Robust Publicly Verifiable Covert Security: Limited Information Leakage and Guaranteed Correctness with Low Overhead

Yi Liu[1], Junzuo Lai[1], Qi Wang[2], Xianrui Qin[3], Anjia Yang[1], and Jian Weng[1]

[1] College of Cyber Security, Jinan University, Guangzhou 510632, China
liuyi@jnu.edu.com, laijunzuo@gmail.com, anjiayang@gmail.com, cryptjweng@gmail.com
[2] Department of Computer Science and Engineering & National Center for Applied Mathematics Shenzhen, Southern University of Science and Technology, Shenzhen 518055, China
wangqi@sustech.edu.cn
[3] Department of Computer Science, The University of Hong Kong, Hong Kong SAR, China
xrqin@cs.hku.hk

Abstract. Protocols with *publicly verifiable covert (PVC) security* offer high efficiency and an appealing feature: a covert party may deviate from the protocol, but with a probability (*e.g.*, 90%, referred to as the *deterrence factor*), the honest party can identify this deviation and expose it using a publicly verifiable certificate. These protocols are particularly suitable for practical applications involving reputation-conscious parties.

However, in the cases where misbehavior goes undetected (*e.g.*, with a probability of 10%), *no security guarantee is provided for the honest party*, potentially resulting in a complete loss of input privacy and output correctness.

In this paper, we tackle this critical problem by presenting a highly effective solution. We introduce and formally define an enhanced notion called *robust PVC security*, such that even if the misbehavior remains undetected, the malicious party can only gain an additional 1-bit of information about the honest party's input while maintaining the correctness of the output. We propose a novel approach leveraging *dual execution* and *time-lock puzzles* to design a robust PVC-secure two-party protocol with *low overhead* (depending on the deterrence factor). For instance, with a deterrence factor of 90%, our robust PVC-secure protocol incurs *only additional* $\sim$10% *overhead* compared to the state-of-the-art PVC-secure protocol.

Given the stronger security guarantees with low overhead, our protocol is highly suitable for practical applications of secure two-party computation.

Keywords: Secure two-party computation · Robust publicly verifiable covert security · 1-bit leakage · Dual execution.

1 Introduction

Secure two-party computation (2PC) allows two mutually distrusted parties to jointly evaluate a common function on their inputs while maintaining input privacy. Traditionally, two main security notions for 2PC, *i.e.*, *semi-honest security* and *malicious security*, have been considered [12]. Protocols with *semi-honest security* could be efficient but only protect against passive attackers who strictly adhere to the prescribed protocols. Alternatively, protocols with *malicious security* provide a much stronger guarantee, preventing attackers from gaining any advantage through deviations from protocols. However, despite progress in the past few years, protocols with malicious security remain significantly complex and incur high overhead compared to those with semi-honest security.

The notion of *covert security* [4] is thereby introduced to serve as a compromise between semi-honest and malicious security. Covert security ensures that a party deviating from the protocol will be caught by the honest party with a fixed probability ϵ (*e.g.*, $\epsilon = 90\%$), referred to as the *deterrence factor*. Achieving covert security entails significantly lower overhead than malicious security [4,14,8,22]. Meanwhile, covert security provides a stronger security guarantee than semi-honest security, as it incorporates the risk of being caught, which can serve as a deterrent to potential cheaters.

Nevertheless, in certain scenarios, the deterrent effect of covert security may be insufficient. Merely catching the cheater does not enable the honest party to effectively persuade others and accuse the

cheater, as the cheater can still deny their misconduct. To address this limitation, Asharov and Orlandi [2] introduced an enhanced notion called *publicly verifiable covert (PVC) security*. Protocols with PVC security allow the honest party to generate a *publicly verifiable certificate* when cheating is detected. The certificate serves as proof of cheating and can be used to convince all other parties, including external entities. The certificate can be utilized for legal proceedings or incorporated into a smart contract that automatically enforces financial penalties against the cheater. This property is particularly compelling in practice, as it is efficient[4] and can expose the cheater's misbehavior publicly and permanently, thereby imposing a reputational risk and providing stronger accountability measures. In recent years, PVC security has garnered substantial attention, resulting in the development of protocols for both general two-party [2,20,17] and multi-party [9,31,10,3] computation. Furthermore, PVC security model is also widely used in specific scenarios, such as financially backed protocols based on blockchain [33,11], secure computation concerning commitment based on blockchain [1], and private function evaluation [24].

While PVC security offers an effective deterrent when cheating is detected, there is a critical issue when misbehavior remains undetected (e.g., with a probability of 10%). In such cases, *the honest party is left without any security guarantees*, potentially resulting in a complete loss of input privacy and output correctness. In other words, the cheater may gain complete knowledge of the honest party's input and manipulate the output. This is unacceptable in many scenarios, particularly for parties less concerned about reputation, who may be more inclined to take risks in pursuit of significant gains. An existing countermeasure to address this problem is to increase the deterrence factor to a relatively high level (*e.g.*, 99% or 99.9%). However, it should be noted that the efficiency of protocols with PVC security is directly related to the deterrence factor, and increasing the deterrence factor comes at the cost of reduced efficiency. Worse, existing protocols with PVC security exhibit diminishing marginal benefits when increasing the deterrence factor. For instance, increasing the deterrence factor from 90% to 99% leads to a *tenfold* increase in protocol execution cost, and the same holds for increasing from 99% to 99.9%.

Therefore, there is still a lack of effective approaches to prevent reputation-insensitive parties from deviating from the protocol without incurring significant overhead.

1.1 Our Contributions

In this paper, we present a compelling countermeasure to address the aforementioned problem. Specifically:

New security notion. We introduce an enhanced notion for PVC security named *robust PVC security* to capture the goal. This notion can be seen as making up for the deficiency of covert security in the other direction, focusing on reducing the benefits of malicious behavior when it goes undetected while maintaining the increased cost for cheater when caught provided by PVC security. Protocols with robust PVC security ensure that *even if the misbehavior remains undetected, the malicious party can only obtain an additional* 1*-bit of information about the honest party's input while simultaneously preserving the correctness of the protocol's output.* By significantly reducing the benefits of successful cheating, our approach goes further to effectively discourages malicious parties from cheating in the protocol.

Protocol with low overhead. We propose a novel approach leveraging *dual execution* and *time-lock puzzles* to design a general *robust PVC-secure* two-party computation protocol with *low overhead* (depending on the deterrence factor). For instance, when the deterrence factor is 90%, our protocol will incur *only additional* $\sim$10% *overhead compared to the state-of-the-art PVC-secure protocol.*

With its stronger security guarantees and low overhead, our protocol is highly suitable for practical applications of secure two-party computation.

1.2 Technical Overview

In this subsection, we commence by reviewing the *derandomization technique*, which has been employed in prior works for the design of PVC protocols. Notably, this technique will also serve as a key

[4] It has been shown that a two-party PVC protocol with deterrence factor 50% incurs only $20-40$% overhead compared to the state-of-the-art semi-honest protocols based on garbled circuits [17].

component in our protocol. Then, we briefly explain our robust PVC security notion and present the basic idea for designing protocols with robust PVC security. Finally, we discuss the main intuition behind our novel approach to achieving robust PVC security with low overhead.

Derandomization in PVC Security Recent general secure computation protocols with PVC security employ the *cut-and-choose* paradigm along with the derandomization technique [17,9,31,10,3], which is simpler and more efficient compared to the signed-OT technique used in earlier work [2,20].

Specifically, in the state-of-the-art PVC-secure two-party computation protocol [17] based on garbled circuits, each party pre-selects a seed for each instance. These seeds determine the randomness used in specific instances, ensuring that the execution of each instance is fully determined by the protocol description and the parties' seeds. At the start of the protocol, the evaluator, who cannot cheat in garbled circuit evaluation, commits to its own seeds and selects one instance for evaluation, leaving the remaining instances for checking. Subsequently, the evaluator, acting as the receiver, engages in classical oblivious transfer (OT) protocols with the garbler to obtain the garbler's seeds for the checking instances. Additionally, the parties sign the transcripts for each instance. Throughout the protocol execution, the evaluator uses the garbler's seeds to verify the correctness of the messages received from the garbler for the checking instances. Upon detecting a deviation from the checking instances, the evaluator combines the previously signed messages and its own seed decommitment to generate the certificate. Any other party can use the seeds to simulate the protocol execution and verify deviations. If no deviation is detected, the evaluator proceeds to evaluate the garbled circuit generated in the evaluation instance.

Importantly, in protocols with PVC security, potential cheaters are unaware of which instances are being checked, making it impossible for them to prevent honest parties from generating certificates. The deterrence factor in these protocols is determined by the number of instances. For example, in the state-of-the-art PVC-secure two-party computation protocol [17], λ instances are involved, with $\lambda - 1$ instances checked by the evaluator and the remaining one instance used for evaluation. Hence, the deterrence factor of this protocol is $\frac{\lambda-1}{\lambda}$. This is the reason why existing protocols exhibit diminishing marginal benefits when increasing the deterrence factor.

Robust PVC Security In this paper, we introduce a new notion called robust PVC security. Protocols with robust PVC security aim to provide security guarantees inherited from PVC security while simultaneously limiting the benefits for cheaters when their deviations go undetected. We require that in such protocols, cheaters can obtain *at most an additional* 1*-bit of information about the honest party's input*, while *ensuring the correctness of the protocol output*, *i.e.*, the output remains untampered.

The term "1-bit information" is used here because our approach leverages the idea of *dual execution*. The dual execution technique was initially introduced by Mohassel and Franklin [27] and later formalized by Huang, Katz, and Evans [18]. This technique has found broad applications in general secure two-party computation with malicious and covert security [19,28,21,29,16] as well as private set intersection (PSI) [30].

The dual execution technique is primarily employed in garbled circuits. The idea behind dual execution is that two parties, $\mathsf{P_A}$ and $\mathsf{P_B}$, execute two protocols for the same evaluation circuit and inputs. In one protocol, $\mathsf{P_A}$ acts as the garbler, and $\mathsf{P_B}$ acts as the evaluator, while in the other protocol, they switch roles, i.e., $\mathsf{P_B}$ becomes the garbler, and $\mathsf{P_A}$ becomes the evaluator. Finally, the two parties run a secure *equality test* protocol on the outputs of the two garbled circuits to determine the final result. If the test passes, both parties obtain the correct evaluation result; otherwise, it indicates that one party deviated from the protocol, leading to termination. By utilizing this dual execution technique, the malicious party can obtain, at most, an additional 1-bit of information about the honest party's input. The intuition is that even if the malicious party deviates from the protocol as the garbler, it cannot cheat in garbled circuit evaluation as the evaluator. Therefore, the information it can obtain from the maliciously generated garbled circuit is limited to the result of the equality test, *i.e.*, the 1-bit information of either true or false, which may depend on the honest party's inputs.

Therefore, a straightforward approach to achieving robust PVC security is by integrating dual execution into PVC-secure protocols. Specifically, two parties $\mathsf{P_A}$ and $\mathsf{P_B}$ can execute a PVC-secure protocol, such as the one proposed in [17], to perform garbled circuit checking and obtain a final garbled circuit for evaluation. Simultaneously, these two parties switch roles and execute the protocol

again to obtain another garbled circuit for evaluation. Then, each party plays the role of the evaluator and evaluates the garbled circuit they have chosen. Finally, a secure equality test protocol is employed to determine the final result. Although this approach may seem viable, we propose a superior solution in this work.

Our Novel Approach If the aforementioned approach is based on the state-of-the-art PVC-secure protocol in [17], achieving robust PVC security with a deterrence factor of $\frac{\lambda-1}{\lambda}$ would require the protocol to generate 2λ garbled circuits. Specifically, each party would need to generate $2\lambda - 1$ garbled circuits and evaluate one garbled circuit. This includes λ circuits generated as the garbler, $\lambda - 1$ garbled circuits generated for simulating garbled circuit generation in instance checking, and one garbled circuit that is not generated but needs to be evaluated. Therefore, achieving robust PVC security based on a PVC-secure protocol incurs a cost that is roughly double that of the PVC-secure protocol itself. As emphasized in [17], the cost of generating garbled circuits (unless the circuit is very small) is the efficiency bottleneck of the protocol. Therefore, this approach is not entirely satisfactory.

It happens that this cost can be significantly reduced by leveraging the special protocol framework when integrating dual execution into a PVC-secure protocol. The idea behind our novel approach stems from the following question:

> *Must each party generate λ garbled circuits as the garbler while generating $\lambda - 1$ different garbled circuits as the evaluator for circuit checking?*

Fortunately, we found that the answer is NO, and parties can combine these two circuit generation processes together. The idea of our approach is to let the two parties jointly select $\lambda + 1$ seeds in a blind fashion. Subsequently, each party randomly obtains λ seeds, while one seed remains hidden. When the sets of seeds obtained by the two parties differ, the intersection of their seeds is of size $\lambda - 1$. Importantly, each party remains unaware of the seeds obtained by the other party. They can use their own λ seeds to generate λ garbled circuits and an additional *dummy garbled circuit* for the hidden seed. Since the parties share $\lambda - 1$ common seeds, they can now *reuse* the materials computed for circuit generation when playing the role of the evaluator to perform the circuit checking.

In summary, there are $\lambda + 1$ seeds, and each party obtains λ seeds, resulting in a seed intersection of size $\lambda - 1$. Using their respective λ seeds, the parties generate λ garbled circuits, where $\lambda - 1$ garbled circuits (derived from seeds in the seed intersection) are identical for both parties. Each party can then reuse the materials generated from these common $\lambda - 1$ seeds to check the $\lambda - 1$ garbled circuits generated by the other party. Consequently, each party is left with one unchecked garbled circuit, which can be used for dual execution.

With this insight, to achieve robust PVC security with a deterrence factor of $\frac{\lambda-1}{\lambda}$, each party only needs to generate λ garbled circuits and evaluate one garbled circuit. Notably, the checking of $\lambda - 1$ garbled circuits generated by the other party no longer requires garbled circuit generation. In comparison to the state-of-the-art protocol with PVC security [17], where a total of $2\lambda - 1$ garbled circuits are generated by both parties, and one is evaluated, our approach requires only one additional garbled circuit generation and one additional garbled circuit evaluation. For instance, for the protocol in [17], achieving a deterrence factor of 90% necessitates the garbler to generate 10 garbled circuits, while the evaluator needs to perform 9 garbled circuit generations and one garbled circuit evaluation. In contrast, our approach requires each party to generate 10 garbled circuits and evaluate one garbled circuit. That is, the protocol in [17] requires 20 garbled circuit generations and evaluations, whereas our approach requires 22, and thus the additional overhead in our approach is less than 10% (given that the cost of evaluations is lower than that of generations). Furthermore, the computation of circuit checking for the protocol in [17] must be conducted after circuit generation, while this computation is already performed during the garbled circuit generation in our approach. Thus, as garbled circuit generations and evaluations for each party can be executed in parallel in our approach, the running time of our approach may outperform the PVC protocol in [17].

However, the realization of this insight is highly non-trivial. We omit intricate details and provide a brief illustration of the idea behind our approach as follows. The detailed description of our protocol can be found in Section 4.

To generate $\lambda + 1$ seeds, two parties, denoted as $\mathsf{P_A}$ and $\mathsf{P_B}$, can each select $\lambda + 1$ seed shares s_i^A and s_i^B, respectively. Subsequently, each party engages in OT protocols with the other party in two directions. As the sender, each party inputs their seed shares, while as the receiver, each party

retrieves λ shares, leaving one share unretrieved. To ensure that each party retrieves all-but-one shares, we incorporate random values called *witnesses* as input in the OT protocols. If a receiving party does not retrieve a share, this party must retrieve the corresponding witness. The retrieved shares and witnesses should be provided by the receiver and verified by the sender later to continue the execution of the protocol. The seeds are then defined as $s_i = s_i^A \oplus s_i^B$, and each party will derive λ seeds. A secure equality test is performed by the two parties to ensure that the sets of their seeds are distinct, and if they are found to be identical, the two parties restart the protocol. Step 1 in Fig. 1 illustrates the scenario where 6 seeds are generated.

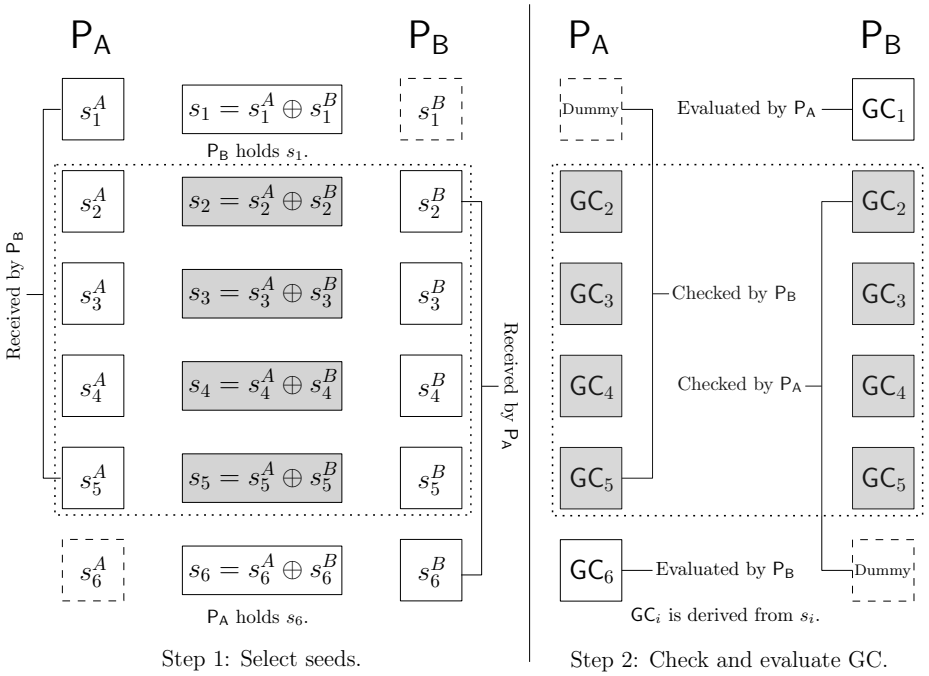

Fig. 1. Seeds generation and garbled circuit evaluation.

