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:	15.02.2021

I confirm that this master's thesis in information systems is my own work and I have documented all sources and material used.

Munich, 15.02.2021 Christopher Harth-Kitzerow

Acknowledgments

I would like to thank my supervisor Prof. Dr. Jörg Ott, for giving me the opportunity to pursue this exciting topic at his chair and for his support along the way. I also want to thank my advisor Mr. Gonzalo Munilla Garrido, for his excellent advice and guidance. In addition, I would like to thank Dr. Nitinder Mohan for providing valuable feedback and suggestions. Finally, many thanks to my parents, who have been supporting and encouraging me all my life.

Abstract

Latency-sensitive Internet of Things (IoT) applications need edge computing to overcome their limited computing capabilities. As idle resources at the edge of a network are not providing any value, they could instead perform an IoT device's computation. 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 whether 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 all participants that they can use 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 comprehensive list of protocol violations that might threaten the integrity or quality of outsourced computation. Our verification scheme provides techniques to prevent or detect each identified protocol violation 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 at most 84 bytes per frame, independent of the frame's size. We also implemented a multi-threaded version of our scripts that performs our verification scheme's tasks parallel to the object detection to eliminate any latency overhead of our scheme. In addition, we 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 our verification scheme's outsourced object detection performance with a regular mid-range GPU and a low-end Edge Accelerator. We were able to achieve more than 60 frames per second with either hardware. Our tests show 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. Compared to other proposed verification schemes, our scheme resists a comprehensive set of protocol violations without sacrificing performance.

Kurzfassung

Latenzempfindliche Internet of Things (IoT)-Anwendungen benötigen Edge-Computing, um ihre begrenzten Rechenkapazitäten zu überwinden. Nicht vollständig ausgelastete Rechenleistung von Hardware am Rande eines Netzwerks könnte genutzt werden, um die Berechnungen von IoT-Geräten in der Nähe zu übernemen. Ein Edge-Computing-Marktplatz könnte es IoT-Geräten (Outsourcern) ermöglichen, Berechnungen an beliebige, teilnehmende Rechen-Geräte (Contractors) in ihrer Nähe auszulagern. Im Gegenzug erhalten diese Rechen-Geräte eine Vergütung für die Bereitstellung von Rechenleistung und Speicherkapazitäten.

Da IoT-Geräte oft rechenschwach sind, ist es für sie ggf. schwierig zu überprüfen, ob die von einem unbekanntem Contractor zurückgegebenen Antworten korrekt sind. Contractor haben sogar einen Anreiz, ein rechnerisch weniger aufwendiges probabilistisches Ergebnis zurückzugeben, um Ressourcen zu sparen und trotzdem eine Entlohnung zu erhalten. Ebenso hat ein Outsourcer keinen Anreiz, einen ehrlichen Auftragnehmer rechtmäßig zu bezahlen, nachdem er alle Berechnungsergebnisse erhalten hat. Mikrotransaktionen dienen hierbei nicht als Lösung, da sie mit hohen Transaktionskosten verbunden sind und ein Latenz-Bottlenceck darstellen könnten.

In dieser Masterarbeit stellen wir ein Verifikationsschema vor, das die Verifikation beliebiger deterministischer Funktionen ermöglicht, indem der Outsourcer zufällige Stichproben an einen zusätzlichen, unbekannten Verifier im Netzwerk sendet und die Antworten mit denen des Contractors vergleicht. Unser entworfenes Verifikationsschema liefert außerdem öffentlich verifizierbare Beweise für alle Teilnehmer, die verwendet werden können, um Zahlungen zu erzwingen, falls alle gesendeten Antworten korrekt waren.

Wir berücksichtigen, dass alle beschriebenen Teilnehmer, die sich auf das Verifikationsschema einlassen, unehrlich oder faul sein könnten. Daher haben wir eine umfassende Liste von Protokollverletzungen zusammengestellt, die die Integrität oder Qualität der ausgelagerten Berechnungen gefährden könnten. Unser Verifikationsschema bietet Techniken, um jede identifizierte Protokollverletzung mit hoher Wahrscheinlichkeit zu verhindern oder zu erkennen.

Objekterkennung ist ein relevanter Anwendungsfall für IoT-Geräte und eine rechenintensive Aufgabe. Wir haben unser Verifizierungsschema mit aktuellen, vortrainierten Convolutional Neural Network-Modellen getestet, die für Objekterkennung entwickelt wurden. Auf allen Geräten erreichte unser Verifikationsschema einen Rechnenaufwand von weniger als 1ms und einen vernachlässigbaren Netzwerkbandbreiten-Overhead von höchstens 84 Byte pro Frame, unabhängig von der Größe des Frames. Wir haben auch eine Multithreading-Version unserer Skripte implementiert, die die Aufgaben unseres Verifikationsschemas parallel zur Objekterkennung ausführt, um jeglichen Latenz-Overhead unseres Schemas zu eliminieren. Zusätzlich haben wir ein nicht blockierendes Nachrichtenmuster verwendet, um neue Frames

parallel zu allen anderen Threads zu empfangen, in den Arbeitsspeicher zu laden und vorzuverarbeiten, um hohe Frame-Raten zu erreichen.

Wir haben die Leistung der ausgelagerten Objekterkennung mit unserem Verifikationsschema mit einer regulären Mittelklasse-GPU und einem Low-End Edge Accelerator evaluiert und konnten mit beiden Geräten mehr als 60 Bilder pro Sekunde erreichen. Unsere Tests haben gezeigt, dass ein kostengünstiges Setup aus einer Low-End-CPU in Kombination mit einem Low-End-Edge Accelerator als Contractor für ausgelagerte Objekterkennung verwendet werden kann. Im Gegensatz zu anderen vorgeschlagenen Verifikationsschemata verhindert unser Schema alle von uns identifizierten Protokollverletzungen ohne Leistungseinbußen.

Contents

Acknowledgments iii

Abstract iv

Kurzfassung v

1. Introduction 1
- 1.1. Assumptions . . . 3
- 1.2. Problem Statement . . . 4
- 1.3. Research Objective . . . 5
- 1.4. Thesis Approach . . . 6

2. Literature Review 8
- 2.1. Components of an Edge Computing Marketplace . . . 8
- 2.2. Verification of Computation . . . 9
  - 2.2.1. Verification Schemes for Arbitrary Functions . . . 9
  - 2.2.2. Verification Schemes for Specific Functions . . . 11
- 2.3. Verification of Convolutional Neural Network Inference . . . 20
- 2.4. Verification of outsourced Computation using Re-execution . . . 20
- 2.5. Protocol Violations . . . 30

3. Design of our Verification Scheme 40
- 3.1. Overview . . . 40
- 3.2. Initial Situation . . . 42
- 3.3. Preparation Phase . . . 42
  - 3.3.1. Randomization . . . 43
  - 3.3.2. Game-theoretic Incentives . . . 44
- 3.4. Execution Phase . . . 48
  - 3.4.1. Sampling . . . 48
  - 3.4.2. Signatures . . . 50
  - 3.4.3. Improving Efficiency of the Execution Phase . . . 52
- 3.5. Closing Phase . . . 56
  - 3.5.1. Review System . . . 57
  - 3.5.2. Blacklisting . . . 57
  - 3.5.3. According to Custom . . . 57
  - 3.5.4. Quality of Service Violations . . . 58
  - 3.5.5. Dishonest Behavior . . . 59

3.5.6. Contestation . . . 59
3.6. Threat Model . . . 61

4. Implementation **68**
4.1. Software Architecture . . . 68
4.2. Improving Run-time of Implementation . . . 77
4.2.1. Benchmark of Digital Signature Algorithms in Python . . . 77
4.2.2. Non-blocking Message Pattern . . . 79
4.2.3. Parallel Execution . . . 79
4.3. Test Setup . . . 84

5. Evaluation **86**
5.1. Results . . . 86
5.1.1. Performance . . . 86
5.1.2. Final Results . . . 88
5.1.3. Multi-threading Performance Difference . . . 92
5.1.4. Network Bandwidth Overhead . . . 95
5.2. Security of the Designed Verification Scheme . . . 95
5.3. Efficiency of the Designed Verification Scheme . . . 97
5.4. Scalability of the Designed Verification Scheme . . . 98
5.5. Comparison with other Verification Schemes . . . 99

6. Conclusion **106**
6.1. Key Findings . . . 106
6.2. Assessment of our Results . . . 107

A. Digital Addenda **109**

List of Figures **110**

List of Tables **112**

Glossary **113**

Bibliography **115**

1. Introduction

Offloading computational tasks from IoT devices to computational resources at the Edge can improve the responsiveness of existing applications and enable the implementation of novel latency-sensitive use cases [1]. 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 with enough computational resources available to execute the assigned function with sufficient Quality of Service (QoS). Resources at the Edge might be unavailable due to the following reasons: First, if there is a high resource demand caused by other jobs, these might temporarily utilize all the computing resources of third-party machines in proximity. Second, there might be no existing hardware near requesting devices in different Edge scenarios such as rural areas. Third, not all nearby third-party hardware might have the necessary hardware or software to process a specific outsourcing task.

A computing marketplace might utilize different techniques such as resource reservation to ensure a certain level of reliability for Outsourcers to find an available Contractor at all times. However, even if, despite these techniques, resources in an edge computing marketplace remain unreliable, they might still provide temporal QoS improvements compared to receiving service of 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 served by an edge server in proximity [2].
2. An IoT device can quickly transfer large amounts of data at low costs [3].
3. An IoT device can offload storage to an Edge Server which might compress the data before sending it over the internet [4].
4. An IoT device can turn off long-range communication-technology to save energy [5].

Figure 1.1 illustrates different temporal QoS improvements. The white clouds in the figure represent cloud servers that serve the IoT devices in case no edge server is available. The black cylinders represent edge servers that serve the IoT devices whenever available to reduce latency, save bandwidth, or save energy.

In this thesis, we introduce a verification scheme that can verify the integrity of arbitrary functions. We evaluated its performance by integrating it into an application that performs object detection based on a real-time image stream.

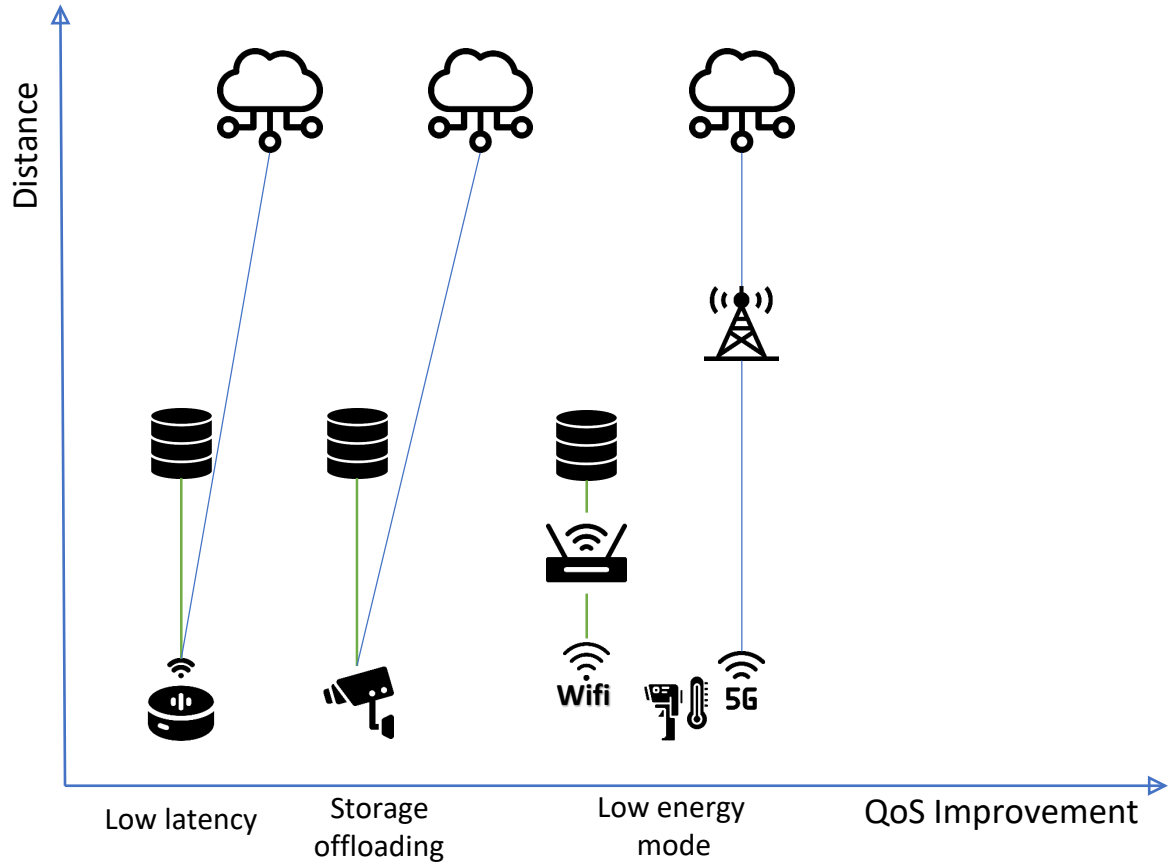

Figure 1.1.: Temporary QoS improvements

Object detection is a suitable application type for IoT devices to outsource to nearby Edge servers due to the following reasons:

First, reliable object detection is complex. The state-of-the-art technique to perform object detection is by performing Inference with a pre-trained 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 (Accelerated Processing Units). These model sizes are suitable to transfer to another third-party device even for one-time outsourcing jobs. Most IoT devices do not have a powerful GPU or APU to perform CNN inference in real-time. Also, the software limitations of an IoT device can make the execution of a pre-trained model difficult.

Second, object detection with real-time image/video streams is a latency-sensitive task for most applications. For example, when moving IoT devices outsource images, real-time analysis can be outdated after just a few milliseconds. Thus, outsourcing all inputs to the cloud is usually not an option as the physical distance to a cloud server is too large to allow for toleratable latency.

Third, image and video files are large and require high bandwidth when sent over a

network, even when compressed. For example, a compressed jpeg image stream with a small resolution of 500X500 can already require more than 5Mb/s of bandwidth. Routing large files over a local network can be more efficient than transferring large files over long distances via the Internet.

Also, there are multiple possible use cases for IoT devices required to perform object detection [6]. The list below provides a few examples:

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 autonomously picks up objects for relocation in a warehouse.

1.1. Assumptions

We assume the following points for outsourced object detection in an edge computing marketplace:

1. Contractors (edge servers or other third-party hardware) are stationary (reappearing actors).
2. Outsourcers (IoT devices) are mobile (reappearing and adhoc actors).
3. There is a 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 players' payoffs are linear functions of their costs and payments/benefits (and risk-neutral payoff function).
6. Outsourcers pay Contractors for each processed image (this only means we assume billing per-image but does not mean that our scheme relies on micro-transactions or synchronous payment).
7. Cloud servers distribute pre-trained models to third-party hardware.

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

We formed these assumptions due to the following reasons: While there are concepts of mobile edge servers, we assume that the majority of powerful third-party hardware remains wired in a fixed location. In contrast, an IoT device might be a moving object, such as a vehicle or a smartphone, or a stationary object such as a CCTV in a fixed location. Therefore we assume IoT devices to be reappearing or adhoc actors. Naturally, an adhoc actor might

become a reappearing actor if it returns or remains in the edge network. Even though IoT devices might be mobile, we assume that task types are usually reappearing. For instance, several smartphone users may use the same application that requires outsourcing.

Similarly, multiple vehicles produced by the same manufacturer might be adhoc actors but still rely on the same type of service. In these cases, an edge server can utilize an already stored CNN model or any other programming to execute the function without transmission delay. However, as explained earlier, a CNN model designed for APUs can be as small as 5MB in size, making it viable for one-time transmission as well.

Our verification scheme is resistant to any dishonest or lazy hardware. However, for a more straightforward analysis, we assume that a dishonest participant aims to maximize its payoff and has a risk-neutral payoff function. This means we assume that a participant would not engage in dishonest behavior if its expected reward of that behavior is negative. It should be noted that despite these assumptions, our verification scheme still detects or prevents dishonest behavior with high confidence even if dishonest participants violate these assumptions.

We assume that Outsourcers pay for each processed image. This design choice maximizes the flexibility and efficiency of outsourcing. For instance, a time-slot reservation of resources might leave reserved resources unutilized if not requested by the reserving Outsourcer, even if other Outsourcers are currently seeking service. Finally, we assume that Cloud servers distribute their pre-trained models or programs to third-party hardware before participants start a contract. This way, a Contractor and a Verifier can always process the most recent version of the applications. An alternative design choice would be to let the IoT device transfer the model or program to the other participants. We decided against this assumption as it leads to unreasonable storage overhead for the IoT device and is more complicated for reappearing task types.

Figure 1.2 illustrates a typical scenario in an edge computing marketplace. As one can see, the smartwatch in the figure is about to leave the illustrated edge network while a smartphone is about to join. All three smartphones are requesting an outsourcing service for the same app. This example illustrates that the same model deployed one time to an edge server can serve multiple adhoc actors such as smartphones as their task types are reappearing.

1.2. Problem Statement

In cloud computing, verification schemes are often not used as Outsourcers assume that cloud servers act honestly. Usually, a cloud server that is caught cheating can be easily traced back to its provider. Also, large scale cloud servers usually have a global reputation at stake when Outsourcers file complaints. In contrast, in an edge computing marketplace with scarce resources, we assume that cherry-picking highly trusted hardware is usually not possible nor advised. A computation marketplace can only maximize the QoS of outsourced applications if it allows all available third-party hardware in proximity to serve as a potential Contractor. This hardware may include resources from untrusted, anonymous, and private machines.

Additionally, Contractors may exploit computationally weak IoT devices that are incapable

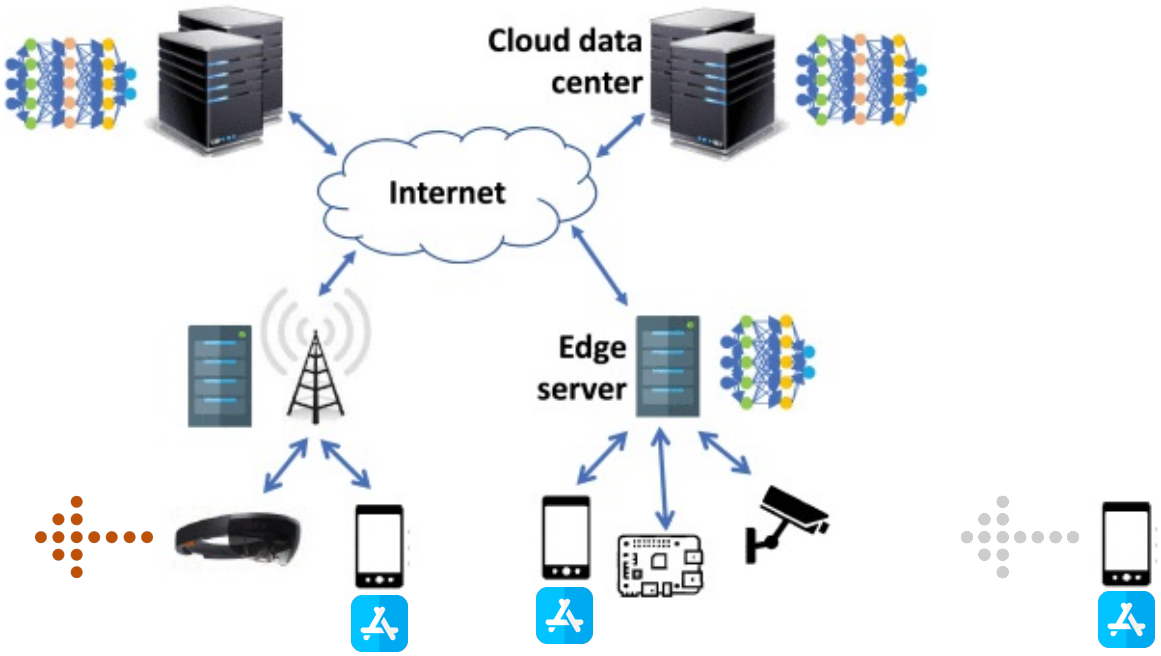

Figure 1.2.: Scenario of an edge computing marketplace with mobile IoT devices outsourcing reappearing object detection task types

of verifying complex computation. Without a global reputation to lose and a variety of adhoc IoT devices seeking service, to act dishonestly can be a rewarding strategy. On the other side, dishonest, adhoc IoT devices might try to refuse to pay honest Contractors. Payment schemes cannot be assumed to be available for low-latency, low-fee micro-transactions at the Edge. Thus, we assume that payment usually needs to be handled in chunks before or after a service is due. Therefore, there is a need for a layer of trust and accountability between all participants.

For these reasons, a secure verification scheme that verifies computations' integrity and records payment obligations is a critical component of an edge computing marketplace.

1.3. 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. Existing verification schemes only address a few threats instead of a comprehensive threat model.

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 TTPs 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 one party's perspective rather than from an ecosystem perspective.

Our research aims 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 novel 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.

1.4. Thesis Approach

There are three research questions (RQs) we address in this thesis:

1. What are state-of-the-art methods to ensure the correctness of outsourced computation?

2. How to implement a secure verification scheme for outsourced computation?

3. How does a prototype that integrates the designed verification scheme from RQ2 perform in terms of scalability and overhead?

In chapter 2 we address RQ1. We first show that a verification scheme is one of the multiple key components that an edge marketplace requires. We then review verification approaches for different functions proposed by the current academic literature. Afterward, we review different techniques used by practical verification schemes in terms of their efficiency and security.

Based on the findings of answering RQ1, we design and implement a verification scheme that solves identified research gaps by combining effective techniques of current verification schemes and by designing new ones if necessary. In our implementation, we develop scripts for three machines in the local network (an Outsourcer, a Contractor, and a Verifier) to outsource, process, and verify object detection on a real-time webcam stream using state-of-the-art pre-trained Convolutional Neural Network (CNN) models. To answer RQ2, each script fully adheres to our designed verification scheme by detecting or preventing any protocol

violation during its execution. A detailed description of our design and implementation choices can be found in chapter 3 and chapter 4.

Finally, in chapter 5, we show detailed benchmarks of our verification scheme and assess its performance. Based on these results, we evaluate if we achieved our objectives of designing a verification scheme that is both secure and scalable. We also compare our verification scheme's security, performance, and third-party involvement with other verification schemes proposed by the current academic literature.

To get a quick overview of all chapters, we recommend reading the first section of chapter 6. In that section, we summarize each chapter's key findings and refer to tables and figures in this thesis that illustrate the most relevant insights.

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 realization of a computing marketplace that specializes in 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 builds an ecosystem of nodes in proximity 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 Outsourcers and Contractors identify each other and negotiate on a price [7]. After they agreed on the contract, the Outsourcer sends inputs to the Contractor to process an assigned function.

Second, it should propose a verification scheme that participants in the ecosystem can use to verify the computation results and if a party is entitled to a payment [8] [9]. 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 [10]. 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.

Edge computing marketplaces are small networks that require less routing than traditional internet applications. Therefore, some authors propose a computation-centric network architecture that focuses on maximizing performance for transmission of outsourced computation at the edge [11].

Additionally, some applications may require privacy preservation of the inputs, functions, or sender information if one of those is confidential [12] [13] [14].

Lastly, a payment scheme should handle payment securely and with low transaction fees [15] [16].

From these identified components, we focused 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 [8] [9]	Secure
	Scalable
	Efficient
Payment [15] [16]	Low fees
	Secure
	Scalable
Matching and price-finding [7]	Fair prices
	High Matching rate
Privacy Preservation [12] [13] [14]	Preserving privacy of sent data
	Preserving privacy of sender information
	Efficient
Network Architecture [11]	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 proposed by current academic literature.

A few verification schemes can be used on arbitrary functions that rely on approaches such as re-execution, trusted hardware, or encryption. Other verification schemes instead only work on specific 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.

2.2.1. 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 (the Contractor), thus preventing any modification.
2. Encrypting all values the function is executed on, thus making it infeasible for the Contractor to generate fake outputs[17] [18].

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

The following paragraphs show a more detailed overview of the different approaches of 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. One of the most widespread types of containers is Docker containers that promise a lightweight and secure option for running code on different platforms and machines. However, Docker containers are vulnerable to multiples attacks such as backdoors, phishing attacks, account hijacking, and unprotected API calls[19].

