Decentralised Repeated Modular Squaring Service Revisited: Attack and Mitigation

Aydin Abadi[†] and Steven J. Murdoch[‡]

University College London

Abstract. Repeated modular squaring plays a crucial role in various time-based cryptographic primitives, such as Time-Lock Puzzles and Verifiable Delay Functions. At ACM CCS 2021, Thyagarajan *et al.* introduced "OpenSquare", a decentralised protocol that lets a client delegate the computation of repeated modular squaring to third-party servers while ensuring that these servers are compensated only if they deliver valid results. In this work, we unveil a significant vulnerability in OpenSquare, which enables servers to receive payments without fulfilling the delegated task. To tackle this issue, we present a series of mitigation measures.

1 Introduction

Repeated modular squaring presents an intriguing attribute: it is (believed to be) sequential. This characteristic assumes a pivotal role in the advancement of time-based cryptographic primitives, such as Time-Lock Puzzles (TLPs) [25,3,20] and Verifiable Delay Functions (VDFs) [8,29,23]. In time-based cryptographic primitives, a solver must continuously perform modular squaring for a (often large) period of time, demanding significant computational power. Generally, to outsource specific computations (like repeated modular squaring) to an untrusted server, Verifiable Computation (VC) schemes can be employed. These schemes have existed for decades, e.g., see [6,17,16]. However, the initial VC schemes did not explicitly account for compensating honest servers. With the emergence of blockchain technology, researchers have devised blockchain-based schemes that offer compensation to servers supplying accurate computation results. As a result, at ACM CCS in 2021, Thyagarajan *et al.* [27] presented "OpenSquare", a blockchain-based protocol that facilitates the delegation of the computation of repeated modular squaring from a client to third-party servers, all while retaining the ability to verify the accuracy of the results generated by these servers. The primary objective of this scheme is to empower the client to remunerate the server solely in the event that the provided result is valid.

Our Contributions. We have identified an attack that can be executed against Thyagarajan *et al.*'s OpenSquare [27]. Moreover, we present potential mitigations to address the aforementioned issue. This attack allows misbehaving entities to receive payments from the client without actually carrying out the sequential squaring task. Our analysis demonstrates that this attack consistently succeeds, except in scenarios where the

[†] aydin.abadi@ucl.ac.uk
[‡] s.murdoch@ucl.ac.uk

number of colluding parties is exceptionally high. In such instances, these parties may secure a diminished share of rewards compared to what they would earn if they were to act honestly. This attack illustrates that OpenSquare cannot attain the two primary objectives it sought to achieve: (1) incentivising servers to consistently provide their services, and (2) guaranteeing a high probability that the client receives a valid result.

We have contacted the authors of OpenSquare and provided them with a copy of the attack description. To maintain fairness, and with the authors' approval, we have incorporated their responses into this paper (in Section 5) too.

Broadly speaking, the matter we identify stems from a single scenario: *insecure delegation of the verification phase*. More precisely, in addition to delegating the task of executing sequential squaring, OpenSquare delegates the task of verifying the results' correctness to third parties, who may consist of rational and colluding adversaries. However, the system falls short in countering potentially rational verifiers who may deviate from the protocol's description and collaborate with others to increase their payoff.

Our findings provide evidence that special care must be taken when the verification phase in a VC scheme is delegated to untrusted parties, especially when they have financial incentives to deviate from the protocol's description.

2 Related Work

2.1 Verifiable Computation: A Superclass of Delegated Repeated Squaring

Verifiable Computation (VC) is a protocol that enables a party to delegate the computation of a specific function to resourceful third-party servers, even if they are potentially untrusted. This delegation allows the party to efficiently verify the computation result without needing to re-execute the function. Some VC schemes, such as those described in [6,17,16]), have been designed to support arbitrary computations. In contrast, others, like the schemes outlined in [4,24,26]), have been specifically tailored to optimise efficiency for particular types of computations.

However, the initial VC schemes did not incorporate mechanisms for automatically compensating honest servers. As a result, various application-specific schemes, as exemplified in [27,1] and generic schemes, like the ones discussed in [2,13], have been proposed. These schemes allow a party to remunerate a server only if the server provides a valid result. In the following section, we will provide further details about OpenSquare, presented in [27].

