serving as a Contractor. In scenarios where not many nodes are available for computation, this can be a problem. Also, the master node causes an additional need for trusted third party resources. The assumptions of this scheme such as no lost messages and zero communication overhead are not realistic for scenarios with weak connection.

Efficient fair conditional payments for outsourcing computations [39] This verification scheme uses a semi-trusted third party to solve disputes between participants. The verification scheme uses true and bogus ringers that were discussed earlier. These lead to efficient verification for certain functions. The verification scheme is designed to prevent dishonest and lazy Contractors from succeeding with their strategy. Only in case of conflict or delays the semi-trusted party is required to participate. Signatures over key communication are used to let the third-party come to a rightful conclusion in case of dispute.

The advantage of this scheme is that it works for dishonest Contractors and to a certain extent also to dishonest Outsourcers.

The disadvantage of this scheme is that it requires a semi-trusted third party that can become the bottleneck of the system and might not be available at all times.

Mechanisms for Outsourcing Computation via a Decentralized Market [12] The authors of this verification scheme propose a smart contract-based protocol running on a Blockchain to set incentives for Outsourcers and Contractors to participate in an outsourcing market place. For verification, their scheme uses semi-trusted third parties. The incentives in the system a set in a way that dishonest behavior is discouraged. For verification, their scheme uses sampling-based re-execution.

The advantage of this scheme is that incentives are set in a way that collusion is made unlikely when dealing with rational pay-off maximizing participants.

The main disadvantage of this scheme is that current transaction fees in Ethereum are too costly for short contracts. Also, Verifiers have to be partially-trusted.

Incentivized outsourced computation resistant to malicious Contractors [40] This verification scheme proposes that an Outsourcer sends it inputs to at least two Contractors performing the identical task. A trusted timestamping server is used to keep records of inputs, results, and timeliness, and a trusted central bank is used to conduct payment. The general procedure of this scheme consists of outsourcing the same job to $n = 2$ Contractors and accepting the result if responses match. If responses do not match, the job gets outsourced again to $n_{new} = n^2$ Contractors until no conflicts arise. The timestamping server records all communication between Contractors and Outsourcer to ensure verifiability of the outcomes. The incentives in this system are sent in a way that being honest is the dominant strategy, and rationally expected pay-off maximizers will not collude.

In their simulations, the authors detected that with a fine-reward ratio of 20% and 25% malicious Contractors, on average 2.04 Contractors are needed until agreement is reached.

The minimum total computation complexity of outsourcing is 200% of the original job plus the overhead of the system which is 2% on average according to the authors.

The advantage of this scheme is that the timestamping server records communication from both the Outsourcer and Contractors. Therefore, it not only prevents dishonest behavior from Contractors but also from the Outsourcer which might try to refuse payment. Also, the quality of service violations can be punished by comparing the times of Outsourcer and Contractor communication on the timeserver.

The main disadvantage of this scheme is that a trusted timestamping server is needed that records nearly all communication of all ongoing contracts of different Outsourcers and Contractors at all times. Thus, the timestamping server can easily become the network or performance bottleneck of the system. Also, the Contractors perform complete re-execution which leads to a minimum computational overhead of 100% per job alone accounted for by that decision.

Approaches to prevent protocol violations In summary, the evaluated verification schemes are mainly designed to prevent dishonest behavior of a Contractor and assume the Outsourcer to behave honest. Only a few examples such as [40] also protect against dishonest behavior from the Outsourcer. The following approaches are used by different schemes to ensure secure verification of outsourced computation and accountability for dishonest behavior:

Complete/Sampling-based re-execution This technique refers to re-executing the outsourced function overall inputs or random sample inputs. If the Outsourcer is not powerful enough to perform re-execution, it can outsource the re-execution to a Verifier (third-party) such as in [40].

Signatures Digital Signatures are a secure way to keep records of communication that can not be tampered with. If an input and related job information is signed by the Outsourcer before sent to the Contractor, the Contractor can prove having received this data from the Outsourcer to any third party, since it is unable to generate a valid signature of the Outsourcer over data by itself. A few schemes such as [1] use Outsourcer signatures to commit to a contract with a specified reward.

Likewise, if the Contractor signs results and related job/input data before sending them back to the Outsourcer, the Outsourcer can prove to any third party which values the Contractor responded with. Schemes such as [35] use Contractor signatures to commit to responses when sending them. This way, if a false response from a dishonest Contractor is detected, the signature can act as proof that the Contractor indeed sent the signed value.

Distributed computing with multi-round inputs This is a special technique introduced by [38]. A job gets outsourced to multiple Contractors with certain overlap and compared by a trusted master node. Inputs are sent in multiple rounds. In each round, Contractors are evaluated based on reliability and performance criteria. This scheme ensures that after a few rounds a near-optimal matching of honest Contractors and a trusted Outsourcer can be achieved. However, it requires a large ability of Contractors to work properly.

Commitment to messages using Merkle Trees Merkle Trees can be used as a data structure to efficiently verify if data is contained in a large collection. The root hash of a Merkle tree can be used to verify if data is contained in the whole tree with $log_2(n)$ steps. Verification schemes such as [37] use this attribute of Merkle trees in combination with signatures or trusted third parties to commit to large amounts of inputs or outputs by only committing to the Merkle tree root hash.

Committing to inputs via a Merkle Tree has the advantage that the Outsourcer can first commit to the inputs by signing their Merkle root hash and sending the actual inputs without adding a digital signature or involving a third party. The disadvantage of committing to inputs is that all inputs must be known in advance as the commitment should be made before sending data. This makes input commitment only feasible for use cases without real-time data.

Committing to responses via a Merkle Tree has the advantage that the Outsourcer can commit to a collection of responses before receiving a proof of membership challenge by the Outsourcer. When receiving a challenge from the Outsourcer, the Contractor is not able to quickly re-compute a response and send it instead since it would not meet the proof of membership challenge. This way, instead of signing all data, it is sufficient for the Contractor to only sign Merkle root hashes and challenges in a specified interval, thus improving efficiency. The disadvantage of commit over responses via Merkle trees is that there is always a certain delay after receiving a response, before it is committed. This can lead to a loss of records in case of a dispute if the verification does not consider this risk.

Use of trusted third parties and Blockchains for records and conflict resolving Some Verification schemes such as [1] use trusted third parties (TTPs) or Blockchains to record the communication of participants. In case of dispute, the record can simply be reviewed by any participant on the Blockchain or by contacting the TTP. Also, the Blockchain or TTP can execute code to settle conflicts without bias. The disadvantage of recording all or key communication on a Blockchain or with the help of the TTP is that the amount of communication from all different contracts of a system can easily lead to them being the performance or bandwidth bottleneck of the system. Also, not in all scenarios, it can be assumed that a TTP is available. On the other hand, a Blockchain may add a large amount of computational complexity to the system, especially when "proof of work" is used.

Reputation system A reputation system is used by verification systems such as [38] to evaluate multiple Contractors based on their reliability and performance. A reputation system can exist both locally at node level or globally at a TTP or Blockchain. The reputation system can give incentives for almost no computational overhead to let participants behave honestly and provide a good quality of service. While in public systems, a negative reputation system

Approach	Advantage	Disadvantage
Complete/Sampling-based re-execution [1] [35] [34] [36] [37] [40]	Efficient verification approach to prevent dishonest behavior, works for arbitrary functions	More efficient verification algorithms are available for certain function types
Signatures [1] [35] [39] [37] [40]	Can be used to record payment promises and responses in a tamperproof manner	Account for computational overhead for each signed value
Distributed computing with multi round inputs [38]	Provides a near optimal performance and reliability matching over time	Requires a large amount of available Contractors
Commitment to messages via Merkle Trees [1] [35] [34] [37]	Can be used to reduce the computational overhead of signatures by signing only Merkle root hash and challenges	Leads to small delays in commitment/secure recording when a value is received
TTPs or Blockchains used for records or conflict resolving [1] [39] [37] [12] [40]	Can be used to securely store records and handle conflicts without bias	Requires an available TTP, or accounts for high comutational complexity wen using a PoW Blockchain
Reputation system [38]	Promotes honest behavior and good quality of service	Works best globally with the need of a TTP or Blockchain to store reputation for each node
Blacklisting [38]	Can be used at node/entity level to punish any form of protocol violation	Works best with a reputation system or node register that prevent or disincentivize genearting a new identity

can be avoided by simply creating a new identity, a positive reputation that cannot be faked may still lead to economical benefits for participants that adhere to the desired protocol.

Blacklisting Blacklisting is a simple technique used by verification systems such as [38] to cancel all future relationships with a node at the entity or node level. Blacklisting is most useful when combined with measurements such as a reputation system or a registry system at a TTP that respectively disincentivize or prevent participants that got blacklisted to resume relationships by generating a new identity. Blacklisting can be used to punish the quality of service violations or dishonest behavior.

The following table illustrates which evaluated verification schemes utilize which techniques.

2.5. Protocol violations

Protocol violations can be categorized into the following two types:

1. Dishonest behavior such as sending back false responses, rejecting legitimate payment, collusion.
2. Quality of Service (QoS) violations such as timeouts, low response rate, or frame loss.

The key difference between both scenarios is that dishonest behavior is always intentional, while QoS violations are usually not intentional. Dishonest behavior can be further categorized into dishonest behavior by a single participant, and dishonest behavior by multiple colluding participants. This section will give an overview of identified possible protocol violations and which of the approaches explained in the last section are used by related work to prevent these scenarios. We assume the scenario, that a contract consists of an Outsourcer, a Contractor, and a Verifier. The Outsourcer outsources computation to the Contractor and random samples to the Verifier.

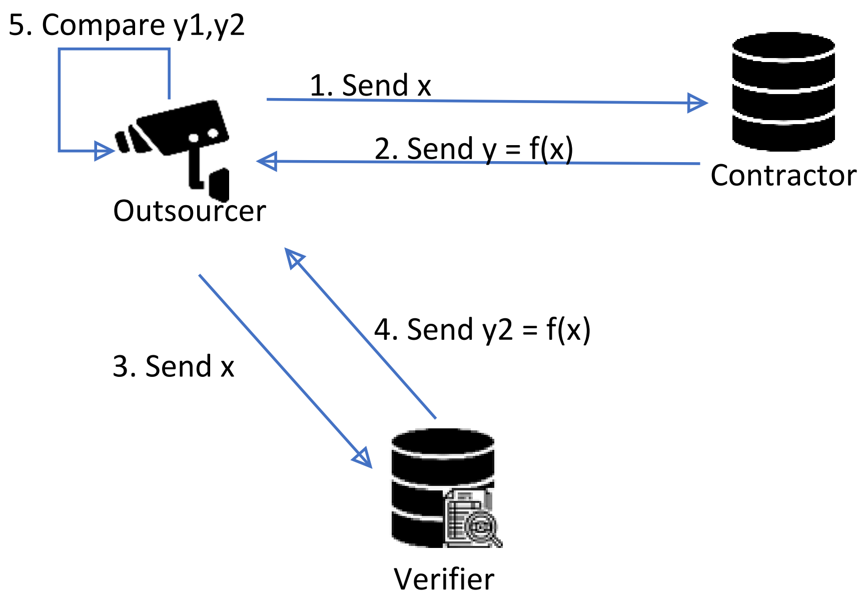


Figure 2.1.: Assumed scenario

Dishonest behavior by one participant Each participant may behave dishonestly on its own if no measurements are set to protect against it. Generally, the incentive of the Outsourcer to act dishonestly is to reject payments to Verifier, or Contractor. The incentives of the Verifier, or Contractor to act dishonestly are to send back false responses that can be generated with less computational effort than performing the function assigned by the Outsourcer. The following scenarios can happen if one party acts dishonestly.

Contractor sends back false responses to save resources In this scenario, the Contractor sends back false responses to the Outsourcer. The false response can be a q-algorithm that returns the correct result with probability q and is computationally less expensive to perform than the function assigned by the Outsourcer. This behavior can be detected by existing verification schemes that use re-execution, often by introducing a third party Verifier such as in [40]. The following figure illustrates the dishonest behavior.

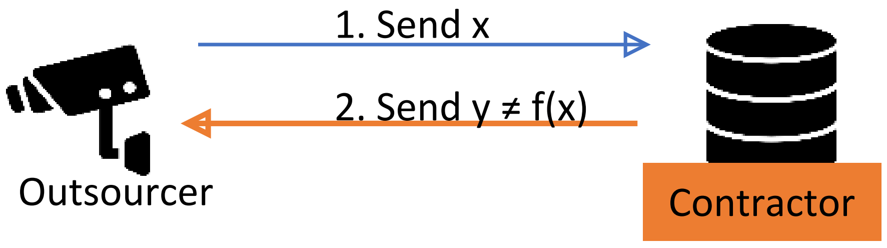


Figure 2.2.: Contractor sends back false responses to save resources

Verifier sends back false response to save resources Likewise, the Verifier might also use a q-algorithm that returns the correct result with probability q and is computationally less expensive to perform than the function assigned by the Outsourcer. As multiple existing schemes assume that Verifiers are honest entities, dishonest behavior can go unnoticed. However, schemes such as [40] outsource the function and inputs of two unequal responses of Contractor and Verifier to additional Verifiers to detect the party that has cheated. The following figure illustrates the dishonest behavior.

Outsourcer sends different input to Contractor and Verifier to refuse payment An Outsourcer has multiple ways to reject payments. One example is, if it sends two different inputs to Verifier and Contractor which leads to unequal responses even if the other participants are behaving honestly. If communication of the Outsourcer is not recorded, this method can be exploited by the Outsourcer to reject payment while claiming that identical inputs were sent to the Contractor and Verifier. While schemes such as [35] use digital signatures we found no evidence that these schemes also propose to digitally sign all inputs and related information sent by the Outsourcer. Also, [40], who proposes a trusted timestamping server to record communication did not specifically mention this scenario. Thus, we consider this potential dishonest behavior as unaddressed by current academic literature. The following figure illustrates the dishonest behavior.

Contractor or Verifier tries to avoid global penalties when convicted of cheating or QoS violations Even when a Verifier or Contractor is detected cheating by an Outsourcer it has multiple ways to avoid system-wide penalties if confronted. While it cannot prevent the Outsourcer from taking actions such as blacklisting in consequence, it might falsely communicate to a third party arbitrator trying to solve the conflict. If communication is not

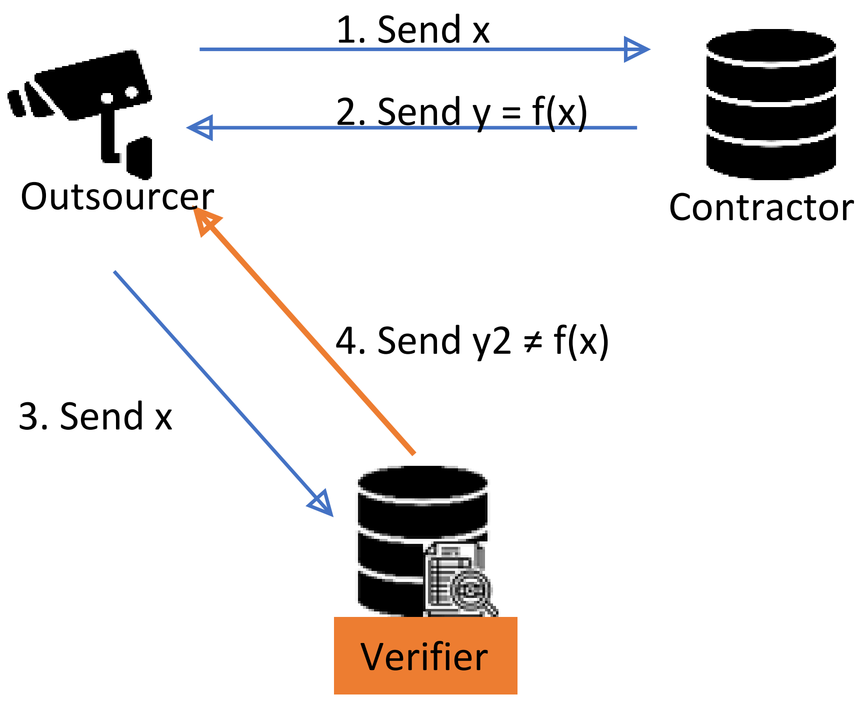


Figure 2.3.: Verifier sends back false responses to save resources

recorded in a tamperproof manner, it can claim to the Arbitrator that it has not received inputs or quickly re-compute a correct response before sending it to the Arbitrator. Verification schemes that utilize digital signatures for Contractors to commit to responses such as [37] can prevent this behavior. The following figure illustrates the dishonest behavior.

Participant refuses to pay even if obliged to by the protocol Being obliged by the protocol to pay a reward or to pay a fine is not sufficient if that reward or fine cannot be enforced. This problem could be solved naively by waving fines and only using real-time payment. However, this strategy leads to a large amount of micro-transaction, large transaction fees, and latency bottlenecks of payment providers. Also, fines can be important disincentives to prevent dishonest behavior to take place. Verification schemes such as [1] solve this problem by utilizing a TTP or Blockchain that supports deposits and payment on another entity's behalf. The following figure illustrates the dishonest behavior.

Dishonest behavior via collusion Collusion refers to the secret collaboration of at least two participants. As Contractor and Outsourcer have fundamentally different interests, collusion is only plausible for either Contractor and Verifier to trick the Outsourcer into believing a false result, or Outsourcer and Verifier to reject payment for the Contractor. The following scenarios can happen if two parties collude:

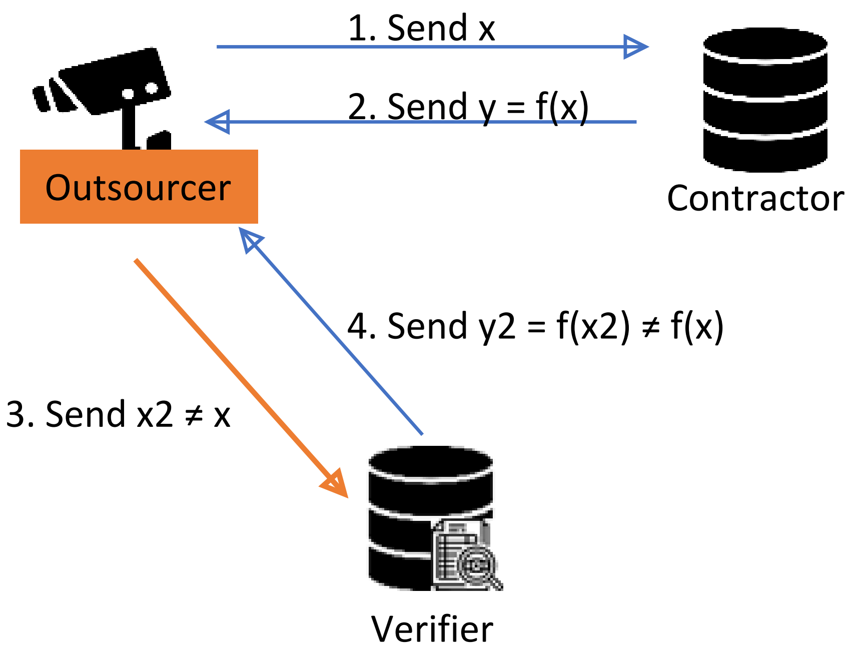


Figure 2.4.: Outsourcer sends different input to Contractor and Verifier to refuse payment

Outsourcer and Verifier collude to refuse payment and save resources Even if inputs of the Outsourcer are recorded via digital signature it cannot prevent that an Outsourcer colludes with a Verifier to send identical inputs to Contractor and Verifier, but receiving a false response from the Verifier, claiming it was the real one. The incentive for the Outsourcer to act dishonestly with this behavior is to reject payment to an honest Contractor. The incentive for the Verifier is to save computational resources by returning a false response and not conducting the assigned function. Potentially, the Verifier could also be rewarded with a proportion of the saved Contractor reward by the Outsourcer. We found no evidence that this scenario is prevented by any of the evaluated verification scheme. Even in schemes such as [40] that outsource unmatching responses to additional Verifier, it seems that (1) this decision is not enforced by the protocol and (2) there seems to be no random assignment of Verifiers which leads to the problem that the Outsourcer can freely pick a colluding Verifier. Thus, we consider this potential dishonest behavior as unaddressed by current academic literature. The following figure illustrates the dishonest behavior.

Contractor and Verifier collude to save resources The biggest risk for the Outsourcer is if the Contractor and the Verifier collude and send false responses to the Outsourcer. Unknowingly, the Outsourcer will continue outsourcing and pay for false responses. The Contractor and Verifier both save resources by being able to use a computationally inexpensive algorithm rather than the function assigned by the Outsourcer. Schemes such as [40] lower the possibility of collusion by designing their incentive structure in a way that being honest is

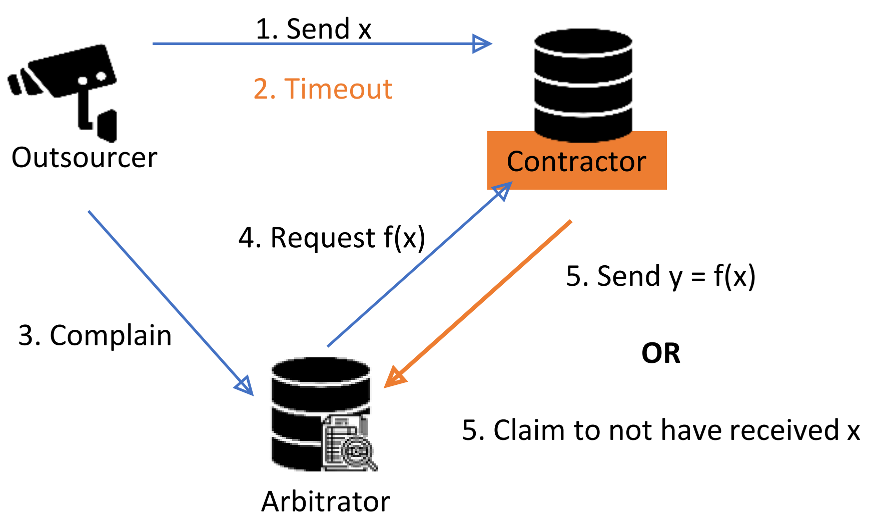


Figure 2.5.: Contractor or Verifier tries to avoid global penalties when convicted of cheating or QoS violations

a dominant strategy for rational expected pay-off maximizing participants. This method only ensures probabilistic protection and may fail if the Verifier and the Contractor are aware of each other's identities. The following figure illustrates the dishonest behavior. The following figure illustrates the dishonest behavior.