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 [20]. TEEs are architectural extensions integrated into processors that provide certain security guarantees. It promises secure execution of code even when it runs on third party hardware. Verification schemes such as [21] rely on these properties to let the secret attestation key of the SGX secure enclave serve as integrity proof.

Apart from high computational overhead, however, 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[22]. Thus, when a response is returned to the outsourcing machine, Intel SGX fails to provide secure proof about the result's validity 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 [17] [18]. 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 are most likely not 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. Either the Outsourcer itself can perform re-execution, or it may outsource

the re-execution to another machine. When comparing an output gained by performing re-execution on the same input, the Outsourcer whether both results are identical[23].

In chapter 3, we show that even with a fraction of re-executed samples, a dishonest Contractor 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 Outsourcer is too computationally weak to perform re-execution of the outsourced samples, a third party (a Verifier) 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 themselves do not have to be modified, and there is almost no computational overhead introduced to the Outsourcer. However, the disadvantage is that currently, there is no entirely secure method available that prevents interference of any kind.

Likewise, the advantage of trusted execution environments is that the Outsourcer does not have to transform inputs and functions. In theory, the security guarantees of trusted execution environments can be higher than those of software virtualization. However, 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. The Outsourcer can easily detect any modification as the Contractor cannot generate valid encrypted outputs based on arbitrary values. However, homomorphic encryption is usually the most computationally expensive verification scheme of all available methods.

The Outsourcer re-execute the outsourced function on all or certain values to verify if results gained from re-executing match received results from the Contractor. The re-execution can be either computed by the machine that receives results from the outsourced computation or a Verifier. This method's advantage 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 Outsourcer 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 functions. Table 2.2 shows a summary of the introduced verification approaches for arbitrary functions.

2.2.2. Verification Schemes for Specific Functions

While the approaches listed in the last section can be used for arbitrary functions, 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 $f(x, y)$ 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 functions.

Table 2.2.: Approaches for verifying the integrity of arbitrary functions

Approach	Advantages	Disadvantages
Software Virtualization [19]	low computational overhead, inputs and function do not have to be transformed	not secure
Trusted Execution Environments [22]	inputs and function do not have to be transformed	not secure
Fully Homomorphic Encryption [17] [18]	highly secure	Extreme computational overhead, often unpractical
Re-execution [23]	low computational overhead when sampling is used, secure	need for a third party if outsourcing machine is not able to re-execute the function

Pre-Image Computation

Pre-Image computation describes function types that, given an input domain D, a function f, and a value y, return all values $x \in D$ for which $f(x) = y$ [24] [25]. 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" [24]. A ringer is a tuple $(x, f(x))$ for a given function f and an arbitrary sample input x. This approach's main idea is to precompute the function for a set of values $x \in D$ (true ringers) and $x \notin D$ (bogus ringers). Only if values x returned by the Contractor for the whole computation match results of these precomputed values, the result is accepted [25].

Magic Numbers Another approach is the use of "magic numbers", which rewards the Contractor 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 a value 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 receive the full reward [26].

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

Matrix Multiplication

Matrix multiplication is a typical operation for image processing and a computationally complex component of convolutional neural network training and inference. The most efficient matrix multiplication algorithm has a time complexity of $O(n^{2.3729})$ [27].

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 [28].

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 confirm 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 [29], and permutation [30] [31]. 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, instead of encrypting the input matrices A and B, is to generate random matrices A' and B' and then sending $A' - A$ and $B' - B$ to one of two remote servers. The other remote server receives different transformed matrices. [32]. As only the Outsourcer knows the original matrices and the two remote servers receive different transformed matrices, collusion should not be possible. This scheme's advantage is that it provides input privacy and creates less than $O(n^2)$ computational complexity. We can conclude that this scheme uses a variation of complete re-execution and provides input privacy through matrix transformation. Also, we analyzed that this scheme also works with sampling-based re-execution.

While the schemes presented above feature input privacy in addition to secure verification, these features come with high computational complexity with the exception to [32]. 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 this verification's performance. Transforming matrices before sending them to remote machines according to [32] might be a useful extension for providing input privacy 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 and a resulting matrix B, the Outsourcer simply has checked whether AXB is equal to the identity matrix. Freivald's algorithm can be used to do this verification more efficiently [28].

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 [33]. Secret sharing has the advantage that it preserves input privacy. 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 [34]. 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 a third party Verifier checks to decide whether computation was executed correctly, without revealing inputs [35]. Digital 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 [36].

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 [37]. To verify if the 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 Network Inference

In deep learning, a Convolutional Neural Network(CNN) is a class of deep neural networks that is usually applied to analyzing digital images. CNNs achieve state-of-the-art performance in classification 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 powerful GPUs or Tensor Processing Units (TPUs). For a weak machine, outsourcing only CNN inference 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 calculation [38]. Essentially, this scheme further optimizes re-execution by only checking a few steps of one iteration instead of re-executing one iteration completely with each sample. However, it should be noted that only verifying the matrix multiplication part is not sufficient to verify the integrity of the whole inference process.

Zero-knowledge Succinct Non-interactive Arguments Another approach is to use Zero-knowledge succinct non-interactive arguments (zk-SNARK). With zk-SNARKs a worker can generate proof of the correctness of computation results without revealing private data. The proof relies on translating a function to constrained polynomials and using homomorphic encryption to hide secret inputs. Zk-SNARKs have the advantage that they can provide privacy of secret inputs and privacy of the model's weights. However, generating a proof of CNN inference with zk-SNARKs takes between 8 hours [39] to 10 years [40] on current hardware. They also require to express the inference function as an arithmetic circuit that is between 80GB to 1400TB large. Hence, while offering valuable privacy-preserving properties on top of verification and outsourcing, zk-SNARKs are far from practical to verify CNN inference.

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 model privacy. When receiving a result Y' based on X' and W' the Outsourcer simply randomly checks if $y'_{i,j}$ equals $w'_{i,:} \cdot x'_{:,j}$ for a random row i and a random column j [41]. 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 [42]. 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 [43]. Hence, 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. Application-specific adjustments used by schemes such as SecureNets can further simplify re-execution. Also, input/output and model privacy can be added at an additional computational cost.

Other functions

Some verification approaches proposed by the current academic literature are only applicable to other functions than mentioned above.

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. This approach can be combined with re-execution to enable secure verification 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 [44].

Arithmetic Circuits Some problems can be represented as an arithmetic circuit. Using arithmetic circuits has the same benefits as fully homomorphic encryption but is less computationally expensive. [45] proposes a verification scheme with a third-party Verifier for problems that can be presented as an arithmetic circuit. This scheme's disadvantage is that computational cost is still high, and it assumes that the Outsourcer is 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 cannot generate a false encrypted response that meets simple integrity checks when decrypted. However, verification schemes based on encryption are also not practical for specific functions. Especially schemes that rely on Homomorphic encryption are far from practical in real-time, even when optimized for specific functions.

Schemes for specific functions that use approaches similar to re-execution are mostly practical. They often use a more efficient verification algorithm than re-executing the whole

functions, such as only re-executing parts of a function, using random challenges, or utilizing efficient algorithms such as Freivald's algorithm. Additionally, input transformation seems a promising technique that works for a range of different functions. It 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. Even if the verified part the computation is correct, incorrect results 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 only on parts of the function.

After assessing multiple verification 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 promising technique. It works for a range of specific functions and preserves input and output privacy. Also, Input transformation, in combination with re-execution, can prevent lazy Verifier behavior and collusion attempts. The Outsourcer can simply use two different random numbers to transform the inputs sent to Verifier and Contractor. Consequently, the Contractor and the Verifier cannot produce two transformed outputs that are equal after re-transforming without knowing the Outsourcer's random numbers. Therefore, collusion between Verifier and Contractor is infeasible.

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. Later sections of this chapter focus on how different verification schemes target threats that can emerge in scenarios with re-execution and Verifiers.

Tables 2.3 and 2.4 summarize verification scheme approaches for specific functions, their use cases, and their trade-offs.

Table 2.3.: Verification scheme approaches for different function types

Function type	Use Case	Approach	Advantage	Disadvantage
Pre-Image Computation	Scientific function evaluations	Ringers [24] [25]	Efficient verification	Computational overhead for Outsourcer to calculate bogus ringers
		Magic Numbers [26]	Efficient verification	Approach assumes honest Outsourcer
Matrix Multiplication	Fundamental operation for image processing and Neural Networks	Freivald's Algorithm [28]	More efficient than re-execution	Requires result matrix as input - promotes lazy behavior
		Secret sharing [29]	More efficient than Homomorphic Encryption	High computational overhead
		Permutation [30] [31]	More efficient than Homomorphic Encryption	Not practical
		Matrix Transformation [32]	Input and output privacy, low computational overhead	-
Matrix Inversion	Scientific and Engineering problems	Freivald's Algorithm [28]	More efficient than re-execution	Requires result matrix as input - promotes lazy behavior
		Secret sharing and monte-carlo verificaiton [33]	More efficient than Homomorphic Encryption	High computational overhead
Linear Equations	Analyzing and Optimizing real-world systems	Homomorphic Encryption [34]	Input and output privacy	Not practical

Table 2.4.: Verification scheme approaches for different function types

Function type	Use Case	Approach	Advantage	Disadvantage
Polynomials	Fundamental functions in engineering and scientific problems	Arithmetic Circuits [35] [36]	More efficient than Homomorphic Encryption	Moderate computational overhead
Regression Analysis	Data analysis	Input transformation [37]	More efficient than Homomorphic Encryption	-
Convolutional Neural Network Inference	Classification tasks, Natural language processing	Partial Verification [38]	More efficient than re-execution	Part of the computation remains unverified
		zk-SNARK [39] [40]	Input, output, and model privacy	Not practical
		Matrix and weights transformation [41]	More efficient than zk-SNARKs	Transforming complex models can be difficult
		Arithmetic Circuits and random challenges [42]	Efficient verification	Only works with certain CNNs
		Homomorphic Encryption [43]	Input, output, and model privacy	Not practical in real-time

2.3. Verification of Convolutional Neural Network Inference

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

As discussed in the previous sections, we evaluated five verification approaches for verifying CNN inference and four approaches for verifying arbitrary functions. As software virtualization and trusted execution environments have still too many security problems [19] [22], we do not consider them as a secure choice for verifying neural network inference. Fully Homomorphic encryption [17] [18], zk-SNARKs [39] [40], and also AlexNet [43] 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. However, transforming an existing CNN into an arithmetic circuit is difficult from a methodological point of view and comes with additional computational overhead. Transforming the input matrix and weights before sending them to a remote server is a promising approach for preserving input, output, and model privacy. Still, 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. Hence, this technique is 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, they may include a third-party Verifier. The possible optimization of only verifying parts of the CNN inference matrix multiplications is not advised due to the risk that other parts of the functions are not executed properly.

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

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

2.4. Verification of outsourced Computation using Re-execution

Many 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. For example, the Verifier and the Contractor might agree on an incorrect response and save computational

resources compared to performing the actual computation. Likewise, there have to be measurements so that the Outsourcer cannot reject payment or falsely accuse Verifier or Contractor of cheating. This section analyzes different verification schemes that use re-execution or similar approaches. Also, it gives an overview of assumptions and measurements these schemes introduce to deal with dishonest behavior or unintended protocol violations such as timeouts.

The roles and interactions between participants during outsourcing of computation has analogies to the principal-agent problem [46]. The Outsourcer can be viewed as the principal 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. The Contractor might be "lazy but honest" [47], lazy and dishonest or malicious to save resources or harm the Outsourcer.
4. The Outsourcer might be dishonest to avoid payment after correct computation.

In chapter 3, we explain the principal-agent analogy and its implications in more detail.

Verification schemes

Uncheatable grid computing [48] This verification scheme introduces commitment-based sampling. The Contractor commits to a Merkle Tree root hash every few intervals and sends it to the Outsourcer. The Outsourcer then randomly selects a few samples to re-compute them and sends a proof of membership challenge to the Contractor. Only if the verified output matches the initial response and only if the proof of membership challenge is successful, the contract is continued. Otherwise, the Contractor is convicted of cheating.

This scheme's primary advantage 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.

This scheme's disadvantage is that it does not feature an ecosystem perspective and, therefore, does not deal with dishonest Outsourcers. Also, 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 the Outsourcer might need a third-party Verifier to perform the re-execution. Other possible contract violations than dishonest behavior such as timeouts or low response rates are not addressed. 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 an incorrect response when detected due to the lack of digital signatures.

2. The Outsourcer can reject payments due to the lack of digital 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. Hence, cheating is 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 [49] This verification scheme uses sampling-based re-execution to verify the results of outsourced computations that a cloud server sends back to the Outsourcer. Within specified intervals, the cloud server commits to signed Merkle roots generated over its results when returning them to the Outsourcer. The cloud user can instruct a Verifier to verify these results by checking if the signed Merkle root's signature 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.

This scheme's main advantage is that no third party is required during the verification process except the Verifier, Outsourcer, and the Contractor. It uses digital signatures 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. It uses sampling-based re-execution to verify results with low computational overhead and high probability.

The major disadvantage of this scheme is that it assumes Verifier and Outsourcer, to be honest. Hence, there are 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 confirm that Contractor responses 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 the payment.

These security problems lead to the conclusion that this verification scheme only works if the Verifier and the Outsourcer are 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 [9] This scheme uses commitment based sampling and assumes that outsourcing jobs are exclusively pre-image computations. Commitment-based sampling refers to committing 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. The Contractor can redeem the deposit if it meets the contract's conditions. Once the Contractor sends results to the Outsourcer that satisfies the challenge, the Contractor gets automatically paid by a 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 crucial communication and automatically enforce payments.

There are, however, also significant disadvantages for most use cases. One major drawback of this scheme is that it exploits trivial verification properties of pre-image computation, making it unusable for other more frequent use cases. Also, the scheme assumes that a trusted third party is 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. Also, it does not contain a dishonest Outsourcer from claiming that it received an invalid computation result.

Integrity verification of cloud-hosted data analytics computations [50] This verification scheme uses artificial data values to enable verification even for a computationally weak Outsourcer. It is mainly targeted at use cases with a large amount of data, such as machine learning or deep learning tasks. This verification scheme's fundamental idea 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 specific position in an image and verify if an object detection response sent by the Contractor detects the artificially added object successfully.

This method's main advantage is that a weak Outsourcer does not need a third-party Verifier for it. Also, the computational overhead of modifying input data is low.

However, there are some disadvantages 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 instead discarded. A Contractor might also detect an irregular change of two consecutive images when an object is added artificially and react with honest computation accordingly. The paper addresses this issue but introduces the additional computational overhead of transforming input data. It should finally be noted that even if an object is added artificially, there might be a certain probability that the CNN performing object detection cannot detect it even if it behaves honestly. Thus, this technique has to be used probabilistically and tolerate

missing detections to a certain extent. One might consider the alternative of not placing the artificial data randomly but falling back on a set of precomputed inputs to ensure detection with 100% probability. However, we do not advise this strategy as a Contractor might learn this limited set of precomputed inputs over time.

If a verification scheme addresses these concerns, adding artificial data values can be a promising technique to replace re-execution in certain use cases due to higher efficiency. However, it should be noted that this technique still has to be supported by other measures that prevent dishonest behavior by the Outsourcer and the 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 [51] This paper modifies an existing verification scheme with re-execution and Verifiers by adjusting game-theoretic incentives and preventing 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 randomly selecting a Contractor and a Verifier and assigning 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.

This scheme's advantage is that it successfully prevents collusion of the Verifier and the Contractor if both parties are rational, expected pay-off maximizing entities. Also, it is the only scheme that we analyzed that addresses Sybil attacks.

This scheme's disadvantage is that preventing Sybil attacks comes with the high overhead of a third party and Ethereum transactions. It also assumes that the Outsourcer, judge, and arbiter are honest, which may not be a valid assumption.

Anticheetah: Trustworthy computing in an outsourced (cheating) environment [52] This paper proposes a system called Anticheetah, which outsources a computation to multiple nodes in a multi-round approach. Different inputs are sent to each node in every round 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, nodes that have the best reputation balance are selected first. Occasionally, a node might get its reputation of reliability decreased despite acting honestly. However, this scenario does not have a significant impact on the overall node balance due to a multi-round approach. The scheme assumes guaranteed message delivery, no lost messages, and zero communication overhead.

This scheme's advantage is that it disincentivizes collusion between multiple nodes due

to its multi-round reputation system. Also, through a reputation system that includes cost-efficiency and timeliness, only nodes that can provide a high quality of service remain as Contractors over time.

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. This scheme's assumptions, such as no lost messages and zero communication overhead, are not realistic for scenarios with a weak connection.

Efficient fair conditional payments for outsourcing computations [47] This verification scheme uses a semi-trusted third party to solve disputes between participants. It uses true and bogus ringers that we 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. Digital signatures over key communication are used to let the third-party come to a rightful conclusion in a dispute.

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

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

Mechanisms for Outsourcing Computation via a Decentralized Market [23] 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 are 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 so that collusion is made unlikely when dealing with rational pay-off maximizing participants.

This scheme's main disadvantage 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 [53] This verification scheme proposes that an Outsourcer sends its 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. This scheme's general procedure consists of outsourcing the same job to $n = 2$ Contractors and accepting the result if the 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 a dominant strategy. Hence, it prevents rationally expected pay-off maximizers from colluding.

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 an agreement is reached. The minimum total computation complexity of outsourcing the original job is 200%. Additionally, the overhead of the system 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 quickly become the network or performance bottleneck of the system. Also, the Contractors perform complete re-execution. This decision 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 honestly. Only a few examples such as [53] also protect against dishonest behavior from the Outsourcer. The following approaches are used by these 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 [53].

Digital Signatures Digital Signatures are a secure way to keep records of communication that cannot be tampered with. Suppose an input and related job information is signed by the Outsourcer before sent to the Contractor. In that case, the Contractor can prove having received this data from the Outsourcer to any third party. It is unable to generate a valid signature of the Outsourcer over data by itself. A few schemes such as [9] 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 [49] use Contractor signatures to commit to responses when sending them. This way, if an incorrect 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 [52]. 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 a near-optimal matching of honest Contractors and a trusted Outsourcer can be achieved after a few rounds. However, it requires a large availability of Contractors.

Commitment to messages using Merkle Trees Merkle Trees can be used as a data structure to verify if data is contained in a large collection efficiently. The root hash of a Merkle tree can be used to verify if data is included in the whole tree with $log_2(n)$ steps. Verification schemes such as [51] 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 cannot 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 only to sign Merkle root hashes and challenges in a specified interval, thus improving efficiency. The disadvantage of committing 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 a dispute if the verification scheme does not consider this risk.

Use of trusted third parties and Blockchains for records and conflict resolving Some Verification schemes such as [9] use trusted third parties (TTPs) or Blockchains to record participants' communication. In a dispute, any participant can review the record 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 critical communication on a Blockchain or with the TTP's help 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. Therefore multiple publications acknowledge that while a Blockchain has the potential to assist P2P computation without the need of a single TTP, there are challenges that remain [54] [55] [56]. However, primarily as a secure payment solution, a Blockchain is used by multiple proposed outsourcing systems [57]

Reputation System Verification schemes such as [52] use a reputation system 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 incentivize participants to behave honestly and provide a good quality of service at almost no computational overhead. While in public networks, a node can avoid a negative reputation by merely creating a new identity, it cannot fake a positive reputation balance. Hence, the reputation system may still lead to monetary benefits for participants that adhere to the desired protocol.

Blacklisting Blacklisting is a simple technique used by verification systems such as [52] 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. These measurements can respectively disincentivize or prevent participants that got blacklisted to resume relationships by generating a new identity. Blacklisting can be used to punish quality of service violations or dishonest behavior.

Table 2.5 shows an evaluation of the different approaches used by verification schemes proposed by current academic literature and lists references for each approach.

Table 2.5.: Techniques utilized by different verification schemes

Approach	Advantage	Disadvantage
Complete/Sampling-based re-execution [9] [48] [49] [50] [51] [53]	Efficient verification approach to prevent dishonest behavior, works for arbitrary functions	More efficient verification algorithms are available for certain function types
Digital Signatures [9] [47] [49] [51] [53]	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 [52]	Provides a near-optimal performance and reliability matching over time	Requires a large number of available Contractors
Commitment to messages via Merkle Trees [9] [48] [49] [51]	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 [9] [23] [47] [51] [53]	Can be used to securely store records and handle conflicts without bias	Requires an available TTP, or accounts for high comutational complexity when using a PoW Blockchain
Reputation system [52]	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 [52]	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

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 violations 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 overview 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 illustrates this scenario and its participants.

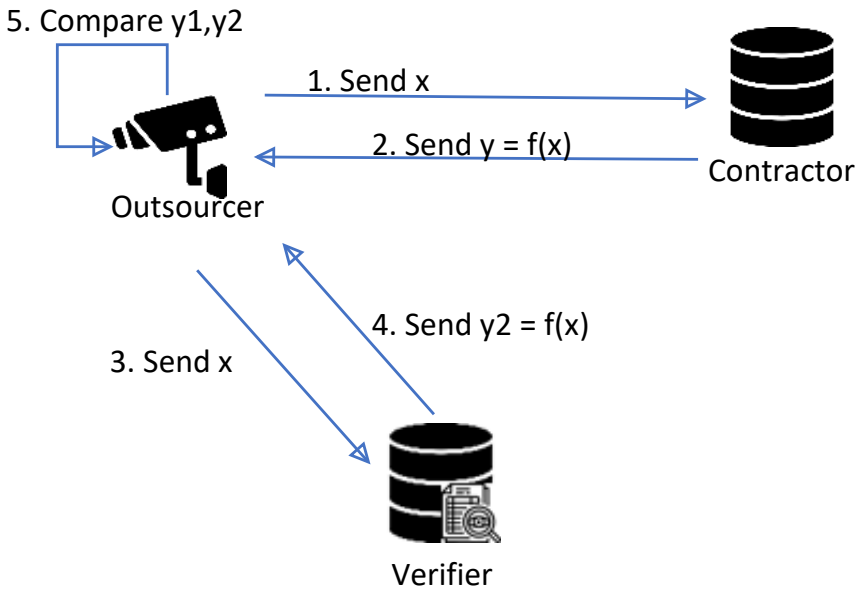

Figure 2.1.: Assumed scenario

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

Number 1: Contractor sends back false responses to save resources In this scenario, the Contractor sends back incorrect responses to the Outsourcer. The wrong 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 [53]. Figure 2.2 illustrates this dishonest behavior.

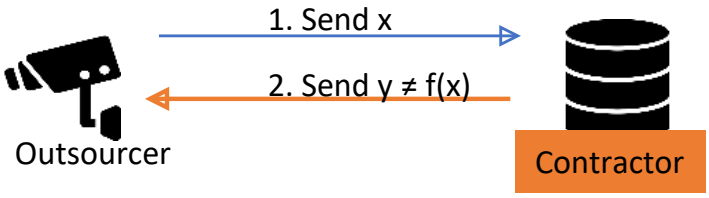

Figure 2.2.: Number 1: Contractor sends back false responses to save resources

Number 2: 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 [53] outsource the function and inputs of two unequal responses of the Contractor and the Verifier to additional Verifiers to detect which party has cheated. Figure 2.3 illustrates this dishonest behavior.

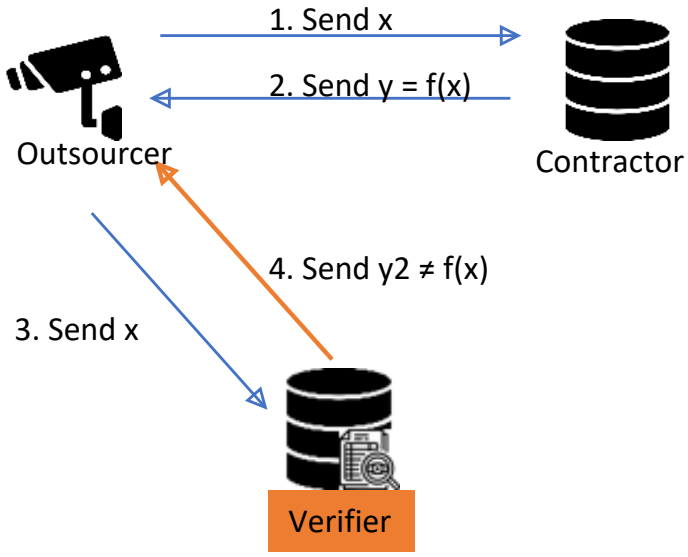

Figure 2.3.: Number 2: Verifier sends back false responses to save resources

Number 3: Outsourcer sends different input to Contractor and Verifier to refuse payment An Outsourcer has multiple ways to reject payments. For instance, it may send two different inputs to the Verifier and the Contractor, which leads to unequal responses even if both participants are behaving honestly. Suppose communication of the Outsourcer is not recorded. In that case, the Outsourcer can exploit this procedure to reject payment while claiming that it sent identical inputs to the Contractor and Verifier. While schemes such as [49] 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, [53], 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. Figure 2.4 illustrates this dishonest behavior.