2.2 Time-Lock Puzzles: An Application of Repeated Modular Squaring

Timothy C. May [21] was the first who proposed the idea of sending information into the future, i.e., time-lock puzzle. Since the scheme that May proposed uses a trusted agent that releases a secret on time for a puzzle to be solved and relying on a trusted agent, Rivest *et al.* [25] proposed an RSA-based TLP scheme. This scheme does not require a trusted agent and relies on *repeated modular squaring*. This RSA-based protocol has been the foundation of many time-lock puzzle schemes that support the encapsulation of an arbitrary message.

Since the introduction of the RSA-based time-lock puzzle, various variants of it have been proposed. For instance, researchers such as Garay *et al.* [15] have proposed time-lock puzzle schemes which consider the setting where a client can be malicious and needs to prove to a solver that the correct solution will be recovered after a certain time. Also, Baum *et al.* [7] developed a composable TLP in the universal composability framework.

Thyagarajan *et al.* [27] at the ACM CCS 2021 have presented a blockchain-based protocol called "OpenSquare" that enables a party (for instance in a TLP scheme) to delegate the computation of sequential squaring to a set of servers/solvers. OpenSquare is of significant importance because (1) it is the first scheme that considers verifiably outsourcing sequential squaring and (2) has been published at a prestigious venue.

In OpenSquare, a client posts the number of squaring required and its related public parameters to a smart contract, say D, and places a certain amount of deposit in D. The solvers (whose identities are not fixed before the protocol's execution) are required to propose a solution before a certain time point. Then, each solver locally computes the solution and related proof of correctness. It sends to D, the solution, proof, and an asking price. The contract allows any users or solvers to check the proof and send their complaints if the proof is invalid. If no complaints are received, then D pays the solver who asked for the lowest price. The scheme uses a proof system which requires at least 4 exponentiations to verify each proof. Since performing a fixed computation on a smart contract, say D, costs more than performing the same computation locally, the scheme requires users (that can include the competing solvers) to locally verify each proof and report it to D if the verification fails. In this case, D verifies the proof itself.

Recently, Abadi *et al.* [3] proposed the notion of the "delegated time-lock puzzle" and its concrete instantiation. The instantiation allows both the client and server/solver to delegate the recourse-demanding tasks to untrusted semi-honest helpers.

TLPs have various applications, including e-voting [12], timely payments in central bank digital currency [19], fair contract signing [10], and timed secret sharing [18].

2.3 Verifiable Delay Function (VDF): An Application of Repeated Modular Squaring

A VDF enables a prover to provide a publicly verifiable proof stating that it has performed a pre-determined number of sequential computations [8,29,9,23]. VDF was first formalised by Boneh *et al* in [8]. They proposed several VDF constructions based on SNARKs along with either incrementally verifiable computation or injective polynomials, or based on time-lock puzzles. Later, Wesolowski [29] and Pietrzak [23] concurrently improved the previous VDFs from different perspectives and proposed schemes based on sequential modular squaring. Most VDFs have been built upon TLP schemes.

VDFs find numerous applications, including their use in decentralised systems to extract reliable public randomness from a blockchain [8], time-stamping [31], and proof of storage [5].

3 Preliminaries

3.1 Commitment Scheme

A commitment scheme involves two parties, sender and receiver. It also includes two phases: commit and open. In the commit phase, the sender commits to a message: x as $\mathsf{Com}(x, r) = \mathsf{Com}_x$, that involves a secret value: $r \xleftarrow{\$} \{0,1\}^\lambda$. In the end of the commit phase, the commitment Com_x is sent to the receiver. In the open phase, the sender sends the opening $x := (x, r)$ to the receiver who verifies its correctness: $\mathsf{Ver}(\mathsf{Com}_x, x) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy two properties: (a) *hiding*: it is infeasible for an adversary (i.e., the receiver) to learn any information about the committed message x, until the commitment Com_x is opened, and (b) *binding*: it is infeasible for an adversary (i.e., the sender) to open a commitment Com_x to different values $x' := (x', r')$ than that was used in the commit phase, i.e., infeasible to find x', *s.t.* $\mathsf{Ver}(\mathsf{Com}_x, x) = \mathsf{Ver}(\mathsf{Com}_x, x') = 1$, where $x \neq x'$.