Then, each party can utilize the λ seeds they obtained to generate λ garbled circuits. In the case where a party does not obtain a particular seed, a *dummy garbled circuit* is generated in its place. The parties can exchange the commitments of their $\lambda + 1$ garbled circuits, where the randomness of each commitment is also derived from the respective seed. With λ seeds in hand, each party can verify the correct generation of $\lambda - 1$ garbled circuits (from the commitments) while identifying the remaining one as the dummy circuit. Finally, each party can evaluate the garbled circuit corresponding to the seed it does not obtain as dual execution and subsequently perform a secure equality test on the outputs of the garbled circuits to determine the final result. Step 2 in Fig. 1 provides an illustrative depiction of this procedure.

One issue encountered in this approach is the potential identification of the dummy garbled circuit when receiving the commitments. More precisely, a malicious party can discern which instances are being checked by the honest party based on the dummy garbled circuit. If its maliciously generated garbled circuit is checked by the other party, the malicious party can promptly abort and refuse to sign the messages required for generating a publicly verifiable certificate. To address this issue, we must ensure that each party remains unaware of which instances are being checked until the necessary materials for certificate generation are ready.

In our approach, we leverage a verifiable time-lock puzzle scheme [10,25] to tackle this challenge. Using this scheme, the puzzle generator can efficiently create a time-lock puzzle for a message, ensuring that the message remains concealed until a specific time has passed, even against parallel adversaries. Essentially, this efficiently generated time-lock puzzle compels a solver to complete a computational task that takes no less time than the specified duration to recover the message. Moreover, once a puzzle is solved, the solver can effectively convince others that the retrieved message originates from

the puzzle. In addition, we also require the option for the puzzle solver to *open the puzzle* at any time, serving as the commitment scheme.

By employing a verifiable time-lock puzzle scheme, each party can generate puzzles for the commitments of the garbled circuits. Therefore, both parties, unaware of the instances being checked, can generate all the necessary materials (*e.g.*, signatures) for generating a publicly verifiable certificate. Afterwards, both parties open their puzzles, verify the garbled circuits, and continue the protocol's execution. If a party refuses to open its puzzles, the other party can solve the puzzle instead. When both parties are honest, no puzzle solving procedure is involved. In the case of detected deviation, the party solving the puzzle can still generate a publicly verifiable certificate as proof of cheating. Given that the puzzle solver can efficiently convince others that the recovered message is derived from the puzzle, the certificate verification process is also efficient. If no deviation is detected, both parties open the respective garbled circuits for evaluation and employ the dual execution approach to evaluate the garbled circuits and determine the evaluation result.

Indeed, similar to the dummy garbled circuit, there exists a potential identification of the dummy instance in the OT protocols for input-wire labels. The solution is also based on time-lock puzzles, but it is more involved (refer to Section 4 for the detailed solution). Additionally, to improve efficiency, we generate time-lock puzzle for messages in a batched fashion, allowing each party to generate only one *real* time-lock puzzle (see Section 2 and 4 for details).

1.3 Organization

In Section 2, we introduce the notations used in this paper and the building blocks that form the foundation of our protocol. Subsequently, in Section 3, we present the formal definition of robust PVC security. Based on this definition, in Section 4, we propose our robust PVC protocol in detail, leveraging the idea outlined in Section 1.2, and provide the security proof. Finally, we discuss the efficiency and potential enhancements of our protocol in Section 5.

2 Preliminaries

We denote the size of a set S as $|S|$ and use the notation $x \leftarrow\!\!\$\, S$ to represent the uniform sampling of an element x from the set S. Additionally, let $[n] = \{1, \ldots, n\}$ for a positive integer n. For a bit string x, the ith bit of x is denoted by $x[i]$. The function $\mathsf{bin}(\cdot)$ returns the bit representation of the input.

Let κ denote the computational security parameter, which is provided in unary format as input to all algorithms. A function f in κ, mapping natural numbers to $[0, 1]$, is considered *negligible* if $f(\kappa) = \mathcal{O}(\kappa^{-c})$ for every constant $c > 0$. Conversely, a function $1 - f$ is deemed *overwhelming* if f is negligible.

Given a $\mathsf{seed} \in \{0,1\}^\kappa$, we can use a pseudorandom function with seed as the key in the Counter (CTR) mode to derive sufficiently many pseudorandom numbers and use them as random coins for operations in protocols.

We denote a (non-interactive) commitment scheme as Com. The scheme utilizes random coins decom for commitment generation and opening. It should satisfy (computational) binding and hiding properties, while also supporting extraction and equivocation. In our protocol, we implement Com using the random oracle $H\colon \{0,1\}^* \to \{0,1\}^\kappa$ by defining $\mathsf{Com}(m) = H(m, \mathsf{decom})$, where $\mathsf{decom} \leftarrow\!\!\$\, \{0,1\}^\kappa$. We employ the collision-resistant hash function H in the protocol. Our protocol will use the signature scheme $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ that is existentially unforgeable under chosen-message attacks (EUF-CMA).

For an execution transcript of a two-party protocol $\mathsf{trans} = (m_1, m_2, m_3, \ldots)$, where the parties send their messages alternately, the *transcript hash* of this execution is defined as $\mathcal{H} = (\mathsf{H}(m_1), \mathsf{H}(m_2), \mathsf{H}(m_3), \ldots)$.

Let Π_{OT} be the protocol that securely realizes a parallel version of the OT functionality $\mathcal{F}_{\mathsf{OT}}$ below with perfect correctness [9], such that the receiver (resp. sender) cannot "equivocate" its view by finding a random tape that produces a different output (resp. uses a different input) from the one in the real execution.

Functionality $\mathcal{F}_{\mathsf{OT}}$

Private inputs: $\mathsf{P_A}$ has input $x \in \{0,1\}^\lambda$ and $\mathsf{P_B}$ has input $\{(A_{i,0}, A_{i,1})\}_{i\in[\lambda]}$.

Upon receiving $x \in \{0,1\}^\lambda$ from $\mathsf{P_A}$ and $\{(A_{i,0}, A_{i,1})\}_{i\in[\lambda]}$ from $\mathsf{P_B}$, send the selected message $\{A_{i,x[i]}\}_{i\in[\lambda]}$ to $\mathsf{P_A}$.

In our protocol, we will use the protocol Π_{Eq} that securely realizes the two-party equality test [7,18] functionality $\mathcal{F}_{\mathsf{Eq}}$ in the following.

Functionality $\mathcal{F}_{\mathsf{Eq}}$

Private inputs: $\mathsf{P_A}$ has input $x \in \{0,1\}^\kappa$ and $\mathsf{P_B}$ has input $y \in \{0,1\}^\kappa$.

Upon receiving $x \in \{0,1\}^\kappa$ from $\mathsf{P_A}$ and $y \in \{0,1\}^\kappa$ from $\mathsf{P_B}$, let $b = \mathsf{true}$ if $x = y$, and $b = \mathsf{false}$ otherwise.

- If both parties are honest, send b to both parties.
- If a party is corrupted by the adversary, send b to this corrupted party. Then if $\mathsf{continue}$ from the adversary is received, send b to the honest party.

Garbling Scheme Our protocol uses a circuit garbling scheme $(\mathsf{Gb}, \mathsf{Eval})$ as follows.

- The algorithm Gb takes as input the security parameter 1^κ and a circuit $\mathcal{C}$ that has $n = n_A + n_B$ input wires and n_O output wires, and outputs input-wire labels $\{(X_{i,0}, X_{i,1})\}_{i\in[n]}$, a garbled circuit GC, and output-wire labels $\{(Z_{i,0}, Z_{i,1})\}_{i\in[n_O]}$.
- The deterministic algorithm Eval takes as input a set of input-wire labels $\{X_i\}_{i\in[n]}$ and a garbled circuit GC. It outputs a set of output-wire labels $\{Z_i\}_{i\in[n_O]}$.

The correctness of the garbling scheme means that for any circuit $\mathcal{C}$ as above and any input $x \in \{0,1\}^n$, we have

$$\Pr\left[\begin{array}{l}\forall i, Z_i = Z_{i,z[i]} \\ \quad \wedge Z_i \neq Z_{i,1-z[i]}\end{array} : \begin{array}{l}(\{(X_{i,0}, X_{i,1})\}_{i\in[n]}, \mathsf{GC}, \{(Z_{i,0}, Z_{i,1})\}_{i\in[n_O]}) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C}) \\ \{Z_i\} \leftarrow \mathsf{Eval}(\{X_{i,x[i]}, \mathsf{GC}\})\end{array}\right]$$

where $z = \mathcal{C}(x)$, except for a negligible probability. We assume that the garbling scheme satisfies the standard security definition [23,5]. We assume that there is a simulator $\mathcal{S}_{\mathsf{Gb}}$ such that for all $\mathcal{C}$ and x, the distribution $\{\mathcal{S}_{\mathsf{Gb}}(1^\kappa, \mathcal{C}, \mathcal{C}(x))\}$ is computationally indistinguishable from

$$\left\{ \begin{pmatrix} \{X_{i,x[i]}\}_{i\in[n]}, \mathsf{GC}, \\ \{(Z_{i,0}, Z_{i,1})\}_{i\in[n_O]} \end{pmatrix} : (\{(X_{i,0}, X_{i,1})\}_{i\in[n]}, \mathsf{GC}, \{(Z_{i,0}, Z_{i,1})\}_{i\in[n_O]}) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C}) \right\}.$$

Verifiable Time-Lock Puzzle Scheme We use a verifiable time-lock puzzle scheme in our protocol. We restate its definition in [10] with minor modification[5] as follows.

Definition 1. *A verifiable time-lock puzzle scheme* TLP *consisting of algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{Verify}, \mathsf{Verify}')$ *with solution space* $\mathbb{S}$ *is as follows.*

- *The algorithm* Setup *takes as input* 1^κ *and a hardness parameter* τ*, and outputs public parameters* pp*.*
- *The algorithm* Gen *takes as input public parameters* pp *and a solution* $s \in \mathbb{S}$ *and then outputs a puzzle* p*, together with an opening* π*.*
- *The deterministic algorithm* Solve *takes as input public parameters* pp *and a puzzle* p *and then outputs a solution* s *and a proof* π'*.*
- *The deterministic algorithm* Verify *takes as input public parameters* pp*, a puzzle* p*, a solution* s*, and an opening* π *and then outputs* true *if the solution* s *is the valid solution for* p*. Otherwise, the algorithm outputs* false*.*

[5] In particular, we include the opening π (*e.g.*, randomness) as the output of Gen for later puzzle opening. We add the algorithm Verify to verify the opening of a puzzle from the puzzle generator.

– *The deterministic algorithm* Verify' *takes as input public parameters pp, a puzzle* p*, a solution* s*, and a proof* π' *and then outputs* true *if the solution* s *is valid. Otherwise, the algorithm outputs* false*. This algorithm must run in total time polynomial in* κ *and* $\log \tau$*.*

A verifiable time-lock puzzle scheme TLP should satisfy *completeness*, *correctness for opening and proof*, *soundness*, and *security*. We provide the definition of these properties in Supplementary Material A.1 and present a verifiable time-lock puzzle scheme [10,25] in Supplementary Material A.2.

In the description of our protocol, we do not explicitly differentiate between puzzle opening and proof, as well as between the algorithms Verify and Verify'. This is because all of these components enable efficient verification of the correctness of a given solution within the protocol.

It is worth noting that we can use a time-lock puzzle scheme to generate time-lock puzzles in a batched manner. Specifically, we can generate a solution $s \in \mathbb{S}$ and a time-lock puzzle p for s. Then, given messages $\{m_i\}$, we can use s as the seed to derive a sufficient number of (symmetric encryption) keys and randomness and encrypt messages $\{m_i\}$ using keys via IND-CPA (or IND-CCA) encryption schemes with generated randomness. For simplicity, we slightly abuse notation and denote the encryption as $\mathsf{Enc}_s(\cdot)$. The puzzle generator can send the puzzle p and the ciphertexts of m_i's to the puzzle receiver. Note that the ciphertexts of m_i's can also be regarded as time-lock puzzles since the puzzle receiver learns nothing about the solution s before p is solved based on the security of the time-lock puzzle and the encryption scheme[6]. After solving p and obtaining s, the receiver can easily derive the messages $\{m_i\}$. In this approach, we refer to the solution s as the *master puzzle key* and the puzzle p as the *master time-lock puzzle*.

In our protocol, we use s as the seed to derive keys, and then use the random oracle with keys in CTR mode to generate enough pseudorandom padding to encrypt messages $\{m_i\}$. We can easily see that the encryption is IND-CPA-secure before s is derived from p. This scheme allows us to program the random oracle to equivocate the encrypted values in the security proof.

Remark 1. We note that given a ciphertext from the ciphertext space and a key, the decryption of the ciphertext always succeeds (although not necessarily yielding the correct result). This property is solely for the convenience of our protocol description. It is very straightforward to modify our protocol to enable honest parties to generate publicly verifiable certificates for decryption failures and attribute blame to malicious parties responsible for generating invalid ciphertexts [25].

3 Definition of Robust PVC Security

Based on the discussion in Section 1.2, we define the ideal functionality $\mathcal{F}_{\mathsf{RobustPVC}}$ for robust PVC security as follows.

Functionality $\mathcal{F}_{\mathsf{RobustPVC}}$ with deterrence ϵ

Public inputs: Both parties agree on a circuit $\mathcal{C}$, which has $n = n_A + n_B$ input wires and n_O output wires.
Private inputs: $\mathsf{P_A}$ has input $x_A \in \{0,1\}^{n_A}$, whereas the other party $\mathsf{P_B}$ has input $x_B \in \{0,1\}^{n_B}$.

Both parties send their inputs to the ideal functionality. If abort from the party corrupted by the adversary is received, send $\perp$ to both parties and terminate.

- If an input of the form $(\mathsf{cheat}, \hat{\epsilon})$, where $\hat{\epsilon} \geq \epsilon$, from the party corrupted by the adversary is received:
  - With probability $\hat{\epsilon}$, send $(\mathsf{corrupted})$ to both parties and terminate.
  - With probability $1 - \hat{\epsilon}$, send $(\mathsf{undetected})$ to the corrupted party. Then ignore any input of the form $(\mathsf{cheat}, \cdot)$ or $(\mathsf{stupidCheat}, \cdot)$.
- If an input of the form $(\mathsf{stupidCheat}, \hat{\epsilon})$ from the party corrupted by the adversary is received:
  - With probability $\hat{\epsilon}$, send $(\mathsf{corrupted})$ to both parties and terminate.
  - With probability $1 - \hat{\epsilon}$, send $\perp$ to both parties and terminate.
- If input $x_A \in \{0,1\}^{n_A}$ from $\mathsf{P_A}$ and $x_B \in \{0,1\}^{n_B}$ from $\mathsf{P_B}$ has been received:
  1. If both parties are honest, give $C(x_A, x_B)$ to them and terminate.
  2. If any party is corrupted, give $C(x_A, x_B)$ to the corrupted party.

[6] In this paper, we use "ciphertext" and "puzzle" interchangeably for this setting.

3. If a boolean function g from the adversary is received and $\mathsf{undetected}$ has been sent, give $g(x_A, x_B)$ to the adversary. Then if $\mathsf{continue}$ from the adversary is received, give $\mathsf{cheating}$ to the honest party if $g(x_A, x_B) = 0$, else give $C(x_A, x_B)$ to the honest party.

In order to align with the security proof, we introduce slight modifications to the definition inherited from PVC security. In the ideal functionality $\mathcal{F}_{\mathsf{RobustPVC}}$, we permit the adversary to cheat with a probability of being caught $\hat{\epsilon}$ that larger than or equal to the specified deterrence factor ϵ, *i.e.*, $\hat{\epsilon} \geq \epsilon$. It is evident that this modification does not compromise security. Additionally, we allow the adversary to engage in *stupid cheating*. With stupid cheating, we do not impose any restrictions on the probability of remaining undetected. In this scenario, even if the adversary is not detected, the ideal functionality will still abort, rendering the adversary unable to gain any advantage. It is also apparent that this modification does not compromise security.

The definition of robust PVC security is given in the following.

Definition 2. *A two-party protocol $\Pi_{\mathsf{RobustPVC}}$ along with algorithms* Blame *and* Judge *achieves* robust publicly verifiable covert (PVC) security with deterrence ϵ *if the following conditions hold.*

Simulatability *The protocol $\Pi_{\mathsf{RobustPVC}}$, where the honest party might send* cert *to the adversary if cheating is detected, securely realizes $\mathcal{F}_{\mathsf{RobustPVC}}$ with deterrence ϵ.*

Public Verifiability *If the honest party outputs* $\mathsf{corrupted}$ *(together with an output certificate* cert*), then the output of* $\mathsf{Judge}(\mathsf{cert})$ *is* true*, except for a negligible probability.*

Defamation Freeness *For every* PPT *adversary $\mathcal{A}$ corrupting a party, if the other party is honest, the probability that $\mathcal{A}$ generates a certificate* cert^* *that blames the honest party and leads to* $\mathsf{Judge}(\mathsf{cert}^*)$ *outputting* true *is negligible.*

Remark 2. In the definition above and the protocol description in Section 4, it is specified that the honest party sends the certificate to the adversary. This specification in the definition is to ensure that the adversary cannot learn any information about the honest party's input even when the certificate is provided.

Remark 3. We note that public verifiability in the definition also implies that if the honest party outputs $\mathsf{corrupted}$, the malicious party cannot hinder the honest party from generating a valid certificate.

4 Our Robust PVC-Secure Protocol

In this section, we introduce the protocol $\Pi_{\mathsf{RobustPVC}}$ that achieves robust PVC security defined in Section 3. This protocol is based on the idea introduced in Section 1.2.

The protocol description here employs λ instances instead of $\lambda + 1$ in Section 1.2 (thus with deterrence factor $\epsilon = \frac{\lambda-2}{\lambda-1}$) for the sake of notation simplicity. The circuit for $\Pi_{\mathsf{RobustPVC}}$ is denoted by $\mathcal{C}$, consisting of $n = n_A + n_B$ input wires and n_O output wires. The input of $\mathsf{P_A}$ is $x_A \in \{0,1\}^{n_A}$ while the input of $\mathsf{P_B}$ is $x_B \in \{0,1\}^{n_B}$.