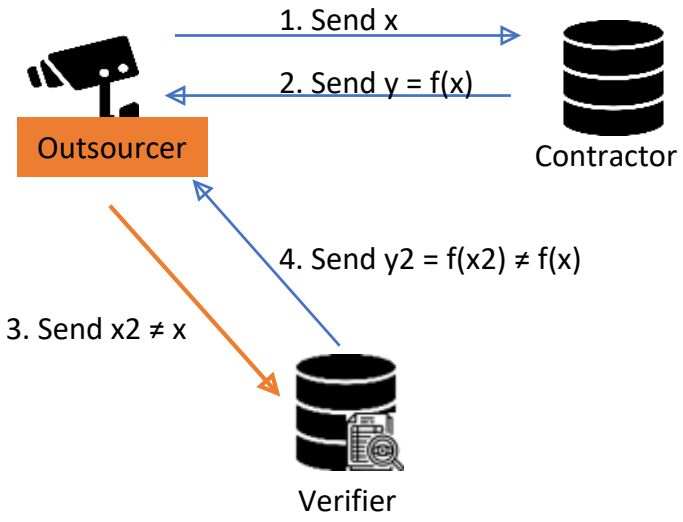

Figure 2.4.: Number 3: Outsourcer sends different input to Contractor and Verifier to refuse payment

Number 4: Contractor or Verifier tries to avoid global penalties when convicted of cheating or QoS violations Even when the Verifier or the Contractor is detected cheating by the 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 with a third party Arbitrator trying to solve the conflict. Suppose communication is not recorded in a tamperproof manner. In that case, it can claim to an 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 [51] can prevent this behavior. Figure 2.5 illustrates this dishonest behavior.

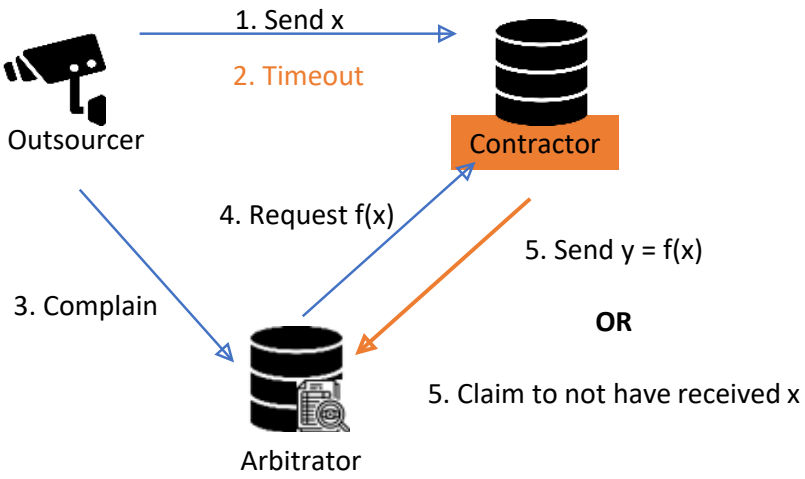

Figure 2.5.: Number 4: Contractor or Verifier tries to avoid global penalties when convicted of cheating or QoS violations

Number 5: 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 these cannot be enforced. This problem can be solved intuitively by waving fines and only using real-time payment. However, this solution leads to a large amount of micro-transaction, large transaction fees, and latency bottlenecks of payment providers. Also, fines can be substantial disincentives to prevent dishonest behavior. Verification schemes such as [9] solve this problem by utilizing a TTP or Blockchain that supports deposits and payment on another entity's behalf. Figure 2.6 illustrates this dishonest behavior.

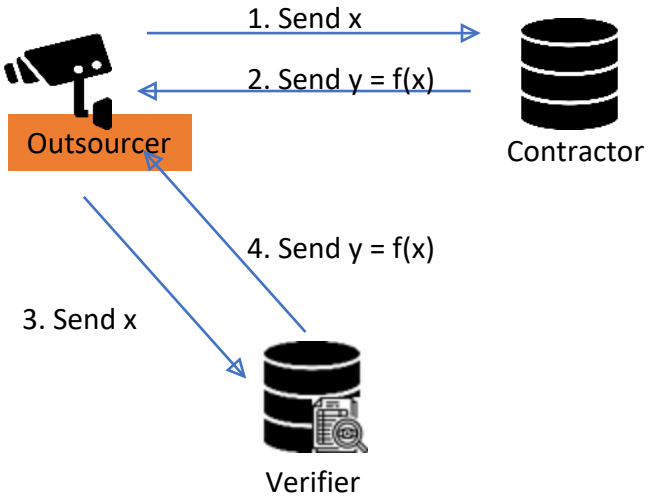

Figure 2.6.: Number 5: Outsourcer refuses to pay even if obliged to by the protocol

Dishonest behavior via collusion Collusion refers to the secret, dishonest collaboration of at least two participants. As Contractor and Outsourcer have fundamentally different interests, collusion is only plausible for (1) either Contractor and Verifier to trick the Outsourcer into believing a false result, or (2) Outsourcer and Verifier to reject payment to the Contractor. The following scenarios can happen if two parties collude:

Number 6: Outsourcer and Verifier collude to refuse payment and save resources Even if the Outsourcer inputs 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 an incorrect response from the Verifier, claiming it was the real one. The Outsourcer's incentive to act dishonestly with this behavior is to reject payment to an honest Contractor. The Verifier's incentive is to save computational resources by returning an incorrect response and not conducting the assigned function. Potentially, the Verifier could also be rewarded by the Outsourcer with a proportion of the saved Contractor reward. We found no evidence that any of the evaluated verification schemes prevents this scenario. Even in schemes such as [53] that outsource unmatching responses to additional Verifiers, it seems that (1) the protocol does not enforce this decision and (2) there appears to be no random assignment of Verifiers. This 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. Figure 2.7 illustrates this dishonest behavior.

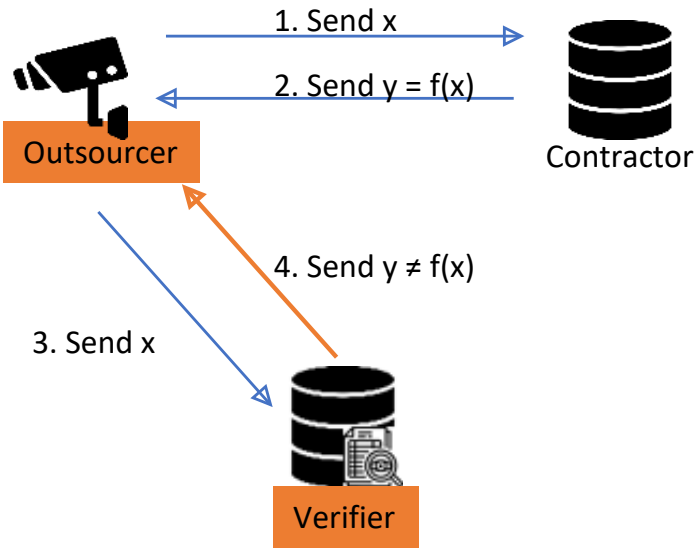

Figure 2.7.: Number 6: Outsourcer and Verifier collude to refuse payment and save resources

Number 7: Contractor and Verifier collude to save resources The most significant 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 incorrect

responses. The Contractor and Verifier both save resources by using a computationally inexpensive algorithm rather than the function assigned by the Outsourcer. Schemes such as [53] lower the possibility of collusion by designing their incentive structure so that being honest is 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 know each other's identities. The following figure illustrates the dishonest behavior. Figure 2.8 illustrates this dishonest behavior.

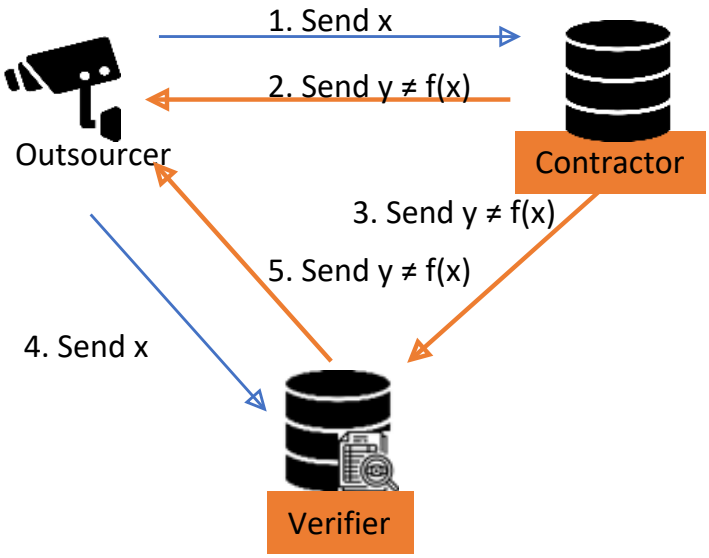

Figure 2.8.: Number 7: Contractor and Verifier collude to save resources

Quality of Service Violations Quality of Service (QoS) 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 connection 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 or to save resources. The problem is that it is hard to tell which of the three cases is true if QoS violations occur. 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.

Number 8: 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 [52] keep local reputation systems that frequently compare all Contractors in their performance. This system 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

timestamping server such as in [53]. These techniques come with additional computational or communicational overhead but can create a globally usable record of when inputs are sent and responses are received. Figure 2.9 illustrates a QoS violation.

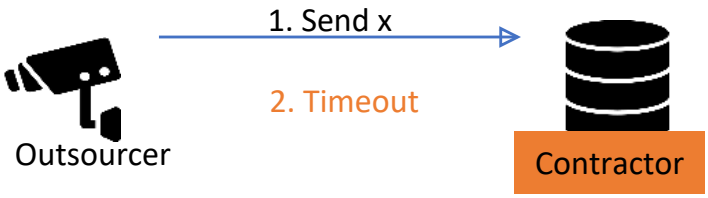

Figure 2.9.: Number 8: Contractor times out

Number 9: Low Response Rates Low response rates can happen due to weak connection or if a participant cannot computationally process a function in the expected time. The same techniques to prevent timeouts can also be used to prevent this behavior.

Number 10: High Response Time High response times can happen due to network latency or if a participant cannot computationally process a function in the expected time. The same techniques to prevent timeouts can also be used to prevent this behavior.

External Threat: Number 11: 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 [47] 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. However, it should be noted that using Merkle trees and signatures lead 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. However, the tampering will be detected once the Merkle root does not match the received inputs. Figure 2.10 illustrates this external threat.

In summary, we identified multiple protocol violations and summed them up by 11 descriptions and four types. It should be noted that while nine out of the identified 11 protocol violations are 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 provides the following attributes:

1. Prevents all of the nine protocol violations that already have been addressed by existing solutions.

2. Proposes solutions for the two unaddressed protocol violations.

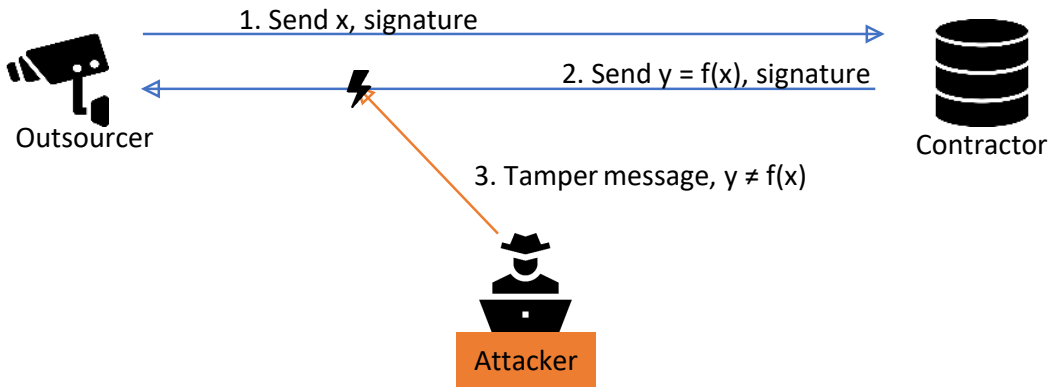

Figure 2.10.: Number 11: Message tampering by an external attacker

3. Minimizes the use of TTPs, Blockchains, and computationally expensive techniques to remain practicable and scalable.

Concluding this chapter, we firstly identified a verification scheme as a key component of an edge computing marketplace. We then evaluated verification approaches for different function types. We focused especially on verification approaches that work for arbitrary functions and CNN inference. We concluded that for both function types, sampling-based re-execution provides the best trade-off between security and performance. To evaluate the security of verification schemes based on re-execution we compiled a list containing 11 possible protocol violations that threaten the integrity or quality of outsourced computation. Our list of protocol violations aims to serve as a comprehensive threat model for any verification scheme based on re-execution used in a computation marketplace with untrusted Outsourcers, Verifiers, and Contractors. Verification schemes proposed by the current academic literature based on sampling-based re-execution provide different techniques, which combined address 9 out of 11 threats in our list of protocol violations.

Table 2.6 provides an overview of the described potential protocol violations and by which techniques the evaluated academic literature addresses them. Figure 2.11 gives a final overview of our literature review. The next chapter introduces our designed verification scheme that aims to prevent all 11 identified protocol violations while remaining efficient, scalable.

Table 2.6.: 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 response 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 to conduct payment on behalf of 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

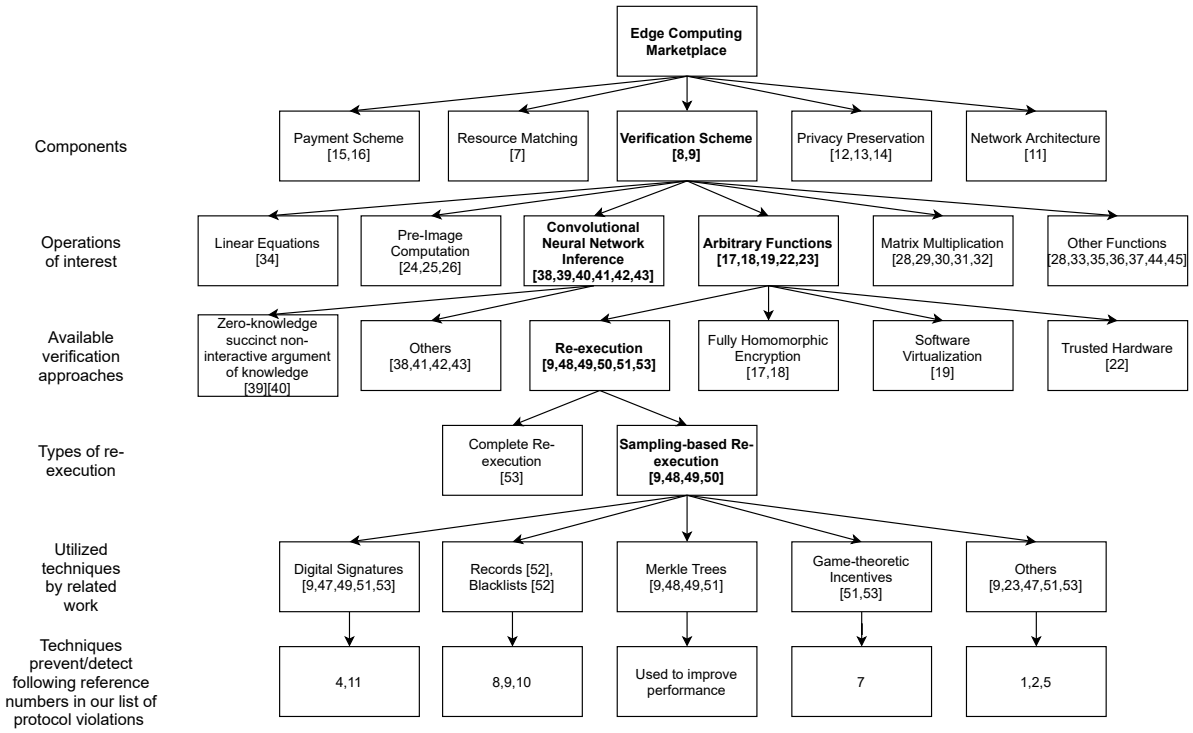

Figure 2.11.: Final overview: Literature review

3. Design of our Verification Scheme

This section will introduce the design of our 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. Our verification scheme prevents all of the 9 identified protocol violations that already have been addressed by the evaluated solutions
2. Our verification scheme proposes solutions for the 2 unaddressed identified protocol violations
3. Our verification scheme 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. While the designed verification scheme is especially suitable for latency-sensitive edge computing marketplaces with scarce resources, it can also be used for cloud computing marketplaces with unverified cloud service providers. A computationally powerful Outsourcer can perform sampling-based re-execution itself instead of relying on a third-party Verifier. This decision does not hurt our scheme's security.

3.1. Overview

Our designed verification scheme begins at the point where an Outsourcer has successfully hired an available Contractor for a specific reward per input. Both participants start 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 $r \mod n$ is picked. To prevent planned collusion, neither Outsourcer nor Contractor can decide on their own which Verifier will be selected. The Contractor is also not informed about the Verifier's index, so it cannot contact it for an ad-hoc collusion attempt. Both Verifier and Contractor generate an identical contract. After that process, our verification scheme checks if all specified fines, rewards, and bounties in both the Outsource-Contract and the Verify-Contract lead to a dominant strategy of being honest from a game-theoretic perspective. 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, 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 lasts until one party terminates 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 new input's signature, performs the assigned function, and digitally signs each response before sending it to the Outsourcer.

Whenever the Outsourcer receives a response from both Verifier and Contractor belonging to the same input, it compares if they are equal. It aborts the contract due to a protocol violation if they are not. Digital signatures are always forged over the input itself, its respective index, and the contract hash to ensure a distinct record for each input. Additionally, the Outsourcer digitally signs with each new message it sends 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 payment to that party. If the acknowledged output-counter does not meet the expected acknowledge rate set by Verifier or Contractor, the respective participant might cancel the contract due to a QoS violation. 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 terminated according to customs, (2) a contract was aborted due to a QoE violation, (3) a contract was aborted due to dishonest behavior. In the first, 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 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 Contractor's response, 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 must pay a fine for cheating. It also needs to pay a fee to all additional Verifiers it 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 in the ecosystem respond with the correct response.

3.2. Initial Situation

In the initial situation, we assume an Outsourcer and a Contractor have agreed to a contract. The contract contains a unique ID that distinguishes it from previous contacts with the same participant and the participants' public keys registered at the payment settlement entity. Also, a reward per processed input and the function/model to be used is specified. Additionally, the Contractor and Outsourcer may agree on fines, deposits, and bounties if a participant is caught cheating to increase the protocol's robustness. Also, we assume multiple available verifiers are available and willing to agree on a contract with the Outsourcer to process random sample inputs. Figure 3.1 illustrates the initial situation.

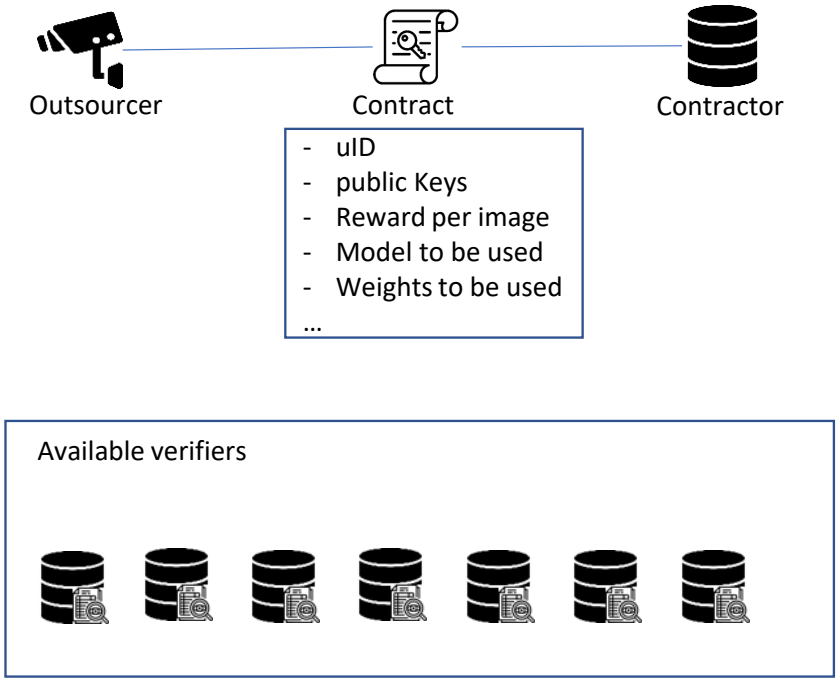

Figure 3.1.: Initial situation

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, the Contractor and Outsourcer engage in a protocol we designed, called Randomization. The result of the protocol is the selection of a random Verifier that cannot be chosen initially by the Outsourcer nor the Contractor.

   We describe ad-hoc collusion as the scenario where a Contractor and a Verifier do not initially know each other but still try to communicate and collude. They may expect individual benefits from collusion, such as spending less computational resources by submitting an incorrect response. This problem also explains our second and third objectives. Thus, we designed Randomization so 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 to the Outsourcer. The Outsourcer proceeds if it approves the Verifier list and the signature sent by the Contractor. By signing the initially sent hash of the Outsourcer, the Contractor commits to have accepted a random number by the Outsourcer without knowing the number.

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 is 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.

Figure 3.2 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,

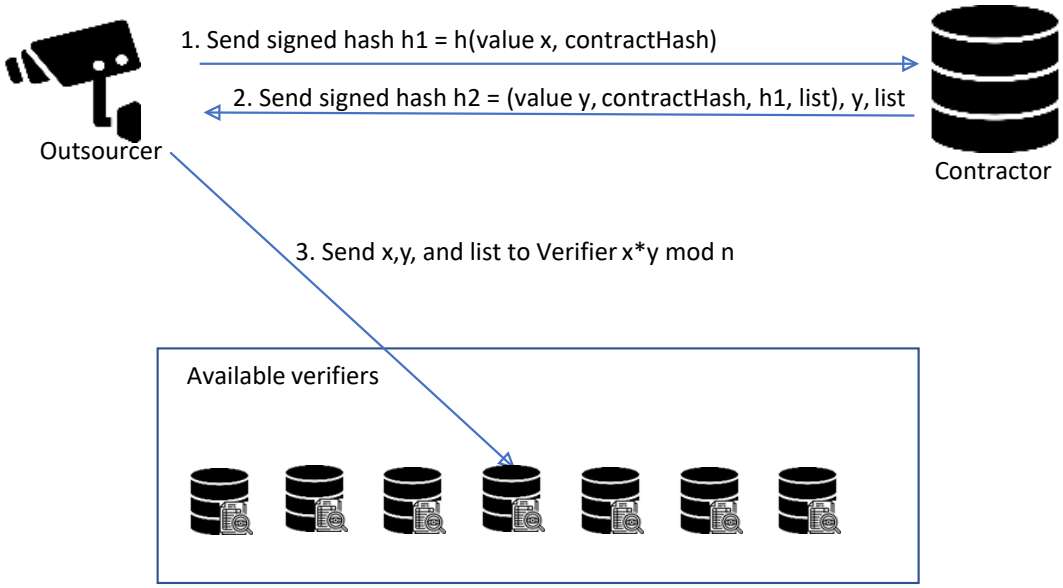

Figure 3.2.: Randomization overview

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 Contractor's number to pick a Verifier by its choice.

The protocol does not add any noticeable overhead to the system. 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 committed 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. Records of the signed random numbers associated with the specific contract are available.