There exist efficient non-interactive commitment schemes both in (a) the standard model, e.g., Pedersen scheme [22], and (b) the random oracle model using the well-known hash-based scheme such that committing is: $\mathtt{G}(x||r) = \mathsf{Com}_x$ and $\mathsf{Ver}(\mathsf{Com}_x, x)$ requires checking: $\mathtt{G}(x||r) \stackrel{?}{=} \mathsf{Com}_x$, where $\mathtt{G} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a collision-resistant hash function; i.e., the probability to find x and x' such that $\mathtt{G}(x) = \mathtt{G}(x')$ is negligible in the security parameter, λ.

3.2 Smart Contract

A smart contract is a computer program/code. It encodes the terms and conditions of an agreement between parties and often contains a set of variables and functions. A smart contract code is stored on a blockchain and is maintained by the miners who maintain the blockchain. When (a function of) a smart contract is triggered by an external party, every miner executes the smart contract's code. Ethereum [30] has been the most predominant cryptocurrency framework that lets users define arbitrary contracts.

3.3 Counter Collusion Smart Contracts

To enable a party, e.g., a client, to efficiently delegate a computation to a couple of potentially colluding third parties (e.g., servers), Dong *et al.* [13] proposed two main smart contracts, "Prisoner's Contract" ($\mathcal{SC}_{\text{PC}}$) and "Traitor's Contract" ($\mathcal{SC}_{\text{TC}}$). $\mathcal{SC}_{\text{PC}}$ is signed by the client and the servers. This contract tries to incentivise correct computation by using the following idea. It requires each server to pay a deposit before the computation is delegated. It is equipped with an external auditor that is invoked to detect a misbehaving server when the servers provide non-equal results.

If a server behaves honestly, then it can withdraw its deposit. If a cheating server is detected by the auditor, then its deposit is transferred to the client. If one of the servers is honest and the other one cheats, then the honest server receives a reward taken from the cheating server's deposit. However, the dilemma, created by $\mathcal{SC}_{\text{PC}}$ between the two servers, can be addressed if they can make an enforceable promise, for instance via a "Colluder's Contract" ($\mathcal{SC}_{\text{CC}}$), in which one party, called "ringleader", would pay its

counterparty a bribe if both follow the collusion and provide an incorrect computation to $\mathcal{SC}_{\text{PC}}$. To counter $\mathcal{SC}_{\text{CC}}$, Dong *et al.* proposed $\mathcal{SC}_{\text{TC}}$, that incentivises a colluding server to betray the other server and report the collusion without being penalised by $\mathcal{SC}_{\text{PC}}$.

4 The Attack

In this section, we will explain the attack that can be launched against OpenSquare. In brief, this attack allows misbehaving parties to receive rewards without actually performing the task of sequential squaring. As we will demonstrate, the flaw (i.e., failing to take consider the possibility of parties colluding with each other, especially during the verification phase) in this scheme creates a financial incentive for parties to collude and neglect the verification of a solution's correctness.

4.1 Overview of the OpenSquare

OpenSquare aims to enable a client to delegate the computation of sequential squaring to a set of servers/solvers. This is because the modular exponentiation (for instance conducted in the original RSA-based TLP [25]) is resource-demanding and a client with limited resources may not be able to meet the demand. There are three types of entities involved in OpenSquare; namely, (i) a client: who is willing to pay for a valid solution (i.e., a predefined number of squaring), (ii) a set of servers: who produce solutions in order to receive rewards, and (iii) a set of verifiers: who check solutions validity. In this scheme, the client is assumed to be fully trusted. However, any subset of the servers and verifiers can be rational adversaries. Briefly, the scheme works as follows:

1. The client sends the details of a sequential squaring operation to a smart contract and simultaneously deposits a specified amount into the same contract.
2. The servers locally perform the sequential squaring. Each server posts to the smart contract both the commitment of the result, which has been watermarked, and proof of the result's correctness, along with the commitment of an asking price. Additionally, each server deposits a certain amount of coins to the smart contract.
3. The verifiers locally check the proof's validity. This is done because the smart contract side verification is computationally expensive. If a verifier concludes that a proof is invalid, then it (i) deposits a certain amount of coins in the smart contract and (ii) sends a complaint to it. The verifiers must send their complaints within a pre-defined period of time.
4. If the smart contract receives a complaint, then it verifies the proof itself (using the expensive verification process). If it concludes that the proof is indeed invalid, the smart contract rewards the verifier and returns its deposit back.
5. If the smart contract does not receive a complaint before a pre-defined time, then it will pay the server(s) with the lowest asking price and return their deposit.