Seed preparation. At the beginning of the protocol, each party generates three random κ-bit strings. For example, $\mathsf{P_A}$ generates seed_j^A, $\mathsf{seed'}_j^A$, and $\mathsf{witness'}_j^A$. Here, $(\mathsf{seed'}_j^A, \mathsf{witness'}_j^A)$ represents $\mathsf{P_A}$'s seed share and witness for the jth instance, as discussed in Section 1.2, and seed_j^A is used to derive randomness for Π_{OT} when $\mathsf{P_A}$ acts as the receiver. Each party commits to their three random strings, and the commitments will be signed by the other party later. This enables a certificate verifier to simulate the protocols execution based on these seeds/witnesses and verify the deviation of the other party, given the signature and the openings of signed commitments to these seeds/witnesses.

Next, each party executes Π_{OT} and retrieves all-but-one, *i.e.*, $\lambda-1$, seed shares from the other party. For example, we assume that the index of the garbled circuit $\mathsf{P_A}$ will evaluate is $\hat{\jmath}_A$, then $\mathsf{P_A}$ retrieves $\{\mathsf{seed'}_j^B\}_{j\in[\lambda]\setminus\{\hat{\jmath}_A\}}$ and $\mathsf{witness'}_{\hat{\jmath}_A}^B$. Each seed is defined as $\mathsf{seed}'_j \leftarrow \mathsf{seed'}_j^A \oplus \mathsf{seed'}_j^B$. Subsequently, both parties execute the equality test protocol Π_{Eq} as mentioned in Section 1.2 to ensure they do not possess the same set of seeds. Furthermore, parties commit to their retrieved seed shares and witnesses, which should be opened and verified by the other party at a later stage.

Circuit and input preparation. After obtaining the $\lambda - 1$ seeds seed'_j's, each party can generate $\lambda-1$ garbled circuits $\{\mathsf{GC}_j\}$ with randomness derived from $\{\mathsf{seed}'_j\}$. To avoid sending the whole garbled circuits, each party generates a commitment to each GC_j, along with its (randomly permuted) input-wire and output-wire labels, using randomness derived from seed'_j. The input-wire labels in each pair

are randomly permuted to hide the semantic values of the labels when the garbled circuit is later opened for evaluation. Additionally, the garbler commits to the index of his/her input-wire labels with respect to the randomly permuted input-wire labels. This step ensures that parties cannot change the input after the instance checking. For the $\hat{\jmath}_A$th (resp. $\hat{\jmath}_B$th) instance where $\mathsf{P_A}$ (resp. $\mathsf{P_B}$) does not possess the seed, dummy materials are generated in their places. As discussed in Section 1.2, a party cannot directly send the commitments of the garbled circuits to the other party. Instead, each party uses the verifiable time-lock puzzle scheme to encrypt those commitments and sends them to the other party.

Then each party acting as the sender executes Π_{OT} with the other party to transmit input-wire labels. The inputs to Π_{OT} are the input-wire labels for the other parties. For each checking instance, it is essential to ensure that each party can verify the validity of the input-wire labels provided by the other party. For example, $\mathsf{P_A}$ needs to verify that $\mathsf{P_B}$ acting as the sender in Π_{OT} uses the correct input-wire labels for $\mathsf{P_A}$. Since $\mathsf{P_A}$ knows seed'_j, she knows all the correct input-wire labels. Therefore, if $\mathsf{P_A}$ knows the randomness used by $\mathsf{P_B}$ in Π_{OT}, she can check whether $\mathsf{P_B}$ deviates from the protocol. However, it is important to note that $\mathsf{P_B}$'s randomness cannot simply be derived from seed'_j or seed'^B_j that are known to $\mathsf{P_A}$. The reason is that, given those materials, $\mathsf{P_A}$ can simulate the execution of Π_{OT} and immediately determine which instance is the $\hat{\jmath}_B$th instance, *i.e.*, the dummy instance of $\mathsf{P_B}$, subsequently identify the checking instances. Hence, similar to the problem mentioned in Section 1.2, we need to ensure that a party can check the instance, but until all materials for certificate generation are ready. In our protocol, we additionally introduce $\mathsf{seed}^A_{j,\mathsf{OT}}$ and $\mathsf{seed}^B_{j,\mathsf{OT}}$ to address this issue. For each instance, $\mathsf{P_A}$ randomly choosees uniform κ-bit string $\mathsf{seed}^A_{j,\mathsf{OT}}$ and uses the time-lock puzzle to encrypt $\mathsf{seed}^A_{j,\mathsf{OT}}$ and input-wire labels of $\mathsf{P_B}$. Then in the execution of Π_{OT} where $\mathsf{P_A}$ acts as the sender, the inputs are time-lock puzzles for input-wire labels of $\mathsf{P_B}$, and the randomness is derived from $\mathsf{seed}^A_{j,\mathsf{OT}}$ for $\mathsf{P_A}$'s $\hat{\jmath}_A$th instance while $\mathsf{seed}'_j \oplus \mathsf{seed}^A_{j,\mathsf{OT}}$ for other instances.

Each party also acts as the receiver and executes Π_{OT} with the other party. For example, the inputs of $\mathsf{P_A}$ to Π_{OT} is 0^{n_A} for checking instance and $x_A \in [n_A]$ for the evaluation instance with index $\hat{\jmath}_A$. The randomness in each execution is derived from seed^A_j. With knowledge of seed'_j and $\mathsf{seed}^B_{j,\mathsf{OT}}$, $\mathsf{P_A}$ can later simulate the execution of $\mathsf{P_A}$ in Π_{OT} and detect $\mathsf{P_B}$'s deviations.

Publicly verifiable evidence creation. Each party proceeds to sign the messages exchanged in each instance. At this stage, all materials necessary for certificate generation are available. Since all materials related to checking deviations are encrypted using time-lock puzzles, both parties are unable to perform the instance checking at this point.

Check of instances. Upon receiving the signature of the other party, each party can open their respective time-lock puzzles, allowing the other party to perform instance checking. If the other party fails to open the puzzles, each party can solve the puzzles themselves. With the commitments of the garbled circuit generated by the other party and the seed for Π_{OT} (*i.e.*, $\mathsf{seed}^A_{j,\mathsf{OT}}$ or/and $\mathsf{seed}^B_{j,\mathsf{OT}}$), both parties can simulate the computation of the other party for their checking instances. Among these instances, one should be a dummy instance, while the remaining instances should be correct. In case a deviation by the other party is detected, the detecting party can generate the certificate based on the signed materials, the openings/proofs for the time-lock puzzles, and the decommitments for the seeds/witnesses used in Π_{OT}.

Circuit evaluation. Assuming the checking instances are correct, two garbled circuits remain for evaluation. First, both parties open the commitments for the seed shares and the witness they obtained at the beginning of the protocol to demonstrate their honesty. Then, they open their respective garbled circuits, input-wire labels, and the mapping of output-wire labels to each other. Both parties verify the openings provided by the other party and proceed to evaluate the garbled circuit to obtain the output-wire labels and the result. For example, $\mathsf{P_A}$ obtains the output-wire labels $\{Z_{\hat{\jmath}_A,i}\}_{i\in[n_O]}$ and the result $z_{\hat{\jmath}_A}$, where $\hat{\jmath}_A$ represents $\mathsf{P_A}$'s index of instance for evaluation. $\mathsf{P_A}$ then sets $\beta_A \leftarrow \mathsf{H}(\bigoplus_{i=1}^{n_O}(Z_{\hat{\jmath}_A,i} \oplus Z_{\hat{\jmath}_B,i,z_{\hat{\jmath}_A}[i]}))$, where $\{Z_{\hat{\jmath}_B,i,b}\}_{i\in[n_O],b\in\{0,1\}}$ denotes the output-wire labels for the garbled circuit generated by $\mathsf{P_A}$ and evaluated by $\mathsf{P_B}$. Similarly, $\mathsf{P_B}$ evaluates the garbled circuit, obtains the output, and computes β_B. Finally, the two parties use the equality test protocol Π_{Eq} to determine the final evaluation result.

The full description of the protocol $\Pi_{\mathsf{RobustPVC}}$ between two parties $\mathsf{P_A}$ and $\mathsf{P_B}$ is given in the following.

Protocol $\Pi_{\mathsf{RobustPVC}}$

Schemes/Protocols: The signature scheme $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ is EUF-CMA-secure. The verifiable time-lock puzzle scheme TLP is secure with respect to the hardness parameter $\tau > 2\tau_c$, where τ_c is the timeout time for execution of Steps 6 and 7. Com is the computational binding and hiding commitment scheme, H is the collision-resistant hash function, the encryption scheme with respect to TLP is IND-CPA-secure (see Section 2 for more information). Π_{OT} is a perfectly correct protocol that realizes $\mathcal{F}_{\mathsf{OT}}$, and Π_{Eq} is an equality test protocol that realizes $\mathcal{F}_{\mathsf{Eq}}$. The garbling scheme used in this protocol is secure.
Public inputs: Both parties agree on parameters κ, λ, and τ, and a circuit $\mathcal{C}$, which has $n = n_A + n_B$ input wires and n_O output wires. Both parties also agree on the public information (*e.g.*, time for communication rounds, algorithms, parameters, and unique id for the execution of this protocol). $\mathsf{P_A}$ and $\mathsf{P_B}$ know keys vk_B and vk_A, respectively, for the signature scheme.
Private inputs: $\mathsf{P_A}$ has input $x_A \in \{0,1\}^{n_A}$ and keys $(\mathsf{vk}_A, \mathsf{sigk}_A)$ for the signature scheme. $\mathsf{P_B}$ has input $x_B \in \{0,1\}^{n_B}$ and keys $(\mathsf{vk}_B, \mathsf{sigk}_B)$ for the signature scheme.

Seed Preparation

1. $\mathsf{P_A}$ goes through the following steps with $\mathsf{P_B}$. In the meantime, they switch their roles to execute the symmetric steps, *i.e.*, $\mathsf{P_A}$ plays $\mathsf{P_B}$'s role and $\mathsf{P_B}$ plays $\mathsf{P_A}$'s.
   (a) $\mathsf{P_A}$ chooses uniform κ-bit strings for seed_j^A, $\mathsf{seed'}_j^A$, and $\mathsf{witness'}_j^A$, and computes $\mathsf{c}_{\mathsf{seed}_j^A} \leftarrow \mathsf{Com}(\mathsf{seed}_j^A)$, $\mathsf{c}_{\mathsf{seed'}_j^A} \leftarrow \mathsf{Com}(\mathsf{seed'}_j^A)$, and $\mathsf{c}_{\mathsf{witness'}_j^A} \leftarrow \mathsf{Com}(\mathsf{witness'}_j^A)$ for $j \in [\lambda]$, and sends these commitments to $\mathsf{P_B}$. $\mathsf{P_B}$ picks $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]$ and sets $b_{\hat{\jmath}_B} = 1$ and $b_j = 0$ for $j \neq \hat{\jmath}_B$.
   (b) $\mathsf{P_A}$ and $\mathsf{P_B}$ run λ executions of the protocol Π_{OT}. In the jth execution, $\mathsf{P_A}$ uses as input $(\mathsf{seed'}_j^A, \mathsf{witness'}_j^A)$ with randomness derived from seed_j^A, while $\mathsf{P_B}$ uses as input b_j with randomness derived from his seed_j^B generated in Step 1a. At the end, $\mathsf{P_B}$ has $\{\mathsf{seed'}_j^A\}_{j \neq \hat{\jmath}_B}$ and $\mathsf{witness'}_{\hat{\jmath}_B}^A$. Denote the transcript of the jth execution by trans_j^A.
   Let $\mathsf{CheckSet}_A = [\lambda] \backslash \{\hat{\jmath}_A\}$, $\mathsf{CheckSet}_B = [\lambda] \backslash \{\hat{\jmath}_B\}$, and $\mathsf{ComCheckSet} = [\lambda] \backslash \{\hat{\jmath}_A, \hat{\jmath}_B\}$.
2. $\mathsf{P_A}$ computes $\mathsf{seed}'_j \leftarrow \mathsf{seed'}_j^A \oplus \mathsf{seed'}_j^B$ for all $j \in \mathsf{CheckSet}_A$ and $h_A \leftarrow \mathsf{H}(\bigoplus_{j \in \mathsf{CheckSet}_A} \mathsf{seed}'_j)$. $\mathsf{P_B}$ performs a similar computation to derive $\{\mathsf{seed}'_j\}$ and h_B according to his $\mathsf{CheckSet}_B$. $\mathsf{P_A}$ and $\mathsf{P_B}$ use the protocol Π_{Eq} to check whether $h_A = h_B$. If it does not hold, they continue the protocol execution. Otherwise, they restart the protocol.
3. $\mathsf{P_A}$ computes $\mathsf{c}_A \leftarrow \mathsf{Com}(\hat{\jmath}_A, \{\mathsf{seed'}_j^B\}_{j \neq \hat{\jmath}_A}, \mathsf{witness'}_{\hat{\jmath}_A}^B)$ and sends it to $\mathsf{P_B}$. Similarly, $\mathsf{P_B}$ computes and sends c_B to $\mathsf{P_A}$.

Circuit and Input Preparation

4. For $j \in \mathsf{CheckSet}_A$, $\mathsf{P_A}$ follows the procedure below, where all randomness in the jth instance is derived from seed'_j.
   (a) $\mathsf{P_A}$ garbles the circuit $\mathcal{C}$. Denote the jth garbled circuit by GC_j, the input-wire labels of $\mathsf{P_A}$ by $\{A_{j,i,b}\}_{i \in [n_A], b \in \{0,1\}}$, the input-wire labels of $\mathsf{P_B}$ by $\{B_{j,i,b}\}_{i \in [n_B], b \in \{0,1\}}$, and the output-wire labels by $\{Z_{j,i,b}\}_{i \in [n_O], b \in \{0,1\}}$.
   (b) $\mathsf{P_A}$ computes her label commitments $h^A_{j,i,b} \leftarrow \mathsf{Com}(A_{j,i,b})$ for all $i \in [n_A]$ and $b \in \{0,1\}$ and also $h^B_{j,i,b} \leftarrow \mathsf{Com}(B_{j,i,b})$ for all $i \in [n_B]$ and $b \in \{0,1\}$.
   (c) $\mathsf{P_A}$ computes the commitment $\mathsf{c}_j \leftarrow \mathsf{Com}(\mathsf{GC}_j, \{(h^A_{j,i,\alpha_i^A}, h^A_{j,i,\bar{\alpha}_i^A})\}_{i \in [n_A]}$, $\{(h^B_{j,i,\alpha_i^B}, h^B_{j,i,\bar{\alpha}_i^B})\}_{i \in [n_B]}, \{(\mathsf{H}(Z_{j,i,0}), \mathsf{H}(Z_{j,i,1}))\}_{i \in [n_O]})$, in which each pair $(h^A_{j,i,0}, h^A_{j,i,1})$ is randomly permuted with respect to $\alpha_i^A \leftarrow\!\!\$\, \{0,1\}$, while $(h^B_{j,i,0}, h^B_{j,i,1})$ is also randomly permuted. Denote the index of $h^A_{j,i,x_A[i]}$ with respect to $x_A[i]$ in $(h^A_{j,i,\alpha_i^A}, h^A_{j,i,\bar{\alpha}_i^A})$ by $\gamma_j^A[i] = \alpha_i^A \oplus x[i]$, and thus the string of indices is denoted by $\gamma_j^A \in \{0,1\}^{n_A}$.
   In parallel, $\mathsf{P_B}$ also follows this procedure as $\mathsf{P_A}$ for $j \in \mathsf{CheckSet}_B$ using randomness derived from seed'_j.
   Then, $\mathsf{P_A}$ lets $\mathsf{c}_{\hat{\jmath}_A} = 0$, $\gamma^A_{\hat{\jmath}_A} = 0$, and $B_{\hat{\jmath}_A,i,b} = 0$ for all $i \in [n_B]$ and $b \in \{0,1\}$. Symmetrically, $\mathsf{P_B}$ lets $\mathsf{c}_{\hat{\jmath}_B} = 0$, $\gamma^B_{\hat{\jmath}_B} = 0$, and $A_{\hat{\jmath}_B,i,b} = 0$ for all $i \in [n_A]$ and $b \in \{0,1\}$.
5. $\mathsf{P_A}$ computes and sends $\mathsf{c}_{\gamma_j^A} \leftarrow \mathsf{Com}(\gamma_j^A)$ for $j \in [\lambda]$ to $\mathsf{P_B}$. Similarly, $\mathsf{P_B}$ computes and sends $\mathsf{c}_{\gamma_j^B} \leftarrow \mathsf{Com}(\gamma_j^B)$ for $j \in [\lambda]$ to $\mathsf{P_A}$.
6. For $j \in [\lambda]$, $\mathsf{P_A}$ chooses uniform κ-bit string $\mathsf{seed}^A_{j,\mathsf{OT}}$ and a random master puzzle key s_j^A, and generates a master time-lock puzzle $\mathsf{p}_{s_j^A}$ for s_j^A with respect to the hardness parameter τ. She also computes puzzles $\mathsf{p}_{\mathsf{c}_j} \leftarrow \mathsf{Enc}_{s_j^A}(\mathsf{c}_j)$, $\mathsf{p}_{\mathsf{seed}^A_{j,\mathsf{OT}}} \leftarrow \mathsf{Enc}_{s_j^A}(\mathsf{seed}^A_{j,\mathsf{OT}})$, and $\mathsf{p}_{B_{j,i,b}} \leftarrow \mathsf{Enc}_{s_j^A}(B_{j,i,b})$ for $i \in [n_B]$ and $b \in \{0,1\}$. Then $\mathsf{P_A}$ sends $\mathsf{p}_{s_j^A}$, $\mathsf{p}_{\mathsf{c}_j}$, and $\mathsf{p}_{\mathsf{seed}^A_{j,\mathsf{OT}}}$ to $\mathsf{P_B}$.