3.3.2. Game-theoretic Incentives

The Outsourcer and the Contractor's role is similar to the widely known principal-agent model [46]. In this model, the principal hires an agent to perform a service but can by itself not completely assess the agent. This analogy is also used by related work to design game-theoretic incentives for the Contractor to act honestly, such as [58] or [59]. In general, the incentive design to verify the agent's job in a principal-agent-model mainly depends on the given maximum punishment and whether collusion is possible or not [60].

Game theory is used to understand the situation in which decision-makers interact [61]. From a game-theoretic perspective, the Outsourcer, the Contractor, and the Verifier are players that engage in a strategic game. In a strategic game, each player has a set of actions, different preferences over the set of action profiles [61]. Following Nash's analysis of strategic games, each player intends to maximize its expected payoff with respect to the actions it can choose. Nash proved that each game with these criteria has an equilibrium in mixed strategies [62]. A Nash equilibrium is a set a stable set of strategies, one for each player, where nobody has a unilateral incentive to deviate from their current strategy [63]. Thus, a rational player who knows all other players' action sets never picks anything but the best response to those actions.

If a player's strategy is at least as good of a response to all other actions of all other players than all alternatives, it is called a dominant strategy. If it is better than each other alternative, it is called a strictly dominant strategy [64].

These fundamentals can be used to describe our scenario from a game-theoretic perspective. After Randomization, there are two players in our game: The Contractor and the Verifier. Both have two strategies. They can either return a correct result or return a false result. Computing a false result may save computational resources but still lead to a reward if it does not get detected by the other party. We want to set the incentives of both participants so that returning the correct result is a strictly dominant strategy for both players.

After Randomization, both the Verifier and the Contractor have received a contract identical to one of the respective contracts of the Outsourcer. Even if 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 coordinates have to be estimated and returned. Suppose there exists a most efficient q-algorithm. In that case, 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.

By designing intuitive incentives, the payoff for each the Contractor and the Verifier can be described as follows:

1. If the Contractor returns a correct result honestly, it receives a reward r from the Outsourcer with 100% certainty (due to Contestation). It has to spend computational cost c_h for calculating an honest response. Thus, its expected payoff for choosing the honest strategy is $r - c_h$ independent of the Verifier's strategy.

2. If the Contractor returns a probabilistic or any other dishonest result, it only receives reward r from the Outsourcer if (1) either the result is correct with a probability of q, or (2) if the Verifier also acts dishonestly and returns the same result. Using a q-algorithm is only rational if it takes less computational resources than performing the assigned computation as it comes with the risk of being detected. Thus, the cost of the q-algorithm c_d is lower than c_h. If the Verifier acts dishonestly, the expected reward for the Contractor is $r - c_d$.

For simplicity, we assume that even if the Verifier and the Contractor do not know each other, they still use the same q-algorithm. In use-cases such as object detection, this is likely because the best naive q-algorithm returns a response without detections. This response has the advantage that it is a valid response for various use cases, and no bounding boxes have to be guessed. By making this assumption, we set even stricter requirements for our scheme.

3. If the Verifier acts honestly, however, and the Contractor's q-algorithm returns an incorrect result, the Contractor has to pay a fine f to the Outsourcer. Thus, its expected payoff depends on the probability q of calculating a correct result with its q-algorithm. Thus the Contractor's expected return is $q * r - (1 - q) * f - c_d$.

4. The Verifier has identical expected payoffs as the Contractor.

We can use these insights to design a payoff matrix that contains all strategies and their expected payoffs for each participant. Figure 3.3 illustrates the payoff matrix for the Contractor. Figure 3.4 illustrates the payoff matrix for the Verifier. The resulting payoff matrices are identical to the two-rational-Contractors-game introduced by [65].

Verifier / Contractor	Honest	Dishonest/Lazy
Honest	$r - c_h$	$r - c_h$
Dishonest/Lazy	$r \times q - f \times (1-q) - c_d$	$r - c_d$

Figure 3.3.: Payoff Matrix of the Contractor with intuitively designed incentives

Contractor / Verifier	Honest	Dishonest/Lazy
Honest	$r - c_h$	$r - c_h$
Dishonest/Lazy	$r \times q - f \times (1-q) - c_d$	$r - c_d$

Figure 3.4.: Payoff Matrix of the Verifier with intuitively designed incentives

In the resulting game, there are two nash equilibria. One nash equilibrium exists when both players act honestly. From this state, if an individual player decides to change its strategy to act dishonestly, its expected payoff changes from $r - c_h$ to $r * q - f * (1 - q) - c_d$. For most applications, due to the lack of a reliable q-algorithm, this results in a lower expected payoff than acting honestly.

However, there also exists a nash equilibrium when both players act dishonestly. In this case, both players receive a payoff of $r - c_d$. If a player changes its strategy to act honestly, it receives only $r - c_h$. As described earlier, one of the properties of a q-algorithm is that it is always cheaper than performing the assigned calculation. Thus, due to $c_d < c_h$, changing its strategy results in a lower expected payoff than acting dishonestly.

It is crucial to design incentives that eliminate the Nash equilibrium of both players acting dishonestly. These incentives transform being honest into a strictly dominant strategy. As

described in chapter 2, there is a solution to this problem. Verification schemes such as [53] and [65] have identified a relationship of incentives with adding fees for cheating players and bounties for dishonest players such that being honest is a dominant strategy from a game-theoretic point of view. When designing incentives correctly, using 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. Figure 3.7 illustrates both contract pairs with a bounty, reward, and fine. The payoff matrix with the use of a bounty are illustrated in figures 3.5 and 3.6.

Verifier / Contractor	Diligent	Dishonest/Lazy
Honest	$r - c_h$	$r - c_h + b$
Dishonest/Lazy	$r \times q - (f+b) \times (1-q) - c_d$	$r - c_d$

Figure 3.5.: Payoff Matrix of the Contractor with honesty-promoting incentives

Contractor / Verifier	Diligent	Dishonest/Lazy
Honest	$r - c_h$	$r - c_h + b$
Dishonest/Lazy	$r \times q - (f+b) \times (1-q) - c_d$	$r - c_d$

Figure 3.6.: Payoff Matrix of the Contractor with honesty-promoting incentives

There are multiple conditions to set the fine and the bounty, such that being honest becomes a strictly dominant strategy for both players. First, $b > c_h + c_b$, where b is the bounty c_h is the cost of performing the assigned function, and c_d is the cost of performing the q-algorithm. If c_h and c_b is unknown, b can be set to $2 * r$, which always satisfies the condition.

Second, $(f + b) * (1 - q) - c_d < r * q - c_h$ where f is the fine when caught cheating, and q is probability of the q-algorithm returning a correct result. If the fine and the bounty are high enough such that the expected payoff of the q-algorithm is negative, to act dishonestly is not rational. If both conditions are satisfied, ad-hoc collusion is not rational. In this case, the expected payoff of a participant to act honestly is the highest independent of the other player's strategy. Not all values in the formulas might be known before, such as the success rate q or the exact costs. Nevertheless, a moderate fine and bounty ratio compared to the reward, such as a 20:1 ratio, are already sufficient for most q-algorithms with $q < 0.95$.

Finally, we use a deposit to ensure that a cheating participant pays its fine in a real-world system. A part of the balance of each participant is locked for spending during the execution of the contract. Our implementation features a warning if incentives are not designed in line with the described best practices. Due to the unlikeliness of ad-hoc collusion with a random Verifier, we chose not to 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-measurable computational overhead.

Figure 3.7 illustrates the incentives-related agreements that can be set in the contracts of our verification scheme.

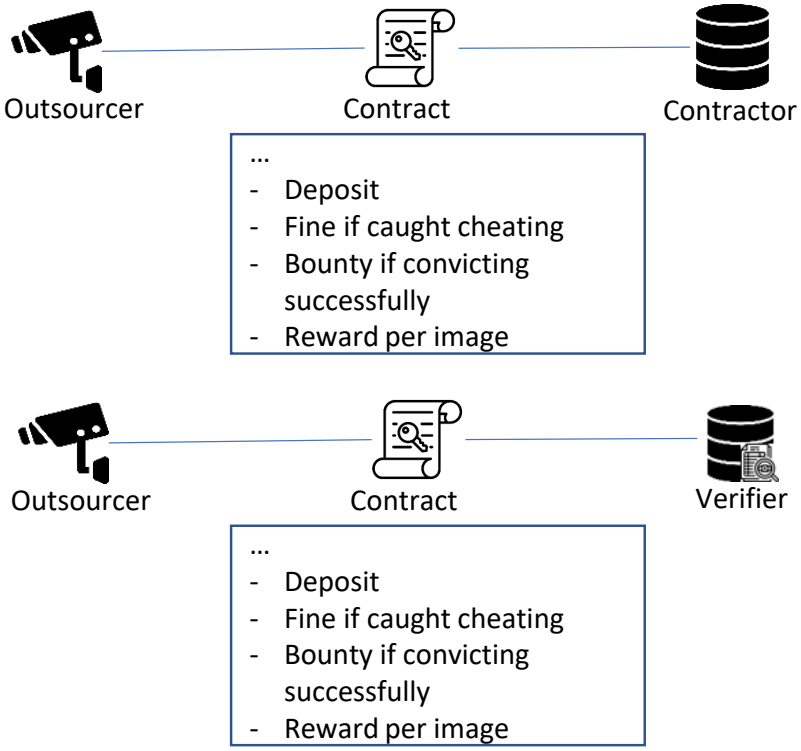

Figure 3.7.: Contracts with incentive related agreements

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 Contractor's response belonging to the same input. We call this process sampling-based re-execution. Sampling-based re-execution has a significant advantage over complete re-execution. Just with a few 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 drawdown.

The following example explains 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. Yet, the computational overhead of the verification is at least 100% compared to not performing any verification. Thus, to achieve 100% confidence, the Verifier needs to recompute all calculations.

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 Verifier. 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 $i >= 43.8$. Hence, even in a difficult and unlikely case, independent of the total number of inputs, the Outsourcer will catch a cheating Contractor with more than 99% confidence using only 44 samples. In our short example-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 rational, dishonest Contractors try to cheat in more than 10% of the cases in real-world scenarios. With a cheating rate of 0.1, they would already need to perform at least 90% of the required computational effort. Still, they 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 terminating 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, our example serves as a stress test. We suppose even less computational overhead in real-world scenarios.

Nevertheless, instead of setting a fixed number of samples, we recommend selecting 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 terminate 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 contract 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 up to 50 inputs locally that can still be verified after a contract ends after a surprisingly short time. This strategy ensures an even distribution of samples and a high-security guarantee even for 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 never to have received any responses from a Contractor or a Verifier. Thus it 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. A payment settlement entity can't solve a dispute and hold entities accountable without tamper-proof records.

Digital signatures can solve this problem. 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 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, suppose the Outsourcer signs how many responses it has received from a Contractor or a Verifier along with other contract-related information. In that case, this record can be used to redeem payment from a payment settlement entity on behalf of the Outsourcer's account.

Figure 3.8 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.

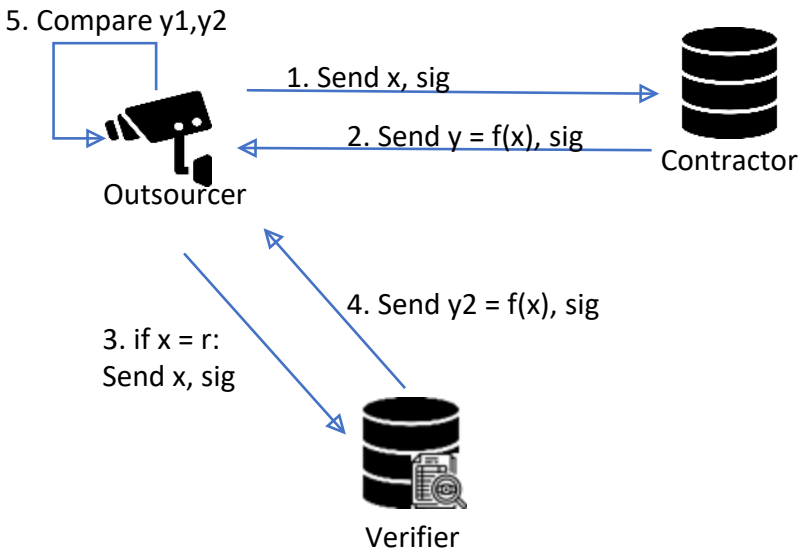

Figure 3.8.: Execution phase with signatures

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

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 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 sent along with the signature and the contract hash can be used to verify the signature (two participants hold an identical copy of the contract hash). If the signature is not valid, they abort the contract according to a QoS violation, 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. It may also leave a bad review to the participant and blacklist it.

Suppose a signed response returned by the Verifier and the Contractor over the same input index is found to be unequal. In that case, the Outsourcer consequently 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. Doing so, it sends 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 incorrect 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 participant sent a message, it was also received and processed by the other participant. Thus, the Outsourcer acknowledges each received response by sending a counter of acknowledged responses from that participant together with each input. It then 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 define a maximum loss rate or a number of consecutively unacknowledged outputs mustn't be revealed to the Contractor. Otherwise, the Contractor may exploit this parameter to 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. No matter if a contract is terminated 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, the 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 3.9.

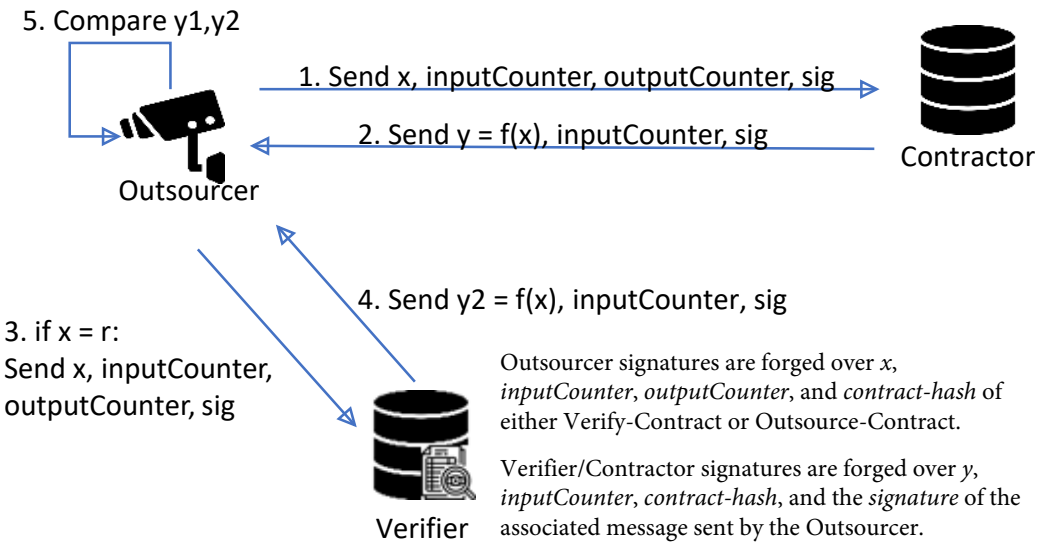

Figure 3.9.: Execution phase with signatures and acknowledged responses

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 fewer responses received also result in fewer rewards for those participants. Nevertheless, we chose not to share any internal parameters between participants that lead to QoS violations. Each participant can set these internal, disclosed parameters according to their demands. In combination with the consequences of aborting a contract due to bad QoS, this makes 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 violations, table 2.6), 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). A message tampered with by an attacker immediately leads to the receiving participant aborting the contract as the attacker cannot forge a valid signature of the tampered message. Also, by using digital signatures, our scheme detects a Contractor or a Verifier that tries to change its record or makes false claims when accused of cheating (number 4 in the list of possible protocol violations).

3.4.3. Improving Efficiency of the Execution Phase

One-time digital signature chains with redundancy codes

Generating a digital signature over each element in a digital stream can cause overhead to a system, especially when the signing device is computationally weak. There are multiple approaches available for signing digital streams efficiently. Approaches range from using

physical unclonable functions [66], aggregating signatures efficiently [67], generating only one signature for a compressed video file [68], or using chains of one-time digital signature algorithms [69].

Digital signature algorithms such as RSA, or ECDSA are secure for signing multiple messages with the same key. One-time digital signature algorithms such as the Lamport or Winternitz scheme instead are only secure when used a limited number of times [70]. In return, they provide faster key-generation and signing/verifying operations [71]. By applying a one-time signature with a new key for each input, a sender can sign a digital stream while, in theory, saving computational resources.

There are multiple challenges when using a new one-time digital signature for each input, however. First, the recipient of the signed message has to know if a received message signed by a new key is still coming from the same entity as before. For this reason, protocols such as [69] only publish the initial public key to the recipient. Every following public key is attached to the last signed message. Thus, a signature chain is generated. The recipient can trace back all new messages to the initial signature that it knows belongs to the sender.

However, this solution comes with additional challenges. A new key-pair has to be generated before signing each message. Thus, a sender can either decide to (1) generate one key-pair before signing each message, or (2) can generate multiple key-pairs in an offline phase to store for later use [72]. The first solution shrinks the advantages of a one-time signature scheme over a multi-use digital signature scheme because performance suffers from generating a new key-pair additionally to signing a message in each iteration. The second solution suffers from a significant storage overhead, especially when many inputs are sent. Also, when a sender runs out of key-pairs generated in the offline-phase, it has to fall back on the first solution.

Another challenge arises if the stream is sent over a lossy network. Since stream authentication relies on an intact signature chain, any loss of a message breaks the chain and, therefore, authentication. A solution that solves this problem with high probability is by using redundancy/combination codes [72]. Depending on the loss rate of the network, a redundancy rate is set. Afterward, according to the redundancy rate, a message may include a combination code. A combination code is bit-wise XOR of multiple inputs. Thus even if one input is lost, it can be recovered using the combination code. In our case, each message contains a unique signature that maintains the integrity of the signature chain. Therefore, it is crucial to receive or recover all signatures.

Figure 3.10 illustrates a 3-input XOR truth table. Let us assume a recipient did not receive message X with a value of 0. It got the messages $W = 1$, $Y = 1$ and $output = 0$ (combination code). 3-way XOR always returns the input that is least frequent among all inputs. Therefore, the recipient can recover $X = 0$ with 100% confidence assuming it received the other inputs and the combination code successfully. The problem with this technique is that it may fail when combination codes are not sent frequently enough, or multiple messages are lost after another (i.e. lost in burst). Thus, a sophisticated trade-off of redundancy and reliability has to be set by participants.

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 3.10.: Truth table of 3 input XOR

Figure 3.11 illustrates a hash chain that utilizes a 3-input XOR as described above. In this example, every third message serves as an input for two different combination codes. Thus, the network-bandwidth overhead in this example is nearly 50%. Therefore, it is only advisable to use this technique if the gained performance improvements provide a higher utility than the bandwidth overhead.

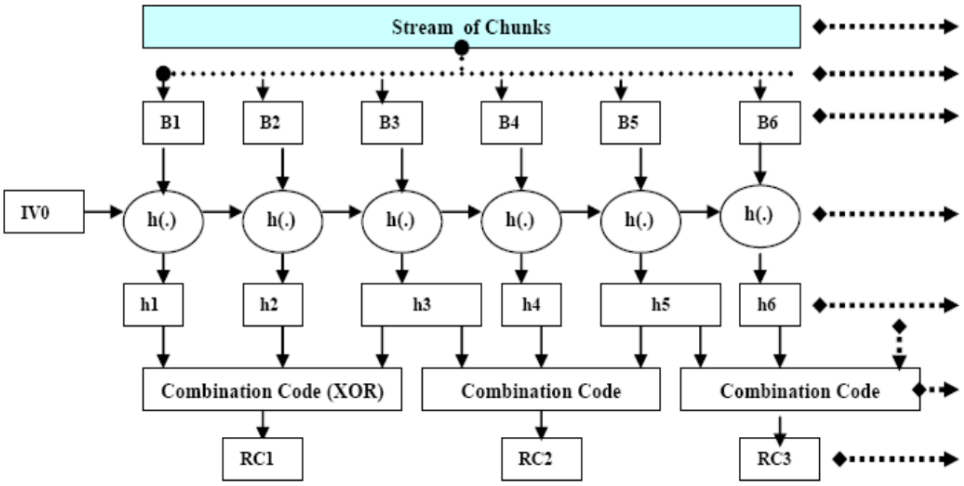

Figure 3.11.: Hash Chain of a digital stream with combination codes using XOR [72]

We conducted a Python benchmark to compare libraries of state-of-the-art multi-use digital signature schemes such as ECDSA, ED25519, or RSA with one-time digital signature schemes such as the Lamport and the Winternitz digital signature schemes. Despite the theoretical performance advantages of one-time signature schemes, implementations of the best performing multi-use digital signature schemes perform slightly better in both signing and verifying than all of the one-time digital signature schemes. This benchmark includes the NaCl ED25519 implementation that we utilize in our verification scheme's implementation. Thus, we do not recommend using one of the benchmarked one-time digital signature algorithms in any Python project. In addition to their inferior performance, they add all the described challenges described in this chapter to the implementation.

Committing to Multiple Messages with one Signature by Using Merkle Trees

Background: Merkle Trees and Proof of Membership Challenges A Merkle tree is a tree that contains the hashes of all its inputs as external-nodes [73]. Each internal node contains the hash of its children [74]. This way, the root of the tree contains a hash that was recursively built from each input.

A proof-of-membership challenge can be performed to verify if the Merkle tree contains a specific value. It requires to inspect the root hash and $log(n)$ hashes of internal nodes, where n is the number of total nodes in the tree. In a proof-membership-challenge, the hashes of each internal node that was recursively built on the input to be verified are returned to the entity seeking verification (prover). The prover calculates the hash of the specific value itself and recalculates each relevant internal node of the Merkle tree using the received hashes. If the calculated root hash matches the originally broadcasted root hash, the proof of membership challenge is successful.

Figure 3.12 illustrates a proof-of-membership challenge. The value to be challenged is H_k while the broadcasted Merkle root is $H_{ABCDEFGHIJKLMNOP}$ in the figure. The proofer requests all nodes marked blue to recalculate the internal nodes with a dotted blue outline. Thus, it only needs to request one hash per level or $log(n)$ nodes. With these nodes, it can verify whether recalculating each node with a dotted blue outline results in the same root hash as the one originally broadcasted.

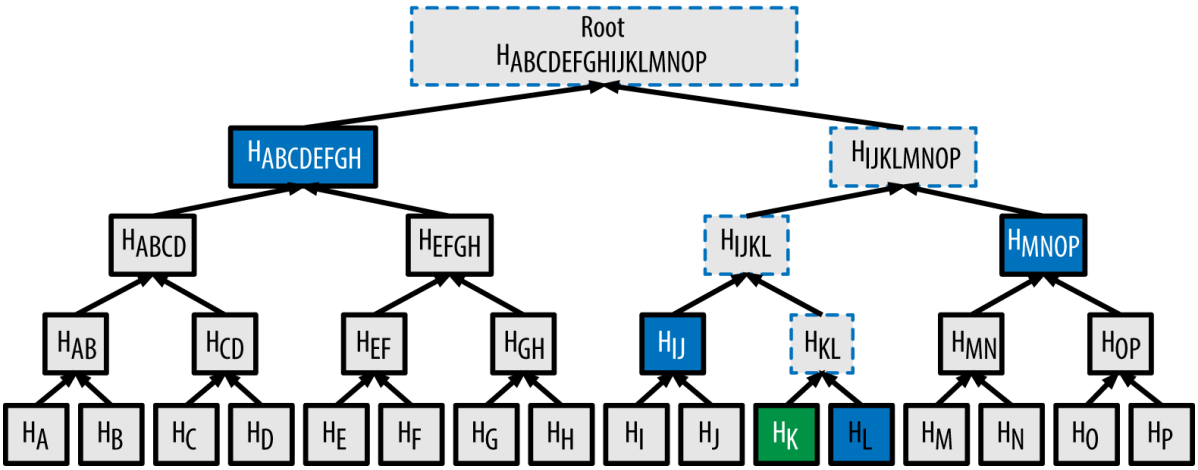

Figure 3.12.: Visualization of a Merkle Tree and a Proof-of-Membership challenge

Use of Merkle Trees for our Verification Scheme Multiple verification schemes proposed by academic literature such as [48] use this property to let the Outsourcer or the Contractor commit to multiple inputs via one signature by signing only the Merkle root hash and potentially returned challenges. Suppose a proof-of-membership challenge performed by another participant turns out to be unsuccessful. In that case, the signature can serve as a record that the participant that broadcasted the signature committed to those inputs. Therefore, a signed Merkle root hash built upon false inputs can be detected and used to hold the signer accountable.