Quality of Service violations Quality of Service (QoS) violations refers to the measurement of overall service performance. An Outsourcer is likely to expect a certain QoS from a Contractor and a Verifier such as maximum response time, or minimum response rate. QoS violations can be (1) unintended in case of weak connections, or unexpected timeouts or (2) reckless in case of overutilizing resources to ensure 100% resource utilization at all time or (3) intended to harm an entity. The problem is that it is hard to tell which of the three cases is true if QoS violations take place. Thus, dishonest behavior may not be assumed, and penalties might be lower or non-existent compared to proven dishonest behavior as described above. The following QoS violations might occur.

Timeouts At any point during a contract, a participant may time out due to network problems, over-utilization, or other problems. While some Verification schemes ignore that problem since an Outsourcer could simply cancel the contract, others such as [38] keep local reputation systems that frequently compare all Contractors in their performance. This implicitly leads to the blacklisting of weak performers. Another approach is that the Outsourcer commits to its inputs via Merkle Trees or sends them to a trusted time-stamping server such as in [40]. These techniques come with computational or communicational overhead but can be used to create a globally usable record of when inputs are sent and responses are received. The following figure illustrates the QoS violation.

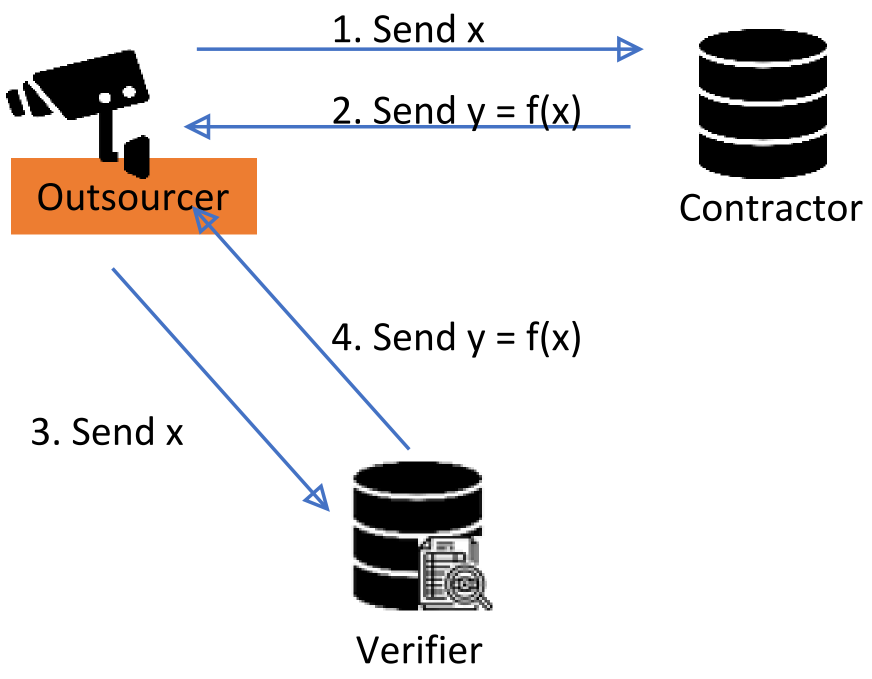


Figure 2.6.: Outsourcer refuses to pay even if obliged to by the protocol

Low Response Rates Low response rates can happen due to weak connection, or if a participant is not able to computationally process a function in the expected time. The same techniques as for timeouts can be used to prevent this behavior.

High Response Time High response times can happen due to network latency, or if a participant is not able to computationally process a function in the expected time. The same techniques as for timeouts can be used to prevent this behavior.

External Threat: Message Tampering While previously explained protocol violations are all caused internally by the participants, there might also be external threats such as an attacker that tries to tamper with messages to harm a participant. We assume that all schemes that utilize signatures such as [39] are resistant to these types of attacks, as an external attacker cannot generate a valid digital signature of a participant over a message it did not send. It should be noted, however, that using Merkle trees along with signatures leads to a brief time window for message tampering between sending an unsigned message and sending the signed Merkle root hash. In this short time window, messages can be tampered with, but the tampering will be detected once the Merkle root does not match received inputs. The following figure illustrates this external threat.

In summary, we identified multiple protocol violations and summed them up by 11 description and 4 types. It should be noted that while 9 out of the identified 11 protocol

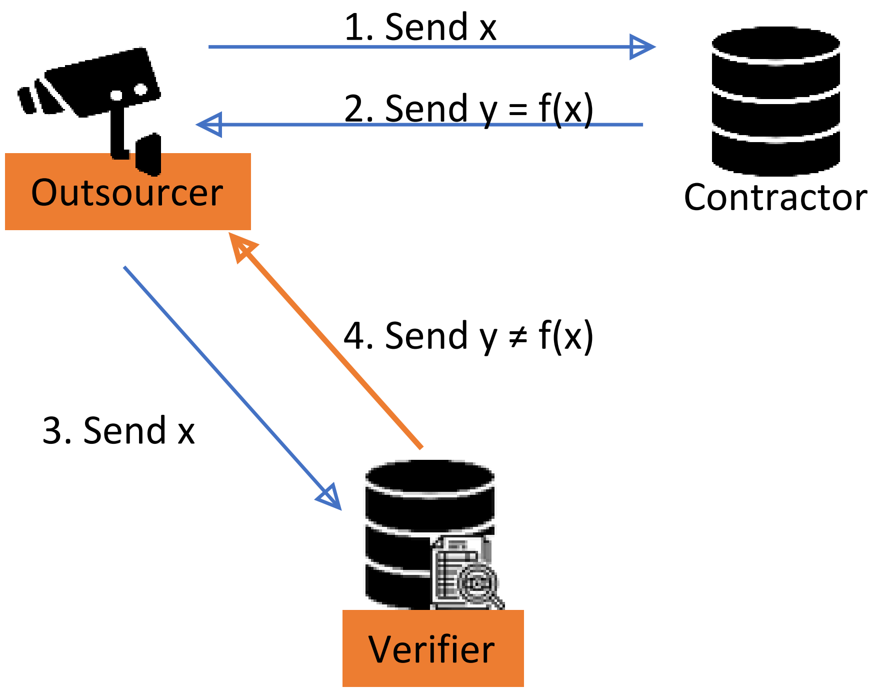


Figure 2.7.: Outsourcer and Verifier collude to refuse payment and save resources

violations seemed to be addressed by techniques used by current academic literature, we found no verification scheme that utilizes at least one technique per identified violation. This leaves the following research practice gaps to design a verification scheme that adheres provides the following attributes:

1. Prevents all of the 9 protocol violations that already have been addressed by existing solutions
2. Proposes solutions for the 2 unaddressed protocol violations
3. Minimizes the use of TTPs, Blockchains, and computationally expensive techniques to remain practicable and scalable

The following table provides an overview of the described potential protocol violations and by which techniques they are addressed by the evaluated academic literature. The next section will introduce our designed verification scheme that aims to prevent all 11 identified protocol violations while remaining efficient, scalable.

Table 2.4.: Protocol Violations and techniques to prevent them proposed by evaluated literature

Type of Violation	Referred Number	Description	Techniques
Dishonest Behavior by Individual	1	Contractor sends back false responses to save resources	Re-execution, utilization of third party Verifiers if required
	2	Verifier sends back false response to save resources	Re-outsourcing input to additional Verifiers if responses do not match
	3	Outsourcer sends different input to Contractor and Verifier to refuse payment	Not addressed
	4	Contractor or Verifier tries to avoid global penalties	Digital Signatures
	5	Participant refuses to pay even if obliged to by the protocol	TTP or Blockchain that is authorized ot conduct payment on behalf on another entity
Dishonest Behavior via Collusion	6	Outsourcer and Verifier collude to refuse payment and save resources	Not addressed
	7	Contractor and Verifier collude to save resources	Incentives designed in a way that being honest is a dominant strategy
QoS Violation	8	Timeout	Blacklisting, Review System,Recording timestamps of Merkle root hashes of inputs and responses to track QoS violations globally
	9	Low Response Rate	Same as Nr 8
	10	High Response Time	Same as Nr 8
External Threat	11	Message Tampering	Digital Signatures

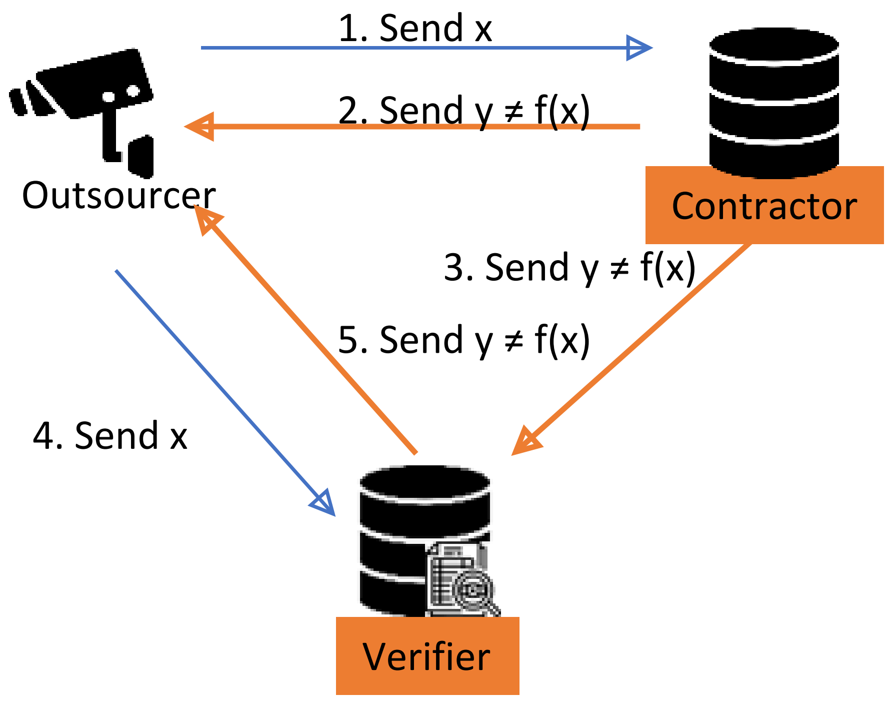


Figure 2.8.: Contractor and Verifier collude to save resources

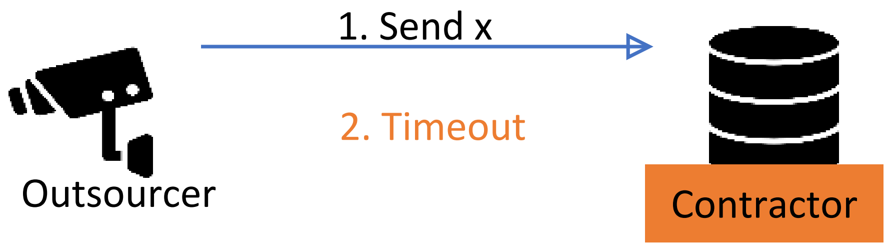


Figure 2.9.: Contractor times out

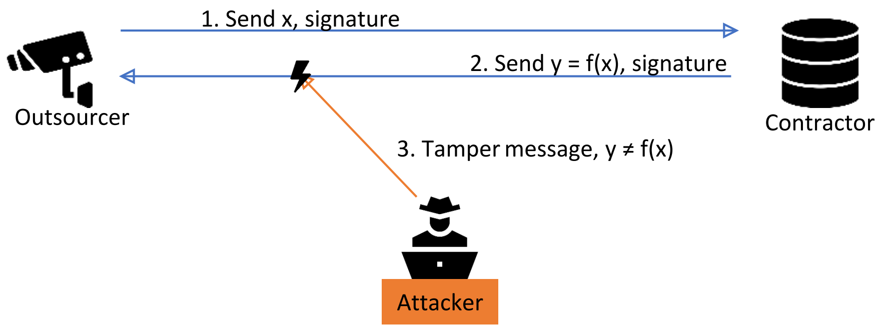


Figure 2.10.: Message Tampering by an external attacker

3. Designed Verification Scheme

This section will introduce our designed verification scheme. According to the research gaps compiled in the last section the following points were of our priority when we designed the scheme:

1. Prevents all of the 9 identified protocol violations that already have been addressed by the evaluated solutions
2. Proposes solutions for the 2 unaddressed identified protocol violations
3. Minimizes the use of TTPs, Blockchains, and computationally expensive techniques to remain practicable and scalable

As our scheme uses re-execution as a verification technique, it can be applied to any deterministic function. Also, the designed verification scheme is suitable for both latency-sensitive Edge Computing marketplaces with scarce resources as well as cloud computing marketplaces with unverified cloud service providers. Nevertheless, we focus on the Edge Computing scenario and outsourced object detection as an application when we introduce assumptions and implementation to provide specific results.

3.1. Overview

Our designed verification scheme begins at the point where an Outsourcer has successfully hired an available contractor for a certain reward per input. Both participants begin with an identical contract object that contains the reward per image, the type of function or model to be used, and other relevant data.

They first engage in a preparation phase. In the preparation phase, both participants generate a random number in a combined manner without the ability for any participant to predict the number. The random number r is only revealed to the Contractor but not to the Outsourcer. We called this process randomization. Randomization ensures that if there is a sorted list of n available Verifiers the random Verifier at position $rmodn$ is picked. To prevent planned collusion, neither Outsourcer nor Contractor can decide on their own which Verifier will be picked. Also, the Contractor is not informed about the choice of the Verifier so it is harder to contact it for an ad-hoc collusion attempt. Both Verifier and Contractor generate an identical contract. After that process, all contracts are checked if their specified fines, rewards, and bounties lead to a dominant strategy of being honest from a game-theoretic incentive. When designing the incentives in such a way, ad-hoc collusion of Verifier and Contractor can be further prohibited. However, adhering to game-theoretic incentives is not enforced by the protocol.

After the preparation phase is done, the execution phase begins. In the execution phase, the Outsourcer digitally signs each input before sending it to the Contractor. The Contractor verifies the signature of the new input, performs the assigned function, and either digitally signs each response before sending it or signs a Merkle root hash over multiple inputs for higher efficiency. The Outsourcer finally verifies the signature of the response. The execution phase is performed as long as one party aborts the contract according to custom or due to a protocol violation. Once in a while, the Outsourcer sends a digitally signed sample input to the Verifier. Like the contractor, the Verifier verifies the signature of the new input, performs the assigned function, and digitally signs each response before sending it. When the Outsourcer receives a response from both Verifier and Contractor, it compares if they are equal and aborts the contract if they are not. Digital signatures are always generated over each input, respective index, and the contract hash to ensure a distinct record of each input. Additionally, the Outsourcer digitally signs with each new message to the Contractor, or the Verifier how many responses it has already received from that participant. Thus, it acknowledges a certain amount of outputs and commits to a payment to that party. If acknowledged output does not match the expected acknowledge rate set by Verifier or Contractor, they might cancel the contract due to protocol violations themselves. In all cases, at the end of the contract, every participant has a clear digitally signed record of honesty and payment promises from each participant it interacts with.

There are three potential scenarios for the closing phase: (1) A contract was aborted according to customs, (2) A contract is aborted due to a QoE violation, (3) a contract is aborted due to dishonest behavior. In the first, and usual case Verifier and Contractor submit their last received input which contains the latest number of acknowledged responses along with a copy of their contract to the payment settlement entity. If signatures match, the Payment settlement entity pays the specified reward per image specified in the contracts on behalf of the Outsourcer to the Verifier and the Contractor. Also, participants can decide to leave a positive review, assuming that the payment settlement entity might as well also feature a global review system. In the second case, a participant may blacklist the party that violated its QoS parameters and leave a negative review at the Payment settlement entity. The third case of dishonest behavior can only happen if a sample response of the Verifier was found unequal to the one of the Contractor. In this case, the Contractor is accused of cheating and receives a fine after a deadline. Within the deadline, the Contractor may perform one of our developed protocols called Contestation. If the Contractor re-outsources the input to additional random Verifiers, and responses match with the response of the Contractor, the Verifier is instead accused of cheating. Verifier and Contractor can perform Contestation as many times as they like at their own cost. The participant that receives the minority of Verifier support for their response is finally accused of cheating and has to pay a fine for cheating and needs to pay all additional verifiers consulted during Contestation. This process ensures that when a participant is accused of dishonest behavior, there is a guarantee to detect the cheating participant if >50% of available Verifiers respond with the correct response.

3.2. Initial Situation

In the initial situation, we assume there are an Outsourcer and a Contractor that have agreed to a contract. The contract contains a unique ID which makes it distinguishable from previous contacts with the same participant, the public keys of the participants that are also registered at the Payment settlement entity. Also, a reward per processed input and the function/model to be used is specified. Additionally, Contractor and Outsourcer may agree on fines, deposits, and bounties if a participant is caught cheating to increase the robustness of the protocol. Also, we assume multiple available verifiers are available and willing to agree on a contract with the Outsourcer to process random sample inputs. The following figures illustrate the initial situation.

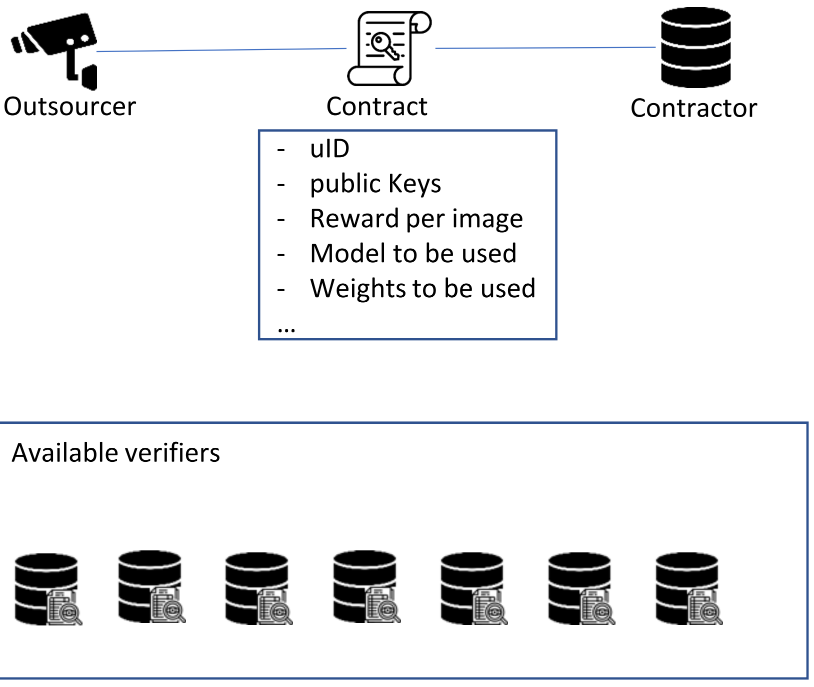


Figure 3.1.: Initial Situation

3.3. Preparation Phase

After the initial situation the objectives for the Preparation Phase are the following:

1. The Outsourcer and the Contractor should commit to a random Verifier
2. The Contractor should not know which Verifier is selected
3. The Verifier should not know which contractor is assigned to the job

   To explain these objectives we first have two distinguish between two types of collusion: Planned collusion and ad-hoc collusion.

We describe planned collusion as the scenario where two participants, either a Contractor and a Verifier, or a Contractor and an Outsourcer know each other initially and try to trick the other participant by colluding. To reduce the risk of planned collusion, Contractor and Outsourcer engage in a protocol we designed and called Randomization. The result of the protocol is the selection of a random Verifier that can not be set initially by the Outsourcer, nor the Contractor.

We describe ad-hoc collusion as the scenario where a Contractor and an Outsourcer do not know each other initially but still try to communicate and collude with each other because they expect individual benefits such as spending less computational resources by submitting a false response. This explains our second and third objectives. Thus, we designed Randomization in a way that Verifier and Contractor do not get to know each other's identity and no communication is present between them.

3.3.1. Randomization

At the start of Randomization, the Outsourcer and Contractor internally generate a large random number. The following steps are executed afterward:

1. The Outsourcer digitally signs the hash of its chosen number x and the contract hash without revealing the chosen number x before sending it to the Contractor.
2. The Contractor digitally signs the received hash by the Outsourcer along with its chosen value y, the contract hash, and a list of available Verifiers sorted by their public keys that meet the conditions specified by the Outsourcer in the contract. Along with this signed hash, the Contractor sends the value y and the list of available Verifiers for the Outsourcer to review. By signing the initially sent hash of the Outsourcer, the Contractor commits to have accepted a random number by the Outsourcer without knowing it.
3. If the list of available Verifier matches the current local list of the Outsourcer with high confidence, and the signature matches with all revealed messages, the Outsourcer contacts the Verifier at $(x * y) \mod n$ in the list of available Verifiers. The Outsourcer sends the signature of unsigned x, y, and the unsigned list to the Verifier. The Verifier might reject the contract if it's not located at the specified position in the list. If the Verifier does not respond, the whole protocol is repeated until an available Verifier agrees to the contract.

The figure below illustrates the protocol.

The result of this process is that a Verifier is selected that adheres to the defined objectives. As both the Outsourcer and the Contractor have signatures over each other's signed numbers, these can later be verified by the payment settlement entity if there is a dispute. Thus, a record is available if the Contractor and Outsourcer adhered to the protocol. This record is necessary to prevent the Outsourcer from ignoring the number of the Contractor to pick a Verifier by its choice.