$\mathsf{P_A}$ and $\mathsf{P_B}$ run λ executions of Π_{OT}. In the jth execution, $\mathsf{P_A}$ uses as input $\{(\mathsf{p}_{B_{j,i,0}}, \mathsf{p}_{B_{j,i,1}})\}_{i\in[n_B]}$, and $\mathsf{P_B}$ uses $x_B \in \{0,1\}^{n_B}$ as input if $j = \hat{\jmath}_B$ and 0^{n_B} otherwise. Here $\mathsf{P_A}$ uses $(\mathsf{seed}'_j \oplus \mathsf{seed}^A_{j,\mathsf{OT}})$ for $j \neq \hat{\jmath}_A$ and $\mathsf{seed}^A_{j,\mathsf{OT}}$ for $j = \hat{\jmath}_A$ to derive her randomness in the jth execution, while $\mathsf{P_B}$ uses seed^B_j. Finally, $\mathsf{P_B}$ obtains $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,x_B[i]}}\}_{i\in[n_B]}$ for the $\hat{\jmath}_B$th execution. Denote the transcript hash for the jth execution of Π_{OT} by $\mathcal{H}^A_j$.
Similarly, $\mathsf{P_B}$ also follows the same procedure as $\mathsf{P_A}$, *i.e.*, $\mathsf{P_B}$ computes and sends $\mathsf{p}_{s^B_j}$, $\mathsf{p}_{\mathsf{c}_j}$, and $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ to $\mathsf{P_A}$. $\mathsf{P_B}$ also runs λ executions of Π_{OT} for input $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ with $\mathsf{P_A}$ as above. Denote the transcript hash for the jth execution of Π_{OT} by $\mathcal{H}^B_j$.

Publicly Verifiable Evidence Creation

7. For each $j \in [\lambda]$, $\mathsf{P_A}$ generates a signature $\sigma^A_j \leftarrow \mathsf{Sig}_{\mathsf{sigk}_A}(id, \mathcal{C}, j, \mathsf{c}_{\mathsf{seed}^B_j}, \mathsf{c}_{\mathsf{seed}'^B_j}, \mathsf{c}_{\mathsf{witness}'^B_j}, \mathsf{trans}^A_j, \mathsf{trans}^B_j, \mathcal{H}^A_j, \mathsf{p}_{s^A_j}, \mathsf{p}_{\mathsf{c}_j}, \mathsf{p}_{\mathsf{seed}^A_{j,\mathsf{OT}}}, \tau, \lambda)$ and sends these signatures to $\mathsf{P_B}$. Then $\mathsf{P_B}$ checks whether σ^A_j is valid for all $j \in [\lambda]$, and aborts with output $\perp$ if not. Similarly, $\mathsf{P_B}$ sends the symmetric signatures σ^B_j for all j to $\mathsf{P_A}$, and $\mathsf{P_A}$ checks their validity.

Check of Instances

8. $\mathsf{P_A}$ and $\mathsf{P_B}$ open the master time-lock puzzle of the master puzzle key to each other. If one party does not open the puzzle, the other party can also solve the puzzle to derive the master puzzle key.
9. For each $j \in [\lambda]$, $\mathsf{P_A}$ decrypts $\mathsf{P_B}$'s $\mathsf{p}_{\mathsf{c}_j}$ and $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ using the master puzzle key s^B_j to obtain c_j and $\mathsf{seed}^B_{j,\mathsf{OT}}$. $\mathsf{P_A}$ also decrypts $\mathsf{p}_{A_{\hat{\jmath}_A,i,x_A[i]}}$ to obtain $A_{\hat{\jmath}_A,i,x_A[i]}$. Note that for $j \in \mathsf{ComCheckSet}$, honest $\mathsf{P_A}$ holds the same seed'_j as honest $\mathsf{P_B}$, and thus the same materials in Step 4. $\mathsf{P_A}$ follows the checking procedure below to check $\mathsf{P_B}$'s $\{\mathsf{c}_j\}$ and $\{\mathcal{H}^B_j\}$. Once a certificate cert is generated during the check, $\mathsf{P_A}$ outputs $\mathsf{corrupted}$, sends cert to $\mathsf{P_B}$, and halts immediately.
   (a) For $j \in \mathsf{CheckSet}_A$, if there exists a c_j from $\mathsf{P_B}$ that is not equal to $\mathsf{P_A}$'s c_j and also $\mathsf{c}_j \neq 0$, $\mathsf{P_A}$ uniformly chooses such a j and uses $\mathsf{Blame}(\mathsf{BadCom}, j)$ to generate cert.
   (b) For $j \in \mathsf{CheckSet}_A$, if c_j from $\mathsf{P_B}$ is equal to $\mathsf{P_A}$'s c_j, $\mathsf{P_A}$ simulates $\mathsf{P_B}$'s computation of $\{\mathsf{p}_{A_{j,i,b}}\}_{i\in[n_A]}$ and the executions of Π_{OT} in Step 6 with $\mathsf{P_B}$'s randomness derived from $\{\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}\}$ to derive the transcript hash $\hat{\mathcal{H}}^B_j$. If $\hat{\mathcal{H}}^B_j \neq \mathcal{H}^B_j$, $\mathsf{P_A}$ uniformly chooses such a j and uses $\mathsf{Blame}(\mathsf{BadOT}, j)$ to generate cert.
   For $j \in \mathsf{CheckSet}_A$, if $\mathsf{P_B}$'s $\mathsf{c}_j = 0$, $\mathsf{P_A}$ simulates $\mathsf{P_B}$'s computation of $\{\mathsf{p}_{A_{j,i,b}}\}$, where $A_{j,i,b} = 0$, and the executions of Π_{OT} in Step 6 with $\mathsf{P_B}$'s randomness derived from $\mathsf{seed}^B_{j,\mathsf{OT}}$ to derive the transcript hash $\hat{\mathcal{H}}^B_j$. If $\hat{\mathcal{H}}^B_j \neq \mathcal{H}^B_j$, $\mathsf{P_A}$ uniformly chooses such a j and uses $\mathsf{Blame}(\mathsf{BadOT}, j)$ to generate cert.
   (c) If there are more than one commitment $\mathsf{c}_j = 0$ for $j \in \mathsf{CheckSet}_A$, uniformly choose two of them and let $J = \{j \mid \mathsf{c}_j = 0\}$ such that $|J| = 2$. $\mathsf{P_A}$ uses $\mathsf{Blame}(\mathsf{BadNum}, J)$ to generate cert.
   (d) If for all $j \in \mathsf{CheckSet}_A$, c_j from $\mathsf{P_B}$ is equal to $\mathsf{P_A}$'s c_j, $\mathsf{P_A}$ aborts with output $\perp$.
   Symmetrically, $\mathsf{P_B}$ follows the same procedure to check instances (and generates cert if needed).

Circuit Evaluation

10. $\mathsf{P_A}$ opens the committed message $(\hat{\jmath}_A, \{\mathsf{seed}'^B_j\}_{j\neq\hat{\jmath}_A}, \mathsf{witness}'^B_{\hat{\jmath}_A})$ inside c_A to $\mathsf{P_B}$, while $\mathsf{P_B}$ opens c_B to $\mathsf{P_A}$. If the other party does not open the commitment or the committed message is incorrect, the party aborts with output $\perp$.
11. $\mathsf{P_A}$ sends $\mathsf{GC}_{\hat{\jmath}_B}$, $\{(h^A_{i,(0)}, h^A_{i,(1)}) = (h^A_{\hat{\jmath}_B,i,\alpha^A_i}, h^A_{\hat{\jmath}_B,i,\bar{\alpha}^A_i})\}$, $\{(h^B_{j,i,\alpha^B_i}, h^B_{j,i,\bar{\alpha}^B_i})\}$ (in the same permuted order as before), and $\{(\mathsf{H}(Z_{\hat{\jmath}_B,i,0}), \mathsf{H}(Z_{\hat{\jmath}_B,i,1}))\}$, together with $\mathsf{decom}_{\mathsf{c}_{\hat{\jmath}_B}}$ to $\mathsf{P_B}$. If $\mathsf{Com}(\mathsf{GC}_{\hat{\jmath}_B}, \{(h^A_{\hat{\jmath}_B,i,\alpha^A_i}, h^A_{\hat{\jmath}_B,i,\bar{\alpha}^A_i})\}, \{(h^B_{j,i,\alpha^B_i}, h^B_{j,i,\bar{\alpha}^B_i})\}, \{\mathsf{H}(Z_{\hat{\jmath}_B,i,b})\}; \mathsf{decom}_{\mathsf{c}_{\hat{\jmath}_B}}) \neq \mathsf{c}_{\hat{\jmath}_B}$ for some i, $\mathsf{P_B}$ aborts with output $\perp$.
    Symmetrically, $\mathsf{P_B}$ sends the corresponding materials with respect to the $\hat{\jmath}_A$th instance to $\mathsf{P_A}$, and $\mathsf{P_A}$ checks the correctness of the materials.
12. $\mathsf{P_A}$ sends $\{A_{\hat{\jmath}_B,i,x_A[i]}\}_{i\in[n_A]}$, $\gamma^A_{\hat{\jmath}_B}$, together with decommitments $\mathsf{decom}_{\gamma^A_{\hat{\jmath}_B}}$ and $\{\mathsf{decom}^A_{\hat{\jmath}_B,i,x_A[i]}\}_{i\in[n_A]}$ to $\mathsf{P_B}$. If $\mathsf{Com}(\gamma^A_{\hat{\jmath}_B}; \mathsf{decom}_{\gamma^A_{\hat{\jmath}_B}}) \neq \mathsf{c}_{\gamma^A_{\hat{\jmath}_B}}$ or $\mathsf{Com}(A_{\hat{\jmath}_B,i,x_A[i]}; \mathsf{decom}^A_{\hat{\jmath}_B,i,x_A[i]}) \neq h^A_{i,\gamma^A_{\hat{\jmath}_B}[i]}$ for some i, $\mathsf{P_B}$ aborts with output $\perp$.
    Otherwise, $\mathsf{P_B}$ evaluates $\mathsf{GC}_{\hat{\jmath}_B}$ with input-wire labels $\{A_{\hat{\jmath}_B,i,x_A[i]}\}_{i\in[n_A]}$ and $\{B_{\hat{\jmath}_B,i,x_B[i]}\}_{i\in[n_B]}$ to obtain the output-wire labels $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$. $\mathsf{P_B}$ can derive the result $z_{\hat{\jmath}_B}$ from $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$ and $\{(\mathsf{H}(Z_{\hat{\jmath}_B,i,0}), \mathsf{H}(Z_{\hat{\jmath}_B,i,1}))\}_{i\in[n_O]}$. If any of the decoded bits is $\perp$, $\mathsf{P_B}$ then lets $z_{\hat{\jmath}_B} = \perp$.

Symmetrically, $\mathsf{P_B}$ sends the corresponding materials with respect to the $\hat{\jmath}_A$th instance to $\mathsf{P_A}$. Then $\mathsf{P_A}$ checks the correctness of the materials and evaluates the $\hat{\jmath}_A$th garbled circuits to derive the result $z_{\hat{\jmath}_A}$.

13. If $z_{\hat{\jmath}_A} = \perp$, $\mathsf{P_A}$ lets β_A be a random κ-bit value. Otherwise, $\mathsf{P_A}$ sets $\beta_A \leftarrow \mathsf{H}(\bigoplus_{i=1}^{n_O}(Z_{\hat{\jmath}_A,i} \oplus Z_{\hat{\jmath}_B,i,z_{\hat{\jmath}_A}[i]}))$. Symmetrically, $\mathsf{P_B}$ computes his β_B. Then $\mathsf{P_A}$ and $\mathsf{P_B}$ use the equality test protocol Π_{Eq} to check whether $\beta_A = \beta_B$. If it holds, $\mathsf{P_A}$ (resp. $\mathsf{P_B}$) outputs $z_{\hat{\jmath}_A}$ (resp. $z_{\hat{\jmath}_B}$). Otherwise, parties abort with output cheating.

The description of the corresponding algorithm Blame that could output a publicly verifiable certificate for cheating is given in the following.

Algorithm Blame

The party P_X runs this algorithm to obtain a certificate cert for the malicious behavior of P_Y. We first define the following notations:

- Denote $\mathsf{msg}_j = (id, \mathsf{trans}_j^Y, \mathsf{trans}_j^X, \mathcal{H}_j^Y, \mathsf{p}_{s_j^Y}, \mathsf{p}_{\mathsf{c}_j}, \mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^Y}, \tau, \lambda)$, where $\mathsf{p}_{\mathsf{c}_j}$ is the ciphertext from P_Y.
- Denote the proof/opening with respect to $\mathsf{p}_{s_j^Y}$ for s_j^Y by π_j[a].
- Denote $\mathsf{v}_j^X = (\mathsf{seed}_j^X, \mathsf{seed}'^X_j, \mathsf{witness}'^X_j)$ and $\mathsf{decom}_j = (\mathsf{decom}_j^X, \mathsf{decom}'^X_j, \mathsf{decom}^X_{\mathsf{witness}'_j})$.

The algorithm generates a certificate cert as follows depending on the received parameters.

- (BadCom, j): Output $\mathsf{cert} = (\mathsf{BadCom}, j, \mathsf{msg}_j, \mathsf{v}_j^X, \mathsf{decom}_j, \sigma_j^Y, s_j^Y, \pi_j)$.
- (BadOT, j): Output $\mathsf{cert} = (\mathsf{BadOT}, j, \mathsf{msg}_j, \mathsf{v}_j^X, \mathsf{decom}_j, \sigma_j^Y, s_j^Y, \pi_j)$.
- (BadNum, J): Output $\mathsf{cert} = (\mathsf{BadNum}, \{j, \mathsf{msg}_j, \mathsf{v}_j^X, \mathsf{decom}_j, \sigma_j^Y, s_j^Y, \pi_j\}_{j \in J})$.

[a] Note that π_j can be provided by P_Y for puzzle opening or generated by P_X if the puzzle is solved by P_X.

In the following, we present the description of the algorithm Judge. The algorithm Judge could take as input a certificate cert generated by the algorithm Blame and output whether the accused party is cheating.

Algorithm Judge

Inputs: A public key vk_Y, a circuit $\mathcal{C}$, and a certificate cert.
Depending on the types of cheating, the algorithm verifies the certificate cert as follows.

- BadCom: Parse the remaining part as $(j, \mathsf{msg}_j, \mathsf{v}_j^X, \mathsf{decom}_j, \sigma_j^Y, s_j^Y, \pi_j)$. Verify the correctness of (s_j^Y, π_j) and output false if π_j is incorrect. Derive c_j from $\mathsf{p}_{\mathsf{c}_j}$. Use v_j^X and decom_j to re-construct commitments $\mathsf{c}_{\mathsf{seed}_j^X}$, $\mathsf{c}_{\mathsf{seed}'^X_j}$, and $\mathsf{c}_{\mathsf{witness}'^X_j}$. Check the correctness of σ_j^Y using vk_Y, and output false if the signature is invalid. Decrypt $\mathsf{p}_{\mathsf{c}_j}$ to obtain c_j. Use v_j^X to simulate the executions of Π_{OT} (in two directions) based on trans_j^X and trans_j^Y and derive seed'^Y_j in Step 1a. Verify the simulation of Π_{OT} using trans_j^X and trans_j^Y. If they do not match, output false. Then use $\mathsf{seed}'_j = \mathsf{seed}'^X_j \oplus \mathsf{seed}'^Y_j$ to simulate Step 4 and obtain $\hat{\mathsf{c}}_j$. If $\mathsf{c}_j \neq \hat{\mathsf{c}}_j$ and $\mathsf{c}_j \neq 0$, output true. Otherwise, output false.
- BadOT: Parse the remaining part as $(j, \mathsf{msg}_j, \mathsf{v}_j^X, \mathsf{decom}_j, \sigma_j^Y, s_j^Y, \pi_j)$. Verify the correctness of (s_j^Y, π_j), and output false if π_j is incorrect. Use v_j^X and decom_j to re-construct commitments $\mathsf{c}_{\mathsf{seed}_j^X}$, $\mathsf{c}_{\mathsf{seed}'^X_j}$, and $\mathsf{c}_{\mathsf{witness}'^X_j}$. Check the correctness of σ_j^Y using vk_Y, and output false if the signature is invalid. Decrypt $\mathsf{p}_{\mathsf{c}_j}$ and $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^Y}$ to obtain c_j and $\mathsf{seed}_{j,\mathsf{OT}}^Y$.
  If $\mathsf{c}_j = 0$, use $\mathsf{seed}_{j,\mathsf{OT}}^Y$ and the master time-lock puzzle key s_j^Y to simulate the execution of Π_{OT} in Step 6, and verify whether P_Y is the first one whose message hash is different from the one in $\mathcal{H}_j^Y$. If yes, output true. Otherwise, output false.
  If $\mathsf{c}_j \neq 0$, use v_j^X to simulate the executions of Π_{OT} (in two directions) based on trans_j^X and trans_j^Y and derive seed'^Y_j in Step 1a. Verify the simulation of Π_{OT} using trans_j^X and trans_j^Y. If they do not match, output false. Compute $\mathsf{seed}'_j \leftarrow \mathsf{seed}'^X_j \oplus \mathsf{seed}'^Y_j$. Then use seed'_j, $\mathsf{seed}'_j \oplus \mathsf{seed}_{j,\mathsf{OT}}^Y$, and the master time-lock puzzle key s_j^Y to simulate the executions of Π_{OT} in Step 6, and verify whether P_Y is the first one whose message hash is different from the one in $\mathcal{H}_j^Y$. If yes, output true. Otherwise, output false.

– BadNum: Parse the remaining part as $\{j, \mathsf{msg}_j, \mathsf{v}_j^X, \mathsf{decom}_j, \sigma_j^Y, s_j^Y, \pi_j\}_{j\in J}$. Verify the correctness of $\{(s_j^Y, \pi_j)\}$ and output false if there exists a tuple (s_j^Y, π_j) that is incorrect. Use $\{\mathsf{v}_j^X\}$ and $\{\mathsf{decom}_j\}$ to re-construct commitments $\{\mathsf{c}_{\mathsf{seed}_j^X}\}$, $\{\mathsf{c}_{\mathsf{seed}'^X_j}\}$, and $\{\mathsf{c}_{\mathsf{witness}'^X_j}\}$. Check the correctness of $\{\sigma_j^Y\}$ using vk_Y, and output false if there exists a signature that is invalid. Decrypt $\{\mathsf{p}_{\mathsf{c}_j}\}$ to obtain $\{\mathsf{c}_j\}$. If $\mathsf{c}_j = 0$ for all $j \in J$, output true. Otherwise, output false.

Theorem 1. *The protocol $\Pi_{\mathsf{RobustPVC}}$ along with algorithms* Blame *and* Judge *is robust publicly verifiable covert secure with deterrence $\epsilon = 1 - \frac{1}{\lambda-1}$.*

Proof. We prove the robust PVC security of $\Pi_{\mathsf{RobustPVC}}$ by showing its simulatability, public verifiability, and defamation freeness.