In our verification scheme, the Outsourcer and the Contractor can agree on an interval length l that is used by the Contractor to only sign one Merkle root hash per interval and attach it to its regular response at that index. By picking a large interval, the total time spent by the Outsourcer to verify signatures of responses and by the Contractor to calculate signatures of responses can be reduced significantly. Instead of signing each response, the Contractor only dynamically builts the Merkle tree over the interval's duration, which requires $(2 * l) - 1$ SHA-256 hashes and one signature for l responses. Additionally, the Contractor signs one Proof-of-Membership challenge per interval when requested by the Outsourcer. Compared to signing l responses, due to the lower computational complexity of hash function over digital signature algorithms, the complexity of using our Merkle root hash signing and verifying protocol is superior to signing and verifying each response for all $l > 2$. In practice, an interval length of $l > 1000$ is viable to save additional computational use cases.

An additional advantage of the described protocol is when the Outsourcer is only interested in receiving certain responses. For example, a CCTV might only be interested in object detection results if a person is detected in the sent frame. In this case, it is crucial to verify that the Contractor also performed any computation when it did not send back a response. By letting the Contractor commit to multiple responses via our protocol, a proof-of-membership challenge can be sent based on a random sample sent to the Verifier, which was not contained in the responses sent by the Contractor. If the sample sent by the Verifier can be recomputed to the initially received Merkle root hash sent by the Contractor, the Contractor proved its honesty. This way, the Outsourcer can verify computation even if no response was received.

3.5. Closing Phase

A contract can be terminated 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 Outsourcer 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 Contractor's response. Each contract also allows for one review of the counter-party specified in the contract, stored by 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 anyway to handle payment, this entity might also feature a review system. While a review system is not necessary to protect our scheme's security, 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. The payment settlement entity might request a deposit from participants that can only be withdrawn with a positive or neutral balance. A participant that generates a new identity when its balance turns negative would give up on its deposit. Alternatively, if the payment scheme that is used along our verification scheme requires a registration process, merely forging 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 cannot be faked. It still provides an incentive for participants to provide good QoS and act honestly. An Outsourcer might also 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 Verifiers might expect a lower reward per response if they can expect that the Outsourcer acknowledges responses at a high rate and does not timeout. Figure 3.13 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 does not require 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 generating a new identity due to a negative review as it might not be aware of being blacklisted.

3.5.3. According to Custom

A participant can terminate a contract at any time if it sends a termination-message to each counter-party. This message is important to not receive a bad review due to unnoticed timeouts. When a party other than the Outsourcer terminates a contract, an early terminate-message also leaves time for the Outsourcer to find a replacement while still being serviced. After the contract was terminated according to custom, the Verifier and the Contractor store their last signed input of the Outsourcer, which contains the latest number of acknowledged

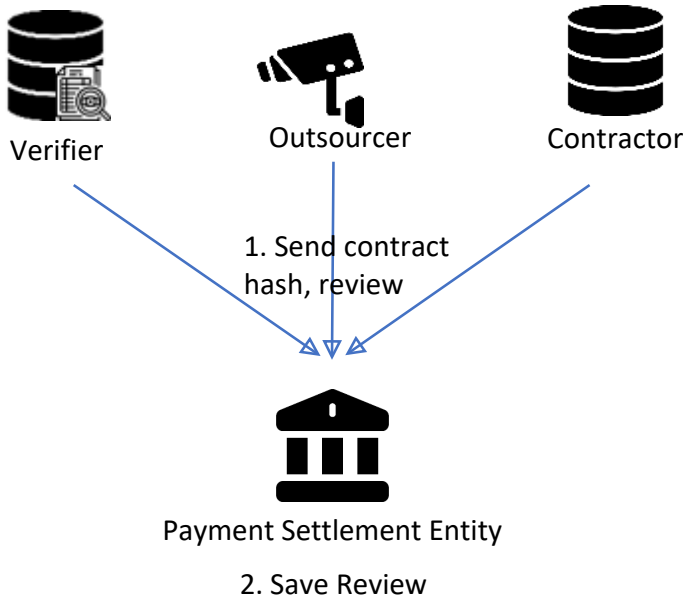

Figure 3.13.: Reviewing a participant after a contract ends

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. Then, it deducts 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 fine instead. If the Outsourcer does not report dishonest behavior, the Contractor and the Verifier are assumed to be honest. Figure 3.14 shows the process of redeeming a payment after a contract ended.

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 essential to distinguish between QoS violations and dishonest behavior as circumstances such as a weak connection may not be the participant's fault and unintended. Additionally, QoS violations can be detected instantly, and their damage is lower than that of dishonest

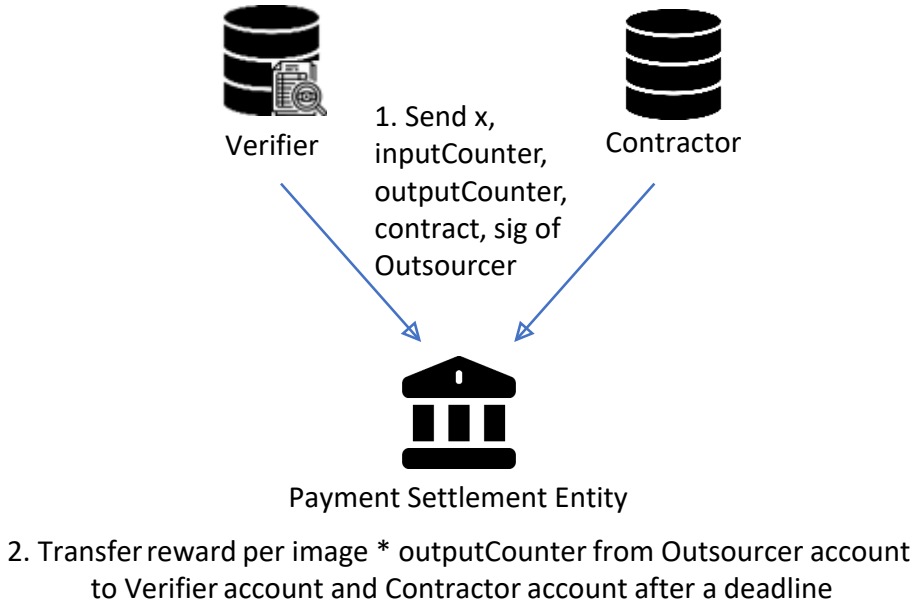

Figure 3.14.: Redeeming payment after closing a contract

behavior. For these reasons, we propose not to fine a participant for a QoS violation. Instead, an adequate consequence is to abort the contract, submit a bad review, and potentially blacklist 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. It 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 call Contestation to prove that the Verifier's response was incorrect instead. Figure 3.15 illustrates the process of reporting two unequal responses.

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. Suppose a Contractor is reported for dishonest behavior because of an incorrect response. In that case, it may decide to re-outsource the

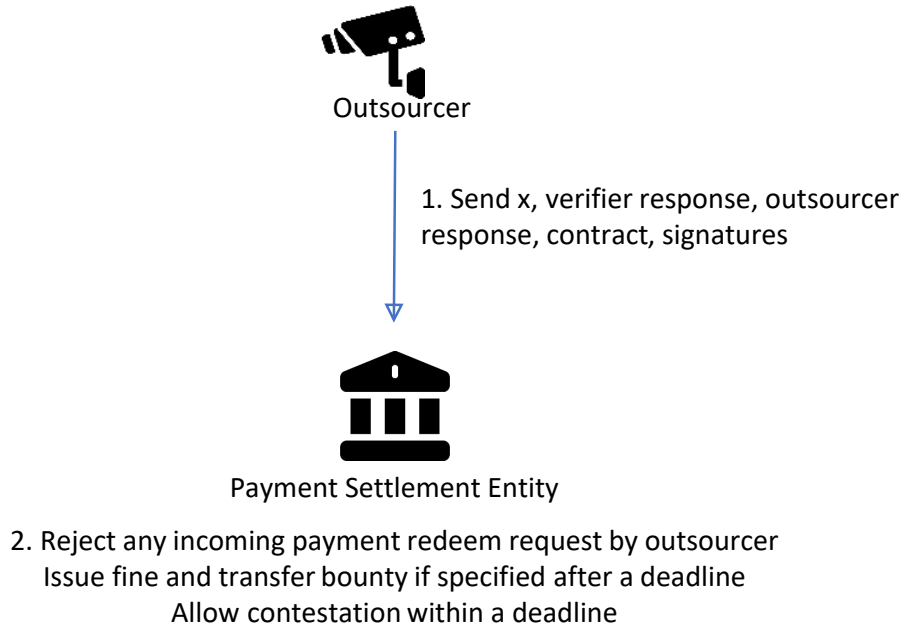

Figure 3.15.: Reporting dishonset behavior

original input to two additional random Verifiers that are available within a deadline. If both random Verifiers return a response that matches the Contractor's response, 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 3.16 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 matching with theirs is assumed to be honest. The other participant is assumed to be dishonest and gets fined. If more than 50% of available Verifiers are non-colluding, 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 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.

Note 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 extra money.

The computational overhead of Contestation is low as only one input has to be recomputed. However, it requires finding multiple available Verifiers in the system. As latency is not

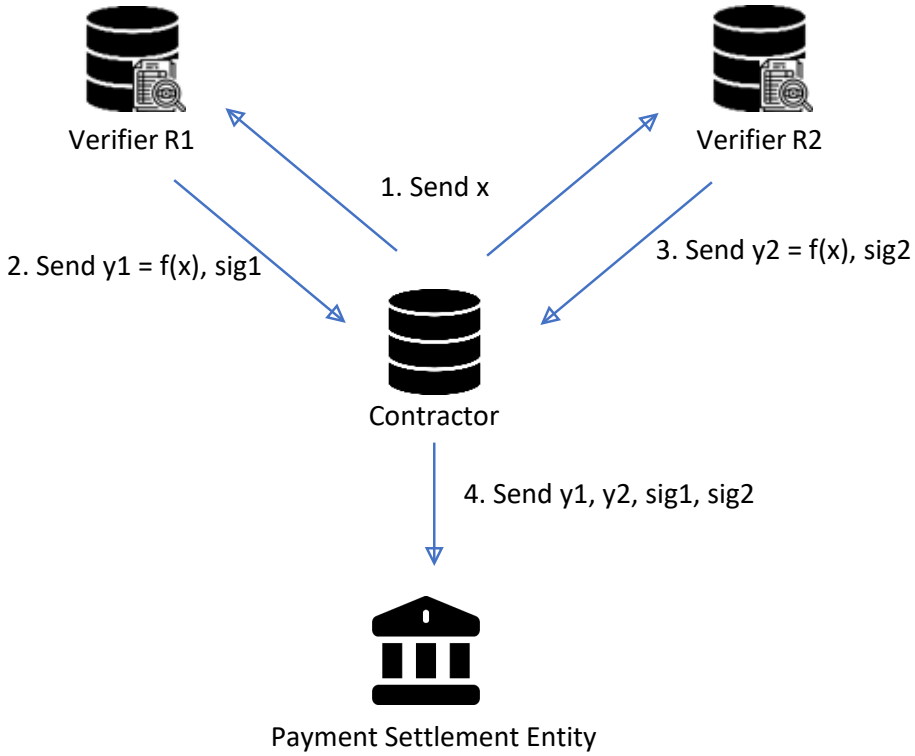

Figure 3.16.: Contestation

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 2.6.

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 2.6. Hence, 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. We include QoS violations in our threat model as Contractors or Verifiers

that accept a contract while knowing that they cannot provide sufficient QoS act dishonestly. In addition, we consider the external threat of an outside attacker trying to tamper with messages sent between participants. Our verification scheme resists all identified threats with high probability. In this section, we explain how our techniques prevent each threat. An explanation of each threat and its motivation can be found in chapter 2.

Number 1: Contractor sends back false responses to save resources

If a Contractor sends back incorrect 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 Outsourcer can adjust the sampling rate, it may also decide to pick a sampling rate of 100% to detect incorrect responses with 100% confidence.

Number 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 instead the Verifier that sent the incorrect response. In the section on Contestation, we explained that if more than 50% of available Verifiers in the whole ecosystem are non-colluding, Contestation guarantees to detect the cheating participant.

Number 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 using 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 two different inputs. Thus, not only do 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 committed to a specific input rather than 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 a 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). It only requires the Outsourcer

to store relevant information about the current sample. This leads to $O(1)$ memory overhead and does not lead to any memory overhead for the Contractor and the Verifier.

Number 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 never to have sent the reported response. The use of digital signatures prevents this threat. 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 with the response, any dishonest behavior aimed to change the record or resubmit a response is detected with 100% confidence.

Number 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. Microtransactions sent for each response are not an option for us. Usually, the payment scheme can become a latency bottleneck, and each transaction comes with transaction costs. Therefore, payment has to be handled after a contract ended. Thus, we assume that the payment scheme used along with our verification scheme supports deposits and payment on other participants' 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 the following requirements can be used with our verification scheme. (1) It deducts a deposit of each participant, (2) It verifies if signatures are valid to form a conclusion after the contract and Contestation deadline end, and (3) It uses the deposits to conduct payments.

Number 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 non-colluding. 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 a dispute.

Additionally, our verification scheme supports contracts that design incentives so that being honest is 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 by considering (1) the

probability of the Verifier being detected, (2) the reward it gets when not being detected, and (3) the fine it gets when being detected, to act honestly 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 these measurements, it is detected by 100% confidence by Contestation.

Number 7: Contractor and Verifier collude to save resources

The Contractor and the Verifier may collude to save computational resources by generating an incorrect 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 incorrect 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 does not inform the Verifier about the Contractor's identity. 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 if it cheats exceeds the payoff if 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 for the described strategy is that Contractor and Verifier have a cheating rate of 100%, an Outsourcer capable of recalculating the assigned function over one input can already detect the collusion with absolute certainty.

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

In our verification scheme, dishonest behavior of the Contractor or the Verifier is punished with the refusal of payment, and, if specified in the contract, with a fine. In contrast, QoS violations such as timeouts, low response rate, or high response time come without monetary consequences. We include QoS violations in our threat model, as participants' hardware might be responsible for bad QoS. Therefore, participating in an ecosystem with insufficient processing or networking capabilities should be discouraged.

We use blacklists, contract abortion, and reviews to punish bad QoS or to promote a good QoS. Whenever a participant receives a message from another participant that exceeds the QoS thresholds specified in its internal parameters, it may abort the current contract due to a QoS violation. It may also blacklist the other participant and submit a negative rating on this participant at the payment settlement entity. Thus, a participant who can not provide good QoS misses out on the current contract's ongoing payments and may receive fewer assignments or less reward due to a bad review and blacklisting. Even though a bad QoS might not be the fault of the other participant, frequent negative and positive reviews ensure that the resulting review balance represents the participant's reliableness.

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 a dishonest review gets submitted once in a while. Nodes with a high QoS may profit from increased rewards due to their proven high performance and honesty.

Thus, we conclude that with high confidence, our scheme can prevent QoS violations before a contract is started 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.

Number 11: Message Tampering

Our verification scheme considers not only 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 message they send. Thus, any participant receiving a message tampered with by an attacker immediately detects that the message is ill-formatted because the attacker cannot generate a valid signature associated with one of the participants. However, it should be noted 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 not to 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.

Table 2.6 summarizes the techniques used by our verification scheme to prevent or detect each protocol violations from our list. The confidence column indicates the probability at which the protocol violation can be prevented or detected by our techniques. Note that Contestation provides a 100% detection rate of associated protocol violations but only if >50% of available Verifiers in the ecosystem are non-colluding by not agreeing on an identical incorrect response. Figure 3.17 shows a summary of our designed verification scheme. Techniques written in a green font indicate that we designed them ourselves.

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 response 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% [1]
	3	Outsourcer sends different input to Contractor and Verifier to refuse payment	Digital Signatures (signature chain), Contestation	100% [1]
	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 of another entity	100%
Dishonest Behavior via Collusion	6	Outsourcer and Verifier collude to refuse payment and save resources	Randomization, Game-theoretic incentives, Contestation	100% [1]
	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%

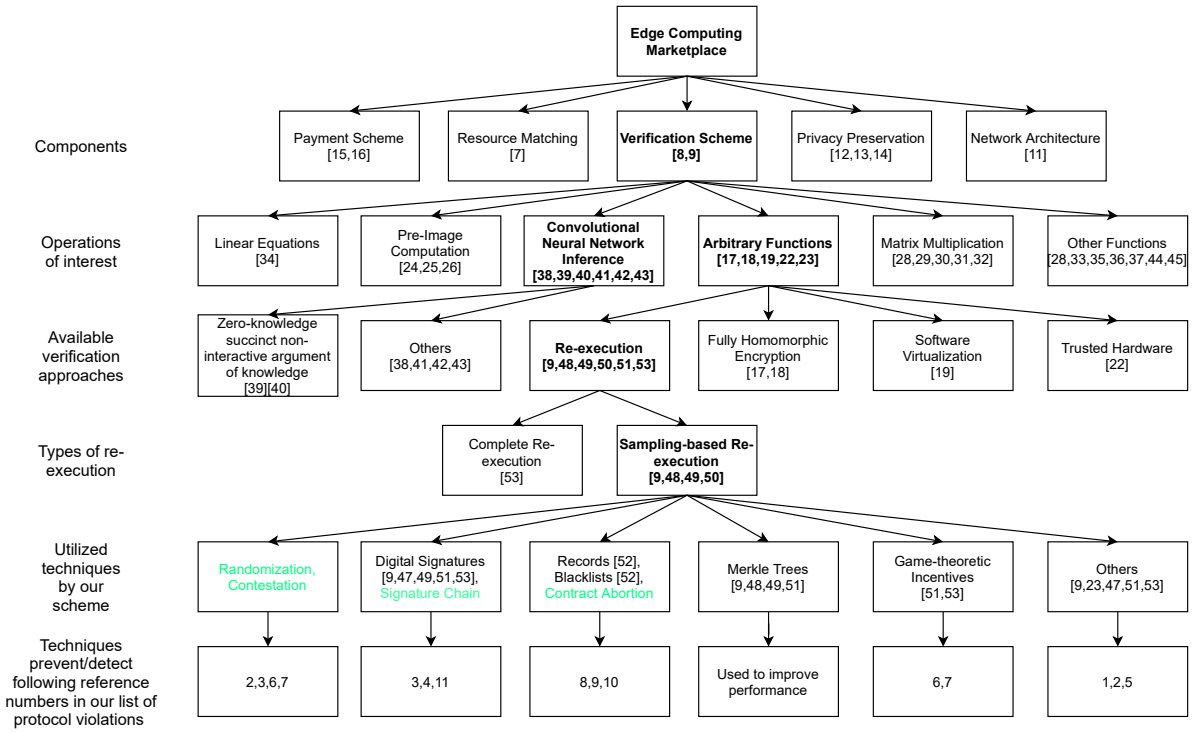

Figure 3.17.: Final overview: Designed verification scheme

[1] if more than 50% of available Verifiers in the ecosystem are non-colluding by not returning an identical false response

4. Implementation

While our verification scheme works with arbitrary functions, our reference implementation focuses on verifying the 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. CNNs are one of the most complex functions that IoT devices might need to outsource and have a high relevance at the edge. Thus, we believe that CNN inference is the perfect function type to stress test if our verification scheme is practical in real-time.

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 on the Microsoft Coco dataset. Tiny weights and custom weights can be used as well. Also, 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 [75].

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 [76]. We chose different versions of TensorFlow as our deep learning frameworks due to their high popularity and state-of-the-art performance. Also, TensorFlow supports the use of Tensor Processing Units (TPUs) such as the Coral USB Accelerator to perform CNN inference [77].

We chose ED25519 as a digital signature algorithm as it provided high security and achieved the best performance in our Python 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 [78].

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-processing pattern that eliminates network wait and a parallelized version that performs the verification tasks parallel to the object detection to increase frames per second (fps). The source code of our implementation with detailed documentation and comments is publicly available on the following website: `https://github.com/chart21/Verification-of-Outsourced-Object-Detection`

4.1. Software Architecture

We implemented our verification scheme in Python. One set of scripts have to be deployed at the machine acting as the Contractor and the machine operating as the Verifier. The other set

of scripts have to be deployed at the device machine as the Outsourcer. Thus, each remote machine executes the full functionality of one participant.

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 multithreading of key tasks, meaning that verification, preprocessing, inference, and postprocessing are conducted sequentially.

Initialization of the Scripts

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 include settings like 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 contract hash is sent with each message, each input can be associated with one unique contract. The contract indicates the function and format to use for 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 a machine have to be set. Each participant sends messages from a specific port and receives messages on another port. The Outsourcer can specify a sampling interval that defines the frequency of sending a random sample 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%. This means within every 200 messages, a random sample gets sent to the Verifier.

Furthermore, each participant sets thresholds for QoS violations, 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 a QoS violation. Finally, a participant has to specify or import a private key associated with the public key specified in the relevant contract used to sign all messages. After contract details and parameters are set, each participant can execute the scripts, thus starting the contract.

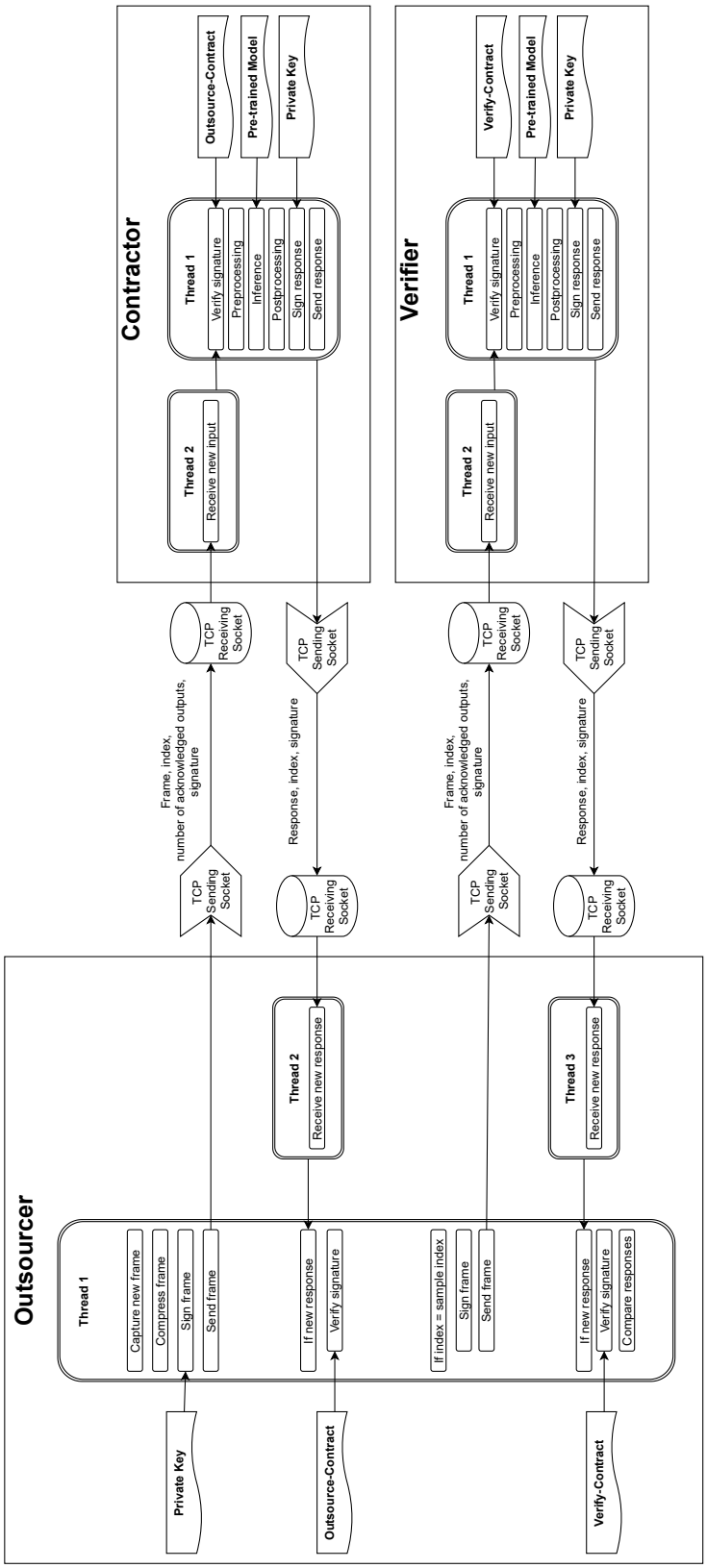

Figure 4.1.: Software architecture without multithreading of key tasks using a regular GPU

Outsourcer Script

The Outsourcer utilizes three threads. The second and third threads are only needed to listen to the TCP sockets of Verifier and Contractor and loading responses into memory, while the main thread is performing other tasks. As receiving responses and loading them into memory is an I/O bound task, these threads do not impact the main thread's performance measurably. This way, network wait is eliminated, assuming 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, it compresses the frame to a JPEG image. The Outsourcer signs the compressed frame, input index, contract hash (Contractor specific), and the number of acknowledged frames (Contractor specific) using its private key. It sends the compressed frame with the signature over a TCP socket to the Contractor. Suppose Merkle trees are set to be used in the contract. In that case, the Outsourcer may additionally send a proof of membership challenge of a previously checked sample, the current Merkle tree interval index, and specify whether it is time to answer the challenge along with a signature over it.