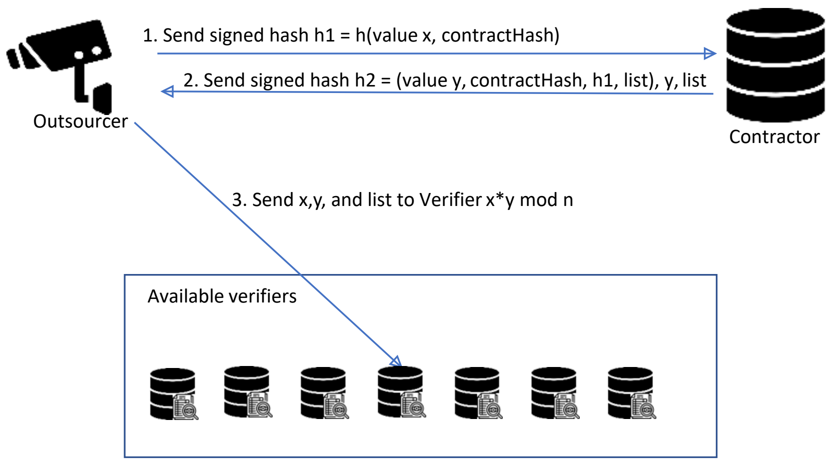

1. Send signed hash h1 = h(value x, contractHash)
2. Send signed hash h2 = (value y, contractHash, h1, list), y, list
Outsourcer
Contractor
3. Send x,y, and list to Verifier x*y mod n
Available verifiers

Figure 3.2.: Randomization Overview

The protocol does not add any noticeable overhead to the system as the preparation phase is only performed once per contract and is expected to take less than 5ms processing time on most machines. The results of the protocol are the following:

1. The Outsourcer and the Contractor should commit to a random Verifier
2. The Contractor does not know which Verifier was selected
3. The Verifier does not know which contractor was assigned to the job
4. Record of signed random numbers associated with the specific contract is available

3.3.2. Game-theoretic Incentives

After Randomization, both the Verifier and the Contractor have received a contract identical to a contract of the Outsourcer. Even when Verifier and Contractor do not know each other there is a theoretical risk for ad-hoc collusion. This is the case, for example, if there exists a q-algorithm that is computationally inexpensive but provides a correct result with a certain probability q. For object detection, this might be the naive response that no object was found and therefore no bounding boxes at specific co-ordinates have to be estimated and returned. If we assume there exists a most efficient q-algorithm, there is a possibility that a lazy Contractor and a lazy Verifier may act rationally by both deciding to use this most efficient, known q-algorithm without the need of ever planning to collude. As described in the literature review chapter 2, there is a solution to lazy behavior and collusion by designing incentives in a certain way. Verification schemes such as [40] have identified a relationship of incentives with adding fees cheating participants and bounties for honest participants such that being honest is a dominant strategy from a game-theoretic point of view. When designing incentives correctly, the use of a q-algorithm is not the decision with the highest expected payoffs, as the fine when detected cheating might be higher than $q * r$ (where r is the reward when not detected cheating). Also, by adding a bounty to the participant that catches the other participant cheating, the expected payoff for honest behavior increases. The following figure illustrates both contract pairs with a bounty, reward, and fine.

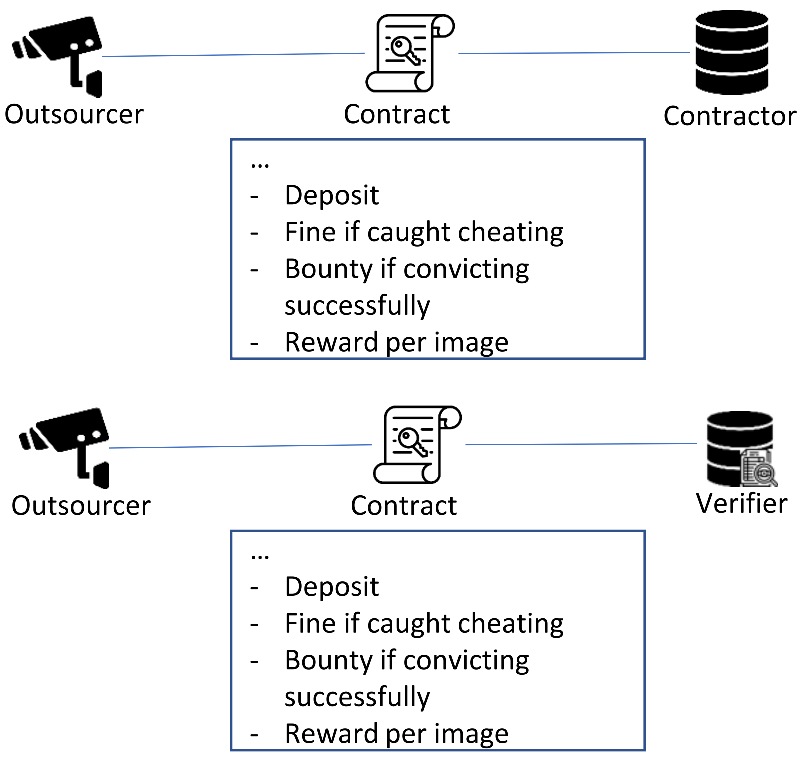
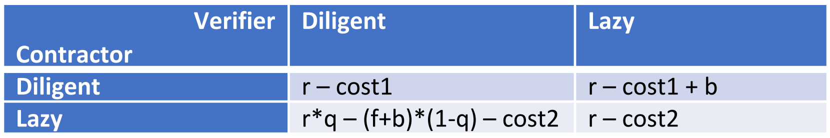

Outsourcer
Contract
Contractor
...
- Deposit
- Fine if caught cheating
- Bounty if convicting successfully
- Reward per image
Outsourcer
Contract
Verifier
...
- Deposit
- Fine if caught cheating
- Bounty if convicting successfully
- Reward per image

Figure 3.3.: Contracts with incentive related agreements

The payoff matrix with the use of a bounty can be seen below. If the bounty $b > cost1 + cost2$ where b is the bounty $cost1$ is the cost of performing the function and $cost2$ is the cost of performing the q-algorithm, and $(f + b) * (1 - q) - cost2 > r * q$ where f is the fine when caught cheating, and q is probability of the q-algorithm returning a correct result, ad-hoc collusion is not rational. In this case, the expected payoff when both participants act honestly is the highest. Even if not all values in the formulas are known before such as the success rate q or the exact costs, a moderate fine and bounty compared to the reward such as a 20:1 ratio are already sufficient with high probability. In the implementation, we will show how we found an upper bound for cost1 and cost2 to utilize for the calculation.

Finally, a deposit ensures in a real-world system that the fine can be paid as part of the balance is locked for spending in the meantime.

Due to the unlikeliness of ad-hoc collusion with a random Verifier, we chose to not enforce contracts adhering to described game-theoretic incentives. Nevertheless, if a payment scheme supports deposits, fines, and bounties game-theoretic incentives can be used to increase security at a tiny, non-measure-able computational overhead.



Verifier Contractor	Diligent	Lazy
Diligent	r – cost1	r – cost1 + b
Lazy	r*q – (f+b)*(1-q) – cost2	r – cost2

Figure 3.4.: Payoff Matrix

3.4. Execution Phase

3.4.1. Sampling

Sampling refers to picking one random input out of a collection of inputs. In our verification scheme, the Outsourcer can send samples to the Verifier to check whether its response matches the response of the Contractor on the same input. We call this process sampling-based re-execution. Sampling-based re-execution has the advantage over complete re-execution that just with a few number of samples, a dishonest Contractor with a cheating rate of c can be detected with nearly 100% confidence. Thus, we can significantly improve the efficiency of the verification process at a neglectable security draw-down.

The following example should explain this statement. We construct a scenario where an Outsourcer sends 10000 inputs to a Contractor. If the 10000 inputs are a video stream at 60 frames per second, this contract is less than 3 minutes long.

Let's suppose an Outsourcer performs complete re-execution, meaning that it sends every input to both the Contractor and the Verifier. Assuming the Verifier is acting honestly in this scenario, a Contractor with any cheating rate c can be found with 100% confidence, but the computational overhead of the verification is at least 100% compared to not performing any verification. Thus, to achieve 100% confidence we need to check all

Let's suppose an alternative strategy, where the Outsourcer splits up the inputs in i intervals and sends only one random sample per interval to the Contractor. In this case the chance p of detecting a cheating attempt is $p = 1 - (1 - c)^i$. If we want to catch a cheating Contractor with at least 99% confidence we simply have to assume a cheating rate and solve the equation for $0.99 >= 1 - (1 - c)^i$. Let's assume a difficult case where the Contractor cheats only in 10% ($c = 0.1$) of all cases. In this case by solving the equation our interval size becomes $i >= 43.8$. This means, independent of the total number of inputs, the Outsourcer will catch a cheating Contractor even in a difficult and unlikely case with more than 99% confidence with only 44 samples. In our short contract with 10000 inputs, this results in a verification overhead of 0.44%. In case we send 100,000 inputs (less than 28 minutes of video at 60fps) this overhead already decreases to 0.044%. Thus, we can conclude that even for short contracts a sampling rate of less than 1% is sufficient to detect a rational, dishonest Contractor with nearly 100% confidence.

We expect that in real-world scenarios rational, dishonest Contractors try to cheat in more than 10% of the cases as with a cheating rate of 0.1 they would still need to perform at least 90% of required computational effort but risk being detected, losing their reward and perhaps even receiving a fine that exceeds their reward expectation. Furthermore, we expect average contracts for most use cases to last more than 3 minutes as aborting a contract according to custom is only useful if (1) a Contractor with a better QoS, or cheaper rates becomes available, (2) the application is stopped, or (3) the Outsourcer moves out of range of the Contractor. Therefore, we consider our example as a stress test and suppose even less computational overhead in real-world scenarios.

Nevertheless, we recommend instead of setting a fixed number of samples, to instead set a sampling rate that checks samples in even intervals. After all, it is hard to predict how long a contract will last, as all parties can cancel the contract according to custom at any time without specifying a reason. The situation should be avoided that (1) either too few samples were sent because the contract was shorter than expected, or (2) that samples were sent too early because the contract was longer than expected. Case 2 either leads to an uneven distribution of samples or demands the Outsourcer to send additional samples whenever the contracts exceeds the expected length. For most use cases, we recommend setting a sampling rate between 0.01% and 1% and from the beginning on always storing 50 inputs locally that can be still verified after a contract ends after a surprisingly short time. This strategy ensures an even distribution of samples, as well as a high-security guarantee even with short contracts. While the resulting computational overhead is already very low further optimization could include lowering the sampling rate, the longer the contract goes on, or adjusting the sampling rate if a global reputation system is present.

3.4.2. Signatures

Since in our verification scheme participants communicate with non-monitored peer-to-peer communication we need a way to securely record payment promises and dishonest behavior. Otherwise, an Outsourcer could claim to never have received any responses from a Contractor or a Verifier, and could accuse another party of faking records that tell otherwise. Likewise Contractor and Verifier could deny that a dishonest response originated from them and could claim to have processed more responses than they did. For a payment settlement entity, it is impossible to solve a dispute and hold entities accountable without tamper-proof records.

Digital signatures can solve this problem as each participant can have a public key registered at the payment settlement entity, and only knows its private key that can be used to sign messages. As forging a digital signature of another entity is infeasible without knowing the associated private key, any message that is signed with the public key of a participant can be used as a verifiable record of communication. Thus, if the Contractor or the Verifier commit to a response by generating a digital signature over it and sending it along with the response, the Outsourcer can prove to any third party that the response indeed originated from that participant. Likewise, if the Outsourcer signs how many responses it has received from a Contractor, or a Verifier along with other contract-related information, this record can be used to redeem payment from a payment settlement entity on behalf of the Outsourcer's account.

Figure 3.5 shows a high-level overview of sampling in combination with digital signatures. $x = r$ refers to the property of our scheme that an input x is only sent to the Verifier if its index is chosen as a random number within the current interval.

When an Outsourcer sends an input with our scheme it is required to always send a

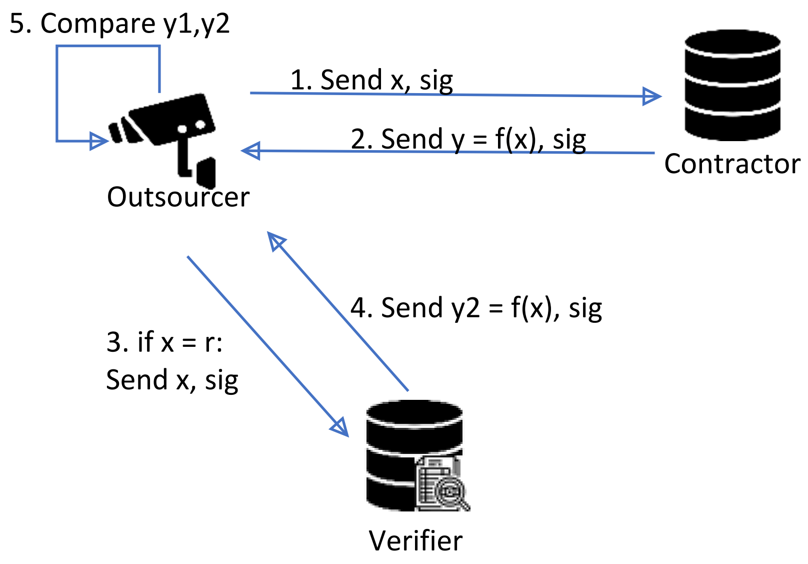


Figure 3.5.: Execution phase with signatures

signature with it, which is signed over the current input index, the contract hash, and the input itself. This ensures that each signature can be traced back to exactly one unique input in a contract context. Even if an identical input is sent later in the contract or in another contract, signatures do not match as the index number or the contract hash would be different. Whenever receiving an input from the Outsourcer, they simply have to check whether the input and input index send along the signature, and the contract hash only know to two participants (which they own a copy of) can be used to verify the signature. If the signature is not valid, they abort the contract according to a QoS violations, leave a bad review to the Outsourcer and blacklist it. The contract cannot be aborted due to dishonest behavior as it cannot be proven where a fake signature originated from.

When the Contractor or the Verifier sends a response to the Outsourcer, they also send the input index that the response is associated with and the input, along with a digital signature forged over contract hash, input index, and the input itself. The Outsourcer verifies each signature and aborts the contract according to a QoS violation if the signature does not match the designated values and may also leave a bad review to the participant and blacklist it. If a signed response returned by the Verifier and the Contractor over the same input index is found to be unequal, the Outsourcer now has a record of both participants that committed to their responses. The Outsourcer always assumes that the Contractor is dishonest in this case and can abort the contract due to dishonest behavior, sending both unequal responses and their signatures as proof to the payment settlement entity. The Contractor has the chance to prove that it was instead the Verifier that sent the false response, which is explained in detail in the section on Contestation.

There also needs to be a way for the Outsourcer, and the Verifier to redeem payment. As we consider scenarios with frame loss, weak participants, and weak network connections, we do not assume that just because a message was sent by a participant it was also received and processed by the other participant. Thus, the Outsourcer acknowledges each received by sending a counter of acknowledged responses from that participant along with each input and integrates the counter into its signature. Verifier and Contractor specify in their internal parameters which loss rate they tolerate before aborting the contract due to Qos violations. All parameters that specify a maximum loss rate or number of consecutively unacknowledged outputs mustn't be revealed to the Contractor. Otherwise, the Contractor may exploit this parameter to always acknowledge only enough inputs so that the other participant continues the Contract. The sent counter of acknowledged outputs is a 32 bit (4 bytes) integer that causes neglectable network overhead and comes with another advantage: When a contract is aborted no matter if according to customs or other reasons, the last input received from the Outsourcer contains a signed number of all acknowledged responses throughout the contract. Thus, Contractor and Verifier only need to store the latest input in their memory and submit this single input after the contract ended to the payment settlement entity to redeem their payment. With this approach, we also do not need any TTP that monitors or records communication between participants. Sending counters alongside inputs and signatures is illustrated in figure 3.6.

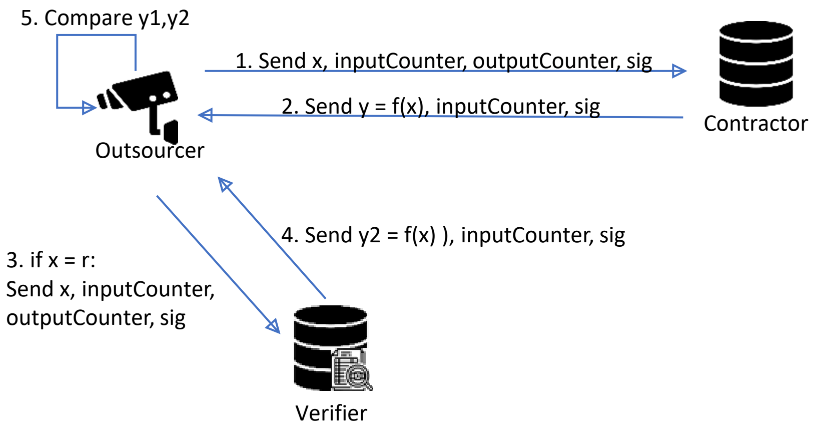

5. Compare y1,y2
1. Send x, inputCounter, outputCounter, sig
2. Send y = f(x), inputCounter, sig
Contractor
Outsourcer
4. Send y2 = f(x)), inputCounter, sig
3. if x = r:
Send x, inputCounter,
outputCounter, sig
Verifier

Figure 3.6.: Execution phase with signatures and acknowledged responses

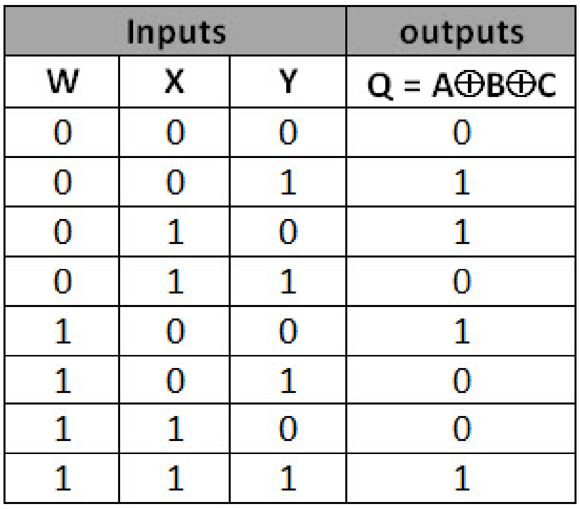
The Outsourcer also sets internal parameters such as a minimum receive/process rate from Contractor and Verifier, a maximum number of consecutive inputs that remain without a response. Unlike the Contractor and the Verifier, it may decide to publish these values as a lower amount of responses received also results in fewer rewards for those participants. Nevertheless, we decided to not share any internal parameters between participants that lead to QoS violations. These, internal, disclosed parameters that can be set by each participant according to their expectation in combination with the consequences of aborting a contract due to bad QoS, make our scheme compatible with lossy networks and weak participants while giving incentives for participants to provide reliable service. These techniques are used to prevent the identified QoS violations (number 8-10 in the list of possible protocol violations2.4), timeouts, low response rate, and high response time.

By utilizing digital signatures, our verification scheme is also resistant to message tampering (number 11 in the list of possible protocol violations 2.4), and Contractor or Verifier trying to change their record or making false claims when accused of cheating (number 4 in the list of possible protocol violations 2.4). A message that is tampered with by an attacker immediately leads to the receiving participant aborting the contract as the attacker can not forge a valid signature of the tampered message.

3.4.3. Improving Efficiency of the Execution Phase

Signing a digital stream via one-time signatures

1. Sign each chunk at index i while sending the newly generated public key with chunk i-1
2. One-time signature schemes are faster than multi-use digital signature schemes in theory
3. Real-world implementations of Lamport and Winternitz one-time digital signatures schemes were found to be slower than ED25519 in the benchmarks



Inputs			outputs
W	X	Y	Q = A⊕B⊕C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 3.7.: Truth table of 3 input XOR

Signing a digital stream via Merkle Trees

1. Instead of signing each response, the outsourcer only signs the root of the Merkle root in a specified interval and the associated challenge sent afterward
2. Implementation reduces total time spent on signing and verifying
3. Merkle trees are also useful when not all responses are significant for the Outsourcer but it still wants to make sure that all inputs are processed

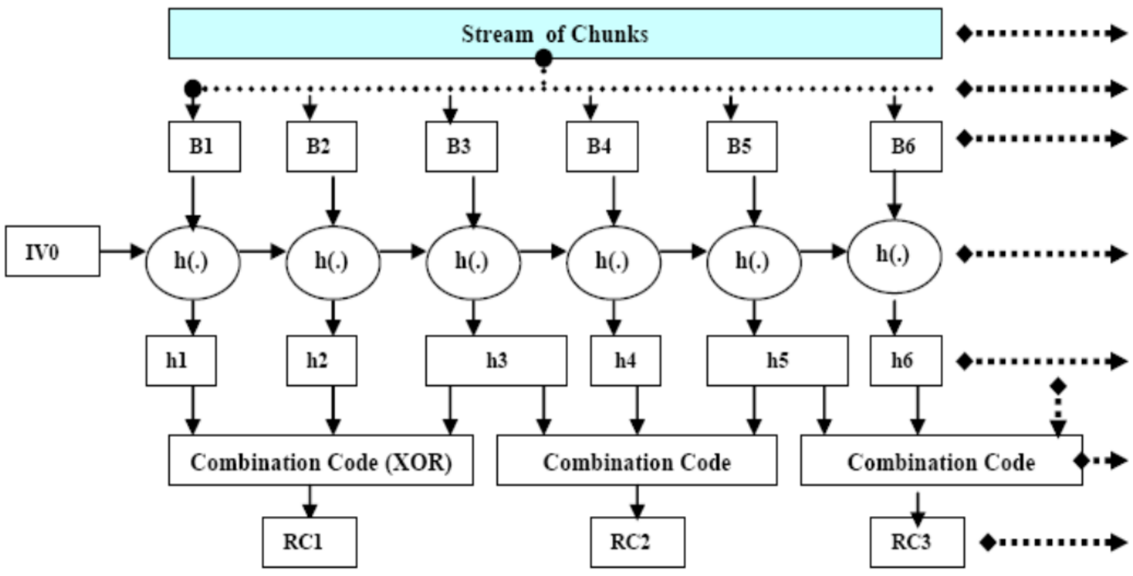


Figure 3.8.: Hash Chain of a digital stream with combination codes using XOR

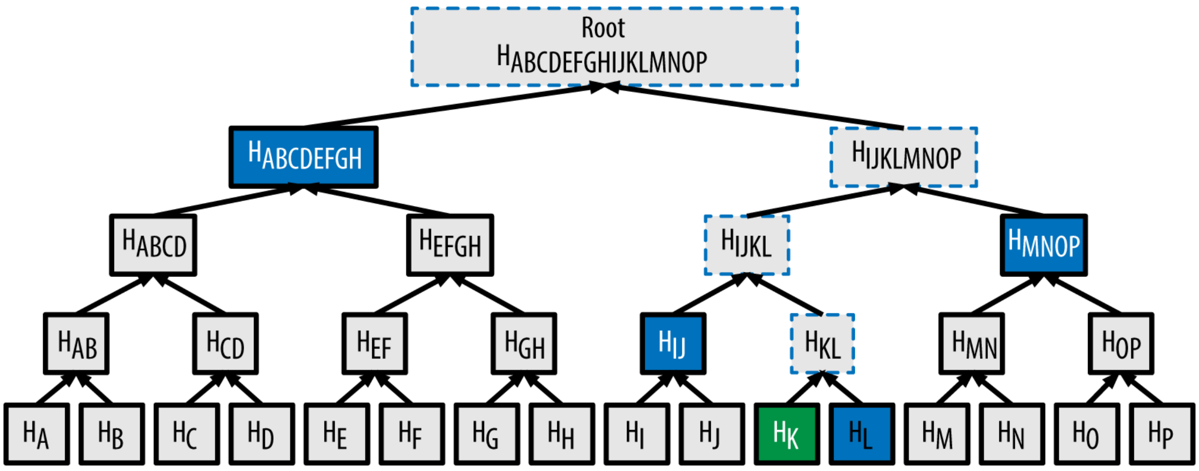


Figure 3.9.: Visualization of a Merkle Tree and a Proof of Membership challenge

3.5. Closing Phase

A contract can be aborted according to custom, due to QoS violations, or due to dishonest behavior. In the closing phase, the Contractor, and the Verifier submit the last signed input from the Contractor to the Payment settlement entity to redeem payment for their acknowledged responses. The Outsourcer may report the Contractor for dishonest behavior if it received a sample response from the Verifier that does not match the response of the Contractor. Each contract also allows for one review of the counter-party specified in the contract at the Payment Settlement entity. The payment settlement entity does not have to be located in proximity to the other machines and can handle payments with an arbitrary delay.