Simulatability We first prove that $\Pi_{\mathsf{RobustPVC}}$ securely realizes $\mathcal{F}_{\mathsf{RobustPVC}}$ with deterrence $\epsilon = 1 - \frac{1}{\lambda-1}$. Without loss of generality, we assume that $\mathsf{P_B}$ is honest and $\mathsf{P_A}$ is corrupted by $\mathcal{A}$. Since the roles of $\mathsf{P_A}$ and $\mathsf{P_B}$ in $\Pi_{\mathsf{RobustPVC}}$ is symmetric, the proof for the scenario where $\mathsf{P_B}$ is corrupted is similar.

For an adversary $\mathcal{A}$ corrupting $\mathsf{P_A}$ in the real world, we construct a simulator $\mathcal{S}$ holding vk_A that runs $\mathcal{A}$ as a sub-routine with auxiliary input z and interacts with $\mathcal{F}_{\mathsf{RobustPVC}}$ in the ideal world. The simulation procedure is presented below.

0. $\mathcal{S}$ uses KGen to generate a pair of key $(\mathsf{vk}_B, \mathsf{sigk}_B)$ and sends vk_B to $\mathcal{A}$.
1. $\mathcal{S}$ goes through the following steps with $\mathcal{A}$.
   (a) $\mathcal{S}$ receives $\{\mathsf{c}_{\mathsf{seed}_j^A}, \mathsf{c}_{\mathsf{seed}'^A_j}, \mathsf{c}_{\mathsf{witness}'^A_j}\}_{j\in[\lambda]}$ from $\mathcal{A}$. In the meantime, $\mathcal{S}$ picks uniform κ-bit strings $\{\mathsf{seed}_j^B, \mathsf{seed}'^B_j, \mathsf{witness}'^B_j\}_{j\in[\lambda]}$, computes $\{\mathsf{c}_{\mathsf{seed}_j^B}, \mathsf{c}_{\mathsf{seed}'^B_j}, \mathsf{c}_{\mathsf{witness}'^B_j}\}_{j\in[\lambda]}$, and sends these commitments to $\mathcal{A}$ as in the protocol.
   (b) For all j, $\mathcal{S}$ playing the role of the receiver uses $b_j = 0$ to retrieve seed'^A_j in Π_{OT} with randomness derived from seed_j^B. $\mathcal{S}$ also plays the role of the sender and uses $\{(\mathsf{seed}'^B_j, \mathsf{witness}'^B_j)\}_{j\in[\lambda]}$ as input in Π_{OT} with randomness derived from seed_j^B.
2. $\mathcal{S}$ computes $\mathsf{seed}'_j = \mathsf{seed}'^A_j \oplus \mathsf{seed}'^B_j$ for all $j \in [\lambda]$. Then $\mathcal{S}$ let $h_B^j \leftarrow \mathsf{H}(\bigoplus_{[\lambda]\setminus j} \mathsf{seed}'_j)$ for all $j \in [\lambda]$. For the execution of Π_{Eq}, $\mathcal{S}$ uses the corresponding simulator $\mathcal{S}_{\mathsf{Eq}}$ to obtain h_A. If there exists h_B^j, such that $h_B^j = h_A$, $\mathcal{S}$ sets $\mathsf{GoodEq} = \mathsf{true}$ and picks $\hat{j}_B \leftarrow\!\!\$\, [\lambda]$. If $h_B^{\hat{j}_B} = h_A$, $\mathcal{S}$ returns true for Π_{Eq} to $\mathcal{A}$ and restarts the protocol, and otherwise returns false for Π_{Eq} to $\mathcal{A}$. If there is no h_B^j, such that $h_B^j = h_A$, $\mathcal{S}$ sets $\mathsf{GoodEq} = \mathsf{false}$ and returns false for Π_{Eq} to $\mathcal{A}$.
3. $\mathcal{S}$ computes and sends dummy commitment c_B to $\mathcal{A}$. $\mathcal{S}$ also receives commitment c_A from $\mathcal{A}$.
4. For all $j \in [\lambda]$, $\mathcal{S}$ prepares garbled circuits and input materials as in Step 4 of the protocol.
5. $\mathcal{S}$ computes and sends dummy commitment $\{\mathsf{c}_{\gamma_j^B}\}$ to $\mathcal{A}$. $\mathcal{S}$ also receives $\{\mathsf{c}_{\gamma_j^A}\}$ from $\mathcal{A}$.
6. $\mathcal{S}$ generates the puzzle $\mathsf{p}_{s_j^B}$ for a random chosen master puzzle key s_j^B as in the protocol for $j \in [\lambda]$. $\mathcal{S}$ also generates dummy ciphertexts for his $\mathsf{p}_{\mathsf{c}_j}$ and $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^B}$ for $j \in [\lambda]$. Then $\mathcal{S}$ sends these puzzles to $\mathcal{A}$.
   $\mathcal{S}$ receives puzzles $\mathsf{p}_{s_j^A}$, $\mathsf{p}_{\hat{\mathsf{c}}_j}$, and $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^A}$ for $j \in [\lambda]$ from $\mathcal{A}$. Since time-lock puzzles can be solved in polynomial time, $\mathcal{S}$ can decrypt these puzzles and derive s_j^A, $\hat{\mathsf{c}}_j$, and $\mathsf{seed}_{j,\mathsf{OT}}^A$ for $j \in [\lambda]$.
   For λ executions of Π_{OT} where $\mathcal{S}$ plays the role of the receiver, $\mathcal{S}$ uses as input 0^{n_B} and randomness derived from seed_j^B and receives $\mathcal{A}$'s $\{\mathsf{p}_{B_{j,i,0}}\}$. Let $\mathcal{H}_j^A$ denote the transcript hash for the jth executions.
   For λ executions of Π_{OT} where $\mathcal{S}$ plays the role of the sender, $\mathcal{S}$ first generates dummy ciphertexts $\{\mathsf{p}_{A_{j,i,b}}\}$ and random κ-bit string seed''_j for all $j \in [\lambda]$. Then $\mathcal{S}$ executes Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from seed''_j.
7. $\mathcal{S}$ generates signatures $\{\sigma_j^B\}$ as an honest $\mathsf{P_B}$ and sends them to $\mathcal{A}$.
   $\mathcal{S}$ also receives $\{\sigma_j^A\}$ from $\mathcal{A}$. If any of the signatures are invalid, $\mathcal{S}$ executes the *finishing touches* procedure as follows and sends abort to $\mathcal{F}_{\mathsf{RobustPVC}}$ to terminate the execution of the protocol.
   (a) Choose $\hat{j}_B \leftarrow\!\!\$\, [\lambda]$ if $\mathsf{GoodEq} = \mathsf{false}$, and otherwise use $\hat{j}_B$ chosen in Step 2 in this simulation.
   (b) Program the random oracle to let the decryption of $\mathsf{p}_{\mathsf{c}_{\hat{j}_B}}$ be 0, decryption of $\mathsf{p}_{\mathsf{c}_j}$ be c_j for $j \in [\lambda]\setminus\{\hat{j}_B\}$, and decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol for $j \in [\lambda]\setminus\{\hat{j}_B\}$ and $j = \hat{j}_B$, respectively.
   (c) Program the random oracle: for the $\hat{j}_B$th instance, let the decryption of $\mathsf{p}_{\mathsf{seed}_{\hat{j}_B,\mathsf{OT}}^B}$ be $\mathsf{seed}_{\hat{j}_B,\mathsf{OT}}^B = \mathsf{seed}''_{\hat{j}_B}$. For $j \in [\lambda]\setminus\{\hat{j}_B\}$, let the decryption of $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^B}$ be the decryption of $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^B}$ be $\mathsf{seed}_{j,\mathsf{OT}}^B = \mathsf{seed}''_j \oplus \mathsf{seed}'_j$.
   Then $\mathcal{S}$ simulates the computation of an honest $\mathsf{P_A}$ playing the role of the sender in the executions of Π_{OT} in Step 6 of the protocol and obtain $\hat{\mathcal{H}}_j^A$. If $\hat{\mathsf{c}}_j = 0$, the simulation is based on $\mathsf{seed}_{j,\mathsf{OT}}^A$ as

the $\hat{\jmath}_A$th instance in the protocol. If $\hat{\mathsf{c}}_j = \mathsf{c}_j$, the simulation is based on $\mathsf{seed}'_j \oplus \mathsf{seed}^A_{j,\mathsf{OT}}$ as other instances in the protocol. Let J_z be the set of indices that $\hat{\mathsf{c}}_j = 0$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the $\hat{\jmath}_A$th instance as described in the protocol. Let J_s be the set of indices that $\hat{\mathsf{c}}_j = \mathsf{c}_j$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the checking instances.

Hence, there are following cases (let $\mathsf{Stupid} = \mathsf{false}$ and $\mathsf{Type} = \perp$ by default):

- If $|J_s| < \lambda - 2$, $\mathcal{S}$ executes the *finishing touches* procedure as in Step 7a-7c of the simulation, sends $(\mathsf{cheat}, 1)$ to $\mathcal{F}_{\mathsf{RobustPVC}}$, receives $\mathsf{corrupted}$, opens the puzzle as in Step 8 of the protocol, generates the certificate cert with respect to j and the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}^A_j$ as in the protocol. $\mathcal{S}$ then sends cert to $\mathcal{A}$ as Step 9 of the protocol to complete the simulation.
- If $|J_s| = \lambda - 2$ and $|J_z| = 2$, $\mathcal{S}$ sets $\mathsf{Stupid} = \mathsf{true}$ and sends $(\mathsf{stupidCheat}, \frac{\lambda-2}{\lambda})$ to $\mathcal{F}_{\mathsf{RobustPVC}}$. If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\mathsf{corrupted}$, $\mathcal{S}$ sets $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{true})$ and $\mathsf{Type} = \mathsf{BadNum}$. If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\perp$, $\mathcal{S}$ sets $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{false})$. Then $\mathcal{S}$ continues the simulation.
- If $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{true}$, $\mathcal{S}$ sends $(\mathsf{cheat}, \frac{\lambda-2}{\lambda-1})$ to $\mathcal{F}_{\mathsf{RobustPVC}}$. If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\mathsf{corrupted}$, $\mathcal{S}$ sets $\mathsf{caught} = (\frac{\lambda-2}{\lambda-1}, \mathsf{true})$ and Type according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}^A_j$. If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\mathsf{undetected}$, $\mathcal{S}$ sets $\mathsf{caught} = (\frac{\lambda-2}{\lambda-1}, \mathsf{false})$. Then $\mathcal{S}$ continues the simulation.
- If $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, $\mathcal{S}$ sends $(\mathsf{cheat}, \frac{\lambda-1}{\lambda})$ to $\mathcal{F}_{\mathsf{RobustPVC}}$. If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\mathsf{corrupted}$, $\mathcal{S}$ sets $\mathsf{caught} = (\frac{\lambda-1}{\lambda}, \mathsf{true})$ and Type according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}^A_j$. If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\mathsf{undetected}$, $\mathcal{S}$ sets $\mathsf{caught} = (\frac{\lambda-1}{\lambda}, \mathsf{false})$. Then $\mathcal{S}$ continues the simulation.
- If $|J_s| = \lambda - 2$ and $|J_z| = 0$, $\mathcal{S}$ executes the *finishing touches* procedure as in Step 7a-7c, sends $(\mathsf{cheat}, 1)$ to $\mathcal{F}_{\mathsf{RobustPVC}}$, receives $\mathsf{corrupted}$, opens the puzzle as in Step 8 of the protocol, generates the certificate cert with respect to j and the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}^A_j$ as in the protocol, and sends cert to $\mathcal{A}$ as in Step 9 of the protocol to complete the simulation.
- If $|J_s| = \lambda - 1$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{true}$, $\mathcal{S}$ sets $\mathsf{caught} = \perp$. Then $\mathcal{S}$ continues the simulation.
- If $|J_s| = \lambda - 1$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, then $\mathcal{S}$ tosses a coin, which with probability $\frac{1}{\lambda}$ outputs false and $\frac{\lambda-1}{\lambda}$ outputs true. If the output is false, then $\mathcal{S}$ executes the *finishing touches* procedure as follows, opens his puzzle, and sends abort to $\mathcal{F}_{\mathsf{RobustPVC}}$ to terminate the execution of the protocol.
  (a) Let $\hat{\jmath}_B$ be the element in J_z.
  (b) Program the random oracle to let the decryption of $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ be 0, decryption of $\mathsf{p}_{\mathsf{c}_j}$ be c_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, and decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ and $j = \hat{\jmath}_B$, respectively.
  (c) Program the random oracle: Let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}}}$ be $\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}} = \mathsf{seed}''_{\hat{\jmath}_B}$. For $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ be $\mathsf{seed}^B_{j,\mathsf{OT}} = \mathsf{seed}''_j \oplus \mathsf{seed}'_j$.

  If the output is true, then $\mathcal{S}$ sets $\mathsf{caught} = \perp$ and continues the simulation.
- If $|J_s| = \lambda - 1$ and $|J_z| = 0$, $\mathcal{S}$ sets $\mathsf{Stupid} = \mathsf{true}$ and then sends $(\mathsf{stupidCheat}, \frac{\lambda-1}{\lambda})$ to $\mathcal{F}_{\mathsf{RobustPVC}}$. If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\mathsf{corrupted}$, $\mathcal{S}$ executes the *finishing touches* by picking $\hat{\jmath}_B \leftarrow\!\!\$\, J_s$, programming the random oracle to let the decryption of $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ be 0, decryption of $\mathsf{p}_{\mathsf{c}_j}$ be c_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, and decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ and $j = \hat{\jmath}_B$, respectively. $\mathcal{S}$ also programs the random oracle: for the $\hat{\jmath}_B$th instance, let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}}}$ be $\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}} = \mathsf{seed}''_{\hat{\jmath}_B}$. For $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ be $\mathsf{seed}^B_{j,\mathsf{OT}} = \mathsf{seed}''_j \oplus \mathsf{seed}'_j$. Then $\mathcal{S}$ opens the puzzle as in Step 8 of the protocol, generates the certificate cert with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}^A_j$ as in the protocol. $\mathcal{S}$ then sends cert to $\mathcal{A}$ as Step 9 of the protocol to complete the simulation.

  If $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\perp$, $\mathcal{S}$ executes the *finishing touches* by picking $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]\backslash J_s$ and programming the random oracle following the same procedure as $\mathcal{F}_{\mathsf{RobustPVC}}$ returns $\mathsf{corrupted}$. Then $\mathcal{S}$ opens his puzzle as in Step 8 of the protocol, and simulates the abortion of the honest $\mathsf{P_B}$ to terminate the simulation.
- If $|J_s| = \lambda$, $\mathcal{S}$ executes the *finishing touches* procedure as in Step 7a-7c in this simulation, opens his puzzle, and sends abort to $\mathcal{F}_{\mathsf{RobustPVC}}$ to terminate the execution of the protocol.

Rewind $\mathcal{A}$ and run steps $1'-7'$ below until[a] $|J'_s| = |J_s|$, $|J'_z| = |J_z|$, $\mathsf{caught}' = \mathsf{caught}$, $\mathsf{GoodEq}' = \mathsf{GoodEq}$, $\mathsf{Stupid}' = \mathsf{Stupid}$, and $\mathsf{Type}' = \mathsf{Type}$.

$1'$. $\mathcal{S}$ goes through the following steps with $\mathcal{A}$.
  (a) $\mathcal{S}$ receives the commitments $\{\mathsf{c}_{\mathsf{seed}^A_j}, \mathsf{c}_{\mathsf{seed}'^A_j}, \mathsf{c}_{\mathsf{witness}'^A_j}\}_{j\in[\lambda]}$ from $\mathcal{A}$. In the meantime, $\mathcal{S}$ playing the role of $\mathsf{P_B}$ randomly chooses $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]$ and then picks uniform κ-bit strings $\{\mathsf{seed}^B_j, \mathsf{seed}'^B_j,$

$\mathsf{witness'}^B_j\}_{j\in[\lambda]}$. $\mathcal{S}$ then computes $\mathsf{c}_{\mathsf{seed}^B_j}$, $\mathsf{c}_{\mathsf{seed'}^B_j}$, and $\mathsf{c}_{\mathsf{witness'}^B_j}$ for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ as in the protocol and let $\mathsf{c}_{\mathsf{seed}^B_{\hat{\jmath}_B}}$, $\mathsf{c}_{\mathsf{seed'}^B_{\hat{\jmath}_B}}$, and $\mathsf{c}_{\mathsf{witness'}^B_{\hat{\jmath}_B}}$ be dummy commitments. Finally, $\mathcal{S}$ sends these commitments to $\mathcal{A}$ as in the protocol.

(b) For all $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, $\mathcal{S}$ playing the role of the receiver uses $b_j = 0$ to retrieve the message $\{\mathsf{seed'}^A_j\}_{j\in[\lambda]\backslash\{\hat{\jmath}_B\}}$ with randomness derived from $\{\mathsf{seed}^B_j\}$. For the $\hat{\jmath}_B$th instance, $\mathcal{S}$ runs the simulator $\mathcal{S}_{\mathsf{OT}}$ for the protocol Π_{OT}, and extracts $(\mathsf{seed'}^A_{\hat{\jmath}_B}, \mathsf{witness'}^A_{\hat{\jmath}_B})$. $\mathcal{S}$ also plays the role of the sender and uses $\{(\mathsf{seed'}^B_j, \mathsf{witness'}^B_j)\}_{j\in[\lambda]}$ as input with randomness derived from seed^B_j.

2′. $\mathcal{S}$ computes $\mathsf{seed}'_j = \mathsf{seed'}^A_j \oplus \mathsf{seed'}^B_j$ for all $j \in [\lambda]$. Then $\mathcal{S}$ let $h^j_B \leftarrow \mathsf{H}(\bigoplus_{[\lambda]\backslash j} \mathsf{seed}'_j)$ for all $j \in [\lambda]$. $\mathcal{S}$ uses the simulator $\mathcal{S}_{\mathsf{Eq}}$ for Π_{Eq} to obtain h_A. If there exists h^j_B, such that $h^j_B = h_A$, $\mathcal{S}$ sets $\mathsf{GoodEq}' = \mathsf{true}$. Then if $h^{\hat{\jmath}_B}_B = h_A$, $\mathcal{S}$ returns true for Π_{Eq} to $\mathcal{A}$ and restart the protocol, and otherwise returns false for Π_{Eq} to $\mathcal{A}$. If there is no $h^j_B = h_A$, $\mathcal{S}$ sets $\mathsf{GoodEq}' = \mathsf{false}$ and returns false for Π_{Eq} to $\mathcal{A}$.