4.2 Description of the Attack

The idea behind the attack is that the servers and verifiers collude with each other and provide arbitrary values (i.e., in place of a valid result and proof) to the smart contract.

However, none of them sends any complaints to the smart contract. After the pre-defined time elapses, the contract sends the reward to them who share it among themselves. Specifically, this attack operates as follows:

1. The verifiers (e.g., the servers or other users) collude with each other and create an enforceable collusion/promise contract. To achieve this, they develop a smart contract, let us call it U, to which the users/servers deposit a certain amount of coins (greater than the amount of reward proposed by the client) and agree to send an arbitrary incorrect result to D; otherwise, they will forfeit their deposit.
   This type of smart contract, which enforces collusion, has been referred to as the "colluder's contract" in [13].
2. The servers send an invalid proof and computation result (e.g., dummy values) to D and ask it to send the reward to U.
3. None of them send any complaints to D.
4. Since no one complained, D rewards one of the servers (which offered the lowest price) by sending a certain amount of coins to U.
5. U distributes the reward among the colluding parties and refunds their deposit.

4.3 Colluders' Payoffs Analysis

Now, we discuss why collusion leads to a higher payoff than not colluding and following the protocol. By definition, the computation resources and time required to perform the sequential squaring are greater than those needed to perform the verification. Accordingly, the reward for the former task is higher than for the latter. Additionally, the scheme does not guarantee that a solver who computes a correct result and provides valid proof will always be rewarded, as other servers may do the same but offer a lower price. In this case, the effort of the former server goes to waste.

Considering these facts and the parties' enforceable promise through the collusion contract, we will discuss the circumstances under which collusion becomes the dominant strategy, resulting in a higher payoff for each of them. For $task_i$, let m_i be the total number of colluding parties, $rew_i^{(squ)}$ be the reward for sequential squaring, and $rew_i^{(com)}$ be the reward for sending a valid complaint. Then, from the perspective of:

- *a verifier*: as long as $\frac{rew_i^{(squ)}}{m_i} > rew_i^{(com)}$, colluding yields a higher payoff than not colluding and sending a complaint.
- *a solver*: although colluding with others and not performing the squaring enables a solver to earn fewer rewards than the original reward offered by the client, collusion does guarantee a payoff and can surpass the original reward if done multiple times with different clients and/or puzzles. More precisely: $\exists(x, i), \sum_{j=1}^{x} \frac{rew_j^{(squ)}}{m_j} \geq rew_i^{(squ)}$, where x is the total number of tasks offered and $i \in [1, x]$.

4.4 Consequences of the Attack

In [27, p. 1] it is stated that "OpenSqaure: (1) incentivises servers to stay available with their services, (2) minimizes the cost of outsourcing for the client, and (3) ensures

the client receives the valid computational result with high probability". However, our attack demonstrates that (i) the former property cannot be met since the scheme does not incentivise rational servers to offer their computational services (instead, it incentivises them to exploit the scheme), and (ii) the latter property, which is the core of the scheme, cannot be satisfied either, as the client must pay without receiving a valid result.

4.5 Naive Rectifications

One may be tempted to rectify the issue by assuming either (a) one of the verifiers is honest or (b) the (honest) client itself verifies the proof. However, we argue that this strong assumption trivialises the OpenSquare design. Below, we elaborate on that.

Assumption 1: There Exist Honest Third-Party Verifiers. There will be two cases under this assumption.

Case 1: The identities of honest verifiers are known in advance. In this case, the scheme does not need to involve (i) the smart contract (and its related verification mechanism), (ii) the watermarking mechanism, (iii) the deposit paradigm, and (iv) the commitment scheme anymore. Instead, the client sends its coins to the honest verifier. The servers also send their solutions, related proofs, and asking price in plaintext directly to the honest verifier who checks the proof and pays the server(s) which provided valid proofs and offered the lowest price.