3.5.1. Review System

Except when found guilty of dishonest behavior, each participant can leave one review per contract to the specified counter-party in the contract. As there needs to be a trusted payment settlement entity anyways to handle payment, this entity might also feature a review system. While a review system is not necessary to protect the security of our scheme, it can be used to filter out reliable, trustable participants that provide a good QoS.

Potentially, when an entities' review balance gets negative it can just generate a new identity with a neutral balance. However, if the payment scheme that is used along our verification scheme requires a registration process, simply generating a new identity and registering a new public key at the payment settlement entity might be prohibited. However, as we focus on the verification scheme in the context of this thesis, we do not make any assumptions beyond that there exists a payment settlement entity that supports our protocol and can make payments on other entities' behalf.

Nevertheless, a review system that allows participants to get rid of a negative balance is still useful if positive reviews can not be faked. It still provides an incentive for participants to provide good QoS, and act honestly. Also, an Outsourcer might choose a higher reward to a reliable Contractor, or Verifier as it can expect a decent Qos, and honest behavior. It may also reduce the sampling rate when it chooses a Contractor with a high reputation. Likewise, Contractors and Versifiers might expect a lower reward per response if they can expect that the Outsourcer acknowledges a large number of responses and does not timeout. Figure 3.10 illustrates our review system, which proposes one review per contract.

3.5.2. Blacklisting

While a review system is ecosystem-wide, a local reputation and blacklist record is still useful for participants to ensure that they do not engage in a contract with another participant again. Also, Blacklisting is not requiring any infrastructure to provide a global review system and a local reputation can be disclosed to other participants. This lowers the chance of a participant to generate a new identity due to a negative review as it might not be aware of being blacklisted.

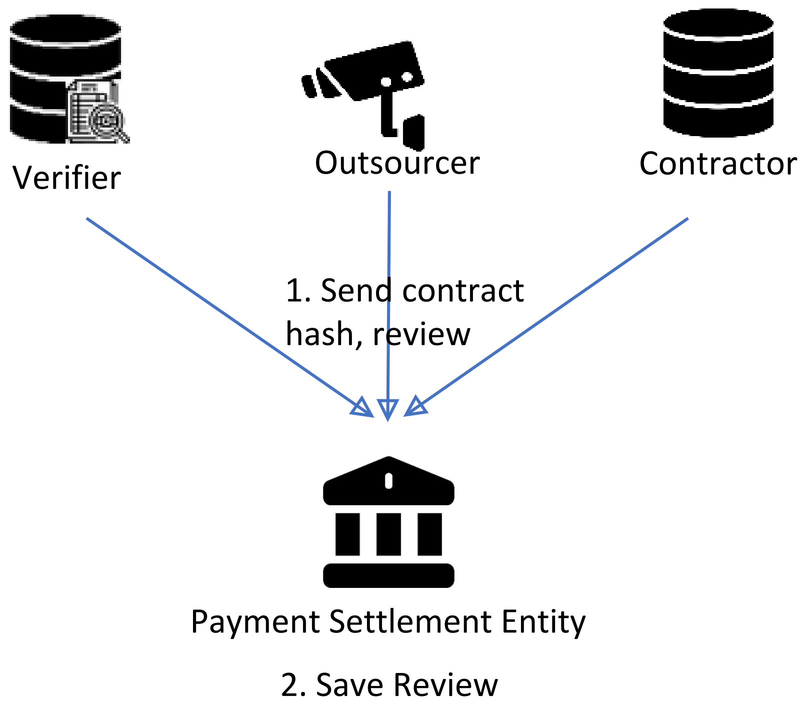


Figure 3.10.: Reviewing a participant after a contract

3.5.3. According to Custom

A participant can abort a contract at any time if it sends an abort message to each counterparty. This message is important to not receive a bad review due to unnoticed timeouts. When a party other than the Outsourcer aborts a contract, an early abort message also leaves time for the Outsourcer to find a replacement and still be serviced. After the contract was aborted according to custom, the Verifier and the Contractor store their last signed input of the Outsourcer, that contains the latest number of acknowledged outputs, and the signed contract hash. They send the whole signed input, along with each unsigned value over which the signature was forged, along with the contract to the payment settlement entity. The payment settlement entity simply has to check whether all inputs can be verified by the sent signature, and deduct the reward per input specified in the contract times the number of acknowledged output contained in the last input, on behalf of the Outsourcer after a deadline. The deadline is needed for the Outsourcer to report dishonest behavior, in which case either the Contractor or the Verifier would receive a sign instead. If the Outsourcer does not report dishonest behavior, the Contractor, and Verifier are assumed to be honest. Figure 3.11 shows the process of redeeming a payment after a contract ended.

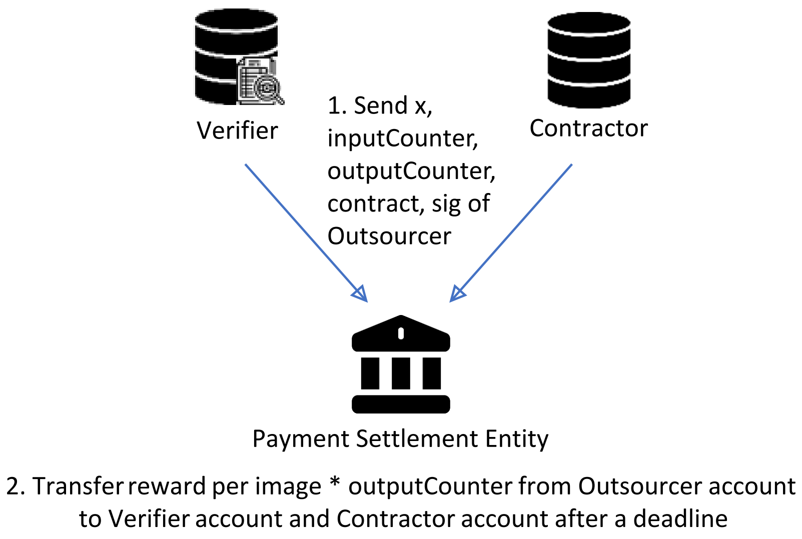

Verifier
1. Send x, inputCounter, outputCounter, contract, sig of Outsourcer
Contractor
Payment Settlement Entity
2. Transfer reward per image * outputCounter from Outsourcer account to Verifier account and Contractor account after a deadline

Figure 3.11.: Redeeming payment after closing a contract

3.5.4. Quality of Service Violations

As described previously, participants can set internal parameters such as a maximum delay or minimum response rate to specify when a contract is aborted due to Quality of Service (QoS) violations. The following list gives an example of QoS violations that might occur.

1. Outsourcer's perspective: Response delay of the Contractor is too high.
2. Contractor's perspective: Outsourcer does not acknowledge enough responses.
3. Verifer's perspective: Outsourcer timed out.

It is important to make a distinction between QoS violations and dishonest behavior as circumstances such as a weak connection may not be the fault of a participant and unintended. Additionally, QoS violations can be detected instantly and their damage is lower than that of dishonest behavior. For these reasons we propose to not fine a participant for a QoS violation but limit its consequence to aborting the contract, submitting a bad review, and potentially blacklisting that participant.

3.5.5. Dishonest Behavior

Only the Outsourcer can abort a contract due to dishonest behavior in case a response sent by the Verifier and a response sent by the Contractor belonging to the same input are unequal. In this case, the Outsourcer sends the input, both responses, their signatures, and the contracts to the payment settlement entity. The Payment Settlement Entity does not re-execute the input, bot only checks if all values match their signatures, and verifies if responses are indeed unequal. Provisionally, the Contractor is accused of cheating. If a fine is set in the contract between Outsourcer and Contractor, the Payment Settlement Entity deducts the fine on the Contractor′s behalf after a deadline. If a bounty is set in the contract between Outsourcer and Verifier, a part of the fee is used to pay that bounty. Within the specified deadline, the Contractor can decide to engage in a protocol we are calling Contestation to prove that response of the Verifier was instead false. Figure 3.12 illustrates the process of reporting two unequal responses.

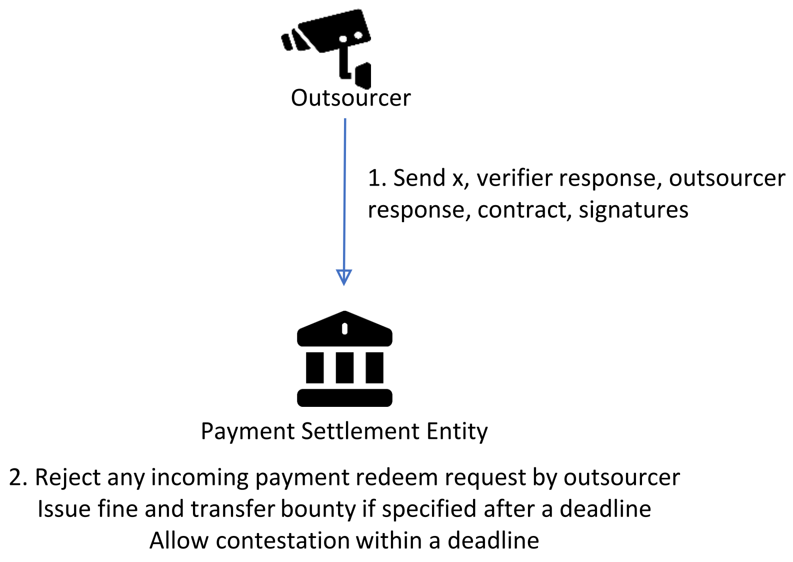

Outsourcer
1. Send x, verifier response, outsourcer response, contract, signatures
Payment Settlement Entity
2. Reject any incoming payment redeem request by outsourcer
Issue fine and transfer bounty if specified after a deadline
Allow contestation within a deadline

Figure 3.12.: Reporting dishonset behavior

3.5.6. Contestation

We designed the Contestation protocol to ensure that an honest participant that is falsely accused of cheating, can prove its innocence. If a Contractor is reported for dishonest behavior because of a false response it may decide to re-outsource the original input to two additional random Verifiers that are available within a deadline. If both random Verifiers return a response that matches the response of the contractor, it presents their responses and signatures to the Payment Settlement Entity. In this case, the Contractor has a majority of random Verifiers supporting its response. Thus, the Verifier is consequently accused of having submitted a false response. Figure 3.13 illustrates Contestation.

If a Verifier is accused of cheating it can use the identical protocol to find two additional random Verifiers to flip the majority of random Verifiers to agree with its response. This protocol might be repeated until no available Verifiers are left. In that case, the participant having the majority of Verifier responses matches with theirs is assumed to be honest, while the other is assumed to be dishonest and gets fined. If more than 50% of available Verifiers are honest, Contestation serves as a guarantee that the participant who responded with a false response is found guilty. In combination with a nearly 100% detection rate of cheating using sampling-based re-execution, any cheating parting will be eventually found guilty with

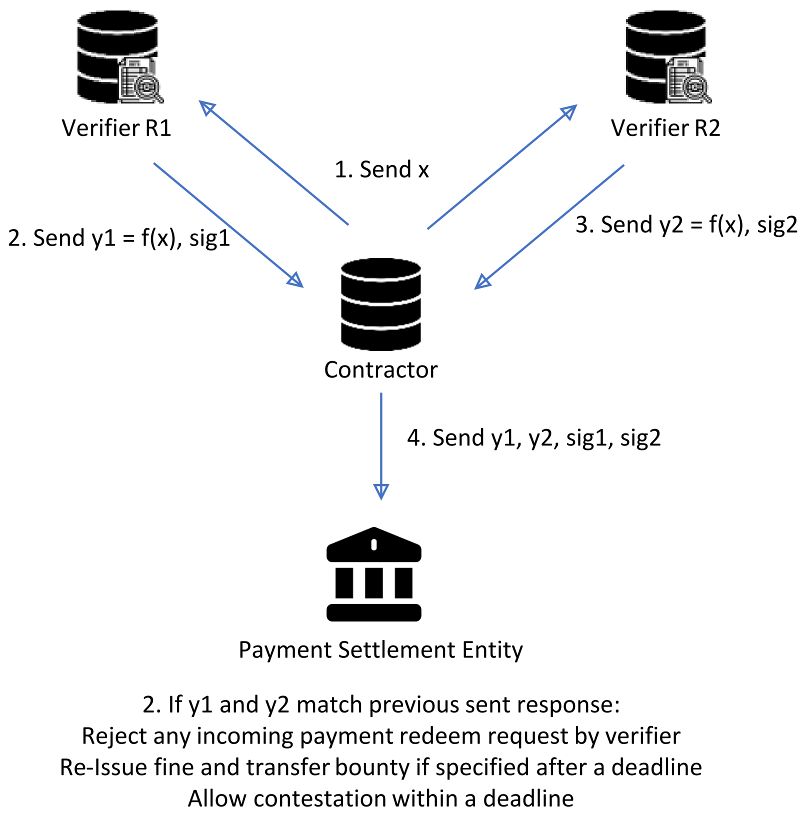

Verifier R1
Verifier R2
1. Send x
2. Send y1 = f(x), sig1
3. Send y2 = f(x), sig2
Contractor
4. Send y1, y2, sig1, sig2
Payment Settlement Entity
2. If y1 and y2 match previous sent response:
Reject any incoming payment redeem request by verifier
Re-Issue fine and transfer bounty if specified after a deadline
Allow contestation within a deadline

Figure 3.13.: Contestation

high probability. The combination of Contestation and sampling is designed to make any cheating attempt irrational. The participant that is found guilty at the end of the protocol has to not only pay the specified fee in its contract but also the additionally consulted Verifiers.

It should be noted, that only for an innocent participant it is rational to perform Contestation as additional random Verifiers have to be paid for their service. Assuming an honest ecosystem, a cheating participant that is performing Contestation is wasting additional money.

The computational overhead of Contestation is low as only one input has to be re-computed. However, it requires to find multiple available Verifiers in the system. As latency is not critical in this scenario, those random Verifiers do not have to be located in proximity and can be computationally weak devices. We expect that Contestation is rarely used or not used at all but provides security for any honest party that it cannot be falsely convicted of cheating due to an unfortunate matching with dishonest participants.

Contestation addresses number 2,3, and 6 in the list of possible protocol violations summarized in table 2.4.

3.6. Threat Model

In our Verification scheme, the Outsourcer, the Contractor, and the Verifier are untrusted and may behave dishonestly. Thus, our threat model considers all possible protocol violations that we identified and listed in table 2.4. This means we consider internal threats of dishonest behavior by one participant, and by collusion. Also, we consider QoS violations such as timeouts and low response rates. In addition, we consider the external threat of an outside attacker trying to tamper with messages sent between participants. Our verification scheme is able to resist all identified threats with high probability. In this section, we will explain how our techniques prevent each threat. An explanation of each threat and motivation behind can be found in the chapter 2.

3.6.1. 1. Contractor sends back false responses to save resources

If a Contractor sends back false responses, the Outsourcer detects this with high probability by sending random samples to the Verifier and comparing if responses belonging to the same input from both participants are equal. In the section on Sampling, we explained that even with a low number of samples, a cheating Contractor can be detected with nearly 100% confidence. Since the sampling rate can be adjusted by the Outsourcer, it may also decide to pick a sampling rate of 100% to detect false responses with 100% confidence.

3.6.2. 2. Verifier sends back false responses to save resources

When the Outsourcer detects two unequal responses from the Verifier and the Contractor, our Verification scheme provisionally accuses the Contractor of cheating. However, the Contractor can perform our developed Contestation protocol to prove that it was the Verifier that sent the false response. In the Section on Contestation, we explained that if more than 50% of available Verifiers in the whole ecosystem are honest, Contestation guarantees to detect which participant was the one cheating.

3.6.3. 3. Outsourcer sends back different inputs to Contractor and Verifier to refuse payment

A cunning dishonest behavior of the Outsourcer, that is not addressed by current academic literature, is to send two different inputs to the Contractor and the Verifier under the same index. Even if the Contractor and Verifier respond honestly, this almost inevitably leads to unequal responses. This behavior can only be detected if a proof is available that the responses were resulting from the different input. Thus, not only the Contractor and Verifier have to sign their responses in our Verification scheme but also the Outsourcer has to sign each sent input with contract-related information. If the Contractor and Verifier also sign the Outsourcer's input signature with their responses, they clearly committed to a specific input and not only an input index. With the help of the information that the Outsourcer has to present when reporting two unequal responses, it can be verified if the signature of the response sent by the Contractor and the Verifier, was built upon the input signatures that the Outsourcer presents in that process. Finally, it can be verified if the input signature that the Outsourcer and Verifier committed on was built upon the same raw input. Once the Contractor initiates Contestation, these proofs get checked to detect this dishonest behavior with 100% confidence. While the process may sound complicated, it is computationally inexpensive (verifying 4 signatures after the contract ended), only requires the Outsourcer to store relevant information of the current sample which leads to $O(1)$ memory overhead, and does not lead to any memory overhead for the Contractor and the Verifier.

3.6.4. 4. Contractor or Verifier tries to avoid global penalties when convicted of cheating or QoS violations

Even when a Verifier or Contractor is detected cheating by an Outsourcer, they may claim to never have sent the response that was reported. This threat is easily prevented with the use of digital signatures. Our verification scheme requires the Contractor and the Verifier to forge the digital signature they send along with the response not only over the output value but also over contract hash, and input index. This way, each input is uniquely identified by its contract (ensured to be unique due to the contract ID), and its index during execution. If a signature of the Contractor or the Verifier is presented along the response, any dishonest behavior aimed to change the record or to resubmit a response is detected with 100% confidence.

3.6.5. 5. Participant refuses to pay even if obliged to by the protocol

Even if the Outsourcer is obliged to reward an honest Contractor, or Verifier, there needs to be a way to enforce the payment. Likewise, the Verifier or the Contractor might try to reject paying a penalty fee when detected cheating. As microtransactions with each response are not an option for us due to a latency bottleneck of the payment scheme, and high transaction costs, payment has to be handled after a contract ended. Thus, we assume that the payment scheme that is used along with our verification scheme supports deposits and payment on other participant's behalf. Also, it needs to be able to verify the signatures of the participants. However, it does not have to be able to recompute any values or execute contract specific functions. Any TTP or Blockchain that supports these requirements can be used with our verification scheme that (1) deducts a deposit of each participant, (2) verifies if signatures are valid to form a conclusion after the contract and Contestation deadline end, and (3) use the deposits to conduct payments.

3.6.6. 6. Outsourcer and Verifier collude to refuse payment and save resources

The Outsourcer and the Verifier may collude to report the Contractor for cheating. This dishonest behavior will be detected by Contestation with 100% confidence if more than 50% of available Verifiers in the whole ecosystem are honest. Nevertheless, we provide additional measurements to avoid this type of collusion. With the use of Randomization, the Outsourcer and the Contractor commit on a random Verifier. It is unlikely that this Verifier by accident turns out to be one that planned to collude with the Outsourcer. If the Outsourcer ignores the Randomization protocol and picks a Verifier itself, it is missing the commitment signature of the Contractor which is revealed in Contestation in case of dispute. Additionally, our verification scheme supports contracts that design incentives leading to being honest as a dominant strategy. If the Verifier is detected cheating by the Contractor through Contestation, it has to pay a fine. If the incentives are set correctly, taking into account the probability of the Verifier being detected, the reward it gets when not being detected and the fine it gets when it gets detected, being honest maximizes the expected payoff.

Thus, it is not rational for the Verifier to collude, and Randomization leads to a random Verifier that is unlikely to collude anyway. If a colluding Verifier gets assigned despite the measurements, it is detected by 100% confidence by Contestation.