3′. $\mathcal{S}$ computes and sends commitment c_B to $\mathcal{A}$. $\mathcal{S}$ also receives commitment c_A from $\mathcal{A}$. $\mathcal{S}$ extracts $(\hat{\jmath}_A, \{\mathsf{seed'}^B_j\}_{j\neq\hat{\jmath}_A}, \mathsf{witness'}^B_{\hat{\jmath}_A})$ from c_A. If this extracted message is correct, store $\hat{\jmath}_A$. Otherwise, let $\hat{\jmath}_A = \perp$.

4′. For all $j \in [\lambda]$, $\mathcal{S}$ prepares garbled circuits and input materials as in Step 4 of the protocol.

5′. $\mathcal{S}$ computes and sends dummy $\{\mathsf{c}_{\gamma^B_j}\}$ to $\mathcal{A}$. $\mathcal{S}$ also receives $\{\mathsf{c}_{\gamma^A_j}\}$ from $\mathcal{A}$.

6′. $\mathcal{S}$ generates puzzles $\mathsf{p}_{s^B_j}$ for a random chosen master puzzle key s^B_j and ciphertexts $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ as in the protocol. $\mathcal{S}$ generates ciphertexts $\mathsf{p}_{\mathsf{c}_j}$ for computed c_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ and let the ciphertext $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ encrypt 0. Then $\mathcal{S}$ sends these puzzles to $\mathcal{A}$.
$\mathcal{S}$ receives puzzles $\mathsf{p}_{s^A_j}$, $\mathsf{p}_{\hat{\mathsf{c}}_j}$, and $\mathsf{p}_{\mathsf{seed}^A_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ from $\mathcal{A}$. Since time-lock puzzles can be solved in polynomial time, $\mathcal{S}$ can decrypt these puzzles and derive s^A_j, $\hat{\mathsf{c}}_j$, and $\mathsf{seed}^A_{j,\mathsf{OT}}$ for $j \in [\lambda]$.
For all $j \neq \hat{\jmath}_B$, $\mathcal{S}$ playing the role of the receiver runs Π_{OT} with $\mathcal{A}$, using input 0^{n_B} and randomness derived from seed^B_j. In this way, $\mathcal{S}$ receives $\mathcal{A}$'s $\{\mathsf{p}_{B_{j,i,0}}\}$. In the $\hat{\jmath}_B$th execution of Π_{OT}, $\mathcal{S}$ playing the role of the receiver uses the simulator $\mathcal{S}_{\mathsf{OT}}$ for Π_{OT} to extract $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,b}}\}_{i\in[n_B],b\in\{0,1\}}$. $\mathcal{S}$ can decrypt $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,b}}\}_{i\in[n_B],b\in\{0,1\}}$ and obtain $\{B_{\hat{\jmath}_B,i,b}\}_{i\in[n_B],b\in\{0,1\}}$.
If $\hat{\jmath}_A = \perp$, $\mathcal{S}$ first generates ciphertexts for $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]$ as in the protocol. Then, $\mathcal{S}$ playing the role of the sender executes Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from $\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}$ or $\mathsf{seed}^B_{j,\mathsf{OT}}$ as in the protocol. If $\hat{\jmath}_A \neq \perp$, $\mathcal{S}$ generates ciphertexts for $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]\backslash\{\hat{\jmath}_A\}$ as in the protocol. For $[\lambda]\backslash\{\hat{\jmath}_A\}$, $\mathcal{S}$ playing the role of the sender executes Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from $\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}$ or $\mathsf{seed}^B_{j,\mathsf{OT}}$ as in the protocol. For $j = \hat{\jmath}_A$, $\mathcal{S}$ runs the simulator $\mathcal{S}_{\mathsf{OT}}$ for Π_{OT} to extract the input x_A and returns dummy ciphertexts $\{\mathsf{p}_{A_{\hat{\jmath}_A,i}}\}$ to $\mathcal{A}$.

7′. $\mathcal{S}$ generates $\{\sigma^B_j\}$ as an honest $\mathsf{P_B}$ in the protocol and sends them to $\mathcal{A}$. $\mathcal{S}$ also receives $\{\sigma^A_j\}$ from $\mathcal{A}$. If any of the signatures are invalid, then return to Step 1′.
Then $\mathcal{S}$ simulates the computation of an honest $\mathsf{P_A}$ playing the role of the sender in the executions of Π_{OT} in Step 6 of the protocol and obtain $\hat{\mathcal{H}}^A_j$. If $\hat{\mathsf{c}}_j = 0$, the simulation is based on $\mathsf{seed}^A_{j,\mathsf{OT}}$ as the $\hat{\jmath}_A$th instance in the protocol. If $\hat{\mathsf{c}}_j = \mathsf{c}_j$, the simulation is based on $\mathsf{seed}'_j \oplus \mathsf{seed}^A_{j,\mathsf{OT}}$ as other instances in the protocol. Let J'_z be the set of indices that $\hat{\mathsf{c}}_j = 0$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the $\hat{\jmath}_A$th instance as described in the protocol. Let J'_s be the set of indices that $\hat{\mathsf{c}}_j = \mathsf{c}_j$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the checking instances.
Hence, there are following cases (let $\mathsf{Stupid}' = \mathsf{false}$ and $\mathsf{Type}' = \perp$ by default):

- If $|J'_s| < \lambda - 2$, then return to Step 1′.
- If $|J'_s| = \lambda - 2$ and $|J'_z| = 2$, $\mathcal{S}$ sets $\mathsf{Stupid}' = \mathsf{true}$. If $\hat{\jmath}_B \in J'_s$, $\mathcal{S}$ sets $\mathsf{caught}' = (\frac{\lambda-2}{\lambda}, \mathsf{true})$ and $\mathsf{Type}' = \mathsf{BadNum}$. Otherwise, $\mathcal{S}$ sets $\mathsf{caught}' = (\frac{\lambda-2}{\lambda}, \mathsf{false})$.
- When $|J'_s| = \lambda - 2$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{true}$, if $\hat{\jmath}_B \in J'_s$, $\mathcal{S}$ sets $\mathsf{caught}' = (\frac{\lambda-2}{\lambda-1}, \mathsf{true})$ and Type' according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}^A_j$. Otherwise, $\mathcal{S}$ sets $\mathsf{caught}' = (\frac{\lambda-2}{\lambda-1}, \mathsf{false})$.
- When $|J'_s| = \lambda - 2$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{false}$, if $\hat{\jmath}_B \in J'_s$ or $\hat{\jmath}_B \in J'_z$, $\mathcal{S}$ sets $\mathsf{caught}' = (\frac{\lambda-1}{\lambda}, \mathsf{true})$ and Type' according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}^A_j$. Otherwise, $\mathcal{S}$ sets $\mathsf{caught}' = (\frac{\lambda-1}{\lambda}, \mathsf{false})$.
- If $|J'_s| = \lambda - 2$ and $|J'_z| = 0$, then return to Step 1′.
- If $|J'_s| = \lambda - 1$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{true}$, $\mathcal{S}$ sets $\mathsf{caught}' = \perp$.
- When $|J'_s| = \lambda - 1$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{false}$, if $\hat{\jmath} \in J_s$, then $\mathcal{S}$ sets $\mathsf{caught}' = \perp$. Otherwise, return to Step 1′.
- If $|J'_s| = \lambda - 1$ and $|J'_z| = 0$, then return to Step 1′.
- If $|J'_s| = \lambda$, then return to Step 1′.

8. If $\hat{\jmath}_A \neq \perp$, $\mathcal{S}$ sends x_A to $\mathcal{F}_{\mathsf{RobustPVC}}$ and receives the output z. Then $\mathcal{S}$ regenerates the garbled circuit via
$$(\{A_{\hat{\jmath}_A,i}\}, \{B_{\hat{\jmath}_A,i}\}, \mathsf{GC}_{\hat{\jmath}_A}, \{Z_{\hat{\jmath}_A,i,b}\}) \leftarrow \mathcal{S}_{\mathsf{Gb}}(1^\kappa, \mathcal{C}, z)$$
and recomputes the materials of the $\hat{\jmath}_A$th instance in Step 4b and 4c of the protocol. Since this garbled circuit is simulated, $\mathcal{S}$ uses/programs dummy commitments/values in case of need, *e.g.*, for $\gamma^B_{\hat{\jmath}_A}$. $\mathcal{S}$ also uses the random oracle to program the opening of $\mathsf{c}_{\hat{\jmath}_A}$ and $\{\mathsf{p}_{A_{\hat{\jmath}_A,i}}\}$ with respect to the simulated garbled circuit. Then $\mathcal{S}$ opens his puzzle to $\mathcal{A}$ as in the protocol. $\mathcal{A}$ may also open her puzzle.
9. If $\mathsf{caught}' = (\cdot, \mathsf{true})$, $\mathcal{S}$ generates the corresponding cert and sends it to $\mathcal{A}$ to complete the simulation.
10. $\mathcal{S}$ sends the opening for c_B to $\mathcal{A}$ as in the protocol. $\mathcal{S}$ also receives the opening for c_A from $\mathcal{A}$. If $\mathsf{Stupid}' = \mathsf{true}$, $\mathcal{S}$ simulates the abortion of the honest party to complete the simulation. If the opening $(\hat{\jmath}_A, \{\mathsf{seed}'^B_j\}_{j\neq\hat{\jmath}_A}, \mathsf{witness}'^B_{\hat{\jmath}_A})$ is not correct, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{RobustPVC}}$ and simulates the abortion of the honest party to complete the simulation.
11. $\mathcal{S}$ sends the simulated garbled circuit $\mathsf{GC}_{\hat{\jmath}_A}$, together with corresponding label hash values, to $\mathcal{A}$. $\mathcal{S}$ also receives the garbled circuit $\mathsf{GC}_{\hat{\jmath}_B}$ and related materials from $\mathcal{A}$. If they are invalid, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{RobustPVC}}$ and simulates the abortion of the honest party to complete the simulation.
12. $\mathcal{S}$ sends the simulated labels, $\gamma^B_{\hat{\jmath}_A}$, and corresponding decommitments to $\mathcal{A}$. $\mathcal{S}$ also receives the labels, $\gamma^A_{\hat{\jmath}_B}$, and corresponding decommitments. If the received materials are invalid, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{RobustPVC}}$ and simulates the abortion of the honest party to complete the simulation.
13. $\mathcal{S}$ uses the simulator $\mathcal{S}_{\mathsf{Eq}}$ for Π_{Eq} to obtain β_A from $\mathcal{A}$. Then $\mathcal{S}$ defines the boolean function g as follows.
    (a) On input $x_B \in \{0,1\}^{n_B}$, select the corresponding input labels for x_B in the label tuples $\{(B_{\hat{\jmath}_B,i,0}, B_{\hat{\jmath}_B,i,1})\}_{i\in[n_B]}$.
    (b) Evaluate $\mathsf{GC}_{\hat{\jmath}_B}$ with the input-wire labels $\{A_{\hat{\jmath}_B,i,x_A[i]}\}_{i\in[n_A]}$ and $\{B_{\hat{\jmath}_B,i,x_B[i]}\}_{i\in[n_B]}$ as in the protocol to obtain $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$ and $z_{\hat{\jmath}_B}$. In particular, if some error occurs in the evaluation of garbled circuit, some hard-coded random values provided by $\mathcal{S}$ are used for $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$ and $z_{\hat{\jmath}_B}$.
    (c) Compute $\beta_B \leftarrow \mathsf{H}(\bigoplus_{i=1}^{n_O}(Z_{\hat{\jmath}_A,i,z_{\hat{\jmath}_B}[i]} \oplus Z_{\hat{\jmath}_B,i,z_{\hat{\jmath}_B}[i]}))$.
    (d) If $\beta_A = \beta_B$, return true. Otherwise, return false.
    $\mathcal{S}$ sends g to $\mathcal{F}_{\mathsf{RobustPVC}}$, receives the output e from $\mathcal{F}_{\mathsf{RobustPVC}}$, and gives e to $\mathcal{A}$. If $\mathcal{A}$ does not abort, $\mathcal{S}$ sends $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{RobustPVC}}$ to complete the simulation.

[a] We could use standard techniques [13,15] to ensure that $\mathcal{S}$ runs in expected polynomial time.

Now it remains to show that the joint distribution of the view of $\mathcal{A}$ simulated by $\mathcal{S}$ and the output of $\mathsf{P_B}$ in the ideal world is computationally indistinguishable from the joint distribution of the view of $\mathcal{A}$ and the output of $\mathsf{P_B}$ in the real world. We provide the detailed proof in Supplementary Material B.

Public Verifiability We argue that whenever an honest party P_X outputs the message $\mathsf{corrupted}$, this party should also be capable of producing a valid certificate for the malicious party P_Y corrupted by the adversary. This implies that once the honest party detects the adversary's cheating behavior, the adversary cannot prevent the honest party from generating the certificate.

Note that in Step 7, without receiving a valid signature, the honest party will not continue the execution of the protocol. Alternatively, if the honest party does receive valid signatures, though this party still cannot obtain the solutions for the time-lock puzzles/ciphertexts at this time, it is guaranteed that the adversary has signed the time-lock puzzles/ciphertexts. Meanwhile, the adversary still cannot access the solution for the time-lock puzzles and remains unaware of the indices of the instances chosen by P_X. Therefore, the adversary cannot base her decision to continue or abort on these indices.

When the adversary gains knowledge of the indices of instances chosen by P_X to verify, the honest party has already gathered sufficient evidence to attribute blame to the adversary. The honest party can locally solve the adversary's puzzles to obtain s^Y_j, c_j, and $\mathsf{seed}^Y_{j,\mathsf{OT}}$ sent by the adversary. These messages, combined with the committed seeds and witness, as well as trans^X_j and trans^Y_j, provide enough information for anyone to simulate the execution of the checked instance and obtain $\hat{\mathsf{c}}_j$ and $\hat{\mathcal{H}}^Y_j$. Anyone can conduct the same verification on $\hat{\mathsf{c}}_j$ and $\hat{\mathcal{H}}^Y_j$ as P_X to determine if malicious P_Y deviated from the protocol execution. Since P_X's signature is provided, the non-repudiation property is ensured.

Defamation Freeness Assuming an honest party P_X is accused by the adversary corrupting P_Y with a valid certificate, it implies that the adversary will provide valid signature(s) of P_X for the message(s)

$$(\mathcal{C}, j, \mathsf{c}_{\mathsf{seed}_j^Y}, \mathsf{c}_{\mathsf{seed}'^Y_j}, \mathsf{c}_{\mathsf{witness}'^Y_j}, \mathsf{trans}_j^X, \mathsf{trans}_j^Y, \mathcal{H}_j^X, \mathsf{p}_{s_j^X}, \mathsf{p}_{\mathsf{c}_j}, \mathsf{p}_{\mathsf{seed}^X_{j,\mathsf{OT}}}, \tau, \lambda)\,.$$

Since P_X is honest, she only signs the puzzles $\mathsf{p}_{s_j^X}$, $\mathsf{p}_{\mathsf{c}_j}$, and $\mathsf{p}_{\mathsf{seed}^X_{j,\mathsf{OT}}}$ she has generated. These puzzles can be decrypted to derive s_j^X, c_j, and $\mathsf{seed}^X_{j,\mathsf{OT}}$. According to the soundness of the verifiable time-lock puzzle scheme, the adversary cannot provide openings/proofs for solutions different from s_j^X, c_j, and $\mathsf{seed}^X_{j,\mathsf{OT}}$.

First, we consider the certificate with type BadCom. In this setting, the adversary aims to demonstrate that c_j does not match the commitment generated from the seed'_j (and $\mathsf{c}_j \neq 0$, which is obvious). Due to the binding property, the adversary can only provide the valid committed values seed_j^Y, seed'^Y_j, and $\mathsf{witness}'^Y_j$ inside $\mathsf{c}_{\mathsf{seed}_j^Y}$, $\mathsf{c}_{\mathsf{seed}'^Y_j}$, and $\mathsf{c}_{\mathsf{witness}'^Y_j}$. A certificate verifier will recompute seed'^X_j from trans_j^X with randomness derived from seed_j^Y. On the one hand, since seed'^X_j is the output of a perfectly correct OT protocol, given the signed transcript of Π_{OT}, there is exactly one valid output for P_Y that is consistent with trans_j^X, regardless of P_Y's randomness and inputs. On the other hand, if the committed values seed'^Y_j and $\mathsf{witness}'^Y_j$ are not the inputs of P_Y playing the role of the sender in the perfectly correct protocol Π_{OT}, these committed values do not match the signed transcript of Π_{OT}. Therefore, $\mathsf{seed}'_j = \mathsf{seed}'^A_j \oplus \mathsf{seed}'^B_j$ should be the same as the one derived by the honest P_X. Consequently, simulation for the computation conducted by P_X always leads to the same c_j, and the algorithm Judge will not output true (except for a negligible probability).

Then we consider the certificate with type BadOT. Similarly, we know that the adversary cannot forge $\mathsf{seed}'_j = \mathsf{seed}'^A_j \oplus \mathsf{seed}'^B_j$, $\mathsf{seed}^X_{j,\mathsf{OT}}$, and s_j^X, and a certificate verifier should derive the same $\{(\mathsf{p}_{Y_{j,i,0}}, \mathsf{p}_{Y_{j,i,1}})\}$ and $\mathsf{seed}'_j \oplus \mathsf{seed}^X_{j,\mathsf{OT}}$. Hence, given the signed $\mathcal{H}_j^X$, the algorithm Judge will not output true except for a negligible probability (when a collision of the hash function is found).

Finally, for the certificate with type BadNum, we know that the adversary cannot forge c_j. Hence, the honest P_X should always have exactly one $\mathsf{c}_j = 0$. Therefore, the algorithm Judge will not output true (except for a negligible probability).