Suppose the current iteration index was assigned by the ongoing generation of random sample indices. In that case, the compressed frame also gets sent to the Verifier, attached with a signature over input index, contract hash (Verifier specific), and the number of acknowledged frames (Verifier specific). Algorithm 1 shows the pseudo-code of processing and sending a new frame to the Contractor and the Verifier if Merkle Trees and Multithreading of key tasks are not used.

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 arrives, and a response resulting from the same input index sent by the other party has arrived before, 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 only needs to verify the Contractor's responses if a Merkle root hash or challenge is sent and signed along with a response. A Merkle root hash must 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 the Outsourcer cancels contracts due to a QoS violation.

Algorithm 2 shows the pseudo-code of processing responses from the Contractor if Merkle Trees and Multithreading of key tasks are not used. Algorithm 3 shows the pseudo-code of processing responses from the Verifier if Merkle Trees and Multithreading of key tasks are not used.

Algorithm 1 Outsourcer Script - Execution Phase, Sending Frames - Without Multithreading, Without Merkle Trees

verifyKey$_{Contractor}$, contractHash$_{Contractor}$ = contract$_{Contractor}$.getDetails()
verifyKey$_{Verifier}$, contractHash$_{Verifier}$ = contract$_{Verifier}$.getDetails()
qosThresholds$_{Verifier}$, qosThresholds$_{Contractor}$ = parameters.getDetails()
generateNewSampleIndex()
while True **do**
    frame = capture()
    compressedFrame = compress(frame)
    frameSig$_{Contractor}$ = sign(privateKey, compressedFrame + frameIndex + contractHash$_{Contractor}$+ acknowledgedOutputs$_{Contractor}$)
    sendToContractor(compressedFrame, frameSig, frameIndex, acknowledgedOutputs$_{Contractor}$)
    if index == SampleIndex **then**
        frameSig$_{Verifier}$ = sign(privateKey, compressedFrame + frameIndex + contractHash$_{Verifier}$ + acknowledgedOutputs$_{Verifier}$)
        sendToVerifier(compressedFrame, frameSig, frameIndex, acknowledgedOutputs$_{Verifier}$)
        generateNewSampleIndex()
    end if
end while

Contract Violations If two responses belonging to the same input are found to be unequal, the Outsourcer aborts the Contracts due to dishonest behavior. Suppose the Contractor or the Verifier has sent a new response during one iteration of Thread 1. In that case, 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:

Algorithm 2 Outsourcer Script - Execution Phase, Processing Contractor Responses - Without Multithreading, Without Merkle Trees

```
1: while True do
2:     if newContractorResponse() == True then response, msg = receive_Contractor()
3:         if msg == 'terminate' then
4:             terminate()
5:         else
6:             sig, frameIndex = decode(msg)
7:             matchingSig = getPastSig(frameIndex, 'Contractor')
8:             if verify(verifyKey_Contractor, response + frameIndex + contractHash_Contractor +
   matchingSig) == False then
9:                 abort_QOS,Contractor()
10:            else
11:                acknowledgedResponses_Contractor+=1
12:                if getSavedResponse(frameIndex) != null then
13:                    if  response != getSavedResponse(frameIndex) then
14:                        abort_dishoestBehavior()
15:                    end if
16:                    if frameIndex == sampleIndex then
17:                        save(frameIndex, response, sig)
18:                    end if
19:                end if
20:            end if
21:        end if
22:    end if
23:    if violation(qosThresholds_Contractor) == True then
24:        abort_QOS,Contractor()
25:    end if
26: end while
```

1. No root hash received for the current interval in time.
2. Merkle tree leaf node does not match the 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 the previously 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.

Algorithm 3 Outsourcer Script - Execution Phase, Processing Verifier Responses - Without Multithreading, Without Merkle Trees

```
1: while True do
2:     if newVerifierResponse() == True then response, msg = receive_Verifier()
3:         if msg == 'terminate' then
4:             terminate()
5:         else
6:             sig, frameIndex = decode(msg)
7:             matchingSig = getPastSig(frameIndex, 'Verifier')
8:             if verify(verifyKey_Verifier, response + frameIndex + contractHash_Verifier + matchingSig) == False then
9:                 abort_QOS,Verfier()
10:            else
11:                acknowledgedResponses_Verifier+=1
12:                if getSavedResponse(frameIndex) != null then
13:                    if response != getSavedResponse(frameIndex) then
14:                        abort_dishoestBehavior()
15:                    end if
16:                else
17:                    save(frameIndex, response, sig)
18:                end if
19:            end if
20:        end if
21:    end if
22:    if violation(qosThresholds_Verifier) == True then
23:        abort()
24:    end if
25: end while
```

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. This way, it receives a signature as commitment once the Contractor signed the proof of membership challenge-response and the next Merkle root hash. 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.

Contractor Script

In the version without multithreading 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. Thread 1 fetches the newest frame in each iteration 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, the number of acknowledged responses, and contract hash. Suppose Merkle trees are set to be used in the Outsource-Contract. In that case, it also expects a random challenge, a variable indicating if it's time to answer a new challenge, and the current Merkle Tree interval. Algorithm 4 shows the pseudo-code of processing messages from the Outsourcer if Merkle Trees and Multithreading of key tasks are not used.

If the Contractor also agrees with the number of acknowledged responses, it continues with prepossessing the frame. This includes resizing, recoloring, and reformatting the frame to create a valid input that the CNN can process. Next, it uses the pre-trained model 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 of 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.

The Contractor signs the formatted response, the contract hash, and the signature sent by the Outsourcer belonging to that frame, using its private key. 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 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 prepared the current response, Thread 2 already downloaded it and loaded it into memory, thus eliminating network wait.

Contract Violations 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.

Algorithm 4 Contractor/Verifier Script - Execution Phase - Without Multithreading, Without Merkle Trees

```
verifyKey, contractHash = contract.getDetails()
qosThresholds, = parameters.getDetails()
while True do
    compressedFrame, msg = receive()
    if msg == 'terminate' then
        terminate()
    else
        sig, imageIndex, acknowledgedOutputs = decode(msg)
        if verify(verifyKey, compressedFrame + frameIndex + contractHash + acknowledgedOutputs, sig) == False then
            abort()
        end if
        if violation(qosThresholds, acknowledgedOutputs) == True then
            abort()
        end if
        frame = decompress(compressedFrame)
        imageData = preprocess(frame)
        cnnOutput = inference(imageData)
        detection = postprocess(cnnOutput)
        detectionSig = sign(privateKey, detection + frameIndex + contractHash + sig)
        respond(detection, frameIndex, detectionSig)
    end if
end while
```

Verifier Script

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. The Contractor and the Verifier participants fetch participant-specific information from internal parameters and the local contract.

Addressing Frame Loss

As our verification scheme supports scenarios with frame loss, each participant has internal parameters tolerating a certain frame loss rate before aborting a contract.

Suppose Merkle trees are set to be used in the Outsource-Contract. In that case, 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, the Contractor sends a new Merkle root hash for a new Merkle tree interval. It repeats sending 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 a termination message, the other participants terminate the contract according to custom and do not report a QoS violation because of a timeout.

4.2. Improving Run-time of Implementation

The run-time of a system attached to our verification scheme mainly depends 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 model's frame input size influences the time that the Outsourcer needs to capture and compress a new frame.

2. Each participant's hardware influences all tasks. The GPU of the Verifier and the Contractor mainly influences CNN inference time.

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

4. The use of a non-blocking, or a blocking message pattern influences the time each participant spends idle waiting for new messages over the network.

5. The use of multithreading, or multiprocessing of main tasks, influences all participants' overall execution time.

While the model, weights, and hardware used significantly impact system performance, these 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. ED25519 is currently considered the best performing secure digital signature

algorithm out of those three without sacrificing security [79]. ECDSA can be implemented with different curves and hashing algorithms [80]. 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 using multi-use digital signature schemes.

As our implementation is built with Python as a programming language, we benchmarked multiple digital signature libraries in Python that implement those different signatures. Note that two libraries implementing a digital signature algorithm with identical inputs and parameters can significantly differ in performance. For example, we found that the Starbank-ECDSA library is almost 10 times slower on 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 (https://github.com/tlsfuzzer/python-ecdsa)
2. RSA (https://github.com/sybrenstuvel/python-rsa)
3. ECDSA using Blake2b (https://github.com/tlsfuzzer/python-ecdsa)
4. ECDSA using Blake2s (https://github.com/tlsfuzzer/python-ecdsa)
5. ECDSA-Blake2b (https://github.com/Matoking/python-ed25519-blake2b)
6. Fast-ECDSA using SHA2 (https://github.com/AntonKueltz/fastecdsa)
7. Fast-ECDSA using SHA3 (https://github.com/AntonKueltz/fastecdsa)
8. Starbank ECDSA (https://github.com/starkbank/ecdsa-python)
9. NaCl (ED25519) (https://github.com/pyca/pynacl)
10. Lamport (https://github.com/Jonas1312/merkle-tree)
11. Winternitz (https://github.com/sea212/winternitz-one-time-signature)

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 frame. 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 signed frames 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. Note that the Outsourcer may use verifying less frequently than signing in our scheme if the Contractor loses frames 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 implementation performance, showing that current libraries cannot exploit their theoretical performance advantage. As one-time signatures also come with the increased complexity of maintaining a signature chain and generating new signatures frequently, we did not consider them for our final implementation.

4.2.2. Non-blocking Message Pattern

A straightforward way to process messages in a network is a blocking request-response message pattern. This means that a client sending data waits idly for a response that either the data was received or already a processed result. We utilize a non-blocking message pattern instead to transfer new messages into memory in parallel to processing the current one. All participants use a Sending TCP socket accessed by one thread to continually send new messages whenever a new one is ready. A Receiving TCP socket is accessed by another thread to continually 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 the identical performance of performing the task locally without any network delay.

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. While we use TCP sockets, one can seamlessly switch to UDP sockets if preferred.

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, multithreading or multiprocessing can 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 finishes inferencing, 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 our verification scheme's tasks.

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, we implemented a script without our verification scheme that runs only local object detection of objects pre-loaded into memory. This implementation has almost identical performance as our system that needs to send two messages over the network per frame and signs and verifies each message.

Comparing Multiprocessing and Multithreading

An important decision to make is whether to use multithreading or multiprocessing of key tasks. The key difference between multiprocessing and multithreading in Python is that multithreading runs tasks in parallel on the same core, switching between tasks in high frequency. Dynamic programming languages such as Python use a Global Interpreter Lock that serializes parallel tasks [81]. 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 [82]. 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 [83]. Thus, creating and maintaining additional processes comes with more computational overhead compared to threads. Multiprocessing is usually recommended for CPU bound tasks [84].

Our tests did not support this theoretical hypothesis. In fact, our multiprocessing version performed up to 50% worse than our single processing version. Moreover, our multithreading version performed up to 70% better than our single processing version. Perhaps 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. Thus, even though the parallel execution of key tasks performs better with multiprocessing, in theory, we use multithreading in our reference implementation. This also has the significant advantage that only one core is fully utilized while the other cores are available for other applications.

Multithreading when using a Regular GPU

In our regular implementation, the main thread is responsible for verifying a new frame's signature, 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 4.2 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, 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, Thread 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.

Multithreading when using a Coral USB Accelerator

Our implementation for Contractors and Verifiers using a Coral 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 4.3 illustrates the software architecture when using multithreading 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 differs for key tasks, library, and model used, we created different scripts for using a Coral USB Accelerator rather than a 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 verification schemes' tasks 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 multithreading instead of multiprocessing, only one core of the machine is fully utilized because of this decision. Combining a non-blocking message pattern and multithreading of key tasks leads to almost identical fps as performing object detection locally without a verification scheme.

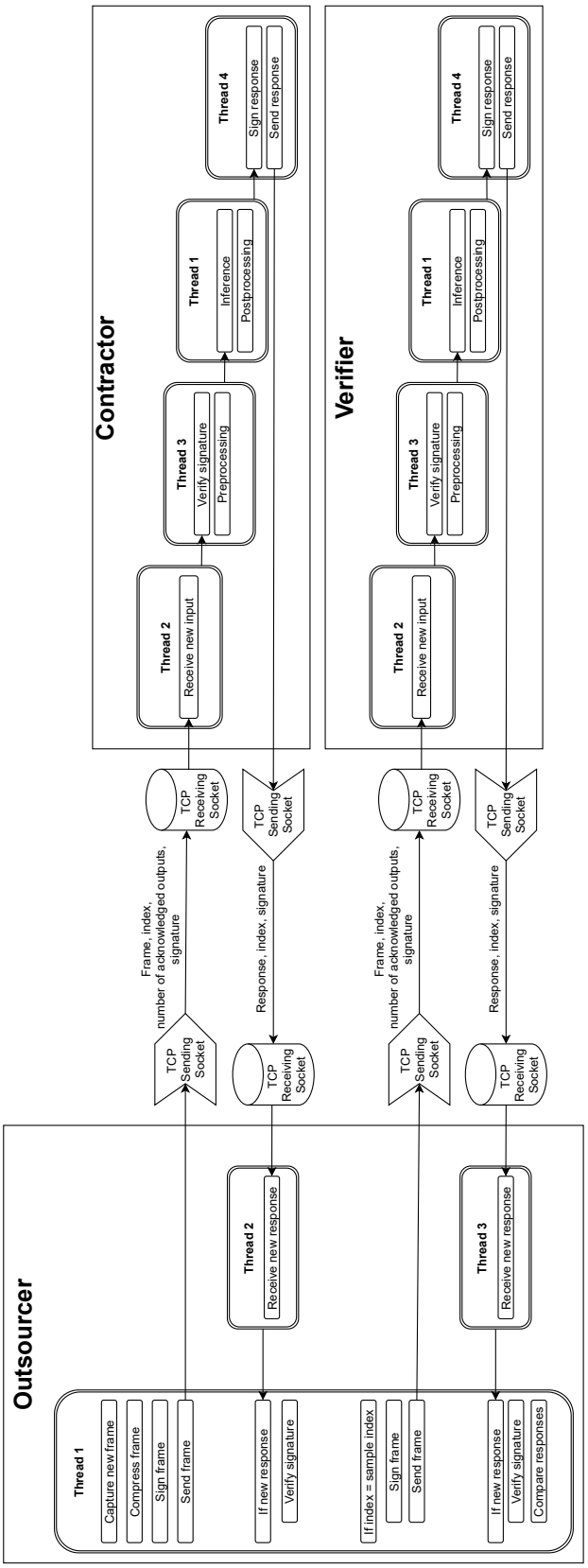

Figure 4.2.: Software architecture with multithreading of key tasks using a regular GPU

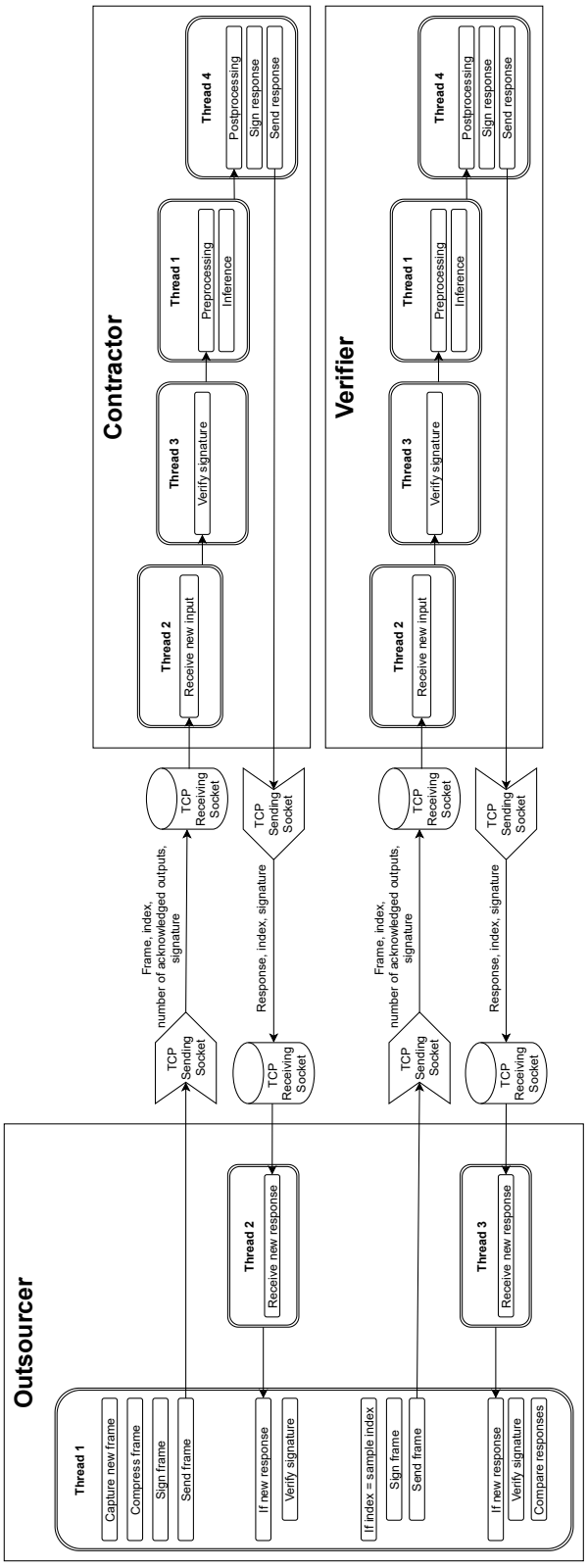

Figure 4.3.: Software architecture with multithreading of key tasks using a Coral USB Accelerator

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. The Coral USB Accelerator is an entry-level Tensor Processing Unit (TPU) that is specifically designed to perform neural network inference [85]. Edge accelerators such as the Coral USB Accelerator or other application-specific integrated circuits (ASICs) are gaining increased popularity and provide an unmatched performance per watts and performance per price ratio. Thus, we believe that to evaluate our implementation for an edge computing scenario properly, 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 Geforce 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. We mostly used tiny weights that were pre-trained on the Microsoft Coco dataset in our test [86]. As recent, more powerful GPUs than ours can achieve more than 1500fps with tiny weights, we recommend using more complex weights for higher accuracy on those devices. This setup's test results show if typical consumer notebooks, Desktop PCs, and a Raspberry Pi have enough performance to run state-of-the-art object detection models with our verification scheme over the local network. Figure 4.4 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: MobileNet SSD V2

The second 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. Figure 4.5 illustrates our second 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 weights. All sizes are reasonable to transfer from a Cloud server to remote hardware at the

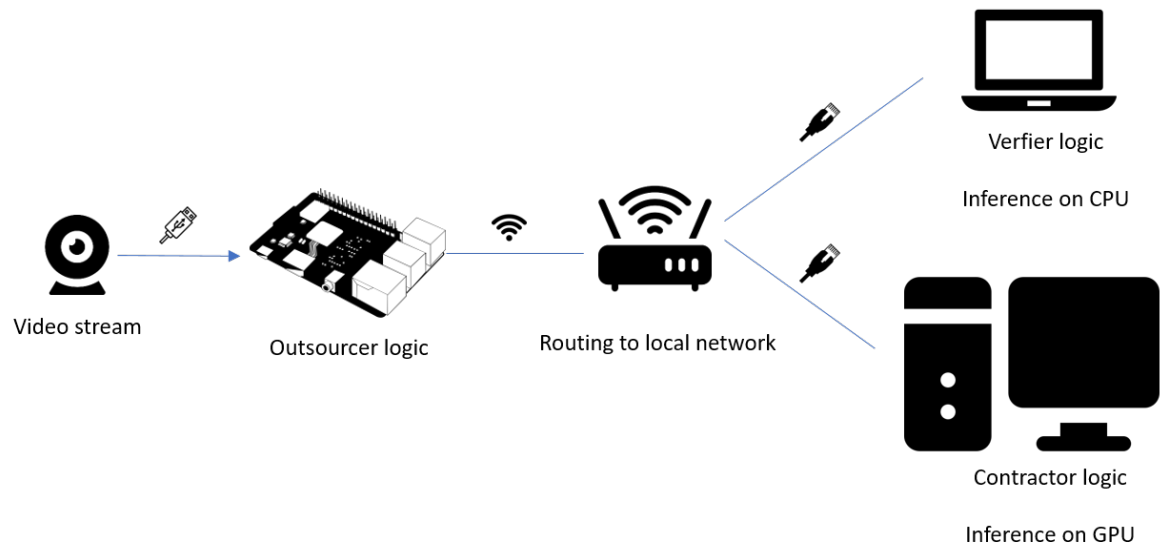

Figure 4.4.: Implementation and testing with a regular GPU and CPU

edge for one or multiple contracts. This setup's test results 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.

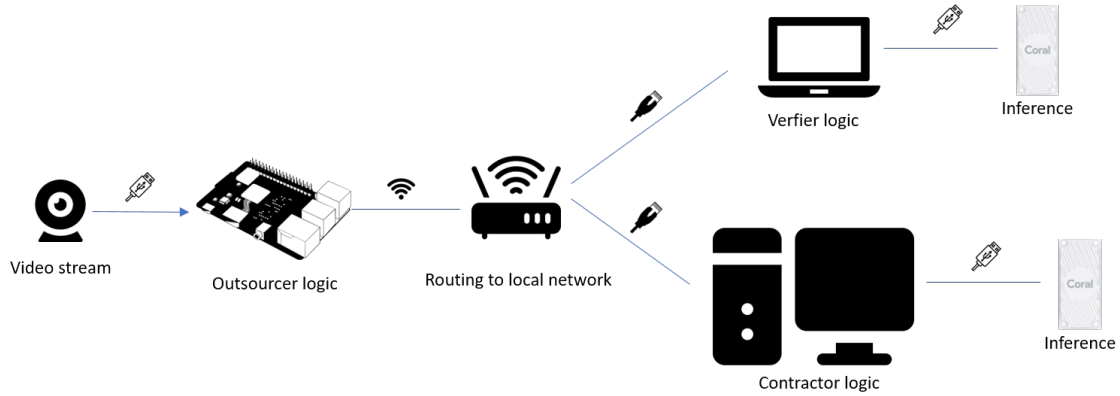

Figure 4.5.: Implementation and testing with an USB Accelerator

In chapter 5 we evaluate the performance of our two test setups. Afterward, we compare our verification scheme with other verification schemes proposed by the current academic literature.

5. Evaluation

In this chapter, we evaluate the performance, scalability, and security of our verification scheme and compare it with other verification schemes proposed by the current academic literature.

5.1. Results

We tested our verification scheme's performance with differently potent hardware and pre-trained CNN models. In all tests, our verification scheme achieved real-time performance and a low network bandwidth overhead.

5.1.1. Performance

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. These include the use of a non-blocking message pattern, more efficient digital signature algorithms, and multithreading.