3.6.7. 7. Contractor and Verifier collude to save re-sources

The Contractor and the Verifier may collude to save computational resources by generating a false response and sending it to the Outsourcer. The Outsourcer checks if both results match, and assumes the responses to be correct. This way it may receive false responses for a long time without ever realizing it. Our verification scheme prevents the communication of Verifier and Contractor through Randomization. As the Contractor and the Outsourcer commit to a random Verifier, planned collusion is highly unlikely to occur. Additionally, the Outsourcer does not reveal the result of Randomization to the Contractor, and also does not inform the Verifier about the identity of the Contractor. The only way for the Contractor and the Verifier to know about each other's identity would be to contact every other available Verifier and Contractor in the network. Therefore, ad-hoc collusion is highly unlikely as well. Additionally, a contract with the right incentives that lead to being honest being a dominant strategy includes a bounty for the Contractor if it detects a cheating Verifier, and a bounty for the Verifier to detect a dishonest Contractor. The detected participant has to pay a fine. If the fine and the bounty are set as described in the section on Game-theoretic incentives, the expected payoff of reporting another party when it cheats exceeds the payoff when colluding. Thus, colluding is not rational for both participants from an individual point of view. This measurement makes ad-hoc collusion between Contractor and Verifier highly unlikely as well. While it is difficult to put the probability of our verification scheme of preventing this type of collusion, we conclude that with high confidence collusion between the Contractor and the Verifier will be prevented. Beyond our verification scheme, an Outsourcer may decide to utilize more than one Verifier, or if it has sufficient computational resources to also re-execute inputs by itself. As the most likely scenario is for the described strategy that Contractor and Verifier have a cheating rate of 100%, an Outsourcer that is capable of re-calculating the assigned function over one input can already detect the collusion with absolute certainty.

3.6.8. Number 8,9,10: Timeouts, Low Response Rate, High Response time

While in our verification scheme dishonest behavior of the Contractor, or the Verifier is punished with refusal of payment, and if specified in the contract with a fine, QoS violations such as timeouts, low response rate, or high response time come without monetary consequences. We use blacklists, contract abortion, and reviews to punish bad QoS or to promote good one. Whenever a participant receives a message from another participant that exceeds the thresholds it specified in its internal parameters, it may abort the current contract due to QoS violations. It may also blacklist the other participant and submit a negative rating on this participant at the payment settlement entity. Thus, a participant that can not provide good quality of service misses out on ongoing payments of the current contract, and also may receive fewer assignments or less reward due to a bad review and blacklisting. While nodes with a neutral review balance might avoid negative reviews by generating a new identity, reviews are still useful to promote good QoS. Additionally, generating a new identity might not be trivial depending on identity checks and requirements when signing up at the payment settlement entity. Over time, the review system will ensure that nodes are accurately sorted by their QoS ecosystem-wide even if once in a while a dishonest review gets submitted. Nodes with a high QoS may profit from increased rewards due to their proven high performance and honesty. Thus, we conclude that our scheme can prevent QoS violations before a contract is started with high confidence over time by utilizing blacklists, contract abortion, and reviews. During a contract these behaviors can be detected with 100% confidence and lead to contract abortion if they violate a threshold set by one of the participants.

3.6.9. Number 11: Message Tampering

Our verification scheme is not only considering internal threats, but also external threats. An external attacker may attempt to tamper with messages sent between participants to harm a participant. In our verification scheme, each participant generates a digital signature over each method they are sending. Thus, any participant receiving a message tampered with by an attacker immediately detects that the message is ill-formatted because the attacker is not able to generate a valid signature associated with one of the participants. It should be noted, however, that if the participants use Merkle trees, and the Contractor only signs a Merkle root and challenges every specified interval size, message tampering is detected with a delay, until the next Merkle root is sent. Therefore, we advise participants that aim to protect themselves from this attack to not utilize Merkle trees. In any case, message tampering can be detected with 100% confidence and leads to abortion of a contract due to QoS violation, as the tampered message is detected as ill-formatted.

The figure 3.14 shows a summary of the literature review and our designed verification scheme.

Table 3.1.: Protocol violations, techniques to prevent them utilized by our verification scheme, and confidence of prevention

Type of Violation	Referred Number	Description	Techniques	Confidence
Dishonest Behavior by Individual	1	Contractor sends back false responses to save resources	Sampling-based re-execution, utilization of a third party Verifier if required	>99% (small sampling size), 100% (sampling rate of 1)
	2	Verifier sends back false response to save resources	Contestation	100% *
	3	Outsourcer sends different input to Contractor and Verifier to refuse payment	Digital Signatures (signature chain), Contestation	100% *
	4	Contractor or Verifier tries to avoid global penalties	Digital Signatures	100%
	5	Participant refuses to pay even if obliged to by the protocol	TTP or Blockchain that is authorized to conduct payment on behalf on another entity	100%
Dishonest Behavior via Collusion	6	Outsourcer and Verifier collude to refuse payment and save resources	Randomization, Game-theorectic incentives, Contestation	100% *
	7	Contractor and Verifier collude to save resources	Randomization, Game-theoretic incentives	High confidence
QoS Violation	8	Timeout	Blacklisting, Review system, Contract abortion	100%
	9	Low Response Rate	Same as Nr 8	100%
	10	High Response Time	Same as Nr 8	100%
External Threat	11	Message Tampering	Digital Signatures	100%

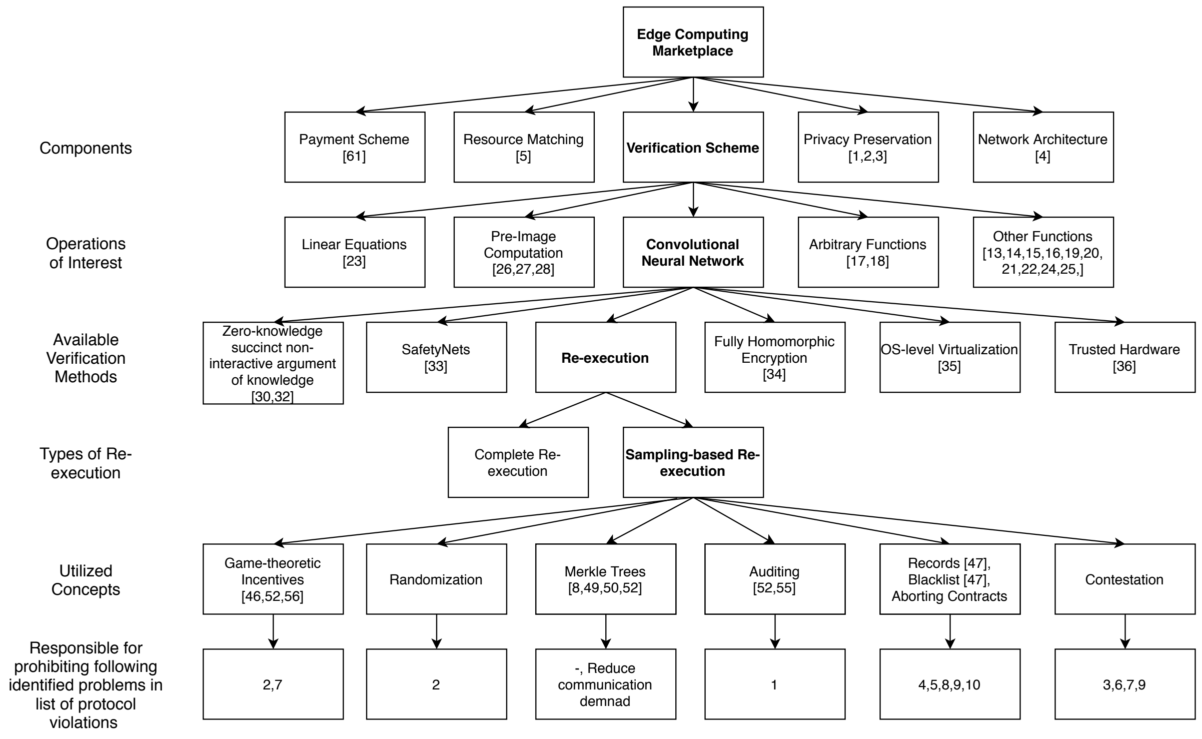

Edge Computing Marketplace
Components
Payment Scheme [61]
Resource Matching [5]
Verification Scheme
Privacy Preservation [1,2,3]
Network Architecture [4]
Operations of Interest
Linear Equations [23]
Pre-Image Computation [26,27,28]
Convolutional Neural Network
Arbitrary Functions [17,18]
Other Functions [13,14,15,16,19,20, 21,22,24,25,]
Available Verification Methods
Zero-knowledge succinct non-interactive argument of knowledge [30,32]
SafetyNets [33]
Re-execution
Fully Homomorphic Encryption [34]
OS-level Virtualization [35]
Trusted Hardware [36]
Types of Re-execution
Complete Re-execution
Sampling-based Re-execution
Utilized Concepts
Game-theoretic Incentives [46,52,56]
Randomization
Merkle Trees [8,49,50,52]
Auditing [52,55]
Records [47], Blacklist [47], Aborting Contracts
Contestation
Responsible for prohibiting following identified problems in list of protocol violations
2,7
2
-, Reduce communication demnad
1
4,5,8,9,10
3,6,7,9

Figure 3.14.: Overview of literature review and designed scheme (outdated reference numbers)

4. Implementation

While our verification scheme works with arbitrary functions, we focused on the verification of Convolutional Neural Network (CNN) inference used for object detection. Real-time object detection is a complex task that exceeds the computational capabilities of many IoT devices. CNNs provide state-of-the-art accuracy for object detection. Due to being one of the most complex functions that IoT devices might need to outsource and their high relevance at the edge, we believe that CNN inference is the perfect function type to stress test if our verification scheme is practical in real-time. Our implementation starts after an Outsourcer and a Contractor agreed on a contract and a Verifier. It proceeds with our preparation phase and our execution phase until one participant decides to abort a contract. We developed scripts with an optimized message pattern that eliminates network wait and a parallelized version that performs the tasks of our verification in parallel to the object detection to increase frames per second (fps).

4.1. Software Architecture

We implemented our verification scheme 100% in Python. As a digital signature algorithm, we chose ED25519. One set of scripts have to be deployed at the machine acting as the Contractor and the machine acting as the Verifier. The other set of scripts have to be deployed at the machine acting as the Outsourcer. Thus, each remote machine executes the full functionality of one participant. The Outsourcer has an "Outsource Contract" and a "Verify Contract" that has to contain an identical set of agreements as the one set by each of the respective participants. The contracts contain settings like a and, the public key of each participant involved, a reward per image sent, the CNN model to use, whether Merkle trees should be used, and more. As a signature over the hash of the contract is sent with each message, each input is associated can be associated with one unique contract that contains all information necessary on which function to use and how to format each response. Also, Each participant has to set various local parameters before starting the contract. At first, the IP addresses of each participant that communicates with this machine have to be set. Each participant sends messages at a specific port and receives messages on another port. The Outsourcer can specify a sampling interval, that defines in which intervals a random sample is sent to the Verifier. An interval size of 1 would result in a sampling rate of 100%, while an interval size of 200 would result in a sampling rate of 0.5%. Furthermore, each participant sets thresholds for QoS violation, such as maximum response delay, and minimum response/acknowledge rate. If one of these thresholds is exceeded during contract execution by another participant, the participant that sets the threshold automatically aborts the contract due to QoS violation.

Finally, a participant has to specify or import a private key that is associated with the public key specified in the relevant contract that is used to sign all messages. After contract details and parameters are set, each participant can proceed to execute the scripts, thus starting the contract. We developed four different versions of our scripts. The object detection models assigned to the Contractor and the Verifier can be run (1) either with a regular CPU or GPU or (2) with an Edge Accelerator. We also developed a multi-threaded version of each variation. Figure 4.1 illustrates the software architecture for each frame when the program is run on a CPU/GPU/Edge Accelerator without multi-threading of key tasks, meaning that verification, preprocessing, inference, and postprocessing is conducted sequentially.

The Outsourcer utilizes three threads. The second and third thread are only needed to listen to the TCP sockets of Verifier and Contractor and load them into memory, while the main thread is performing other tasks. As downloading responses and loading them into memory is an I/O bound task, these threads do not impact the performance of the main thread measurably. This way, network wait is eliminated, considering network delay is lower than the time it takes the Outsourcer to perform one iteration in the main thread, In one iteration of the main thread, the Outsourcer captures a new frame with its camera. Before sending the frame over the network gets compressed to a JPEG image. Using its private key, the Outsourcer signs the compressed frame, input index, contract hash (Contractor specific), and number of acknowledged frames (Contractor specific). It sends the compressed image with the signature over a TCP socket to the Contractor. If Merkle trees are set to be used in the contract, it additionally may send a proof of membership challenge of a previously checked sample, the current Merkle tree interval index, and specifies whether it's time to answer the challenge along with a signature over. If the current iteration index was assigned by the ongoing generation of random sample indices, the compressed frame also gets sent to the Verifier, attached with a signature over input index, contract hash (Verifier specific), and number of acknowledged frames (Verifier specific). Next, it checks via an object it shares with Thread 2 and Thread 3 respectively if new responses arrived from the Contractor or the Verifier. If this is the case, it verifies the signatures using the public key specified in the Outsource-Contract, or the Verify-Contract. Whenever a response of the Contractor or the Verifier arrive, and a response resulting from the same input index arrived before by the other party, the two responses are compared whether they are equal. If they are equal, the main thread proceeds with the next iteration. If Merkle trees are used, the Outsourcer does only need to verify responses by the Contractor if a Merkle root hash or challenge is sent and signed along with a response. A Merkle root hash has to be sent by the Contractor every n frames specified in the Outsource contract. Once the Outsourcer receives a Merkle root hash in time, it sends a proof-of-membership challenge of a previously checked sample to the Contractor. Only if the proof of membership challenge is successful, the Outsourcer continues the contracts. In each iteration, thresholds set in the internal parameters associated with QoS violations such as response delay, response rate, number of consecutive inputs that remained unanswered, and more get checked to decide whether contracts are canceled by the Outsourcer due to a QoS violation. If two responses belonging to the same input are found to be unequal, the Outsourcer aborts the Contracts due to dishonest behavior. Assuming the

Contractor or the Verifier has sent a new response during one iteration of Thread 1, then Thread 2, or Thread 3 respectively already downloaded it and loaded it into memory, thus eliminating network wait. During execution, whenever necessary the Outsourcer checks if the following QoS violations occurred:

1. Contractor did not connect in time.
2. Verifier did not connect in time.
3. Contractor response is ill formated.
4. Verifier response is ill formated.
5. Contractor signature does not match response.
6. Verifier signature does not match response.
7. Contractor response delay rate is too high.
8. Verifier has failed to process enough samples in time.

Checks only applicable if Merkle trees are used:

1. No root hash received for the current interval in time.
2. Merkle tree leaf node does not match earlier sent response.
3. Contractor signature of challenge-response is incorrect.
4. Leaf is not contained in Merkle Tree.
5. Contractor signature of root hash received at challenge response does not match previous signed root hash.
6. Merkle Tree proof of membership challenge-response was not received in time.

Also, it checks the following dishonest behaviors whenever it receives a response belonging to the same input as an earlier sent response by the other participant. Note that

1. Merkle Tree of Contractor is built on responses unequal to responses of the Verifier (if Merkle trees are used).
2. Contractor response and Verifier sample are not equal. (if Merkle trees are not used).

Note that if Merkle trees are used, and the Contractor finds two unequal responses belonging to the same input, that it sends a proof of membership challenge to the Contractor first over that input to receive a signature as commitment once proof of membership challenge and the next Merkle root hash are signed by the Contractor. Then it can cancel the Outsource-Contract and successfully report the Contractor with both signatures.

The described checks are used to detect all the dishonest behavior and QoS violations from an Outsourcer perspective during the execution phase we identified in our list of protocol violations in table 3.1.

In the version without multi-threading of key tasks, the Contractor utilizes two threads. Like for the Outsourcer, Thread 2 is only responsible for implementing the non-blocking message pattern, meaning that it receives new frames in parallel to object detection to eliminate network delay. Thus, the computational complexity of Thread 2 is neglectable as receiving frames from the TCP socket is an I/O bound task.

Thread 1 is responsible for all tasks that deal with computation. In each iteration, Thread 1 fetches the newest frame via a shared object between Thread 1 and Thread 2. First, the Contractor verifies the signature attached to the frames and checks if all attached information such as the contract hash is accurate. It uses the public key of the Outsourcer specified in the Outsource-Contract to verify the signature generated over the compressed frame, index, number of acknowledged responses, and contract hash. If Merkle trees are set to be used in the Outsource-Contract, it also expects a random challenge, a variable indicating if it's time to answer a new challenge, and the current Merkle Tree interval. If the Contractor also agrees with the number of acknowledged responses, it continues with prepossessing the frame. This includes tasks like resizing, recoloring, and reformatting the frame to create a valid input that the CNN can process. Next, it uses the pre-trained model that is specified in the Outsource-Contract to perform CNN inference on that frame. Inference is the task that usually takes the longest amount of time compared to the tasks from all participants. After inference, postprocessing is used to create a formatted output that contains the frame index, each object found, the confidence, and the bounding box coordinates where the object is located in the frame. Using its private key, the Contractor signs the formatted response, the contract hash, and the signature sent by the Outsourcer belonging to that frame. As the latest frame sent by the Outsourcer contains the signed, total number of acknowledged responses associated with the Contract, the Contractor only needs to store the latest frame in memory, which leads to a memory overhead of $O(1)$. If Merkle trees are set to be used in the Outsource-Contract, the Contractor only signs a frame if it either intends to send a Merkle root hash or a proof-of-membership challenge response along with it. These have to be sent once per Merkle tree interval, also specified in the Outsource-Contract. Finally, it sends the response and its attached signature via a TCP socket to the Outsourcer. Assuming, the Outsourcer has already sent a new frame, while Thread 1 processed the current one, Thread 2 already downloaded it and loaded it into memory, thus eliminating network wait.

During execution, whenever necessary the Contractor checks if the following QoS violations occurred:

1. Outsourcer signature does not match the input.
2. Outsourcer did not acknowledge enough outputs.
3. Outsourcer timed out.

The Verifier logic is identical to the Contractor logic with the difference of using a Verify-Contract instead of an Outsource Contract and having different internal parameters. Thus, our implementation uses identical execution scripts for both participants, that fetch participant-specific information from internal parameters and the local contract.

As our verification scheme supports scenarios with frame loss, each participant has internal parameters tolerating a certain frame loss rate before aborting a contract. Additionally, in case Merkle trees are set to be used in the Outsource-Contract critical information such as Merkle root hash, Merkle tree challenge, and Merkle tree challenge-response get sent multiple times until the receiving participant sends a message that indicates that the initial critical message was received. For instance, when a new Merkle root hash has to be sent by the Contractor for a new Merkle tree interval, it sends the Merkle root hash with every new frame until it receives a frame from the Outsourcer with an increased interval counter, indicating that the Outsourcer has received and confirmed the new Merkle root. Thus, it is ensured that all critical information gets delivered eventually with minimal delay in all network states. In our implementation, if a participant sends an abort message, the other participants cancel the contract according to custom and does not report a QoS violation because of a timeout.

In our current implementation, supported models for object detection on a regular GPU and CPU are Yolov4 and Yolov3 using Tensorflow, TFLite, and TensorRT (only deterministic) as the framework. By default, we use weights, pre-trained with the Microsoft Coco dataset. Tiny weights and custom weights can be used as well. Any input size can be used. The supported model for object detection on a Coral USB Accelerator is Mobilenet SSD V2 with an input size of 300x300 pixels. We chose Yolov4, Yolov3, and Mobilenet SSD V2 as these are all state-of-the-art object detection models with a reasonable trade-off of performance and accuracy. We chose different versions of TensorFlow as our deep learning frameworks due to their high popularity and state-of-the-art performance. We chose ED25519 as a digital signature algorithm as it provides high security, and resulted in the best performance in our benchmark. ED25519 uses SHA-512 as its hashing algorithm which results in 512-bit large signatures. For the contract hash, we use SHA3-256 due to its high security.

4.2. Improving Run-time of Implementation

The run-time of a system attached to our verification scheme can mainly depend on the following different factors.

1. The CNN model and pre-trained weights used influence the time for each frame that the Contractor and the Verifier spend on prepossessing, inference, and postprocessing. Additionally, the frame input size of the model influences the time that the Outsourcer needs to capture and compress a new frame.

2. The used hardware of each participant influences all tasks. The GPU is mainly influencing CNN inference time.

3. The used digital signature and hash algorithms influence all signing and verifying tasks of all participants.

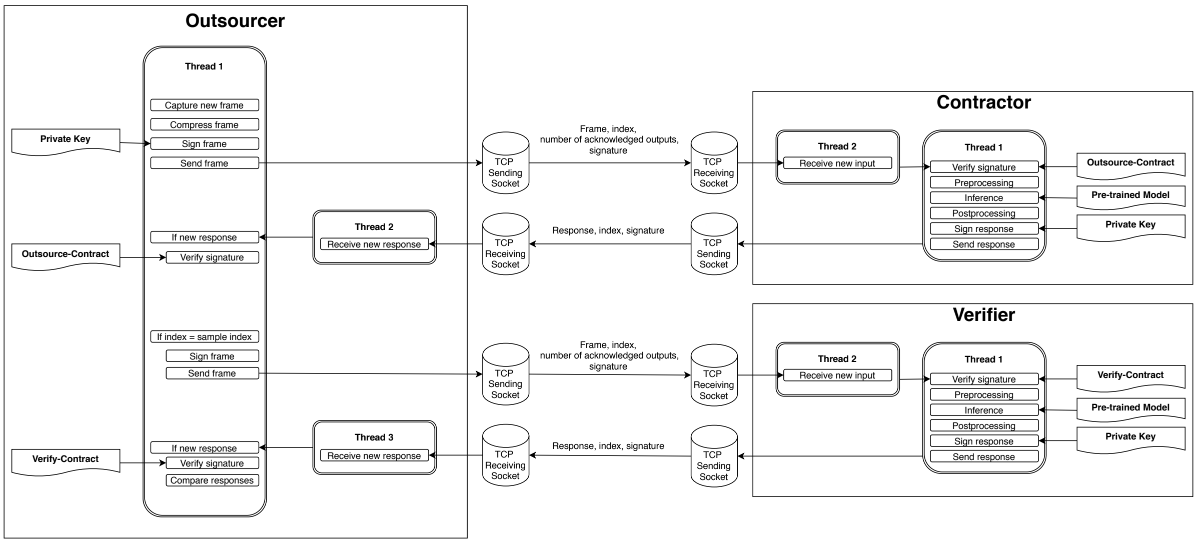


Figure 4.1.: Software Architecture without multithreading of key tasks using a regular GPU

4. The use of a non-blocking, or a blocking message pattern influences the time that each participant spends idle waiting for new messages over the network.
5. The use of multi-threading, or multi-processing of main tasks influences the overall execution time of all participants.