Therefore, the protocol $\Pi_{\mathsf{RobustPVC}}$ along with algorithms Blame and Judge is robust publicly verifiable secure with deterrence $\epsilon = 1 - \frac{1}{\lambda - 1}$. □

5 Discussion

In this section, we discuss the performance and possible enhancement of our protocol $\Pi_{\mathsf{RobustPVC}}$.

5.1 Comparison

The protocol $\Pi_{\mathsf{RobustPVC}}$ is building upon the state-of-the-art secure two-party computation protocol with PVC security [17]. As discussed in Section 1.2, our protocol incurs low additional overhead. We can easily observe that, compared to the protocol in [17], our protocol only requires an additional garbled circuit generation and an additional garbled circuit evaluation (see Table 1). Interestingly, as the deterrence factor increases, the additional overhead for garbled circuit generation/evaluation in our protocol compared to the protocol in [17] decreases. As emphasized in [17], the cost of generating garbled circuits (unless the circuit is very small) is the efficiency bottleneck of their protocol, making the number of garbled circuit generations in our protocol highly desirable.

Table 1. Number of garbled circuit generation and evaluation needed to achieve different deterrence factor ϵ, along with the additional overhead of our $\Pi_{\mathsf{RobustPVC}}$ compared to the protocol in [17]. Since evaluations should be faster than generations, the additional cost is actually overestimated.

ϵ	PVC-secure protocol [17]			Our protocol $\Pi_{\mathsf{RobustPVC}}$			Additional
	$\mathsf{P_A}$	$\mathsf{P_B}$	Total	$\mathsf{P_A}$	$\mathsf{P_B}$	Total	overhead
80%	$5+0=5$	$4+1=5$	10	$5+1=6$	$5+1=6$	12	20%
87.5%	$8+0=8$	$7+1=8$	16	$8+1=9$	$8+1=9$	18	12.5%
90%	$10+0=10$	$9+1=10$	20	$10+1=11$	$10+1=11$	22	10%
95%	$20+0=20$	$19+1=20$	40	$20+1=21$	$20+1=21$	42	5%

In the following, we argue that each phase of our protocol incurs low additional overhead compared to the protocol in [17].

Seed Preparation It is easy to verify that the expected number of executions for this phase is $2-\epsilon$, where ϵ is the deterrence factor. For $\epsilon = 90\%$, the expected number of executions is only 1.1. For $\epsilon = 95\%$, the expected number is reduced to only 1.05. Additionally, the secure equality test can be completed very quickly [18,7]. Therefore, the overhead of this phase is low.

Circuit and Input Preparation In our protocol, two parties can perform garbled circuit generation in parallel during this phase. For the same deterrence factor, the number of garbled circuits generated in [17] is the same as in $\Pi_{\mathsf{RobustPVC}}$ (for each party). Therefore, the running time for garbled circuit generation is approximately the same as that of the protocol in [17]. We also note that time-lock puzzle generation is efficient. Indeed, for *all* instances, a single master puzzle key (inside a *real* time-lock puzzle) is sufficient. Hence, the overhead of this phase is low.

Publicly Verifiable Evidence Creation This phase is almost identical to the protocol in [17].

Check of Instances If both parties are honest and the master time-lock puzzle is correctly opened, the computation performed by both parties (in parallel) mainly involves the simulation of Π_{OT}, which incurs a similar running time as the protocol in [17]. Moreover, compared to the protocol in [17], no simulation of garbled circuit generation is needed. Therefore, our protocol may outperform the protocol in [17] in this phase.

Circuit Evaluation In this phase, both parties can evaluate the garbled circuit in parallel. Hence, the running time required for garbled circuit evaluation is almost the same as that of the protocol in [17]. Additionally, secure equality tests can be completed quickly. Therefore, the overhead of this phase is low.

Based on the implementation presented in [17], as an example, when $\epsilon = 87.5\%$, *i.e.*, the number of garbled circuit is 8, the execution time of their PVC-secure protocol for an SHA-256 circuit is 71.31 ms in LAN and 2436 ms in WAN. In comparison, the corresponding semi-honest protocol takes 38.04 ms in LAN and 1080ms in WAN, while the maliciously secure protocol takes 611.7 ms in LAN and 17300 ms in WAN. Given that the performance of our protocol is similar to that of the protocol in [17], it is evident that our $\Pi_{\mathsf{RobustPVC}}$ achieves comparable performance to the semi-honest protocol while significantly outperforming the maliciously secure protocol.

5.2 Size of Certificate

In $\Pi_{\mathsf{RobustPVC}}$, we have chosen not to include the circuit description $\mathcal{C}$ in the certificate. Instead, we make the assumption that the circuit description is commonly known. This practice aligns with the convention followed in PVC-secure protocols [17,9,10,31,3]. Alternatively, one could opt to include the circuit description $\mathcal{C}$ in the certificate. It is worth noting that the description of $\mathcal{C}$ can be significantly shorter than the full circuit, such as representing it in high-level code or utilizing an ID number from a widely used "reference circuits" database.

Due to the exclusion of the circuit description in the certificate, the size of publicly verifiable certificates generated in the $\Pi_{\mathsf{RobustPVC}}$ protocol remains constant.

5.3 Potential Enhancements

In our protocol $\Pi_{\mathsf{RobustPVC}}$, we do not aim to prevent an attacker from learning the evaluation result $\mathcal{C}(x_A, x_B)$ even when the honest party outputs cheating at the end of the protocol. This occurs when the attacker generates a malicious garbled circuit but not being caught (with probability $1-\epsilon$) or when the attacker causes an abortion in the final equality test protocol.

We note this issue is not the focus of our protocol. In fact, it is well known that two-party computation cannot guarantee fairness in general [6]. But there are still some countermeasures that can be readily incorporated into our protocols.

For instance, one approach is to introduce an additional circuit that processes the output-wire labels before they are used in the equality test, effectively concealing the semantic values associated with the garbled circuit's output. Another technique, known as *progressive revelation*, allows for a gradual release of the evaluation result bit-by-bit, preventing the attacker from obtaining the complete input while leaving the honest party with nothing. For more detailed information on these countermeasures, we refer interested readers to the work [18].

Acknowledgments. We would like to express our sincere appreciation to the anonymous reviewers for their valuable comments. Yi Liu was supported by National Natural Science Foundation of China under Grant No. 62302194. Junzuo Lai was supported by National Natural Science Foundation of China under Grant No. U2001205, Guangdong Basic and Applied Basic Research Foundation (Grant No. 2023B1515040020). Qi Wang was supported by Shenzhen Key Laboratory of Safety and Security for Next Generation of Industrial Internet under Grant No. ZDSYS20210623092007023 and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation under Grant No. 2020B121201001. Anjia Yang was supported by National Key Research and Development Program of China under Grant No. 2021ZD0112802 and National Natural Science Foundation of China under Grant No. 62072215. Jian Weng was supported by National Natural Science Foundation of China under Grant Nos. 61825203, 62332007 and U22B2028, Major Program of Guangdong Basic and Applied Research Project under Grant No. 2019B030302008, Guangdong Provincial Science and Technology Project under Grant No. 2021A0505030033, Science and Technology Major Project of Tibetan Autonomous Region of China under Grant No. XZ202201ZD0006G, National Joint Engineering Research Center of Network Security Detection and Protection Technology, Guangdong Key Laboratory of Data Security and Privacy Preserving, Guangdong Hong Kong Joint Laboratory for Data Security and Privacy Protection, and Engineering Research Center of Trustworthy AI, Ministry of Education.

References

1. Agrawal, N., Bell, J., Gascón, A., Kusner, M.J.: Mpc-friendly commitments for publicly verifiable covert security. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. pp. 2685–2704. ACM (2021)
2. Asharov, G., Orlandi, C.: Calling out cheaters: Covert security with public verifiability. In: Wang, X., Sako, K. (eds.) Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7658, pp. 681–698. Springer (2012)
3. Attema, T., Dunning, V., Everts, M.H., Langenkamp, P.: Efficient compiler to covert security with public verifiability for honest majority MPC. In: Ateniese, G., Venturi, D. (eds.) Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13269, pp. 663–683. Springer (2022)
4. Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient protocols for realistic adversaries. J. Cryptol. **23**(2), 281–343 (2010)
5. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. pp. 784–796. ACM (2012)
6. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: Hartmanis, J. (ed.) Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA. pp. 364–369. ACM (1986)
7. Couteau, G.: New protocols for secure equality test and comparison. In: Preneel, B., Vercauteren, F. (eds.) Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10892, pp. 303–320. Springer (2018)
8. Damgård, I., Geisler, M., Nielsen, J.B.: From passive to covert security at low cost. In: Micciancio, D. (ed.) Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings. Lecture Notes in Computer Science, vol. 5978, pp. 128–145. Springer (2010)
9. Damgård, I., Orlandi, C., Simkin, M.: Black-box transformations from passive to covert security with public verifiability. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12171, pp. 647–676. Springer (2020)
10. Faust, S., Hazay, C., Kretzler, D., Schlosser, B.: Generic compiler for publicly verifiable covert multi-party computation. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12697, pp. 782–811. Springer (2021)
11. Faust, S., Hazay, C., Kretzler, D., Schlosser, B.: Financially backed covert security. In: Hanaoka, G., Shikata, J., Watanabe, Y. (eds.) Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13178, pp. 99–129. Springer (2022)

12. Goldreich, O.: The Foundations of Cryptography - Volume 2: Basic Applications. Cambridge University Press (2004)
13. Goldreich, O., Kahan, A.: How to construct constant-round zero-knowledge proof systems for NP. J. Cryptol. **9**(3), 167–190 (1996)
14. Goyal, V., Mohassel, P., Smith, A.D.: Efficient two party and multi party computation against covert adversaries. In: Smart, N.P. (ed.) Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4965, pp. 289–306. Springer (2008)
15. Hazay, C., Lindell, Y.: Efficient Secure Two-Party Protocols - Techniques and Constructions. Information Security and Cryptography, Springer (2010)
16. Hazay, C., Shelat, A., Venkitasubramaniam, M.: Going beyond dual execution: MPC for functions with efficient verification. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12111, pp. 328–356. Springer (2020)
17. Hong, C., Katz, J., Kolesnikov, V., Lu, W., Wang, X.: Covert security with public verifiability: Faster, leaner, and simpler. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11478, pp. 97–121. Springer (2019)
18. Huang, Y., Katz, J., Evans, D.: Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA. pp. 272–284. IEEE Computer Society (2012)
19. Huang, Y., Katz, J., Evans, D.: Efficient secure two-party computation using symmetric cut-and-choose. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II. Lecture Notes in Computer Science, vol. 8043, pp. 18–35. Springer (2013)
20. Kolesnikov, V., Malozemoff, A.J.: Public verifiability in the covert model (almost) for free. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9453, pp. 210–235. Springer (2015)
21. Kolesnikov, V., Mohassel, P., Riva, B., Rosulek, M.: Richer efficiency/security trade-offs in 2pc. In: Dodis, Y., Nielsen, J.B. (eds.) Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9014, pp. 229–259. Springer (2015)
22. Lindell, Y.: Fast cut-and-choose-based protocols for malicious and covert adversaries. J. Cryptol. **29**(2), 456–490 (2016)
23. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two-party computation. J. Cryptol. **22**(2), 161–188 (2009)
24. Liu, Y., Wang, Q., Yiu, S.: Making private function evaluation safer, faster, and simpler. In: Hanaoka, G., Shikata, J., Watanabe, Y. (eds.) Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13177, pp. 349–378. Springer (2022)
25. Liu, Y., Wang, Q., Yiu, S.: Towards practical homomorphic time-lock puzzles: Applicability and verifiability. In: Atluri, V., Pietro, R.D., Jensen, C.D., Meng, W. (eds.) Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13554, pp. 424–443. Springer (2022)
26. Malavolta, G., Thyagarajan, S.A.K.: Homomorphic time-lock puzzles and applications. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11692, pp. 620–649. Springer (2019)
27. Mohassel, P., Franklin, M.K.: Efficiency tradeoffs for malicious two-party computation. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3958, pp. 458–473. Springer (2006)
28. Mohassel, P., Riva, B.: Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II. Lecture Notes in Computer Science, vol. 8043, pp. 36–53. Springer (2013)
29. Rindal, P., Rosulek, M.: Faster malicious 2-party secure computation with online/offline dual execution. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 297–314. USENIX Association (2016)

30. Rindal, P., Rosulek, M.: Malicious-secure private set intersection via dual execution. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1229–1242. ACM (2017)
31. Scholl, P., Simkin, M., Siniscalchi, L.: Multiparty computation with covert security and public verifiability. Cryptology ePrint Archive, Report 2021/366 (2021), https://ia.cr/2021/366
32. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11478, pp. 379–407. Springer (2019)
33. Zhu, R., Ding, C., Huang, Y.: Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 633–650. ACM (2019)

Supplementary Material

A Verifiable Time-Lock Puzzle

A.1 Definition

Definition 3. *A verifiable time-lock puzzle scheme* TLP *should satisfy* completeness, correctness for opening and proof, soundness, *and* security *as follows.*

Completeness *For all* κ, τ, $pp \leftarrow \mathsf{TLP.Setup}(1^\kappa, \tau)$, *and all* $s \in \mathbb{S}(\kappa)$, *we have*

$$(s, \cdot) \leftarrow \mathsf{TLP.Solve}(\mathsf{TLP.Gen}(pp, s))\,.$$

Correctness for Opening *For all* κ, τ, $pp \leftarrow \mathsf{TLP.Setup}(1^\kappa, \tau)$, $s \in \mathbb{S}$, *and* $(\mathsf{p}, \pi) \leftarrow \mathsf{TLP.Gen}(pp, s)$, *we have*

$$\mathsf{TLP.Verify}(pp, \mathsf{p}, s, \pi) = \mathsf{true}\,.$$

Correctness for Proof *For all* κ, τ, $pp \leftarrow \mathsf{TLP.Setup}(1^\kappa, \tau)$, $s \in \mathbb{S}$, *and* $(\mathsf{p}, \pi) \leftarrow \mathsf{TLP.Gen}(pp, s)$, *if* $(s, \pi') \leftarrow \mathsf{TLP.Solve}(pp, \mathsf{p})$, *we have*

$$\mathsf{TLP.Verify}'(pp, \mathsf{p}, s, \pi') = \mathsf{true}\,.$$

Soundness *For all* κ, τ, *and all* PPT *adversary* $\mathcal{A}$, *the following probability is negligible in* κ*:*

$$\Pr\left[\begin{array}{l}\left(\begin{array}{l}\mathsf{TLP.Verify}(pp, \hat{\mathsf{p}}, \hat{s}, \hat{\pi}) = \mathsf{true} \\ \vee\ \mathsf{TLP.Verify}'(pp, \hat{\mathsf{p}}, \hat{s}, \hat{\pi}) = \mathsf{true}\end{array}\right) \\ \quad \wedge\ \hat{s} \neq s\end{array} : \begin{array}{l} pp \leftarrow \mathsf{TLP.Setup}(1^\kappa, \tau(\kappa)); \\ (\hat{\mathsf{p}}, \hat{s}, \hat{\pi}) \leftarrow \mathcal{A}(1^\kappa, pp, \tau(\kappa)); \\ (s, \cdot) \leftarrow \mathsf{TLP.Solve}(pp, \hat{\mathsf{p}}); \end{array}\right].$$

Security *The verifiable time-lock puzzle scheme* TLP *is secure with gap* $\varepsilon < 1$ *if there exists a polynomial* $\tilde{\tau}(\cdot)$, *such that for all polynomials* $\tau(\cdot) \geq \tilde{\tau}(\cdot)$ *and all polynomial-size adversaries* $(\mathcal{A}_1, \mathcal{A}_2) = \{(\mathcal{A}_1, \mathcal{A}_2)_\kappa\}_{\kappa \in \mathbb{N}}$, *where the depth of* $\mathcal{A}_2$ *is bounded from above by* $\tau^\varepsilon(\kappa)$, *we have*

$$\Pr\left[b \leftarrow \mathcal{A}_2(pp, \mathsf{p}, \tau) : \begin{array}{l}(\tau, s_0, s_1) \leftarrow \mathcal{A}_1(1^\kappa, \tau(\kappa)); \\ pp \leftarrow \mathsf{Setup}(1^\kappa, \tau(\kappa)); \\ b \leftarrow_{\$} \{0,1\}; (\mathsf{p}, \cdot) \leftarrow \mathsf{Gen}(pp, s_b);\end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\kappa)\,,$$

where $s_0, s_1 \in \mathbb{S}$.

A.2 Construction

In the following, we present the construction of a verifiable time-lock puzzle scheme. This scheme draws inspiration from Wesolowski's public verifiable approach for verifiable delay functions (VDF) as proposed in [32] for generating the proof of solution (see more in [26,10,25]).

Construction Verifiable Time-Lock Puzzle

$\mathsf{Setup}(1^\kappa, \tau)$ Sample two strong RSA modulus $N = pq$, where $p = 2p' + 1$ and $q = 2q' + 1$. Then pick a random generator g of $\mathbb{J}_N$, where $\mathbb{J}_N$ is a subgroup of $\mathbb{Z}_n^*$ for elements with Jacobi symbol $+1$. Compute $h \leftarrow g^{2^\tau} \bmod N$, where $2^\tau \bmod 2p'q'$ can be computed first to speed up the computation. Output $pp = (\tau, N, g, h)$.

$\mathsf{Gen}(pp, s)$ On input $pp = (\tau, N, g, h)$ and $s \in \mathbb{J}_n$, pick $r \leftarrow_{\$} [\lceil n/2 \rceil]$, compute $u \leftarrow g^r \bmod N$ and $v \leftarrow s \cdot h^r \bmod N$. Output $\mathsf{p} = (u, v)$ and $\pi = r$.

$\mathsf{Solve}(pp, \mathsf{p})$ Parse $\mathsf{p} = (u, v)$. Compute $w \leftarrow u^{2^\tau} \bmod N$ by repeated squaring. Then compute $s \leftarrow w^{-1} v \bmod N$. Generate a prime $\ell \leftarrow \mathsf{H}_{\mathsf{prime}}(\mathsf{bin}(\mathsf{p})) \in \mathsf{Prime}(2\kappa)$, where $\mathsf{Prime}(2\kappa)$ is the set of the first $2^{2\kappa}$ primes and $\mathsf{H}_{\mathsf{prime}}$ is a hash function mapping arbitrary strings to $\mathsf{Prime}(2\kappa)$. Write $2^T = q\ell + r'$, where $q, r' \in \mathbb{Z}$ and $0 \leq r' < \ell$, and compute[a] $\pi' \leftarrow u^q \bmod N$. Output s and π'

$\mathsf{Verify}(pp, \mathsf{p}, s, \pi)$ Parse $\mathsf{p} = (u, v)$. Check whether $\pi \in [\lceil n/2 \rceil]$, $u = g^r \bmod N$ and $v = s \cdot h^r \bmod N$. It they hold, output true. Otherwise, output false.