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 ECDSA as a signature algorithm. "Display" in the figure's legend stands for the decision to output object detection results on the screen, which takes 2.4ms. This is only useful for demo purposes and not required in an actual setup. With this setup, we achieved 32.9 fps. We measure fps by 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 transferred into the Outsourcer's memory.

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 integrating 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 improve performance significantly. It leads to a fraction of the required amount of signing and verification operations when signing and verifying each message instead. Using Merkle trees in our implementation, we increased the average performance to 40.3 fps from 38.7 fps.

We switched from the ECDSA library to the NaCl library in Python that performed best in our benchmark. It uses ED25519 instead of ECDSA, which is considered to be more secure and faster than ECDSA. 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.

We chose not to show the object detection results on the Contractor's display for further benchmarks 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 the biggest performance improvement by far using multithreading of key tasks as introduced in chapter 4. We increased fps from 46 to around 68 fps with our test setup using multithreading of key tasks.

In summary, we achieved (1) major performance improvements with a non-blocking message pattern (-4.55ms, +5.8fps, 17.6%) and multithreading (-6.94ms, +21.8fps, 47.3%), (2) a moderate performance improvement with the use of ED25519, which was the most efficient in our benchmark (-1.8ms, +2,9fps, 7.5%), and (3) a minor performance improvement when using Merkle trees instead of signing every message (-1.03ms, +1,6fps, 4.1%, sometimes less improvement, dependant on the interval size). In total, we improved 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%).

Table 5.1 below shows the performance improvements by introducing each of the different techniques. Figure 5.1 illustrates the performance results of the Contractor with varying setups using a regular GPU. All performance improvements are relative to the implementation that uses all performance-enhancing techniques in the lines above. For instance, compared to the baseline using a non-blocking message pattern improved fps by 17.6%, and Merkle trees improved fps by another 4.1%. The product of both percentages leads to the relative fps improvement compared to the baseline.

Table 5.1.: Relative fps improvement of utilizing our performance-enhancing techniques on a regular GPU, compared to previous lines

Technique	Performance Improvement
Non-blocking message pattern	17.6%
Merkle trees	4.1%
Best performing digital signature library	7.5%
Multithreading of key tasks	47.3%
All techniques combined	106.9%

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

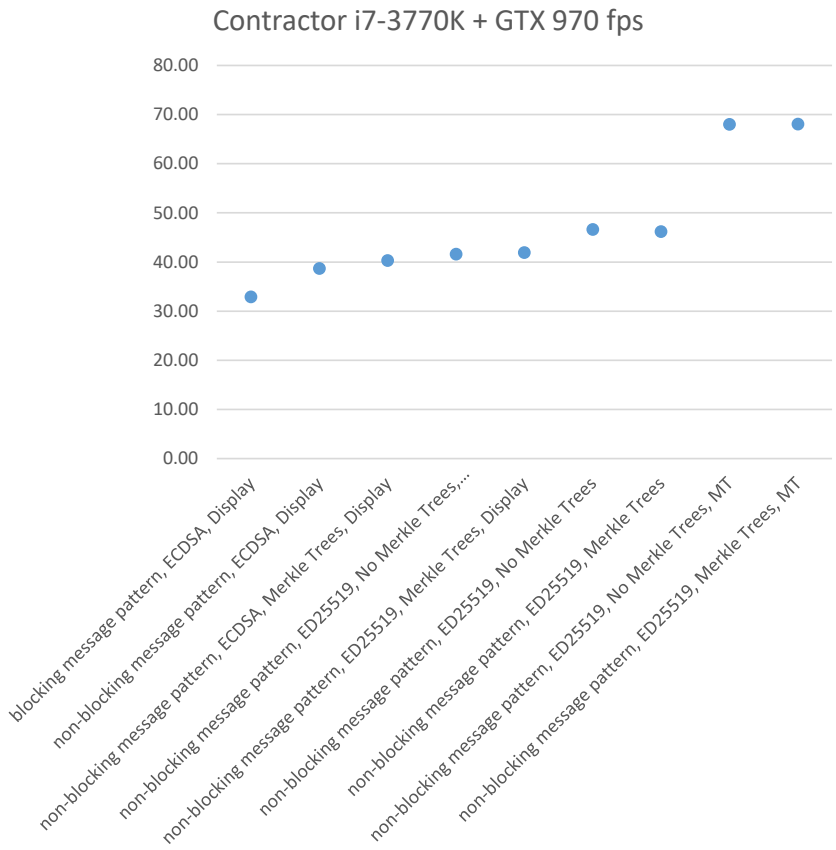

Figure 5.1.: Contractor performance with different setups on regular GPU

As one can see, our implementation using no multithreading 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 multithreading 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 improved performance significantly (-1.65ms, 10.4%). The reason for multithreading 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 key tasks can only increase performance by roughly 10% as the other tasks are less computationally expensive.

5.1.2. Final Results

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

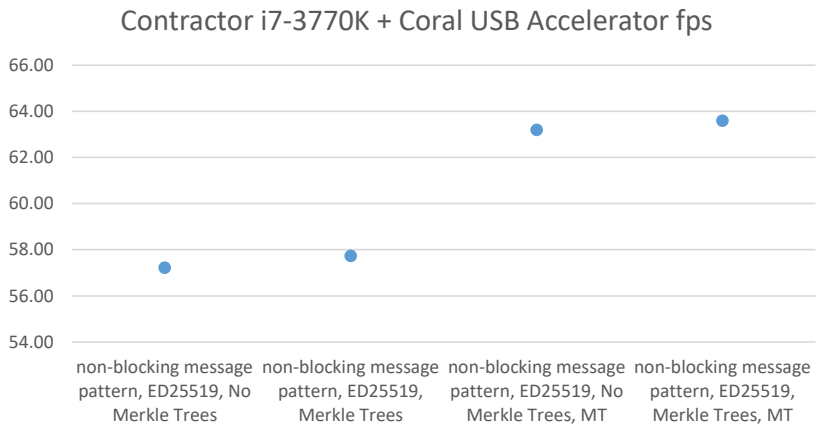

Figure 5.2.: Contractor performance with different setups on Edge Accelerator

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 also representative for other state-of-the-art CNN models that usually use an input size of 300x300 pixels. The Outsourcer's performance depends mainly on image size since capturing and compressing a large frame takes more time. 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. Note 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 affect the performance of one cycle. Even in our implementation that does not use multithreading of key tasks, the verification scheme only takes 0.36ms. This latency 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.

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 multithreading 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.

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 multithreading 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.

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 Verifier performs worse than the Contractor with the same hardware because 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 scripts can be easily adjusted to utilize the same message pattern as 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 each participant's individual performances 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, which 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, assuming it can capture, compress, and sign frames faster than the Contractor needs for one iteration. However, if the contractor's response rate is even lower than the minimum response rate specified in the Outsourcer's parameters, it aborts the Contract due to a QoS violation. In our test setup, the Contractor is the system's bottleneck, 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	% 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	0.00	78.70	21.30	0.90
Outsourcer	Raspberry Pi Model 4B			Yolov4 tiny 416*416	146.90	0.00	85.10	14.90	1.01
Contractor	Desktop PC	Core i7 3770K	GTX 970	Yolov4 tiny 416*416	68.06	0.00	100.00	0.00	0.00
Contractor	Desktop PC	Core i7 3770K	Coral USB Accelerator	MobileNet SSD V2 300*300	63.59	0.00	100.00	0.00	0.00
Contractor	Notebook	Core i5 4300U	Coral USB Accelerator	MobileNet SSD V2 300*300	49.30	0.00	100.00	0.00	0.00
Verifier	Notebook	Core i5 4300U	Coral USB Accelerator	MobileNet SSD V2 300*300	28.75	-	0.64	-	-

5.1.3. Multi-threading Performance Difference

This section explains 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 two threads. Thread 2 is constantly transferring 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 transfers 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 Outsourcer's signature, 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 transferred 3 new messages sent by the Outsourcer into memory. Thus, Thread 1 discards two messages and fetches the third message 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.

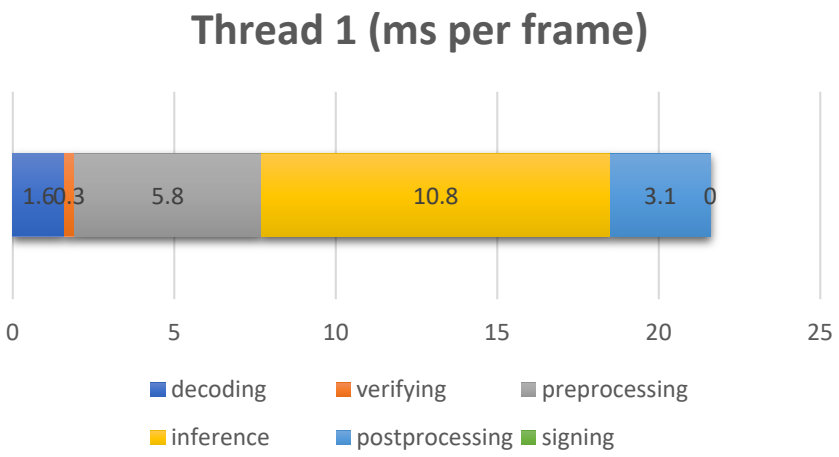

Figure 5.3.: Main thread in regular implementation

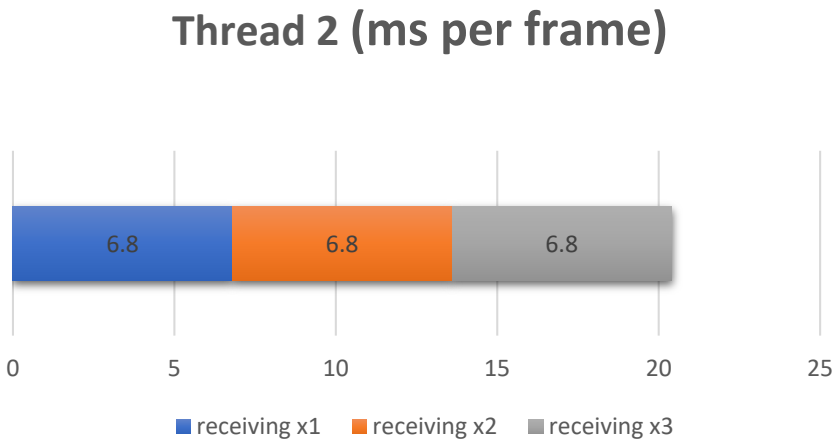

Figure 5.4.: Thread 2 in regular implementation

With multithreading of key tasks, tasks are run in parallel Threads to eliminate idle waiting time. As in the last 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 tasks can thus be respectively aggregated into one thread. As long as their total computation is less than 13.9ms, this limits all threads' total execution time to the bottleneck tasks' performance. Therefore, we end up four three different threads.

Figure 5.5 illustrates the newly added threads. Figure 5.6 illustrates Thread 2. Thread 2 is identical to Thread 2 in the previously explained version. It constantly transfers a new

message from the Outsourcer into memory. Thread 3 receives 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 transferring 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.

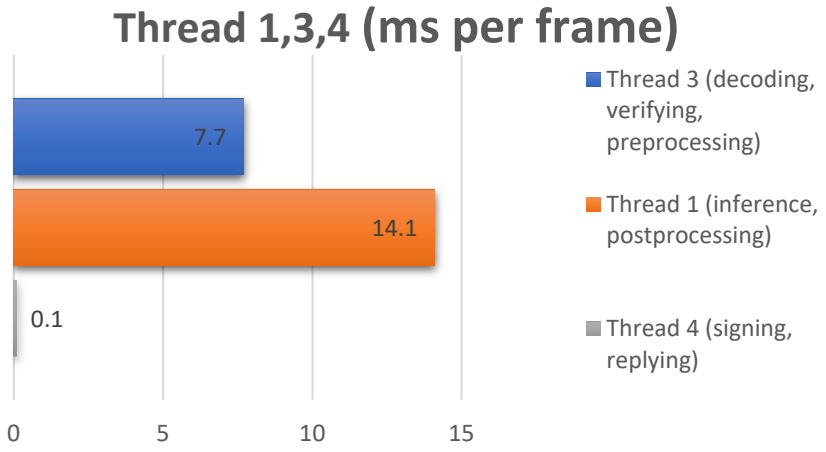

Figure 5.5.: Main threads in mutlithreading implementation

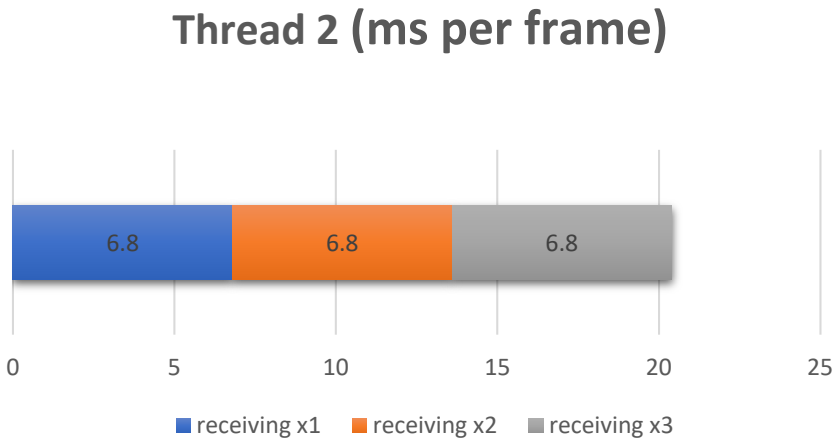

Figure 5.6.: Thread 2 in mutlithreading implementation

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, multithreading

of key tasks improved performance from 21.6ms (46.3 fps) to 14.9ms (70.9 fps) per message (+53.1% performance difference).

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. Therefore, we recommend using our scripts with multithreading 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 bottleneck thread's performance with only a slight overhead.

The tasks of the verification scheme (signing and verifying) are exclusively performed in Thread 1 and Thread 3. As 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 multithreading overhead of 1.39% cannot be ascribed to our verification scheme. Therefore, our verification scheme does not add any processing time to the overall Contractor' and Verifier' performance.

By using multithreading of key tasks, our verification scheme's performance overhead 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 multithreading version of the Outsourcer. Capturing and especially compressing a frame takes significantly longer than signing it and verifying responses. By running the verification scheme's tasks 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. It consists 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 the Designed Verification Scheme

Our research aimed 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 2.6. 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. They 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 an incorrect 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 non-colluding. 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 collude or use 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 incorrect responses (number 1), sampling-based re-execution only needs a small number of samples to detect this violation with more than 99% confidence. If confidence of 100% is preferred, our scheme and reference 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 have shown 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. Note that this violation

is a highly invested attack, as it requires at least two colluding participants. 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 more than 50% of available Verifiers in the ecosystem are non-colluding, 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 all possible protocol violations are detected or prevented with almost 100% confidence in a typical scenario.

QoS Violations such as frame loss, low response rate, or timeouts are eventually prevented with high confidence before starting a contract 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 verify the signatures of participants and prevent Sybil attacks by registering new Verifiers that join the ecosystem. Optionally it may feature a reward system. While a reward system is not needed to ensure our verification scheme's security, 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 the Designed Verification Scheme

Our verification scheme added less than 1ms latency per input to the overall system performance in our tests. Implementing multithreading of key tasks reduces the added latency to 0 by running the verification scheme's tasks 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: 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. Using the NaCl ED25519 library

over ECDSA-python ECDSA improves the performance of the verification scheme moderately.

4. Multithreading of key tasks (optional). Multithreading of key tasks can be used to perform all tasks of the verification scheme in a parallel thread to the main thread. If the outsourced function is the performance bottleneck among all Contractor's tasks, multithreading of key tasks can eliminate any performance overhead caused by our verification scheme.

Our test setup used state-of-the-art CNNs for object detection. While being a very computationally demanding task, 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. Combining all performance-enhancing techniques 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 the Designed Verification Scheme

Our verification scheme features increased security if a larger number of machines participate in the ecosystem. This is for two reasons. First, when an Outsourcer starts a Contract, a larger number of available Contractors and Verifiers in proximity reduces 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 non-colluding. A larger number of available Verifiers in the ecosystem increase the difficulty for potential attackers to gain a majority of the available Verifiers supporting their response.

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. Local blacklists also punish bad QoS as long as the associated node does not register with a new identity. We assume that the payment scheme that is implemented along with our scheme handles registering. Thus, it ideally prevents Verifiers from 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, the matching rate and reliability also increase with more participants.

The payment scheme is not likely to become the system's bottleneck for 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-independent. 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.

A computationally powerful Outsourcer can perform sampling-based re-execution itself instead of relying on a third-party Verifier. This decision increases our scheme's scalability without hurting its security.

We conclude that the performance and security of our system improve 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 a contract's length. 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

There are three main differences of our verification scheme with schemes proposed by current academic literature that we analyzed: (1) security from protocol violations, (2) requirements for trusted third parties (TTPs), (3) Computational overhead.

Security In table 2.6 we summarised which protocol violations other verification scheme are resistant to. In table 3.1 we summarized which protocol violations our scheme is resistant to. Note that the techniques listed in table 2.6 are taken from multiple papers. Thus, even though all techniques introduced by academic literature combined can prevent or detect 9 out of 11 identified protocol violations, the individual proposed schemes provide significantly less security.

Our verification scheme combines these effective techniques from other papers and adds three new ones: Randomization, Contestation, and signature chains of length 2. Randomization is mainly used to increase the probability of preventing collusion. Contestation and signature chains improve overall security and solve the two unaddressed protocol violations that the current literature left out.

In protocol violation 3, the Outsourcer sends different inputs to the Contractor and the Verifier to refuse payment. The Contractor and the Verifier, however, sign the signature of the Outsourcer's initial message. This generates a signature chain of length 2. If an Outsourcer reports the Contractor for cheating, the Contractor can prove during Contestation that the values leading to its received signature chain were all correct. As a consequence, the values

leading to the signature chain of the Verifier show that the raw inputs sent to both participants are unequal.

Contestation not only provides security for another unaddressed violation (number 6) but also offers improved performance over other techniques such as re-outsourcing inputs to additional Verifiers during the execution phase. Thus, we conclude that the three described techniques uniquely introduced by our verification scheme improve our verification scheme's security and performance compared to others. Also, to the best of our knowledge, there is no verification scheme proposed by current academic literature that prevents or detects all 11 identified protocol violations.

Table 5.3 shows a comparison of collected techniques used by the verification schemes analyzed in chapter 2 and ours.

Requirements for trusted third parties (TTPs) Apart from performance, a verification scheme's scalability is influenced by the requirements it defines for trusted third parties. While all verification schemes use a source of trust such as a Blockchain or a TTP, the amount of computation that the TTP is involved in differs.

Our scheme only requires one TTP or Blockchain that issues the payments according to the protocol. It does not have to be located at the Edge and is not needed during the Execution phase or re-execution. It only has to be able to verify signatures and perform payments on a participant's behalf. All communication conducted by the TTP can be delayed as it is only involved in the Closing Phase and not in the latency-sensitive Execution phase.

We can compare our scheme with other verification schemes using the following criteria:

1. The number of third parties required.
2. The number of third parties actively involved in the execution phase.
3. The computational complexity of operations performed by the trusted third party.

Some verification schemes such as [53] require a TTP to be actively involved in the execution phase. These solutions do not scale well, as we can assume that trusted resources are scarce. Frequent computations during the execution phase use a significant amount of resources from the TTP. Also, schemes that use a Blockchain such as [9] for contract-related transactions do not scale well due to high transactions costs, high latency, and low performance of current public Blockchains. The majority of verification schemes evaluated by us, such as [48] also assume the honesty of at least one participant, usually the Outsourcer. While this assumption can be utilized to reduce the need for TTPs and reduce complexity, it is not realistic in a public ecosystem.

Table 5.4 shows a comparison of the verification schemes analyzed in chapter 2 based on the above mentioned criteria. We conclude that the use of TTPs in our verification scheme scales better than the majority of schemes compared to. As our TTP is only responsible for issuing payments, and optionally a reward system, the computational complexity is low. Also, any participant in our scheme can act dishonestly. Thus, we made no artificial assumptions to improve the scheme's scalability at the cost of real-world security.

Table 5.3.: Techniques to prevent or detect protocol violations used by current academic literature compared with ours

Referred Number of Violation	Other Techniques	Our Techniques	Comparison
1	Re-execution, utilization of third-party Verifiers if required	Sampling-based re-execution, utilization of a third party Verifier if required	Identical
2	Re-outsourcing input to additional Verifiers if responses do not match	Contestation	Contestation improves performance
3	Not addressed	Digital Signatures (signature chain), Contestation	Combination of length 2 signature chain and Contestation can detect this violation
4	Digital Signatures	Digital Signatures	Identical
5	TTP or Blockchain that is authorized to conduct payment on behalf of another entity	TTP or Blockchain that is authorized to conduct payment on behalf of another entity	Identical
6	Not addressed	Randomization, Game-theoretic incentives, Contestation	Contestation provides reliable detection of this violation
7	Incentives designed in a way that being honest is a dominant strategy	Randomization, Game-theoretic incentives	Randomization increases probability of prevention
8	Blacklisting, Review System, Recording timestamps of Merkle root hashes of inputs and responses to track QoS violations globally	Blacklisting, Review System, Recording timestamps of Merkle root hashes of inputs and responses to track QoS violations globally	Identical
9	Same as Nr 8	Same as Nr 8	Identical
10	Same as Nr 8	Same as Nr 8	Identical
11	Digital Signatures	Digital Signatures	Identical

Table 5.4.: Comparison of verification schemes based on the TTPs required

Related Work	Amount of TTPs	TTPs involved during execution phase	Computational complexity of TTP operations	Additional Assumptions
Our verification scheme	1	0	Low	-
Uncheatable grid computing [48]	1	1	High	Outsourcer is fully trusted, Outsourcer is capable of re-execution
Security and privacy for storage and computation in cloud computing [49]	2	2	High	Outsourcer and Verifier are trusted entities
Bitcoin-based Fair Payments for Outsourcing Computations of Fog Devices [9]	2	1	Moderate	Outsourcer is capable of re-execution
Integrity verification of cloud-hosted data analytics computations [50]	1	1	Low	Outsourcer is fully trusted
Game-Theoretic Analysis of an Incentivized Verifiable Computation System [51]	>3	1	High	Outsourcer is semi-trusted
Anticheetah: Trustworthy computing in an outsourced (cheating) environment [52]	>2	>2	High	Multiple Contractors available for one Contract, pool of trusted nodes available
Efficient fair conditional payments for outsourcing computations [47]	1	0	Very low	Can only be used for certain functions
Mechanisms for Outsourcing Computation via a Decentralized Market [23]	1	1	High	Verifiers are semi-trusted
Incentivized outsourced computation resistant to malicious Contractors [53]	2	1	Very high	Trusted timestamp server is available

Computational overhead. The computational overhead of a verification scheme can be divided into network bandwidth overhead and processing time overhead. Network bandwidth overhead is caused by adding redundancy, signatures, or additional information to messages. Processing time overhead is caused by signing data, verifying signatures, redundant operations, or other operations specific to the verification scheme.

Multiple authors such as [53] measured the processing time overhead of their scheme with specific applications. However, these empirical results cannot be used to accurately compare the performance of different verification schemes due to different test setups and application parameters. For instance, even in our internal tests with identical test setup and application, we only changed the frame size from 300X300 to 416X416. This already reduced our verification scheme's performance by 10.9% due to digitally signing and verifying larger inputs. Hence, only with a currently non-existent standardized benchmark and test setup for verification schemes, an empirical analysis would be useful. Therefore, we limit our analysis to theoretical estimations.

Three major decisions influence the processing time overhead of a verification scheme in theory:

1. Whether it uses sampling-based or complete re-execution.
2. Whether it uses digital signatures, and utilizes any performance-enhancing techniques such as only signing Merkle roots in fixed intervals instead of singing all data.
3. Whether it uses a Blockchain or a TTP during its execution phase (as analyzed earlier).

Our verification scheme utilizes sampling-based re-execution, digital signatures, and performance-enhancing techniques such as committing to an interval of messages by only signing their Merkle root hash. Thus, we conclude that, in theory, it only adds low computational overhead to the system. While our empiric performance results discussed previously cannot be used for comparison with other verification schemes, they do show that the overhead of less than 1ms per message for very complex operations and large messages was small enough to be utilized in real-time.