While the model, weights, and hardware used to have a significant impact on system performance, both factors are dependant on the use-case and devices used with our verification scheme. Thus, we used different hardware and high-performing models for testing our implementation, but focus on factors 3-5 in this chapter to explain how we achieve high fps in our implementation.

4.2.1. Benchmark of Digital Signature Algorithms in Python

We considered three different state-of-the-art digital signature algorithms: RSA, ECDSA, ED25519. ED25119 is currently considered the most secure digital signature algorithm out of those three. ECDSA can be implemented with different curves and hashing algorithms. In general Nist P-192 is the most efficient curve and blake2s are the most efficient hashing algorithms. We also tested one-time digital signature schemes such as Lamport and Winternitz digital signature schemes. In theory, signing and verifying with one-time signature schemes should be computationally less expensive than with multi-use digital signature schemes. As our implementation is built with Python as a programming language, we benchmarked multiple digital signature libraries available in Python that implement those different signatures. It should be noted that two libraries implementing a digital signature algorithm with identical inputs and parameters can have significantly different performance. For example, we found that the Starbank-ECDSA library is almost 10 times slower with our machines than the ECSDA library in Python. We benchmarked the following libraries on a Raspberry Pi Model 4B and an Intel Core i7-3770K @ 3.8Ghz.

1. ECDSA using SHA2
2. RSA
3. ECDSA using Blake2b
4. ECDSA using Blake2s
5. ECDSA-Blake2b
6. Fast-ECDSA using SHA2
7. Fast-ECDSA using SHa3
8. Starbank ECDSA
9. NaCl (ED25519)
10. Lamport
11. Winternitz

We tested the signature libraries with different images in different resolutions. While RSA only needed 0.34ms to verify signatures on the Raspberry Pi, it needed up to 9ms to sign a large image. The implementations of ECDSA differed significantly with the ECSA library being the best one needing around 1ms to sign and verify. The best performances were achieved by the NaCl library implementing ED25519, and the Lamport one-time digital signature algorithm. Both could achieve signing and in less than 0.3ms even on our Raspberry Pi. While the Lamport digital signature scheme only needed 0.1ms for signing, it needed 0.9ms for verifying on the Raspberry Pi. Thus, ED25519, implemented by the NaCl library with 0.23ms signing and 0.6ms verifying performed the best in our benchmark. It should be noted, that the Outsourcer may use verifying less frequently than signing in our scheme if the Contractor loses frame or cannot keep up with the frame rate of the Outsourcer. Thus, fast signing is critical for our verification scheme. We conclude that ED25519 is not only the most secure digital signature algorithm of the considered ones' but also the one with the best performance in Python according to our benchmarks. One-time digital signature schemes resulted in lower performance in implementation, showing that current libraries cannot exploit their theoretical performance advantage. As one-time signatures also come with increased complexity of maintaining a signature chain and generating new signatures frequently, we chose to not consider them for our final implementation.

4.2.2. Non-blocking Message Pattern

When sending a real-time stream to other devices the default message pattern is a blocking request-response pattern. This means that a client sending data waits idly for a response that either the data was received or already a processed result. The main advantage of a blocking message pattern is its easy logic and debugging. Even though networking applications are by nature asynchronous due to independent hardware and different processing speeds of clients and servers, a blocking message pattern synchronizes processes of multiple devices by letting them wait for new data over the network. If we used a blocking message pattern in our system, the Outsourcer would wait idle for the processed response of the Contractor after sending a frame. Likewise, the Contractor would wait idle after sending a response until a new response arrived over the network. This sequential operation has the advantage of simple implementation and the need for only one main thread. The major disadvantage, however, is the increased time spent on each frame by waiting for messages over the network. Figure 4.2 illustrates a blocking message pattern used between the Outsourcer and the Verifier.

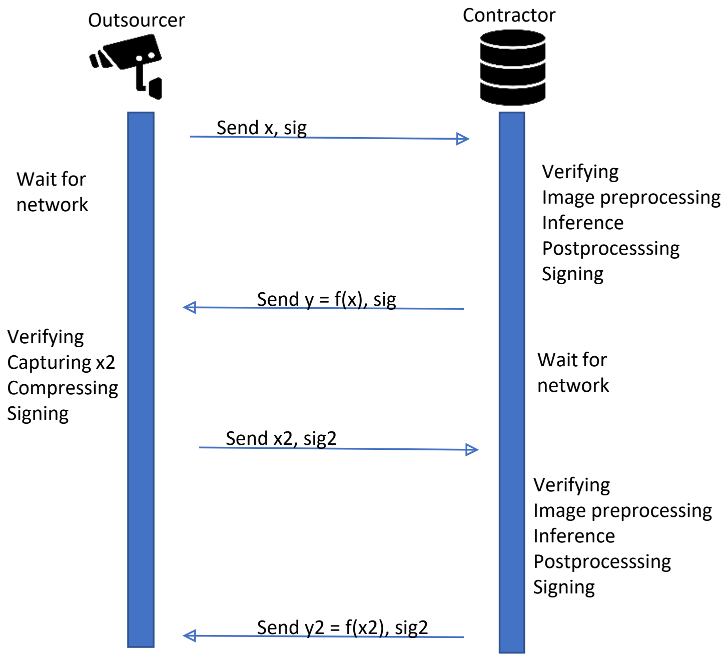


Figure 4.2.: Blocking Message pattern

As one can see, it would be more efficient for the Outsourcer to not wait for a network response but instead to already capture, compress, and sending a new frame while waiting for a response over the network. If it utilizes a separate thread that constantly fetches the most recent response from the Contractor and loads it into memory, it can eliminate network wait. Likewise, the Contractor can use a separate thread to load new frames into memory while performing object detection. In the next iteration, it can consume the new frame without the need of idle waiting in the main thread. As the added thread only wait for I/O they do not influence the performance of the main thread measurably even if run on the same processing core. This idea can be implemented using a non-blocking message pattern. Instead of waiting for responses, all participants use a Sending TCP socket accessed by one thread to constantly send new messages whenever a new one is ready, and a Receiving TCP socket accessed by another thread to constantly receive new messages whenever a new one gets sent over the network. Assuming the network delay is lower than the time either participant needs for its local message processing, our system can achieve identical performance of performing the task locally without any network delay. When processing time is the bottleneck in each iteration, parallel network operations do not increase processing time. Figure 4.3 illustrates how a non-blocking message pattern eliminates any idle waiting for the network. As one can see, the Outsourcer only verifies a response in its main loops, if a new response arrived since the last iteration. Further optimization could include moving the verification of a new frame to a separate thread or process. This can increase fps if the Contractor has sufficient hardware capabilities to process higher fps. Another advantage of a non-blocking message pattern is that it is easily compatible with frame loss scenarios. As each message is not strictly associated with a response, messages or responses that get lost do not cause a problem by default. However, this also means that a non-blocking message pattern leads to a more difficult development process as the Outsourcer has to check constantly if the response rate and delay of the Contractor is still matching QoS expectations. Also, Verifier and Contractor responses belonging to the same frame can arrive at a time where the other participant has already sent more recent frames. A program that deals properly with these challenges can significantly increase the performance and flexibility of the system. Thus, we implemented our system with a non-blocking message pattern. While we use TCP sockets, one can seamlessly switch to UDP sockets if preferred.

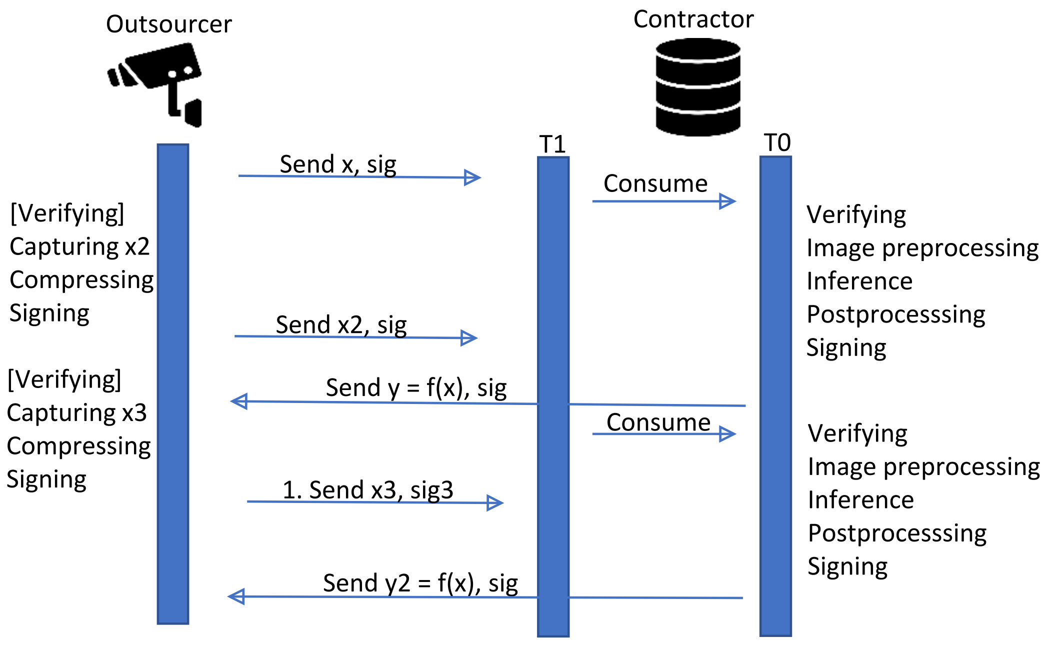


Figure 4.3.: Non-Blocking Message pattern

4.2.3. Parallel Execution

Using a non-blocking message pattern, our system already utilizes parallel I/O bound threads to eliminate network wait. Additionally, multi-threading or multi-processing has the potential to also parallelize the execution of computationally more expensive tasks. For instance, if the Contractor receives a new frame from the Outsourcer during CNN inference which is a GPU bound task, it can already verify its signature and do prepossessing which are CPU and I/O bound tasks. Once it is finished with inference, its GPU does not need to spend any time idle, but can immediately process the next frame. Not only has parallel execution of key tasks the potential to decrease the processing time of object detection (preprocessing, inference, postprocessing) significantly but it can also eliminate any time waiting for tasks of our verification scheme. If signing and verifying are done in parallel to inference, our verification scheme does not increase the overall processing time of the Contractor at all as inference is the performance bottleneck in most systems. Thus, it is most efficient to aggregate all other CPU bound and I/O bound tasks to threads or processes that take slightly less time than inference and run in parallel. This way, the overhead of initializing new threads and processes is minimized. Together with the non-blocking message pattern, our system exclusively depends on inference time for total performance. In a test scenario running the implementation exclusively local object detection without verification of objects that are already loaded into memory leads to almost identical performance of our system that needs to send two messages over the network per frame and has to sign and verify each message. An important decision to make is whether to use multi-threading or multi-processing of key tasks. The key difference between multiprocessing and multithreading is that multithreading runs tasks in parallel on the same core, switching between tasks in high frequency. Thus, it is only pseudo-parallel but has the advantage of sharing the memory between threads and low computational overhead. Multithreading is usually recommended for I/O bound tasks. Multiprocessing runs tasks in parallel on different CPU cores. Thus it is truly parallel but has the disadvantage of not sharing memory between cores in Python. This means that communication between processes is slow and each process has to copy all objects into its local memory during initialization. Thus, creating and maintaining additional processes comes with more computational overhead compared to threads. Multi-processing is usually recommended for CPU bound tasks. Our tests did not support this theoretical hypothesis. In fact, our multi-processing version performed up to 50% worse than our single processing version while our multi-threading version performed up to 70% better than our single processing version. We suspect the following reasons for this measurement.

1. The TensorFlow library that handles CNN inference already utilizes the GPU and multiple CPU cores and is disturbed if other cores are utilized for computationally expensive tasks.
2. Preprocessing, Postprocessing, signing, and verifying are more I/O bound and less CPU bound than anticipated.
3. Python is not optimized for multiprocessing tasks with low latency.

4. Communication between cores becomes the bottleneck of the system.

We believe that reason 1 and 2 are the most plausible explanation for our measurements. Thus, even though the parallel execution of key tasks performs better with multi-processing in theory, we use multi-threading in our implementation. This also has the major advantage that only one core is fully utilized while the other cores are available for other applications.

In our regular implementation, the main thread is responsible for verifying the signature of a new frame, decompressing, preprocessing, inference, postprocessing, signing the response, and sending the response. In our multi-threaded implementation that uses a regular GPU or CPU for inference, we split up the main thread from our base version into three different ones. Figure 4.4 illustrates the software architecture. In Thread 3, a signature attached to a new frame gets verified, the frame gets decompressed, preprocessed, and stored in a thread-safe object. Thread 1 fetches the latest preprocessed frame and performs inference, post-processes the frame, and stores it in another thread-safe object. The reason for using postprocessing is that this task relies on the GPU for efficient array operations and uses TensorFlow. Thus, it cannot be moved to a different thread. Finally, Threat 4 signs the response from postprocessing and sends it to the Outsourcer. Thread 1 took significantly longer than Thread 3 and Thread 4 in our test. Thus, the run time of this implementation is only dependant on the speed of inference and preprocessing. It does not depend on the speed of the verification scheme or network wait.

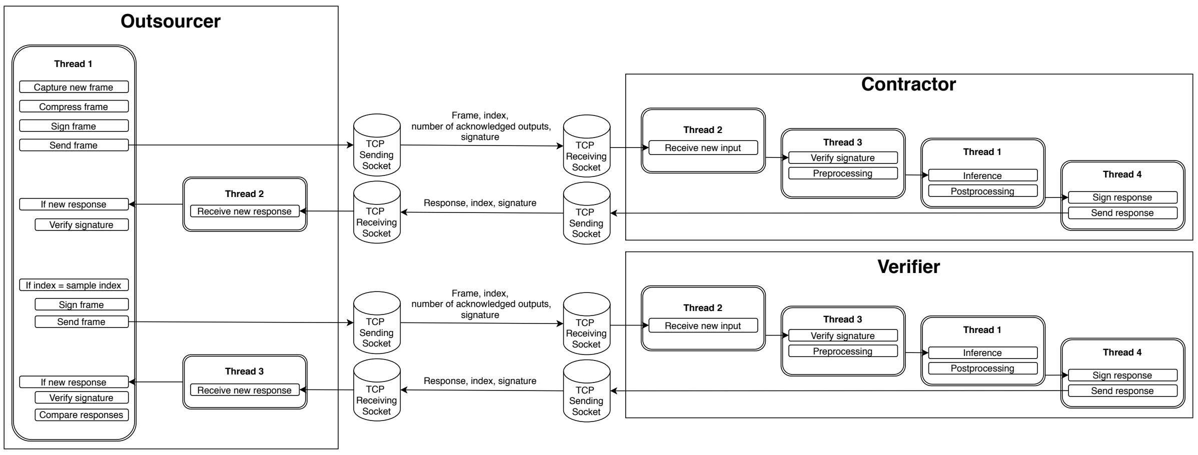


Figure 4.4.: Software Architecture with multithreading of key tasks using a regular GPU

Our implementation for Contractors and Verifiers using a Coal USB Accelerator is slightly different from the version running on a regular GPU/CPU. In this case, the TensorFlow Lite library is used in prepossessing. Postprocessing on the other hand does neither need the library nor the GPU. Thus, we moved Preprocessing from Thread 3 to Thread 1 and Postprocessing from Thread 1 to Thread 4. Figure 4.5 illustrates the software architecture when using multi-threading of key tasks with a Coral USB Accelerator.

In summary, we implemented different versions of the Contractor and Verifier scripts. As implementation on a Coral USB Accelerator and a regular CPU/GPU fundamentally

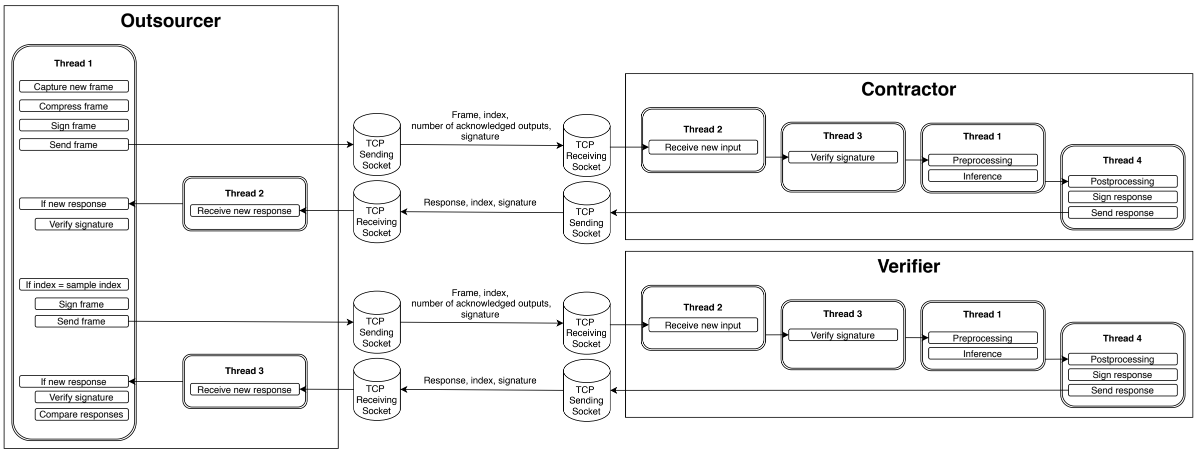


Figure 4.5.: Software Architecture with multithreading of key tasks using a Coral USB Accelerator

differs for key tasks, library, and model used we created different scripts for using a Coral USB Accelerator rather than regular CPU/GPU. For each of the two different scripts, we implemented a single-threaded and a multi-threaded version. All versions use a non-blocking message pattern by implementing an additional thread that loads new messages into memory in parallel to key tasks. This feature eliminates network wait. As inference is the bottleneck of all key tasks, our multi-threaded versions run the tasks of the verification schemes and other key tasks in parallel to inference. This feature maximizes fps and eliminates any latency overhead that our verification scheme adds to the execution time. As we use multi-threading instead of multi-processing, only one core of the machine is fully utilized because of this decision. The combination of a non-blocking message pattern and multi-threading of key tasks leads to almost identical fps as performing object detection locally without a verification scheme.

4.3. Test Setup

We tested our implementation with two different setups. One using a regular GPU/CPU for inference and one using a Coral USB Accelerator. Edge accelerators such as the Coral USB Accelerator or other application-specific programming units (APUs) are gaining increased popularity and provide an unmatched performance per watts and performance per price ratio. Thus, we believe that to properly evaluate our implementation for an edge computing scenario, we need to include specialized edge computing hardware in our tests.

Our first test setup uses the following hardware:

- Contractor Hardware: Intel Core i7-3770K @3.8GHz + Nvidia Geforece GTX 970
- Verifier Hardware: Intel Core i5-4300U @1.9GHz
- Outsourcer: Raspberry Pi Model 4b + RPI Camera V2
- Object Detection Model: YOLOv3, YOLOv4, tiny and non-tiny versions

The first test setup represents typical consumer hardware. The contractor hardware consists of a mid-range Desktop CPU and GPU. The Verifier Hardware uses a low-End Notebook CPU with an integrated GPU. The Raspberry Pi Model 4b represents an IoT device. Yolov3 and Yolov4 are state-of-the-art object detection models that feature a reasonable trade-off between accuracy and performance. We mostly used tiny weights that were pre-trained on the Microsoft Coco dataset in our test. As recent, more powerful GPUs can achieve more than 1500fps with tiny weights, we recommend using more complex weights for higher accuracy on those devices. The test results of this setup show if typical consumer notebooks, Desktop PCs, and a Raspberry Pi deliver have enough performance to run state-of-the-art object detection models with our verification scheme over the local network. Figure 4.6 illustrates our first test setup.

Our second test setup uses the following hardware:

- Contractor Hardware: Intel Core i7-3770K @3.8GHz + Coral USB Accelerator
- Verifier Hardware: Intel Core i5-4300U @1.9GHz + Coral USB Accelerator
- Outsourcer: Raspberry Pi Model 4b + RPI Camera V2
- Object Detection Model: YOLOv3, YOLOv4, tiny and non-tiny versions

The first test setup represents a cheap setup of mid-range CPU and an entry-level edge accelerator to perform inference. The hardware used is identical to the first test, except for using a Coral USB Accelerator instead of a regular CPU/GPU for inference. Mobilenet SSD V2 is a state-of-the-art object detection models that also features a reasonable trade-off between accuracy and performance. Figure 4.7 illustrates our first test setup.