Case 2: The identities of honest verifiers are not known in advance. In this case, the flawed scheme may run for a certain number of times, and those verifiers who have produced valid complaints are identified by the smart contract. Consequently, the scheme will be reduced and trivialised to the aforementioned Case 1.

Assumption 2: Honest Client Acts as the Verifier. This case even more trivialises the solution (compared to the cases under Assumption 1), as it falls in the server-client setting, where the server directly sends the result, proof, and asking price to the client which pays the server(s) that provided valid proofs and offered the lowest price.

4.6 Candidate Mitigation

Mitigation 1: Disincentivising Collusion. As we discussed in Section 3.3, Dong *et al.* in [13] proposed a counter-collusion mechanism (consisting of two smart contracts, $\mathcal{SC}_{\text{PC}}$ and $\mathcal{SC}_{\text{TC}}$) that allows a client to outsource resource-intensive computations to a couple of servers, who are potentially rational and may collude with each other. Therefore, the counter-collusion mechanism presents a potential solution to address the attack identified in OpenSquare. In the remainder of this section, we elucidate the utilisation of this mechanism within the OpenSquare framework. For more comprehensive information, interested readers are encouraged to consult [13].

First, all servers and verifiers need to sign $\mathcal{SC}_{\text{PC}}$ and deposit a predetermined amount of coins into $\mathcal{SC}_{\text{PC}}$. The client also deposits into $\mathcal{SC}_{\text{PC}}$ (rather than OpenSquare's smart contract) the amount it wants to pay to the honest servers and verifier.

In the event that one of the colluding parties intends to betray the others, it must engage with the client in a signing process of $\mathcal{SC}_{\text{TC}}$, requiring the client to deposit a specific amount of coins into $\mathcal{SC}_{\text{TC}}$. The betraying party is also obligated to make a deposit into $\mathcal{SC}_{\text{TC}}$. Furthermore, the betraying party needs to provide a "correct result" to $\mathcal{SC}_{\text{TC}}$. In cases where the betraying party is a verifier, the correct result pertains to the verification output. However, if the betraying party is a server, the correct result corresponds to the output of the repeated modular squaring.

When no verifier sends any valid complaint or no party betrays other parties, then OpenSquare operates as before and honest servers are paid from the client's deposit. Moreover, when a verifier sends a valid complaint, then OpenSquare operates as before and pays only the honest verifier. In the event of an inconsistency in the result or an act of betrayal, the counter-collusion mechanism's auditor is summoned to detect and penalise the parties involved in misconduct. The original counter-collusion mechanism may need slight adjustments to align with the setting of OpenSquare. This is because the original mechanism, along with its game-theoretic analysis, was initially designed for a two-server setting, whereas OpenSquare operates within a multi-party environment.

Mitigation 2: Enabling the Smart Contract to Always Check Solutions' Correctness. OpenSquare delegates the verification task to third-party verifiers due to the computational expense associated with verification itself, which incurs a substantial financial cost when executed by the smart contract. Nonetheless, OpenSquare has the potential to benefit from the efficient verification mechanisms employed by the schemes described in [3,1]. These schemes' verification algorithms use an efficient hash-based commitment scheme (as presented in Section 3.1) and do not require modular exponentiation. In such a scenario, OpenSquare would not rely on external verifiers; instead, the smart contract could handle verification once a solution is provided. This verification may require further analysis to ensure that other features of OpenSquare, such as unlinkability, are also upheld.

5 The Response of the OpenSquare's Authors

We reached out to the authors of OpenSquare and forwarded them a copy of the attack description. In the interest of fairness and with the authors' consent, we have included their responses in this section. As there have been several rounds of communication between us, we have organised all communications in chronological order.