Note that from the verification schemes we evaluated, all schemes that do not use digital signatures (or Blockchain transactions that automatically utilize digital signatures) assumed at least one participant to be honest. Thus, our performance comparison also includes the kinds of additional assumptions that the compared verification scheme makes in contrast to ours. For instance, even though the verification scheme proposed by [52] features very low performance-overhead, its requirements of an available pool of trusted nodes are hard to meet.

Table 5.4 shows a comparison of the verification schemes analyzed in chapter 2 based on the above mentioned criteria. We conclude that our verification scheme performs better than the majority of verification schemes in theory and provides the best trade-off between honesty-assumption and performance.

Table 5.5.: Comparison of verification schemes based on their computational overhead

Related Work	Integrity verification	Digital Signatures, Merkle Trees used	Computational Overhead	Additional Assumptions
Our verification scheme	Sampling-based re-execution	yes, yes	Low	-
Uncheatable grid computing [48]	Sampling-based re-execution	yes, yes	Low	Outsourcer is fully trusted, Outsourcer is capable of re-execution
Security and privacy for storage and computation in cloud computing [49]	Sampling-based re-execution	yes,yes	Low	Outsourcer and Verifier are trusted entities
Bitcoin-based Fair Payments for Outsourcing Computations of Fog Devices [9]	Sampling-based re-execution	yes (Blockchain), yes	Moderate	Outsourcer is capable of re-execution
Integrity verification of cloud-hosted data analytics computations [50]	Adding artificial data	no, no	Very low	Outsourcer is fully trusted
Game-Theoretic Analysis of an Incentivized Verifiable Computation System [51]	Sampling-based re-execution	yes (Blockchain), yes	Moderate	Outsourcer is semi-trusted
Anticheetah: Trustworthy computing in an outsourced (cheating) environment [52]	Overlapping	no, no	Low	Multiple Contractors available for one Contract, pool of trusted nodes available
Efficient fair conditional payments for outsourcing computations [47]	True ringers and Bogus ringers	yes, no	Very low	Can only be used for certain functions
Mechanisms for Outsourcing Computation via a Decentralized Market [23]	Sampling-based re-execution	yes (Blockchain), no	Moderate	Verifiers are semi-trusted
Incentivized outsourced computation resistant to malicious Contractors [53]	Complete re-exeuction	yes, no	High	Trusted timestamp server is available

Summary of comparisons Our goal was to design a secure and scalable verification scheme for arbitrary functions.

In our first comparison, we used our compiled list of protocol violations from chapter 2 to compare our verification scheme with verification schemes proposed by current academic literature. According to our analysis' results, only our scheme can prevent a comprehensive list of possible protocol violations that can occur in a computation marketplace with untrusted participants. We made three main contributions in security:

1. We compiled a comprehensive list of protocol violations that can occur in computation marketplaces with untrusted participants.
2. We combined several techniques proposed by multiple proposed verification schemes to detect or prevent most violations.
3. We designed two new techniques, Randomization and Contestation, to detect or prevent the currently unaddressed violations and improve the detection and prevention confidence of already addressed violations.

Thus, we conclude that our proposed verification scheme improves the state-of-the-art verification schemes in terms of security.

In our second comparison, we compared multiple verification schemes in terms of their TTP involvement. A minimum number of TTPs, amount of TTP computations and communication, and computational complexity of TTP tasks are desired to ensure the system's scalability. Otherwise, the lack of TTPs or overutilized TTP resources can become the bottleneck of a system that aims to scale with a large number of participants. Compared to other verification schemes, our verification scheme requires only a low complexity of TTP operations. Also, it only requires one TTP that is not involved during the execution phase. Therefore it requires lower TTP involvement than the majority of compared verification schemes. Our main contribution to TTP involvement is featuring lower TTP involvement than most of the compared verification schemes without sacrificing security. This was achieved mainly by our Contestation protocol that uses untrusted Verifiers for reliable conflict resolving instead of a TTP. While there are schemes that even require less TTP involvement than ours, they do this at lower security guarantees than our schemes. Note that none of the compared verification schemes was designed without a trusted party or Blockchain.

In our third comparison, we compared the computational overhead of multiple verification schemes. In order to scale well, a verification scheme should provide minimal overhead to all participants for the following reasons: 1. The Outsourcer should be able to perform all tasks of the verification scheme even if it is a computationally weak device 2. The Contractor and the Verifier should have as many resources as possible left for other jobs. Our verification scheme performs better than the majority of verification schemes in theory and provides the best trade-off between honesty-assumption and performance. The combination of digital signatures, Merkle trees, and sampling-based re-execution was also used by other verification schemes to combine high security and good performance. Our verification scheme improves the state-of-the-art verification schemes by offering higher security at a similar performance than schemes that currently provide the best security-performance trade-off.

6. Conclusion

In this thesis, we designed and implemented a verification scheme for outsourced arbitrary functions in a computation marketplace. The verification scheme is resistant to a comprehensive list of protocol violations that might occur in a computation marketplace with untrusted participants. Even when used to verify complex operations such as CNN inference, our verification scheme adds less than 1ms latency overhead to the system and only causes neglectable network bandwidth overhead. These performance and security characteristics make it an ideal choice to be used within a latency-sensitive edge computing marketplace that matches untrusted third party resources to computationally weak, untrusted IoT devices. Compared to the verification schemes proposed by the current academic literature, our verification scheme provides security against a larger variety of attacks and requires a lower involvement of TTPs. Both performance and security of our verification scheme improve with an increasing number of participants. Thus, we conclude that our verification scheme is secure and scalable.

6.1. Key Findings

Based on the research gaps in related academic literature we formulated the objective of our verification scheme to (1) prevent or detect all possible threats in a computation marketplace with untrusted participants, (2) with minimal use of trusted third parties, and (3) low latency and computational overhead. We made the following key findings in our thesis that address these objectives.

Identification of Practical and Secure Approaches for Verification Schemes In chapter 2 we identified a verification scheme that verifies the integrity of outsourced computation as one of the key components for an edge computing marketplace (table 2.1). We concluded that partial re-execution of computation, which works for arbitrary functions, is currently the most practical approach to verify arbitrary functions.

Compilation of a Comprehensive List of all possible Threats in Computation Marketplaces We noticed non of the reviewed literature aimed to provide a comprehensive list of potential protocol violations that might occur in computation marketplaces with untrusted participants. Thus, we compiled such a comprehensive list ourselves that considers dishonest behavior by individuals and collusion, quality of service violations, and external attacks as a threat model. We analyzed that by combining techniques introduced by multiple verification schemes, only 9 out of 11 identified protocol violations can be detected or prevented (table

2.6). We found the two remaining identified protocol violations to be unaddressed by the current academic literature.

Design of a Verification Scheme that addresses all Identified Threats We combined the following techniques proposed by other verification schemes in our scheme: Sampling-based re-execution, digital signatures, Merkle trees. Optionally, our scheme also uses game-theoretic incentives, a review system, and blacklisting. To prevent or detect the two currently unaddressed protocol violations, we designed two new protocols: Randomization and Contestation. These protocols are also used to improve the prevention and detection rate of protocol violations already addressed by the current academic literature. We also explained how our verification scheme prevents or detects each possible protocol violation from our compiled list. For the majority of protocol violations, our verification scheme provides 100% confidence of detection or prevention (table 3.1).

Performance Results of our Verification Scheme In chapter 5, we evaluated the characteristics and results of measuring our proposed verification scheme's performance on consumer hardware and TPUs. On all tested machines, our verification scheme achieves less than 1ms of latency overhead per frame (table 5.2). Using our scripts that utilize parallel execution, the overhead can be reduced to 0 by running the verification scheme's tasks in a parallel thread to the main threads (figure 5.5). The network bandwidth overhead of our scheme is neglectable, at most 84 bytes per frame. Our test results also showed that our performance-enhancing techniques provided performance improvements of over 100% compared to our unoptimized implementation (table 5.1).

Comparison of our Verification Scheme with other Verification Schemes In comparison with verification schemes proposed by the current academic literature, our verification scheme provides higher security by preventing or detecting all identified protocol violations (table 5.3). At the same time, it requires less third party involvement (table 5.4) and performs better than the majority of verification schemes compared with, based on theoretical analysis (table 5.5).

6.2. Assessment of our Results

Based on our key findings, we evaluated our verification scheme's efficiency, along with its scalability and security. We concluded that our scheme provides practical performance for real-time applications and is secure by preventing or detecting all identified protocol violations with high confidence. Also, its security improves with an increased number of participants. It only requires one TTP for settling payments, which does not have to be located at the edge, is not utilized during the execution phase, and only performs computationally inexpensive operations. Thus, we concluded that our scheme also scales well.

Limitations We identified two limitations that come along with our verification scheme. These are based on specific functionalities that our verification scheme requires a compatible payment scheme to contain. First, the payment scheme used alongside our verification scheme has to provide specific functionalities like verifying signatures and optionally featuring a review system. Even though the required operations are computationally inexpensive and can be handled at an arbitrary delay, this requires a modification of general-use payment systems.

Second, a trusted entity should issue identity-checks of new Verifiers to prevent Sybil-attacks, as discussed in chapter 5. Existing payment settlement entities usually already provide identity checks. Ideally, the payment settlement entity would also provide such an identity-check to ensure the system's integrity. On the other hand, contractors and Outsourcers can join the ecosystem with an arbitrary amount of identities without hurting the system's security.

Future Work In this thesis, we designed a scheme that verifies the integrity of arbitrary functions. We also defined how a trusted payment scheme should be used alongside the verification scheme to enable reliable payment according to the protocol. While we consider our verification scheme to be finalized, it only implements one essential component of a fully functioning edge computing marketplace. As explained in chapter 2, we extracted the following remaining components identified by the current academic literature to make an edge computing marketplace viable:

1. Payment.
2. Matching and price-finding.
3. Privacy preservation.
4. Specific network architecture.

While all of these components have been addressed by other related work, they have to be compared, perhaps improved, and combined with a verification scheme. Future work may first identify the best solution proposed for each component and then aggregate these components to an end-to-end system that can be deployed as a standalone edge computing marketplace for arbitrary functions.

A. Digital Addenda

Our Digital Addenda includes full-size figures, tables, our source code, and documentation. It is accessible attached to the thesis or via the following website:
`https://github.com/chart21/Verification-of-Outsourced-Object-Detection`
It contains the following elements:

Figures

1. Overview of our literature review
2. Overview of our designed verification scheme
3. Software Architecture without multithreading of key tasks
4. Software Architecture with multithreading of key tasks on a regular GPU
5. Software Architecture without multithreading of key tasks on a Coral USB Accelerator

Tables

1. Key Results
2. Full Results

Demonstration

1. GIF of executing the scripts for a regular GPU using YOLOv4 tiny 416*416
2. GIF of executing the scripts for a Coral Edge Accelerator using MobileNet SSD 300*300

List of Figures

1.1. Temporary QoS improvements . . . 2
1.2. Scenario of an edge computing marketplace with mobile IoT devices outsourcing reappearing object detection task types . . . 5

2.1. Assumed scenario . . . 30
2.2. Number 1: Contractor sends back false responses to save resources . . . 31
2.3. Number 2: Verifier sends back false responses to save resources . . . 31
2.4. Number 3: Outsourcer sends different input to Contractor and Verifier to refuse payment . . . 32
2.5. Number 4: Contractor or Verifier tries to avoid global penalties when convicted of cheating or QoS violations . . . 33
2.6. Number 5: Outsourcer refuses to pay even if obliged to by the protocol . . . 33
2.7. Number 6: Outsourcer and Verifier collude to refuse payment and save resources 34
2.8. Number 7: Contractor and Verifier collude to save resources . . . 35
2.9. Number 8: Contractor times out . . . 36
2.10. Number 11: Message tampering by an external attacker . . . 37
2.11. Final overview: Literature review . . . 39

3.1. Initial situation . . . 42
3.2. Randomization overview . . . 44
3.3. Payoff Matrix of the Contractor with intuitively designed incentives . . . 46
3.4. Payoff Matrix of the Verifier with intuitively designed incentives . . . 46
3.5. Payoff Matrix of the Contractor with honesty-promoting incentives . . . 47
3.6. Payoff Matrix of the Contractor with honesty-promoting incentives . . . 47
3.7. Contracts with incentive related agreements . . . 48
3.8. Execution phase with signatures . . . 50
3.9. Execution phase with signatures and acknowledged responses . . . 52
3.10. Truth table of 3 input XOR . . . 54
3.11. Hash Chain of a digital stream with combination codes using XOR [72] . . . 54
3.12. Visualization of a Merkle Tree and a Proof-of-Membership challenge . . . 55
3.13. Reviewing a participant after a contract ends . . . 58
3.14. Redeeming payment after closing a contract . . . 59
3.15. Reporting dishonset behavior . . . 60
3.16. Contestation . . . 61
3.17. Final overview: Designed verification scheme . . . 67

4.1. Software architecture without multithreading of key tasks using a regular GPU 70

4.2. Software architecture with multithreading of key tasks using a regular GPU . 82
4.3. Software architecture with multithreading of key tasks using a Coral USB Accelerator 83
4.4. Implementation and testing with a regular GPU and CPU 85
4.5. Implementation and testing with an USB Accelerator 85

5.1. Contractor performance with different setups on regular GPU 88
5.2. Contractor performance with different setups on Edge Accelerator 89
5.3. Main thread in regular implementation 93
5.4. Thread 2 in regular implementation 93
5.5. Main threads in mutlithreading implementation 94
5.6. Thread 2 in mutlithreading implementation 94

List of Tables

2.1. Components of an edge computing marketplace 9
2.2. Approaches for verifying the integrity of arbitrary functions 12
2.3. Verification scheme approaches for different function types 18
2.4. Verification scheme approaches for different function types 19
2.5. Techniques utilized by different verification schemes 29
2.6. Protocol violations and techniques to prevent them proposed by evaluated literature 38

3.1. Protocol violations, techniques to prevent them utilized by our verification scheme, and confidence of prevention 66

5.1. Relative fps improvement of utilizing our performance-enhancing techniques on a regular GPU, compared to previous lines 87
5.2. Key results 91
5.3. Techniques to prevent or detect protocol violations used by current academic literature compared with ours 101
5.4. Comparison of verification schemes based on the TTPs required 102
5.5. Comparison of verification schemes based on their computational overhead . . 104

Glossary

CNN A convolutional neural network (CNN) marks a class of deep neural networks that are usually applied to analyze digital images. 6

Contestation In our verification scheme, Contestation is a protocol we designed that guarantees the correct detection of the cheating entity. 41

Contractor In our verification scheme, the Contractor is the machine that computes an assigned function over sent inputs of the Outsourcer for a fee. 1

dominant strategy If a player's strategy is at least as good of a response to all other players' actions than all alternatives, it is called a dominant strategy. If it is better than each other alternative, it is called a strictly dominant strategy. 25

ECDSA Elliptic Curve Digital Signature Algorithm (ECDSA) is a digital signature algorithm based on elliptic-curve cryptography. 53

ED25519 ED25519 is a digital signature algorithm based on twisted Edward-curves that offers both faster performance and higher security than ECDSA. It uses SHA-512 and Curve25519. 54

edge computing Edge computing brings computation and data storage closer to the location where it is needed to reduce latency and save network-bandwidth. 1

Inference Inference applies knowledge from a trained neural network model to infer a result. 2

IoT Internet of Things (IoT) devices are equipped with sensors and other technologies to exchange data with other devices and systems over the Internet. 1

Merkle Tree A Merkle Tree enables efficient and secure verification of the contents of large data structures. 21

Outsourcer In our verification scheme, the Outsourcer is a computationally weak device that outsources a computational task to a third-party machine. It sends all inputs to a Contractor and random samples to a Verifier. 1

Randomization In our verification scheme, Randomization is a protocol we designed that lets the Contractor and the Outsourcer agree on a random available Verifier. 40

RSA RSA (Rivest–Shamir–Adleman) is a digital signature algorithm based on elliptic-curve cryptography based on the factoring problem using two large primes. 53

SHA The Secure Hash Algorithms (SHA) are a family of cryptographic hash functions. The reference implementation of our verification scheme uses the SHA-256, SHA3-256 (256-bit message size), and the SHA-512 (512-bit message size) hash algorithms. 68

TTP A trusted third-party (TTP) is assumed to always act honestly. 6

Verifier In our verification scheme, the Verifier is a randomly selected machine that computes an assigned function over random samples sent by the Outsourcer for a fee. 4

Bibliography

[1] W. Tang, X. Zhao, W. Rafique, L. Qi, W. Dou, and Q. Ni. "An offloading method using decentralized P2P-enabled mobile edge servers in edge computing". In: *Journal of Systems Architecture* 94 (2019), pp. 1–13.

[2] M. Satyanarayanan. "The emergence of edge computing". In: *Computer* 50.1 (2017), pp. 30–39.

[3] M. T. Beck, M. Werner, S. Feld, and S. Schimper. "Mobile edge computing: A taxonomy". In: *Proc. of the Sixth International Conference on Advances in Future Internet*. Citeseer. 2014, pp. 48–55.

[4] P. Mach and Z. Becvar. "Mobile edge computing: A survey on architecture and computation offloading". In: *IEEE Communications Surveys & Tutorials* 19.3 (2017), pp. 1628–1656.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[6] J. Ren, Y. Guo, D. Zhang, Q. Liu, and Y. Zhang. "Distributed and Efficient Object Detection in Edge Computing: Challenges and Solutions". In: *IEEE Network* 32.6 (2018), pp. 137–143. DOI: 10.1109/MNET.2018.1700415.

[7] 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.

[8] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang. "A Survey on the Edge Computing for the Internet of Things". In: *IEEE Access* 6 (2018), pp. 6900–6919. DOI: 10.1109/ACCESS.2017.2778504.

[9] 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.

[10] R. Gennaro, C. Gentry, and B. Parno. "Non-interactive verifiable computing: Outsourcing computation to untrusted workers". In: *Annual Cryptology Conference*. Springer. 2010, pp. 465–482.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, X. Lin, S. Hu, and M. Du. "VeriML: Enabling Integrity Assurances and Fair Payments for Machine Learning as a Service". In: *arXiv preprint arXiv:1909.06961* (2019).

[17] 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.

[18] 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.

[19] 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.

[20] V. Costan and S. Devadas. "Intel SGX Explained". In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 86.

[21] F. Tramer and D. Boneh. "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware". In: *arXiv preprint arXiv:1806.03287* (2018).

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

[23] 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).

[24] 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.

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

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

[27] J. Alman. "Limits on the universal method for matrix multiplication". In: *arXiv preprint arXiv:1812.08731* (2018).

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

[29] 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.

[30] 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.

[31] 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).

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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].

[44] 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.

[45] 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.

[46] J.-J. Laffont and D. Martimort. *The theory of incentives: the principal-agent model*. Princeton university press, 2009.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

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

[54] R. Yang, F. Yu, P. Si, Z. Yang, and Y. Zhang. "Integrated Blockchain and Edge Computing Systems: A Survey, Some Research Issues and Challenges". In: *IEEE Communications Surveys & Tutorials* 21 (2019), pp. 1508–1532.

[55] J. Zhang, W. Cui, J. Ma, and C. Yang. "Blockchain-based secure and fair crowdsourcing scheme". In: *International Journal of Distributed Sensor Networks* 15 (July 2019), p. 155014771986489. DOI: 10.1177/1550147719864890.

[56] C. Chedrawi and P. Howayeck. "Audit in the Blockchain era within a principal-agent approach". In: *Information and Communication Technologies in Organizations and Society (ICTO 2018):"Information and Communications Technologies for an Inclusive World* (2018).

[57] Y. Zhang, R. H. Deng, X. Liu, and D. Zheng. "Blockchain based efficient and robust fair payment for outsourcing services in cloud computing". In: *Information Sciences* 462 (2018), pp. 262–277.

[58] V. Pham, M. Khouzani, and C. Cid. "Optimal contracts for outsourced computation". In: *International Conference on Decision and Game Theory for Security*. Springer. 2014, pp. 79–98.

[59] B. Pejo and Q. Tang. "To cheat or not to cheat: A game-theoretic analysis of outsourced computation verification". In: *Proceedings of the Fifth ACM International Workshop on Security in Cloud Computing*. 2017, pp. 3–10.

[60] I. Dittmann. "How reliable should auditors be?: optimal monitoring in principal–agent relationships". In: *European Journal of Political Economy* 15.3 (1999), pp. 523–546.

[61] M. J. Osborne et al. *An introduction to game theory*. Vol. 3. 3. Oxford university press New York, 2004.

[62] M. Aghassi and D. Bertsimas. "Robust game theory". In: *Mathematical Programming* 107.1-2 (2006), pp. 231–273.

[63] C. A. Holt and A. E. Roth. "The Nash equilibrium: A perspective". In: *Proceedings of the National Academy of Sciences* 101.12 (2004), pp. 3999–4002.

[64] R. C. Picker. "An introduction to game theory and the law". In: (1994).

[65] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, and A. Lysyanskaya. "Incentivizing outsourced computation". In: *Proceedings of the 3rd international workshop on Economics of networked systems*. 2008, pp. 85–90.

[66] M. Z. Shahrak, M. Ye, V. Swaminathan, and S. Wei. "Two-way real time multimedia stream authentication using physical unclonable functions". In: *2016 IEEE 18th International Workshop on Multimedia Signal Processing (MMSP)*. IEEE. 2016, pp. 1–4.

[67] A. A. Yavuz, A. Mudgerikar, A. Singla, I. Papapanagiotou, and E. Bertino. "Real-time digital signatures for time-critical networks". In: *IEEE Transactions on Information Forensics and Security* 12.11 (2017), pp. 2627–2639.

[68] Y. Wu and R. H. Deng. "Scalable authentication of MPEG-4 streams". In: *IEEE Transactions on Multimedia* 8.1 (2006), pp. 152–161.

[69] R. Gennaro and P. Rohatgi. "How to sign digital streams". In: *Annual International Cryptology Conference*. Springer. 1997, pp. 180–197.

[70] G. Becker. "Merkle signature schemes, merkle trees and their cryptanalysis". In: *Ruhr-University Bochum, Tech. Rep* (2008).

[71] M.-H. Chang and Y.-S. Yeh. "Improving Lamport one-time signature scheme". In: *Applied mathematics and computation* 167.1 (2005), pp. 118–124.

[72] E. Abd-Elrahman, M. Boutabia, and H. Afifi. "Video streaming security: reliable hash chain mechanism using redundancy codes". In: *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*. 2010, pp. 69–76.

[73] R. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: vol. 293. Aug. 1987, pp. 369–378. ISBN: 978-3-540-18796-7. DOI: 10.1007/3-540-48184-2_32.

[74] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo. "Fractal Merkle Tree Representation and Traversal". In: *Topics in Cryptology — CT-RSA 2003*. Ed. by M. Joye. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 314–326. ISBN: 978-3-540-36563-1.

[75] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[76] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". In: *arXiv preprint arXiv:2004.10934* (2020).

[77] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[78] M. J. Dworkin. *SHA-3 standard: Permutation-based hash and extendable-output functions*. Tech. rep. 2015.

[79] M. A. Mehrabi and C. Doche. "Low-cost, low-power FPGA implementation of ED25519 and CURVE25519 point multiplication". In: *Information* 10.9 (2019), p. 285.

[80] D. Johnson, A. Menezes, and S. Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1.1 (2001), pp. 36–63.

[81] R. Meier and A. Rigo. "A way forward in parallelising dynamic languages". In: *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*. 2014, pp. 1–4.

[82] T. Khot. "Parallelization in Python". In: *XRDS: Crossroads, The ACM Magazine for Students* 23.3 (2017), pp. 56–58.

[83] A. Marowka. "On parallel software engineering education using python". In: *Education and Information Technologies* 23.1 (2018), pp. 357–372.

[84] S. An. "Python and Parallel Programming". In: ().

[85] A. Ghosh, S. A. Al Mahmud, T. I. R. Uday, and D. M. Farid. "Assistive Technology for Visually Impaired using Tensor Flow Object Detection in Raspberry Pi and Coral USB Accelerator". In: *2020 IEEE Region 10 Symposium (TENSYMP)*. 2020, pp. 186–189. DOI: 10.1109/TENSYMP50017.2020.9230630.

[86] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.