The Coral USB Accelerator needs a special compiled Tensorflow Lite version of the object detection weights that are only 6MB in size for Mobilenet SSD V2. In comparison, YOLOv4 weights trained on the same dataset are 23MB in size for tiny weights and 246MB for regular

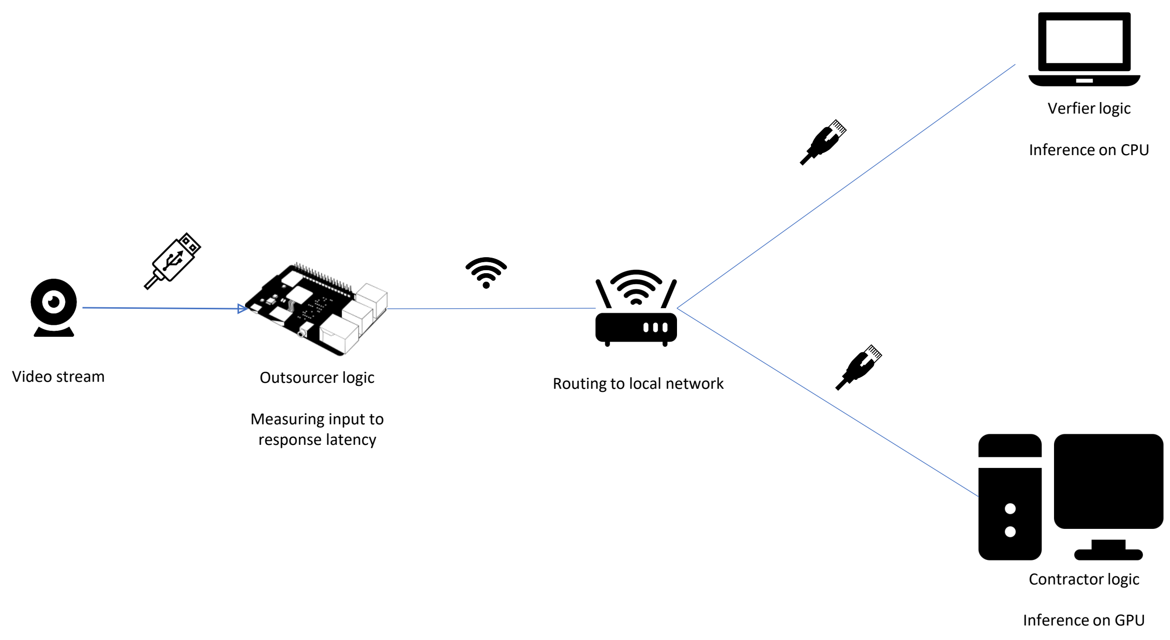


Figure 4.6.: Implementation and Testing with regular GPU and CPU

weights. All sizes, are reasonable to transfer from a Cloud server to remote hardware at the edge for one or multiple contracts. The test results of this setup show if a mid-range CPU and an entry-level edge accelerator can already deliver sufficient performance to run state-of-the-art object detection models with our verification scheme over the local network. We also swapped Contractor and Verifier hardware in further tests to evaluate if even a low-range CPU can be used along with the edge accelerator for real-time object detection.

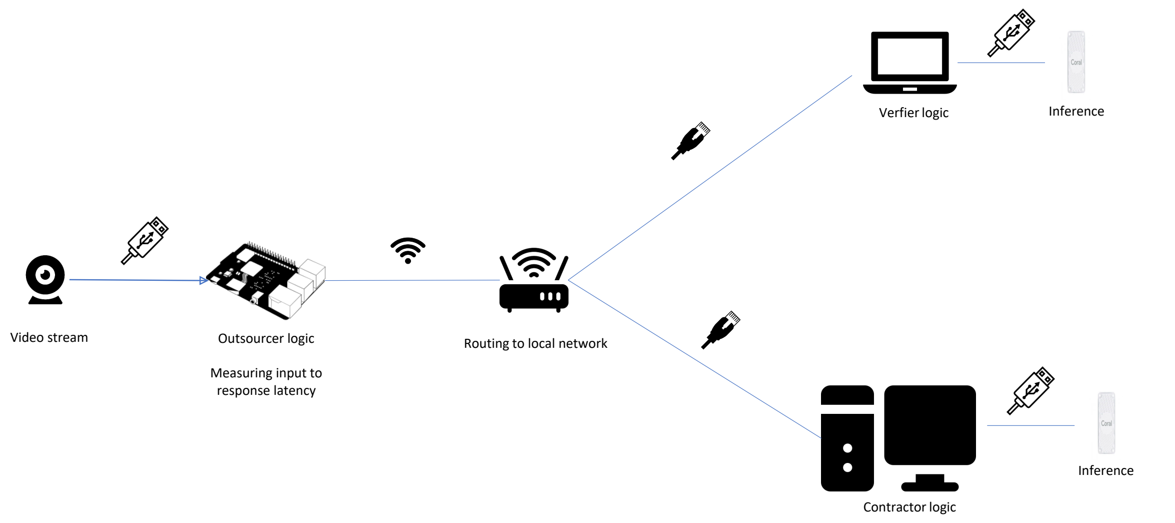


Figure 4.7.: Implementation and Testing with Edge Accelerator

5. Evaluation

5.1. Results

In this chapter we show our performance results and compare different versions and techniques of our implementation.

5.1.1. Performance

In this section, we show performance measurements of our different test setups. Before showing our final results, we first present the increase in performance that we achieved with the performance-enhancing techniques, introduced in chapter 3 and 4 such as non-blocking message pattern, more efficient digital signature algorithms, and multi-threading.

Execution time progress over time

Figure 5.1 illustrates the performance results of the Contractor in our test setup with different implementation approaches using a mid-range consumer GPU. Our first implementation utilized a blocking message pattern and used ECDSA as a signature algorithm. "Display" just stands for the decision to output object detection results on the screen which takes 2.4ms but is only useful for demo purposes and not required in an actual setup. With this setup, we achieved 32.9 fps. We measure fps with the average time passed between the Outsourcer receiving a response from the Contractor, and the Outsourcer receiving the next response from the Contractor. Thus it covers the whole cycle of a new frame loaded by the Contractor until the response is processed, sent over the network, and is loaded into the memory of the Outsourcer.

While achieving over 30fps is already practical for real-time object detection, we decided to further optimize key parts of the implementation with the potential of increasing performance.

Our initial solution suffered from network delay as it used a blocking message pattern. Each time the Contractor sent a response it had to wait more than 5ms to receive the next frame from the Outsourcer. By introducing a non-blocking message pattern as introduced in chapter 4 we increased the fps from 32.9 to 38.7.

As explained in chapter 3, using Merkle trees to only sign and verify a Merkle root hash and challenge in a specified interval can lead to a fraction of the required amount of signing and verification operations when signing and verifying each message instead. When using Merkle trees in our implementation, we increased the average performance to 40.3 fps from 38.7 fps.

Technique	Performance Improvement
Non-blocking message pattern	17.6%
Merkle trees	4.1%
Best performing digital signature library	3.2%
Multithreading of key tasks	47.3%
All techniques combined	106.9%

Table 5.1.: Relative performance improvement of utilizing our performance-enhancing techniques on a regular GPU

We switched from the ECDSA library to the NaCl library in Python that performed best in our benchmark, to use ED25519 instead of ECDSA which is not only considered to be more secure but also faster than ECDSA. By using ED25519, we increased performance from 38.7 fps to 41.6 without using Merkle trees, and from 40.3 fps to 41.9 fps with using Merkle trees.

For further benchmarks, we chose to not show the object detection results on the Contractor's display to achieve more realistic results. This decision increased performance to around 46 fps for both using Merkle trees and not using Merkle Trees.

We achieved by far the biggest performance improvement when using multithreading of key tasks as introduced in chapter 4. We increased fps from 46 to around 68 fps with our test setup by using multithreading of key tasks.

In summary, from our performance-enhancing techniques, we achieved major performance improvements with a non-blocking message pattern (-4.55ms, 17.6%) and multithreading(-6.94ms, 47.3%), a moderate performance improvement with the use of ED25519 which was the most efficient in our benchmark (-1.8ms, 3.2%), and a minor performance improvement when using Merkle trees instead of signing every message (-1.03ms, 4.1% sometimes less improvement, dependant on interval size). In total, we improve performance with our performance-enhancement techniques by about 15ms and were able to more than double the performance of our initial implementation from 32.9 fps to 68.06 fps (106.9%).

The table below shows the performance improvements by introducing each of the different techniques.

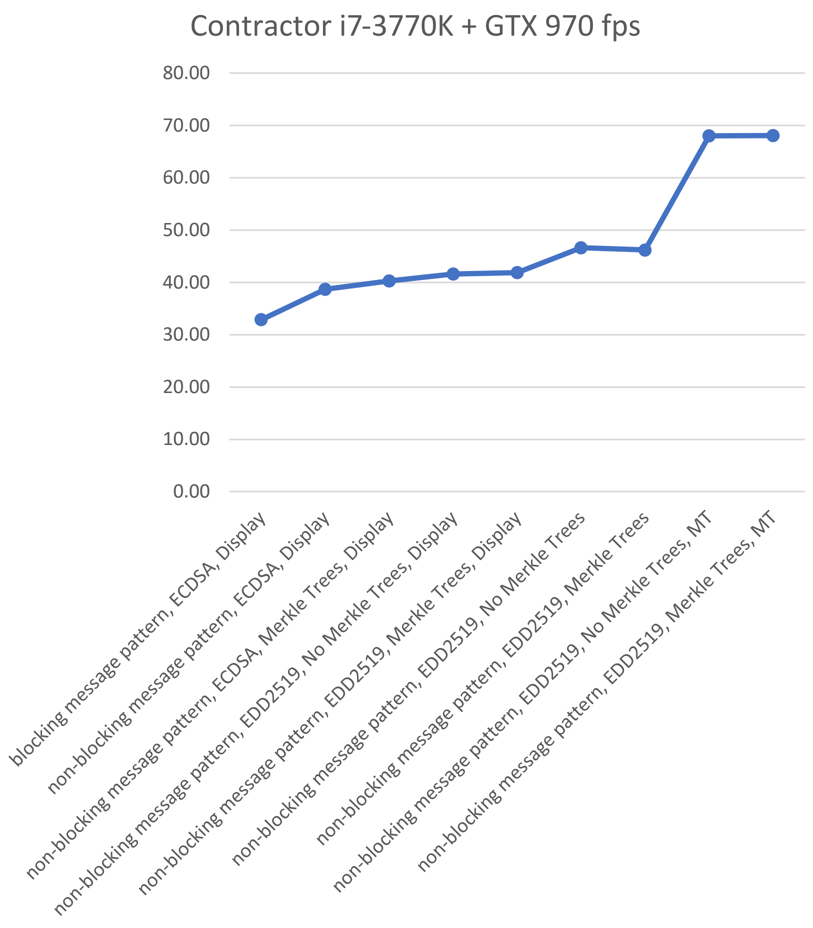


Figure 5.1.: Contractor performance with different setups on regular GPU

Figure 5.2 illustrates the performance results of the Contractor with our four final implementation scripts using a Coral USB Accelerator.

As one can see, our implementation using no multi-threading and no Merkle trees results in 57.22 fps. By using Merkle Trees, the performance improves slightly to 57.73fps. When using multithreading of key tasks and no Merke Trees, performance increases from 57.22 fps to 63.19. We achieved the best performance by using Merkle Trees, and multi-threading of key tasks with 63.59 fps.

In summary, we achieved a minor performance improvement when using Merkle trees instead of signing every message (-0.15ms or less, 0.9%). By using multithreading of key tasks, we can improve performance significantly (-1.65ms, 10.4%). The reason for multi-threading being less effective in our tests using a Coral USB accelerator is that inference accounts for roughly 90% of overall processing time instead of roughly 50% in our tests utilizing a regular GPU. Thus, parallelizing of key tasks can only increase performance by roughly 10% as the other tasks are less computationally expensive.

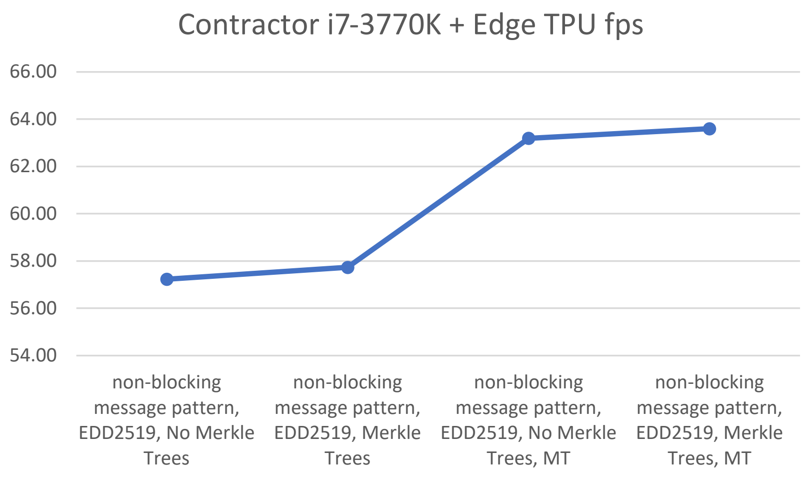


Figure 5.2.: Contractor performance with different setups on Edge Accelerator

5.1.2. Final Results

Table 5.2 shows the key results of our test implementation. More results can be found in the Appendix.

Outsourcer performance

As one can see, the Outsourcer achieved 236 fps with a frame size of 300x300 pixels and 146.9 fps with a frame size of 416x416 pixels. The frame sizes are representative also for other state-of-the-art CNN models which use an input size of 300x300 pixels in majority. The performance of the Outsourcer is mainly dependant on image size since capturing and compressing a frame takes more time for large frames. Due to the non-blocking message pattern, our results show that no time is spent on network wait. The verification scheme takes 0.9ms per iteration with a frame size of 300X300 pixels. This is mainly caused by signing frames and related information and verifying responses of the Contractor. It should be noted that with a higher performance of the Contractor and the Verifier the Outsourcer has to spend more time on average on verifying responses in an iteration.

Contractor performance

On a regular mid-range GPU and CPU, the Contractor achieved 68.06 fps using Yolov4 with tiny weights and a 416X416 input size. Due to tasks of the verification running in parallel to the bottleneck thread performing object detection, our verification scheme did not decrease the performance of one cycle. Even in our implementation that does not use multi-threading of key tasks, the verification scheme only takes 0.36ms. This is caused by verifying the frame and related information sent by the Outsourcer and signing the response and related information. Thus, in the parallel version, the verification scheme adds 0% latency to the implementation, while the sequential version adds 1.7% of latency to the implementation.

On the Coral USB Edge Accelerator and a mid-range CPU, the Contractor achieved 63.59 fps using Mobilenet SSD V2 and a 300X300 input size. Due to tasks of the verification running in parallel to the bottleneck thread performing object detection, our verification scheme did not decrease the performance of one cycle. Even in our implementation that does not use multi-threading of key tasks, the verification scheme only takes 0.30ms. This is caused by verifying the frame and related information sent by the Outsourcer and signing the response and related information. Thus, in the parallel version, the verification scheme adds 0% latency to the implementation, while the sequential version adds 1.4% of latency to the implementation.

On the Coral USB Edge Accelerator and a low-range Notebook CPU, the Contractor achieved 49.3 fps using Mobilenet SSD V2 and a 300X300 input size. Due to tasks of the verification running in parallel to the bottleneck thread performing object detection, our verification scheme did not decrease the performance of one cycle. Even in our implementation that does not use multi-threading of key tasks, the verification scheme only takes 0.46ms. This is caused by verifying the frame and related information sent by the Outsourcer and signing the response and related information. Thus, in the parallel version, the verification scheme adds 0% latency to the implementation, while the sequential version adds 1.3% of latency to the implementation.

Verifier performance

On the Coral USB Edge Accelerator and a low-range Notebook CPU, the Contractor achieved 28.75 fps using Mobilenet SSD V2 and a 300X300 input size. The reason why the Verifier performs worse than the Contractor with the same hardware is that a blocking message pattern is used. As sampling-based re-execution usually requires less than 1 fps, a blocking message pattern allows for simpler response handling of the Outsourcer. Since the performance improvement of a non-blocking message pattern is less than 50%, it only improves the rate at which an Outsourcer processes Verifier responses if complete re-execution is used. In this case, our implementation can be easily adjusted to utilize the same message pattern as for the Contractor to achieve nearly identical performance at a slightly more complex sampling logic for the Outsourcer.

Overall System performance

In an expected real-world application, the Outsourcer sends every frame to the Contractor and less than 1% of frames to the Verifier. Considering the performances of each participant individually in our test setup, the Outsourcer can achieve significantly more fps (up to 236 fps) than the Contractor (up to 68.06 fps). Thus, most frames sent by the Outsourcer get discarded by the Contractor since it cannot keep up with the high framerate of the Outsourcer. For this reason, we added a frame-sync option to the Outsourcer scripts that limits sent frames to the framerate that the Contractor can process. This framerate is calculated and adjusted over time in case of temporal performance changes automatically. A frame-sync option saves network bandwidth and computational resources of the Outsourcer in case it can capture, compress, and sign frames faster than the Contractor needs for one iteration. In our test setup, the Contractor is the bottleneck of the system, limiting overall performance to 68.06 fps. As we only used a mid-range GPU and an entry-level edge accelerator in our tests, a faster Contractor can be easily realized with more potent hardware to increase overall system performance to more than 200fps.

Table 5.2.: Key Results

Participant	**Device**	**CPU**	**GPU**	**Model**	**Frames per second**	**Milliseconds per frame**	**% spent on network wait**	**% spent on application processing**	**% spent on verification scheme**	**ms spent on verification scheme**
Outsourcer	Raspberry Pi Model 4B			Mobilenet SSD V2 300*300	236.00	4.24	0.00	78.70	21.30	0.90
Outsourcer	Raspberry Pi Model 4B			Yolov4 tiny 416*416	146.90	6.81	0.00	85.10	14.90	1.01
Contractor	Desktop PC	Core i7 3770K	GTX 970	Yolov4 tiny 416*416	68.06	14.69	0.00	100.00	0.00	0.00
Contractor	Desktop PC	Core i7 3770K	Coral USB Accel-erator	Mobilenet SSD V2 300*300	63.59	15.73	0.00	100.00	0.00	0.00
Contractor	Notebook	Core i5 4300U	Coral USB Accel-erator	Mobilenet SSD V2 300*300	49.30	20.28	0.00	100.00	0.00	0.00
Verifier	Notebook	Core i5 4300U	Coral USB Accel-erator	Mobilenet SSD V2 300*300	28.75	34.78	?	0.64	?	?

5.1.3. Multi-threading performance difference

In this section, we explain in more detail, how we split up key tasks to threads running in parallel based on their individual performance. In our regular Contractor scripts that do not use multithreading of key tasks, we use 2 threads. Thread 2 is constantly loading new frames from the network into memory, that are sent by the Outsourcer. Thread 1 performs all tasks necessary to verify messages, calculate responses, and sending responses back to the Outsourcer.

Figure 5.3 illustrates Thread 1, and figure 5.4 illustrates Thread 2. As one can see, Thread 2 loads new frames and related information sent by the Outsourcer into memory every 6.8ms. Thread 1 performs all key tasks consisting of decoding/decompressing a frame, verifying the signature of the Outsourcer, prepossessing the frame, running CNN inference, postprocessing object detection results into a formatted response, signing the response, and sending it to the Outsourcer over the network. All key tasks combined take 21.6 ms. In the meantime, Thread 2 already loaded 3 new messages sent by the Outsourcer into memory. Thus, two messages get discarded and the third message is fetched by the next iteration of Thread 1 to repeat all tasks.

In our implementation using a regular GPU, the GPU is required during inference and postprocessing. These tasks take 13.9ms out of the 21.6ms necessary to perform a whole iteration. Thus, the GPU waits idly for 7.7ms before performing the next object detection as it has to wait for the next frame to be fetched and processed. Likewise, a message that arrives every 6.8ms is ignored until the Contractor finished a whole iteration.

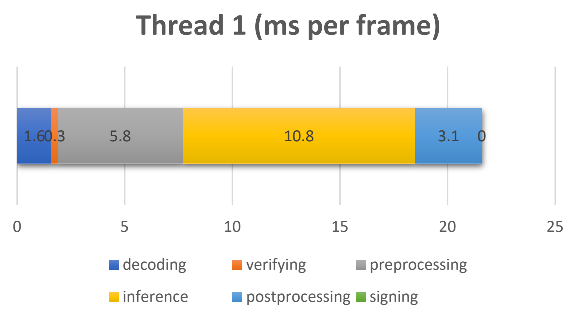


Figure 5.3.: Main thread in regular implementation

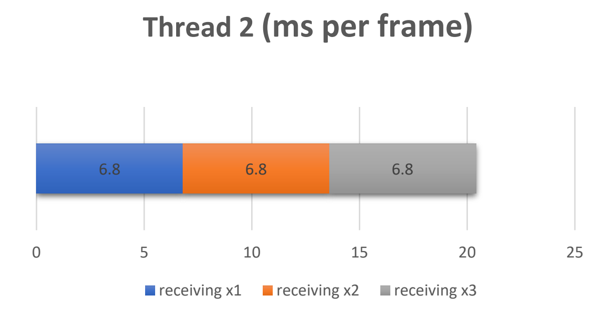


Figure 5.4.: Thread 2 in regular implementation

With multi-threading of key tasks, tasks are run in parallel Thread to eliminate idle waiting time. As in this example, the GPU and deep learning library are needed for inference and postprocessing, both tasks have to be executed by the same thread and cannot be separated. Thus, the bottleneck thread takes 13.9ms plus additional overhead that maintaining multiple threads comes with. All other tasks positioned before or after the bottleneck can thus be aggregated into one thread respectively, limiting total execution time to the performance of the bottleneck tasks, as long as their total computation is less than 13.9ms. Therefore, we end up four three different threads.

Figure 5.5 illustrates the newly added threads. Figure **??** Thread 2 that is identical to Thread 2 in the previous version. Thread 2 still constantly loads a new message from the Outsourcer into memory. Thread 3 loads these messages decodes, verifies, preprocesses them, and stores the result in a thread-safe object. This takes 7.7ms. Since it takes 6.8ms for loading a new message into memory, Thread 3 always catches the latest frame. Thread 1 performs inference fetches the latest preprocessed frame and performs inference and postprocessing. This thread takes 14.1ms. The 0.2ms added time can from the previous version can be explained by the slight computational overhead that running two threads in parallel comes with. Finally, Thread 4 signs the latest object detection result and sends it to the Outsourcer. This thread takes less than 0.1ms.