- Our Comment #1: We noticed an issue/attack in the OpenSquare scheme and we highlighted it in our paper. Can we have your opinion on the issue?
- Their Response #1: *In OpenSquare, we do not explicitly have a separate verifier set. Rather anyone in the decentralised network can act as a verifier. This includes servers, regular nodes, or the client who requested the puzzle himself. It is in their financial interest to act as a verifier and complain about incorrect solutions. Therefore, assuming rational players, there will be at least one online verifier who will make the complaint. When we wrote the paper, we let the client himself be this complainant. Another salient assumption we make is the liveness of the network.*

Therefore, any user wanting to post a transaction on the blockchain can do so within a bounded time. Therefore, a verifier's complaint will go to the OpenSquare contract within the complaint phase. Now contrary to Assumption 4.5 that you have in Section 4.5, the client is not honest, but rather mutually distrustful and rational. We do not consider him to be blatantly irrational and assume he does not go out of the way to harm other users even when his own utility is hurt.

- Our Comment #2: As we discussed in Section 4.5, the **attack works for any relational verifiers unless we assume the client always acts as the verifier**. However, under this assumption (that the client is [not fully trusted and] always involved in the verification), the solution in OpenSquare boils down to the solution proposed in Section 4 "Contingent Service Payment" in [11] (published at ACM CCS 2017). Furthermore, **assuming the client is rational (not trusted) creates an opportunity for a new attack**, as I will explain. In OpenSquare, the client itself generates the parameters for squaring and posts them on the smart contract. The solvers/servers do not check these parameters. In the OpenSquare paper, there exists no mechanism that allows the solvers/servers to check these parameters. Ill-formed parameters can prevent honest solvers from generating the correct result on time. This means, one of the servers (say server A) can act as a client to generate ill-formed parameters and put that on the smart contract. This can increase the rational server A's payoff because other solvers allocate resources to compute a correct result but they never manage to do that, because the parameters have been ill-formed (e.g., they have been set up to be solved in 10 years' time). But server A can work on the request posted by other clients and get the reward for that.
- Their Response #2: *Recall that the request is essentially to perform T sequential squarings from a starting value g in the modular group. This is strictly a deterministic computation. However, if the group parameters are ill-formed, it could potentially lead to some targeted DoS attacks, as we may not be able to guarantee anything about the copy-prevention of the solutions. This is just intuition, but it has to be formally checked if this indeed is possible for the Wesolowski VDF. But one could quickly fix this problem if the client adds proof that the group parameters were generated correctly. Concretely, in the RSA case, the client has to add proof that the N is a valid RSA modulus. This will not affect the cost of OpenSquare, as the contract does not do the verification.*
  Note that the problem does not arise for the unlinkable extension of OpenSquare as we have a public HTLP setup available. Regarding point 1, Campanelli et al.'s work [11] does not exactly fit our setting. Recall that in OpenSquare, the client is the first one to make a move and never interacts with the servers again. To our understanding, what they have in Section 4 does not fit as there is interaction between A and B, or it's the seller making the first move, or the verification is private.

6 Discussion on OpenSquare's Authors' Reply

According to the OpenSquare's authors' response:

– *"any party can be a verifier"*. We argue (as discussed in Section 4.2 and comment #2 in Section 5) that this is not the case. Because our attack works for any relational

verifiers unless we assume the client always acts as the verifier. This is an important and **strong assumption** which has **not been stated in the paper**. This assumption itself must rely on one of the following assumptions: (i) the client is fully honest or (ii) the client is rational. As we discussed in Section 4.5, in the former case, the solution offered by OpenSquare can be trivialised. However, OpenSquare cannot deal with the latter case, as it requires further proving and verification algorithms (as we will discuss shortly), which are **not present in the OpenSquare paper**.

- *"the client can be a rational adversary"*. We argue that (as discussed in comment #2 in Section 5) assuming the client is rational creates an opportunity for a new attack which OpenSquare cannot deal with.
- *"additional proving and verification mechanisms must be in place to ensure a rational client cannot provide ill-formed parameters to the servers"*. We highlight that the OpenSquare paper does not offer these additional mechanisms. Furthermore, even if OpenSquare were to offer these mechanisms (to deal with a rational client), then (i) it would still require the client to consistently verify the correctness of solutions provided by the servers, which contradicts OpenSquare's initial assumption that any party can be a verifier, and (ii) the service offered by OpenSquare could easily be provided by the "Contingent Service Payment (CSP)" method in [11], which was published four years prior to the OpenSquare paper. Shortly, we will delve into further discussion on the latter point.
- *"the work of Campanelli et al. [11] does not exactly fit OpenSquare's setting"*. We argue that this is not the case and CSP in [11] fits the OpenSquare setting, because (i) in both settings, the client as a verifier must check the solutions provided by the servers and (ii) in the CSP the (zero-knowledge) proving and verification algorithms can be made non-interactive by relying on Fiat-Shamir heuristic [14]. Moreover, CSP works for publicly verifiable schemes as well.