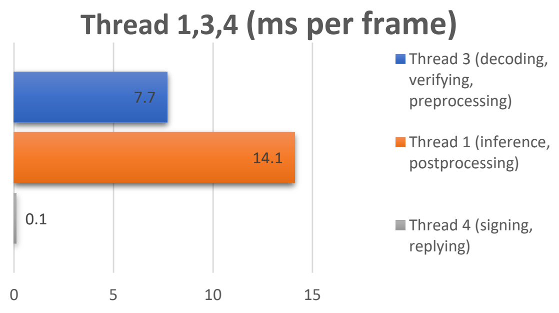


Figure 5.5.: Main threads in mutlithreading implementation

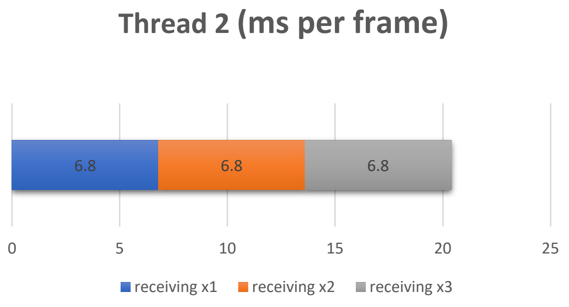


Figure 5.6.: Thread 2 in mutlithreading implementation

The illustration shows, that the total processing time of one iteration was reduced to the time it takes for the bottleneck thread to finish executing. In this example, multi-threading of key tasks improved performance from 21,6ms (46.3 fps) to 14,9ms (70.9 fps) per message (+53,1%).

As explained in chapter 4, in contrast to multiprocessing, multithreading only utilizes one CPU core. Thus, the Contractor should be able to still perform other applications on different CPU cores with identical performance to the previous version. Thus, we recommend to use our scripts with multi-threading of key tasks on all machines, even if they run multiple applications at once. Only, if an application runs on the same CPU core, it may suffer in performance or interfere with the performance improvement gained from the multithreading of key tasks. In our tests, maintaining four threads in parallel only caused an overhead of 0.2ms within the bottleneck thread (1.39%). Thus, we can conclude that the multithreading of key tasks reduces total performance to the performance of the bottleneck thread with only a slight overhead.

The tasks of the verification scheme (signing, and verifying) are exclusively performed in Thread 1 and Thread 3. Since Thread 1 and Thread 3 also perform application-specific tasks such as preprocessing and sending responses, Thread 1 and Thread 3 are necessary even without our verification scheme. Thus, even the multi-threading overhead of 1.39% can not be ascribed to our verification scheme. Therefore, our verification scheme does not at any processing time to the overall Contractor and Verifier performance.

By using multi-threading of key tasks, the performance overhead of our verification scheme to the overall system is only dependant on the Outsourcer performance. This means the total overhead of our verification scheme can be reduced to less than 1ms per frame. As the Contractor was the bottleneck machine in our test, we did not implement a multi-threading version of the Outsourcer. Capturing and especially compressing a frame takes significantly longer than signing it and verifying responses. By running the tasks of the verification scheme in a parallel thread to capturing and compressing a new frame, the Outsourcer can also eliminate the overhead of our verification scheme and achieve even more than 236 fps. Thus, our verification scheme adds less than 1ms per frame to the overall system performance but can be optimized to eliminate any performance overhead.

5.1.4. Network Bandwidth Overhead

The average 416x416 jpg frame was 120 KB in size in our tests. Network bandwidth overhead is neglectable at a maximum of 84 bytes per frame consisting of a 512 bit (64 bytes) large signature, and at most five 32 bit (4 bytes each) integers such as image index, acknowledged responses, and other contract-related information.

5.2. Security of Designed Verification Scheme

The goal of our research was to design and implement a secure, efficient, and scalable verification scheme. To address security, we compiled a comprehensive list of possible protocol violations that serves as our threat model. In chapter 2, we explained each of the 11 protocol violations and summarized them in table 2.4. The violations can be separated into dishonest behaviors of an individual, dishonest behavior via collusion, QoS violations, and external threats.

We also analyzed which techniques are used by other verification schemes proposed by the current academic literature to prevent these behaviors. Out of the 11 identified violations, nine are addresses by current academic literature. For designing our verification scheme, we combined the most promising techniques from different papers, that also meet our criteria in terms of efficiency and scalability. Namely, we use sampling-based re-execution, game-theoretic incentives, blacklisting, a review system, and digital signatures.

Additionally, we designed two new protocols, Randomization and Contestation to prevent the remaining two violations that are currently unaddressed by academic literature. Randomization and Contestation also improve upon existing techniques for other protocol violations in terms of efficiency and added security. A summary of all techniques we use and which protocol violations they prevent can be found in table 3.1 of chapter 3.

As analyzed in chapter 3, our verification scheme detects or prevents most violations with 100% confidence, except the following cases:

1. Whenever the Verifier sends back a false response (number 2 and number 6 in the list), the Contractor can perform Contestation. Contestation has a detection rate of 100% if more than 50% of available Verifiers in the ecosystem are honest. The only way for Contestation to fail is if more than 50% of available Verifiers in the ecosystem are dishonest or lazy and all send back an identical response. This can only happen if all random Verifiers selected during Contestation are colluding, or are using the same q-algorithm to generate a probabilistic result. Both scenarios are highly unlikely in practice and can be neglected.

2. If a contractor sends back false responses (number 1), sampling-based re-execution only needs a small number of samples to detect this violation with more than 99% confidence. If a confidence of 100% is preferred, our scheme and implementation allow the Outsourcer and Verifier to perform complete re-execution to always detect these violations at the expense of efficiency. Thus, the Outsourcer can set any sampling rate to find its preferred security-efficiency trade-off. We showed in chapter 3 that a sampling rate of less than 1% can detect most dishonest Contractors with almost 100% confidence. Therefore, the default setting of our implementation performs sampling-based re-execution with a sampling rate of 1%.

3. We prevent a scenario where the Contractor and the Verifier collude (number 7) with high confidence. First, our verification scheme uses Randomization to prevent any planned collusion and ad-hoc collusion with high confidence. Second, our verification scheme checks for all contracts if they specify optimal incentives. With optimal incentives, any ad-hoc collusion can be prevented as well assuming the Contractor and the Verifier are rational, expected payoff maximizers. Even if this assumption turns out to be false, the Contractor and the Verifier do not know each other's identity due to

Randomization. Therefore, they cannot agree on a false response to trick the Outsourcer. With a higher number of available Contractors and Verifiers and a higher network honesty rate, the effectiveness of designed techniques increases. It should be noted that this violation is a highly invested attack, as it requires at least two colluding participants. While the probabilities of successful prevention of these violations due to the described techniques are hardly quantifiable, we conclude that for a sufficiently large ecosystem, the chance of this violation being successful is neglectable.

In summary, even in the worst case, our verification scheme prevents or detects 6 (number 4,5,8,9,10,11) out of 11 protocol violations with 100% confidence. If greater than 50% of available Verifiers in the ecosystem are honest, it prevents or detects 9 (additionally number 2,3,6) out of 11 protocol violations with 100% confidence. 2 (1,7) out of 11 protocol violations can be detected with nearly 100% confidence. Thus, we can conclude that in a typical scenario, all possible protocol violations are detected or prevented with almost 100% confidence.

QoS Violations such as frame loss, low response rate, or timeouts are eventually prevented before the start of a contract with high confidence due to the review system and blacklisting. During a contract, they are detected with 100% confidence. External threats are detected with 100% confidence as each internal participant signs each message with a digital signature before sending it.

Like all other state-of-the-art verification schemes, our verification scheme requires a trusted payment settlement entity to conduct payment once a contract is finished. This payment settlement entity needs to be able to verify signatures of participants and prevent Sybil attacks by registering new participants that join the ecosystem. Optionally it may feature a reward system. While a reward system is not needed to ensure the security of our verification scheme, it can incentivize good QoS. The payment settlement entity can be any central authority or Blockchain and does not have to perform any latency-critical communication.

5.3. Efficiency of Designed Verification Scheme

In our tests, our verification scheme added less than 1ms latency per input to the overall system performance. By implementing multithreading of key tasks, the added latency can be reduced to 0 by running tasks of the verification scheme in parallel to the outsourced computation or capturing of new inputs.

The low overhead was achieved by utilizing four different performance-enhancing techniques:

1. Merkle trees (optional): The Contractor and the Outsourcer respectively only sign and verify responses that contain a Merkle root hash or a Merkle tree challenge. Merkle root hashes are sent every few responses in a specified interval to commit to multiple responses at once. Thus, Merkle trees can improve overall performance slightly.

2. Non-blocking message pattern: In a traditional blocking request-response message pattern, participants wait idly for a response over the network whenever they send a

message. We use a non-blocking message pattern to load new messages sent by other participants into the memory by a thread running in parallel to the main thread. This eliminates network wait as the main thread can always fetch the newest message from memory in a new iteration without any delay. The use of a non-blocking message pattern improved performance in our low-latency test environment significantly. In high-latency environments even more performance improvement can be expected.

3. Best performing digital signature algorithm: We conducted a benchmark to identify the fastest digital signature algorithm in Python. ED25519 resulted not only to be the fastest but also a highly secure digital signature algorithm that relies on SHA-512 and Curve25519. Curve25519 is considered the new standard of elliptic curves for asymmetric encryption due to its high security. Using the NaCl ED25519 library over ECDSA-python ECDSA improves the performance of the verification scheme moderately.

4. Multi-threading of key tasks (optional). Multi-threading of key tasks can be used to perform all tasks of the verification scheme in a parallel thread to the main thread. Depending on the application, also other key tasks can be moved to parallel threads or processes to improve performance. Usually, the outsourced function is the performance bottleneck of the implementation, as only complex tasks need to get outsourced by computationally weak machines. By running the outsourced function in the main thread it becomes the bottleneck thread and dictates the overall performance of the system. Our tests have shown that running the tasks of our verification scheme in parallel to a complex bottleneck thread only decreases the performance of the bottleneck thread slightly. Also, the tasks of the verification scheme can be added to other non-bottleneck threads. Thus, multithreading of key tasks can eliminate any performance overhead caused by our verification scheme. In contrast to multiprocessing, using multithreading fully utilizes one CPU core due to high-frequency thread switching but does not affect other CPU cores that may be used for different applications. Multithreading of key tasks improves performance significantly.

Our test setup used state-of-the-art CNNs for object detection. While being a very computationally demanding tasks, we achieved up to 68 fps for Contractors running state-of-the-art object detection models with both an edge accelerator and a GPU. The Outsourcer even achieved up to 236 fps. All performance-enhancing improved the performance of our initial implementation by more than 100%. However, even without performance-enhancing techniques, our test system achieved more than 30 fps and can be considered practically for real-time applications.

The network bandwidth overhead per message is between at most 84 bytes large depending on which participant sends a message and if Merkle trees are used. It consists of a 64 bytes large signature and additional contract- or message-related 4 bytes large integers. In our use case with an input size of 120kb, this network bandwidth overhead is neglectable.

5.4. Scalability of Designed Verification Scheme

Our verification scheme features increased security with an increased number of machines participating in the ecosystem due to two reasons. First, a larger number of available Contractors and Verifiers in proximity when an Outsourcer starts a Contract decreases the possibility that Randomization matches a colluding Contractor and a colluding Verifier that know each other's identity. Second, the security of Contestation increases with each available Verifier. Contestation detects a cheating participant with 100% confidence if at least 50% of available Verifiers in the ecosystem are honest. A large amount of available Verifiers in the ecosystem increases the difficulty of potential attackers to gain the majority of the available Verifiers.

Our verification scheme also features increased performance with a large number of participants and past contracts. Due to the optional review system, nodes that provide a good QoS will be filtered eventually and may profit from more contracts and better rates. Also, local blacklists punish bad QoS as long as the associated node does not register with a new identity. Since we assume, that the payment scheme that is implemented along with our scheme handles registering, it ideally prevents creating an arbitrary number of identities. Additionally, every latency-critical computation marketplace naturally improves performance with more participants as the probability for the Outsourcer to locate a Contractor and a Verifier in proximity increases. Thus, also the matching rate and reliability increases with more participants.

The payment scheme is not likely to become the bottleneck of the system due to the following reasons: First, it only has to verify two signatures per contract in an honest case and handle the payment. If the optional review system is used it also stores up to one review per participant per contract. Second, the payment settlement entity is never involved in re-execution and is therefore function-independently. Thus, its computational overhead per contract is constant. Third, the payment settlement entity can conduct payments with an arbitrary delay as our verification scheme does not rely on latency-sensitive microtransactions. Thus it does not need to be located in proximity to the participants and only has to handle 2-3 payments per contract. Finally, even in a dishonest case it only needs to verify up to two additional signatures of additional Verifiers per iteration of Contestation.

We conclude that the performance and security of our system improves with an increasing number of participants. The payment scheme used alongside our verification scheme only has to perform a few computationally cheap tasks independent of the length of a contract. It does not have to be located in proximity to the participants and can conduct payments with a large delay.

5.5. Comparison with other Verification Schemes

6. Conclusion

6.1. Future Work

6.2. Summary

A. General Addenda

A.1. Detailed Addition

B. Figures

B.1. Example 1

✓

B.2. Example 2

✗

List of Figures

List of Tables

Bibliography

[1] H. Huang, X. Chen, Q. Wu, X. Huang, and J. Shen. “Bitcoin-based fair payments for outsourcing computations of fog devices”. In: *Future Generation Computer Systems* 78 (Dec. 2016). DOI: `10.1016/j.future.2016.12.016`.

[2] R. Rahmani, Y. Li, and T. Kanter. “A Scalable Distriubuted Ledger for Internet of Things based on Edge Computing”. In: *Seventh International Conference on Advances in Computing, Communication and Information Technology-CCIT 2018, Rome, Italy, 27-28 October, 2018*. Institute of Research Engineers and Doctors (IRED). 2018, pp. 41–45.

[3] A. Zavodovski, S. Bayhan, N. Mohan, P. Zhou, W. Wong, and J. Kangasharju. “DeCloud: Truthful Decentralized Double Auction for Edge Clouds”. In: May 2019. DOI: `10.1109/ICDCS.2019.00212`.

[4] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu. “Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues”. In: *IEEE Access* 6 (2018), pp. 18209–18237.

[5] Y. Wang, Z. Tian, S. Su, Y. Sun, and C. Zhu. “Preserving Location Privacy in Mobile Edge Computing”. In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. May 2019, pp. 1–6. DOI: `10.1109/ICC.2019.8761370`.

[6] M. Gheisari, Q.-V. Pham, M. Alazab, X. Zhang, C. Fernández-Campusano, and G. Srivastava. “ECA: An Edge Computing Architecture for Privacy-Preserving in IoT-based Smart City”. In: *IEEE Access* PP (Aug. 2019), pp. 1–1. DOI: `10.1109/ACCESS.2019.2937177`.

[7] I. Psaras. “Decentralised Edge-Computing and IoT through Distributed Trust”. In: June 2018, pp. 505–507. DOI: `10.1145/3210240.3226062`.

[8] Y. C. Chunming Tang. *Efficient Non-Interactive Verifiable Outsourced Computation for Arbitrary Functions*. Cryptology ePrint Archive, Report 2014/439. `https://eprint.iacr.org/2014/439`. 2014.

[9] C. Xiang and C. Tang. “New verifiable outsourced computation scheme for an arbitrary function”. In: *International Journal of Grid and Utility Computing* 7 (Jan. 2016), p. 190. DOI: `10.1504/IJGUC.2016.080187`.

[10] T. Combe, A. Martin, and R. Di Pietro. “To Docker or Not to Docker: A Security Perspective”. In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[11] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. *SGAxe: How sgx fails in practice*. 2020.

[12] S. Eisele, T. Eghtesad, N. Troutman, A. Laszka, and A. Dubey. “Mechanisms for Outsourcing Computation via a Decentralized Market”. In: *arXiv preprint arXiv:2005.11429* (2020).

[13] B. Carbunar and M. Tripunitara. “Fair Payments for Outsourced Computations”. In: *2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*. 2010, pp. 1–9.

[14] B. Carbunar and M. V. Tripunitara. “Payments for Outsourced Computations”. In: *IEEE Transactions on Parallel and Distributed Systems* 23.2 (2012), pp. 313–320.

[15] P. Golle and I. Mironov. “Uncheatable Distributed Computations”. In: vol. 2020. Apr. 2001, pp. 425–440. DOI: `10.1007/3-540-45353-9_31`.

[16] R. Freivalds. “Probabilistic Machines Can Use Less Running Time.” In: *IFIP congress*. Vol. 839. 1977, p. 842.

[17] M. J. Atallah and K. B. Frikken. “Securely Outsourcing Linear Algebra Computations”. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’10. Beijing, China: Association for Computing Machinery, 2010, pp. 48–59. ISBN: 9781605589367. DOI: `10.1145/1755688.1755695`. URL: `https://doi.org/10.1145/1755688.1755695`.

[18] X. Lei, X. Liao, T. Huang, and F. H. Rabevohitra. “Achieving security, robust cheating resistance, and high-efficiency for outsourcing large matrix multiplication computation to a malicious cloud”. In: *Inf. Sci.* 280 (2014), pp. 205–217.

[19] Z. Cao and L. Liu. “A note on "achieving security, robust cheating resistance, and high-efficiency for outsourcing large matrix multiplication computation to a malicious cloud"”. In: (Mar. 2016).

[20] D. Benjamin and M. J. Atallah. “Private and Cheating-Free Outsourcing of Algebraic Computations”. In: *2008 Sixth Annual Conference on Privacy, Security and Trust*. 2008, pp. 240–245.

[21] X. Lei, X. Liao, T. Huang, H. Li, and C. Hu. “Outsourcing Large Matrix Inversion Computation to A Public Cloud”. In: *IEEE TRANSACTIONS ON CLOUD COMPUTING* 1 (July 2013), pp. 78–87. DOI: `10.1109/TCC.2013.7`.

[22] C. Wang, K. Ren, J. Wang, and Q. Wang. “Harnessing the Cloud for Securely Outsourcing Large-Scale Systems of Linear Equations”. In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (2013), pp. 1172–1181.

[23] W. Song, B. Wang, Q. Wang, C. Shi, W. Lou, and Z. Peng. “Publicly Verifiable Computation of Polynomials Over Outsourced Data With Multiple Sources”. In: *IEEE Transactions on Information Forensics and Security* 12.10 (2017), pp. 2334–2347.

[24] X. Wang, K.-K. R. Choo, J. Weng, and J. Ma. “Comments on “Publicly Verifiable Computation of Polynomials Over Outsourced Data With Multiple Sources””. In: *IEEE Transactions on Information Forensics and Security* PP (Aug. 2019), pp. 1–1. DOI: `10.1109/TIFS.2019.2936971`.

[25] J. Meena, S. Tiwari, and M. Vardhan. "Privacy preserving, verifiable and efficient outsourcing algorithm for regression analysis to a malicious cloud". In: *Journal of Intelligent & Fuzzy Systems* 32 (Apr. 2017), pp. 3413–3427. DOI: `10.3233/JIFS-169281`.

[26] H. Chabanne, J. Keuffer, and R. Molva. "Embedded Proofs for Verifiable Neural Networks". In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 1038.

[27] S. Lee, H. Ko, J. Kim, and H. Oh. "vCNN: Verifiable Convolutional Neural Network". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 584.

[28] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: May 2016, pp. 305–326. ISBN: 978-3-662-49895-8. DOI: `10.1007/978-3-662-49896-5_11`.

[29] X. Chen, J. Ji, L. Yu, C. Luo, and P. Li. "SecureNets: Secure Inference of Deep Neural Networks on an Untrusted Cloud". In: *ACML*. 2018.

[30] Z. Ghodsi, T. Gu, and S. Garg. "SafetyNets: Verifiable Execution of Deep Neural Networks on an Untrusted Cloud". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 4675–4684. ISBN: 9781510860964.

[31] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar. *Towards the AlexNet Moment for Homomorphic Encryption: HCNN, theFirst Homomorphic CNN on Encrypted Data with GPUs*. 2018. arXiv: `1811.00778 [cs.CR]`.

[32] X. Hu and C. Tang. "Secure outsourced computation of the characteristic polynomial and eigenvalues of matrix". In: *Journal of Cloud Computing* 4 (Dec. 2015). DOI: `10.1186/s13677-015-0033-9`.

[33] G. Xu, G. T. Amariucai, and Y. Guan. "Delegation of Computation with Verification Outsourcing: Curious Verifiers". In: *IEEE Transactions on Parallel and Distributed Systems* 28.3 (2017), pp. 717–730.

[34] W. Du, J. Jia, M. Mangal, and M. Murugesan. "Uncheatable grid computing". In: *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE. 2004, pp. 4–11.

[35] L. Wei, H. Zhu, Z. Cao, X. Dong, W. Jia, Y. Chen, and A. V. Vasilakos. "Security and privacy for storage and computation in cloud computing". In: *Information sciences* 258 (2014), pp. 371–386.

[36] H. Wang. "Integrity verification of cloud-hosted data analytics computations". In: *Proceedings of the 1st International Workshop on Cloud Intelligence*. 2012, pp. 1–4.

[37] M. Nabi, S. Avizheh, M. V. Kumaramangalam, and R. Safavi-Naini. "Game-Theoretic Analysis of an Incentivized Verifiable Computation System". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2019, pp. 50–66.

[38] R. Di Pietro, F. Lombardi, F. Martinelli, and D. Sgandurra. "Anticheetah: Trustworthy computing in an outsourced (cheating) environment". In: *Future Generation Computer Systems* 48 (2015), pp. 28–38.

[39] X. Chen, J. Li, and W. Susilo. “Efficient fair conditional payments for outsourcing computations”. In: *IEEE Transactions on Information Forensics and Security* 7.6 (2012), pp. 1687–1694.

[40] A. Küpçü. “Incentivized outsourced computation resistant to malicious contractors”. In: *IEEE Transactions on Dependable and Secure Computing* 14.6 (2015), pp. 633–649.