7 Future Work

Recently, researchers introduced a "payment channel" for the "Monero" cryptocurrency, in [28]. This payment channel leverages OpenSquare service, enabling a party to outsource the task of opening "verifiable timed linkable ring signatures" commitments to servers. Consequently, this approach empowers the party to effectively manage numerous Monero payment channels concurrently, removing the limitation imposed by its computational capacity. To further enhance the understanding of this proposition, future research could delve into exploring the potential impacts of our findings on the security aspects of the proposed payment channel.

Acknowledgements

Aydin Abadi was supported in part by REPHRAIN: The National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online, under UKRI grant: EP/V011189/1. Steven J. Murdoch was supported by REPHRAIN and The Royal Society under grant UF160505. We would like to thank Dan Ristea for his comments.

References

1. Abadi, A., Kiayias, A.: Multi-instance publicly verifiable time-lock puzzle and its applications. In: FC (2021)
2. Abadi, A., Murdoch, S.J., Zacharias, T.: Recurring contingent service payment. In: IEEE EuroS&P (2023)
3. Abadi, A., Ristea, D., Murdoch, S.J.: Delegated time-lock puzzle. arXiv preprint arXiv:2308.01280 (2023)
4. Abadi, A., Terzis, S., Dong, C.: VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In: FC (2016)
5. Ateniese, G., Chen, L., Etemad, M., Tang, Q.: Proof of storage-time: Efficiently checking continuous data availability. In: NDSS. The Internet Society (2020)
6. Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: ACM STOC (1991)
7. Baum, C., David, B., Dowsley, R., Nielsen, J.B., Oechsner, S.: TARDIS: A foundation of time-lock puzzles in UC. In: EUROCRYPT (2021)
8. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO'18
9. Boneh, D., Bünz, B., Fisch, B.: A survey of two verifiable delay functions. IACR Cryptol. ePrint Arch. (2018)
10. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000
11. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS'17
12. Chen, H., Deviani, R.: A secure e-voting system based on RSA time-lock puzzle mechanism. In: BWCCA'12 ,
13. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In: CCS (2017)
14. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO (1986)
15. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02
16. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: CRYPTO (2010)
17. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: ACM STOC (2008)
18. Kavousi, A., Abadi, A., Jovanovic, P.: Timed secret sharing. Cryptology ePrint Archive (2023)
19. Kiayias, A., Kohlweiss, M., Sarencheh, A.: Peredi: Privacy-enhanced, regulated and distributed central bank digital currencies. In: ACM CCS (2022)
20. Malavolta, G., Thyagarajan, S.A.K.: Homomorphic time-lock puzzles and applications. In: CRYPTO'19
21. May, T.C.: Timed-release crypto (1993), `https://cypherpunks.venona.com/date/1993/02/msg00129.html`
22. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91
23. Pietrzak, K.: Simple verifiable delay functions. In: 10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
24. Ren, Y., Ding, N., Zhang, X., Lu, H., Gu, D.: Verifiable outsourcing algorithms for modular exponentiations with improved checkability. In: ACM AsiaCCS (2016)

25. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep. (1996)
26. Shacham, H., Waters, B.: Compact proofs of retrievability. In: ASIACRYPT. pp. 90–107 (2008)
27. Thyagarajan, S.A.K., Gong, T., Bhat, A., Kate, A., Schröder, D.: Opensquare: Decentralized repeated modular squaring service. In: CCS (2021)
28. Thyagarajan, S.A.K., Malavolta, G., Schmid, F., Schröder, D.: Verifiable timed linkable ring signatures for scalable payments for monero. In: ESORICS (2022)
29. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT'19
30. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014)
31. Wu, Q., Xi, L., Wang, S., Ji, S., Wang, S., Ren, Y.: Verifiable delay function and its blockchain-related application: A survey. Sensors (2022)