$\mathsf{Verify}'(pp, \mathsf{p}, s, \pi')$ Parse $\mathsf{p} = (u, v)$. Compute $r' \leftarrow 2^T \bmod \ell$. Output true if $\pi' \in \mathbb{Z}_n$ and $v = s \cdot \pi'^{\ell} u^{r'} \bmod N$, and false otherwise.

[a] Note that $\pi' \leftarrow u^q \bmod n$ can be efficiently computed by $\tau \cdot 3/\log(\tau)$ group operations that are allowed to be parallelized (see [32] for more information).

B Proof of Simulatability for Theorem 1

Proof. Based on the simulation in the proof of Theorem 1, our goal now is to show that the joint distribution of the view of $\mathcal{A}$ and the output of $\mathsf{P_B}$ in the ideal world is computationally indistinguishable from the joint distribution of the view of $\mathcal{A}$ and the output of $\mathsf{P_B}$ in the real world. We define the following experiments and let the output of each experiment be the view of $\mathcal{A}$ and the output of $\mathsf{P_B}$.

Expt$_0$ This is the ideal-world execution between $\mathcal{S}$ (as described in the proof of Theorem 1) and the honest party $\mathsf{P_B}$ holding some input x_B, both interacting with $\mathcal{F}_{\mathsf{RobustPVC}}$. We inline the actions of $\mathcal{S}$, $\mathcal{F}_{\mathsf{RobustPVC}}$, and $\mathsf{P_B}$, and rewrite the experiment as follows.

0. Use KGen to generate a pair of key $(\mathsf{vk}_B, \mathsf{sigk}_B)$ and send vk_B to $\mathcal{A}$.
1. Go through the following steps with $\mathcal{A}$.
   (a) Receive $\{\mathsf{c}_{\mathsf{seed}_j^A}, \mathsf{c}_{\mathsf{seed}'^A_j}, \mathsf{c}_{\mathsf{witness}'^A_j}\}_{j\in[\lambda]}$ from $\mathcal{A}$. Meanwhile, pick uniform κ-bit strings $\{\mathsf{seed}_j^B, \mathsf{seed}'^B_j, \mathsf{witness}'^B_j\}_{j\in[\lambda]}$, compute $\{\mathsf{c}_{\mathsf{seed}_j^B}, \mathsf{c}_{\mathsf{seed}'^B_j}, \mathsf{c}_{\mathsf{witness}'^B_j}\}_{j\in[\lambda]}$, and send these commitments to $\mathcal{A}$ as in the protocol.
   (b) For all j, play the role of the receiver and use $b_j = 0$ to retrieve seed'^A_j in Π_{OT} with randomness derived from seed_j^B. Plays the role of the sender and use $\{(\mathsf{seed}'^B_j, \mathsf{witness}'^B_j)\}_{j\in[\lambda]}$ as input in Π_{OT} with randomness derived from seed_j^B.
2. Compute $\mathsf{seed}'_j = \mathsf{seed}'^A_j \oplus \mathsf{seed}'^B_j$ for all $j \in [\lambda]$. Then let $h_B^j \leftarrow \mathsf{H}(\bigoplus_{[\lambda]\setminus j} \mathsf{seed}'_j)$ for all $j \in [\lambda]$. For the execution of Π_{Eq}, use the corresponding simulator $\mathcal{S}_{\mathsf{Eq}}$ to obtain h_A. If there exists h_B^j, such that $h_B^j = h_A$, set $\mathsf{GoodEq} = \mathsf{true}$ and pick $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]$. If $h_B^{\hat{\jmath}_B} = h_A$, return true for Π_{Eq} to $\mathcal{A}$ and restart the protocol, and otherwise return false for Π_{Eq} to $\mathcal{A}$. If there is no h_B^j, such that $h_B^j = h_A$, set $\mathsf{GoodEq} = \mathsf{false}$ and return false for Π_{Eq} to $\mathcal{A}$.
3. Compute and send dummy commitment c_B to $\mathcal{A}$. Receive commitment c_A from $\mathcal{A}$.
4. For all $j \in [\lambda]$, prepare garbled circuits and input materials as in Step 4 of the protocol.
5. Compute and send dummy commitment $\{\mathsf{c}_{\gamma_j^B}\}$ to $\mathcal{A}$. Receive $\{\mathsf{c}_{\gamma_j^A}\}$ from $\mathcal{A}$.
6. Generate the puzzle $\mathsf{p}_{s_j^B}$ for a random chosen master puzzle key s_j^B as in the protocol for $j \in [\lambda]$. Generate dummy ciphertexts for his $\mathsf{p}_{\mathsf{c}_j}$ and $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^B}$ for $j \in [\lambda]$. Then send these puzzles to $\mathcal{A}$. Receive puzzles $\mathsf{p}_{s_j^A}$, $\mathsf{p}_{\hat{\mathsf{c}}_j}$, and $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^A}$ for $j \in [\lambda]$ from $\mathcal{A}$. Since time-lock puzzles can be solved in polynomial time, decrypt these puzzles and derive s_j^A, $\hat{\mathsf{c}}_j$, and $\mathsf{seed}_{j,\mathsf{OT}}^A$ for $j \in [\lambda]$.
   For λ executions of Π_{OT}, play the role of the receiver, use as input 0^{n_B} and randomness derived from seed_j^B, and receive $\mathcal{A}$'s $\{\mathsf{p}_{B_{j,i,0}}\}$. Let $\mathcal{H}_j^A$ denote the transcript hash for the jth executions.
   For λ executions of Π_{OT}, play the role of the sender, generate dummy ciphertexts $\{\mathsf{p}_{A_{j,i,b}}\}$, and random κ-bit string seed''_j for all $j \in [\lambda]$. Then execute Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from seed''_j.
7. Generate signatures $\{\sigma_j^B\}$ as an honest $\mathsf{P_B}$ and send them to $\mathcal{A}$.
   Receive $\{\sigma_j^A\}$ from $\mathcal{A}$. If any of the signatures are invalid, execute the *finishing touches* procedure as follows and simulate the abortion of the honest $\mathsf{P_B}$ to terminate the simulation.
   (a) Choose $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]$ if $\mathsf{GoodEq} = \mathsf{false}$, and otherwise use $\hat{\jmath}_B$ chosen in Step 2 in this simulation.
   (b) Program the random oracle to let the decryption of $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ be 0, decryption of $\mathsf{p}_{\mathsf{c}_j}$ be c_j for $j \in [\lambda]\setminus\{\hat{\jmath}_B\}$, and decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol for $j \in [\lambda]\setminus\{\hat{\jmath}_B\}$ and $j = \hat{\jmath}_B$, respectively.
   (c) Program the random oracle: Let the decryption of $\mathsf{p}_{\mathsf{seed}_{\hat{\jmath}_B,\mathsf{OT}}^B}$ be $\mathsf{seed}_{\hat{\jmath}_B,\mathsf{OT}}^B = \mathsf{seed}''_{\hat{\jmath}_B}$. For $j \in [\lambda]\setminus\{\hat{\jmath}_B\}$, let the decryption of $\mathsf{p}_{\mathsf{seed}_{j,\mathsf{OT}}^B}$ be $\mathsf{seed}_{j,\mathsf{OT}}^B = \mathsf{seed}''_j \oplus \mathsf{seed}'_j$.
   Then simulate the computation of an honest $\mathsf{P_A}$ playing the role of the sender in the executions of Π_{OT} in Step 6 of the protocol and obtain $\hat{\mathcal{H}}_j^A$. If $\hat{\mathsf{c}}_j = 0$, the simulation is based on $\mathsf{seed}_{j,\mathsf{OT}}^A$ as the $\hat{\jmath}_A$th instance in the protocol. If $\hat{\mathsf{c}}_j = \mathsf{c}_j$, the simulation is based on $\mathsf{seed}'_j \oplus \mathsf{seed}_{j,\mathsf{OT}}^A$ as other instances in the protocol. Let J_z be the set of indices that $\hat{\mathsf{c}}_j = 0$ and $\hat{\mathcal{H}}_j^A$ matches the simulation of Π_{OT} for the $\hat{\jmath}_A$th instance as described in the protocol. Let J_s be the set of indices that $\hat{\mathsf{c}}_j = \mathsf{c}_j$ and $\hat{\mathcal{H}}_j^A$ matches the simulation of Π_{OT} for the checking instances.
   Hence, there are following cases (let $\mathsf{Stupid} = \mathsf{false}$ and $\mathsf{Type} = \perp$ by default):

- If $|J_s| < \lambda - 2$, executes the *finishing touches* procedure as in Step 7a-7c of the simulation, open the puzzle as in Step 8 of the protocol, generate the certificate cert with respect to j and the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$ as in the protocol. Then send cert to $\mathcal{A}$ as Step 9 of the protocol and output $\mathsf{corrupted}$ to complete the simulation.
- If $|J_s| = \lambda - 2$ and $|J_z| = 2$, set $\mathsf{Stupid} = \mathsf{true}$. With probability $\frac{\lambda-2}{\lambda}$ set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{true})$ and $\mathsf{Type} = \mathsf{BadNum}$. With the remaining probability, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{false})$. Then continue the simulation.
- If $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{true}$, with probability $\frac{\lambda-2}{\lambda-1}$ set $\mathsf{caught} = (\frac{\lambda-2}{\lambda-1}, \mathsf{true})$ and Type according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$. With the remaining probability, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda-1}, \mathsf{false})$. Then continue the simulation.
- If $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, with probability $\frac{\lambda-1}{\lambda}$, set $\mathsf{caught} = (\frac{\lambda-1}{\lambda}, \mathsf{true})$ and Type according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$. With the remaining probability, set $\mathsf{caught} = (\frac{\lambda-1}{\lambda}, \mathsf{false})$. Then continue the simulation.
- If $|J_s| = \lambda - 2$ and $|J_z| = 0$, execute the *finishing touches* procedure as in Step 7a-7c. Then open the puzzle as in Step 8 of the protocol, generate the certificate cert with respect to j and the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$ as in the protocol, and send cert to $\mathcal{A}$ as in Step 9 of the protocol to complete the simulation.
- If $|J_s| = \lambda - 1$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{true}$, set $\mathsf{caught} = \perp$. Then continue the simulation.
- If $|J_s| = \lambda - 1$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, then toss a coin, which with probability $\frac{1}{\lambda}$ outputs false and $\frac{\lambda-1}{\lambda}$ outputs true. If the output is false, then execute the *finishing touches* procedure as follows, open the puzzle, and simulate the abortion of the honest $\mathsf{P_B}$ to terminate the simulation.
  (a) Let $\hat{\jmath}_B$ be the element in J_z.
  (b) Program the random oracle to let the decryption of $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ be 0, decryption of $\mathsf{p}_{\mathsf{c}_j}$ be c_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, and decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ and $j = \hat{\jmath}_B$, respectively.
  (c) Program the random oracle: for the $\hat{\jmath}_B$th instance, let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}}}$ be $\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}} = \mathsf{seed}''_{\hat{\jmath}_B}$. For $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ be $\mathsf{seed}^B_{j,\mathsf{OT}} = \mathsf{seed}''_j \oplus \mathsf{seed}'_j$.
  If the output is true, then set $\mathsf{caught} = \perp$ and continues the simulation.
- If $|J_s| = \lambda - 1$ and $|J_z| = 0$, set $\mathsf{Stupid} = \mathsf{true}$.
  With probability $\frac{\lambda-1}{\lambda}$, execute the *finishing touches* by picking $\hat{\jmath}_B \leftarrow\!\!\$\, J_s$, programming the random oracle to let the decryption of $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ be 0, decryption of $\mathsf{p}_{\mathsf{c}_j}$ be c_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, and decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ and $j = \hat{\jmath}_B$, respectively. Program the random oracle: for the $\hat{\jmath}_B$th instance, let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}}}$ be $\mathsf{seed}^B_{\hat{\jmath}_B,\mathsf{OT}} = \mathsf{seed}''_{\hat{\jmath}_B}$. For $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, let the decryption of $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ be $\mathsf{seed}^B_{j,\mathsf{OT}} = \mathsf{seed}''_j \oplus \mathsf{seed}'_j$. Then open the puzzle as in Step 8 of the protocol, generates the certificate cert with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$ as in the protocol. Then send cert to $\mathcal{A}$ as Step 9 of the protocol to complete the simulation.
  With the remaining probability, execute the *finishing touches* by picking the index $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]\backslash J_s$ and programming the random oracle following the same procedure as above. Then open the puzzle as in Step 8 of the protocol, and simulate the abortion of the honest $\mathsf{P_B}$ to terminate the simulation.
- If $|J_s| = \lambda$, execute the *finishing touches* procedure as in Step 7a-7c in this simulation, open the puzzle, and simulate the abortion of the honest $\mathsf{P_B}$ to terminate the simulation.

Rewind $\mathcal{A}$ and run steps $1' - 7'$ below until $|J'_s| = |J_s|$, $|J'_z| = |J_z|$, $\mathsf{caught}' = \mathsf{caught}$, $\mathsf{GoodEq}' = \mathsf{GoodEq}$, $\mathsf{Stupid}' = \mathsf{Stupid}$, and $\mathsf{Type}' = \mathsf{Type}$.

$1'$. Go through the following steps with $\mathcal{A}$.
  (a) Receive $\{\mathsf{c}_{\mathsf{seed}^A_j}, \mathsf{c}_{\mathsf{seed}'^A_j}, \mathsf{c}_{\mathsf{witness}'^A_j}\}_{j\in[\lambda]}$ from $\mathcal{A}$. Meanwhile, choose $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]$ and uniform κ-bit strings $\{\mathsf{seed}^B_j, \mathsf{seed}'^B_j, \mathsf{witness}'^B_j\}_{j\in[\lambda]}$. Then compute $\mathsf{c}_{\mathsf{seed}^B_j}$, $\mathsf{c}_{\mathsf{seed}'^B_j}$, and $\mathsf{c}_{\mathsf{witness}'^B_j}$ for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ as in the protocol and let $\mathsf{c}_{\mathsf{seed}^B_{\hat{\jmath}_B}}$, $\mathsf{c}_{\mathsf{seed}'^B_{\hat{\jmath}_B}}$, and $\mathsf{c}_{\mathsf{witness}'^B_{\hat{\jmath}_B}}$ be dummy commitments. Finally, send these commitments to $\mathcal{A}$ as in the protocol.
  (b) For all $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, play the role of the receiver and use $b_j = 0$ to retrieve $\{\mathsf{seed}'^A_j\}_{j\in[\lambda]\backslash\{\hat{\jmath}_B\}}$ with randomness derived from $\{\mathsf{seed}^B_j\}$. For the $\hat{\jmath}_B$th instance, run $\mathcal{S}_{\mathsf{OT}}$ for the protocol Π_{OT}, and extract $(\mathsf{seed}'^A_{\hat{\jmath}_B}, \mathsf{witness}'^A_{\hat{\jmath}_B})$. Play the role of the sender and use $\{(\mathsf{seed}'^B_j, \mathsf{witness}'^B_j)\}_{j\in[\lambda]}$ as input with randomness derived from seed^B_j.

$2'$. Compute $\mathsf{seed}'_j = \mathsf{seed}'^A_j \oplus \mathsf{seed}'^B_j$ for all $j \in [\lambda]$. Then let $h^j_B \leftarrow \mathsf{H}(\bigoplus_{[\lambda]\backslash j} \mathsf{seed}'_j)$ for all $j \in [\lambda]$. Use the simulator $\mathcal{S}_{\mathsf{Eq}}$ for Π_{Eq} to obtain h_A. If there exists h^j_B, such that $h^j_B = h_A$, set $\mathsf{GoodEq}' =$

true. Then if $h_B^{\hat{\jmath}_B} = h_A$, return true for Π_{Eq} to $\mathcal{A}$ and restart the protocol, and otherwise return false for Π_{Eq} to $\mathcal{A}$. If there is no $h_B^j = h_A$, set $\mathsf{GoodEq}' = \mathsf{false}$ and return false for Π_{Eq} to $\mathcal{A}$.

3′. Compute and send commitment c_B to $\mathcal{A}$. Receive commitment c_A from $\mathcal{A}$. Then extract $(\hat{\jmath}_A, \{\mathsf{seed}'^B_j\}_{j\neq\hat{\jmath}_A}, \mathsf{witness}'^B_{\hat{\jmath}_A})$ from c_A. If this extracted message is correct, store $\hat{\jmath}_A$. Otherwise, let $\hat{\jmath}_A = \perp$.

4′. For all $j \in [\lambda]$, prepare garbled circuits and input materials as in Step 4 of the protocol.

5′. Compute and send dummy $\{\mathsf{c}_{\gamma_j^B}\}$ to $\mathcal{A}$. Receive $\{\mathsf{c}_{\gamma_j^A}\}$ from $\mathcal{A}$.

6′. Generate puzzles $\mathsf{p}_{s_j^B}$ for a random chosen master puzzle key s_j^B and ciphertexts $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ as in the protocol. Generate ciphertexts $\mathsf{p}_{\mathsf{c}_j}$ for computed c_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ and let the ciphertext $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ encrypt 0. Then send these puzzles to $\mathcal{A}$.
Receive puzzles $\mathsf{p}_{s_j^A}$, $\mathsf{p}_{\hat{\mathsf{c}}_j}$, and $\mathsf{p}_{\mathsf{seed}^A_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ from $\mathcal{A}$. Since time-lock puzzles can be solved in polynomial time, decrypt these puzzles and derive s_j^A, $\hat{\mathsf{c}}_j$, and $\mathsf{seed}^A_{j,\mathsf{OT}}$ for $j \in [\lambda]$.
For all $j \neq \hat{\jmath}_B$, play the role of the receiver and run Π_{OT} with $\mathcal{A}$, using input 0^{n_B} and randomness derived from seed_j^B. In this way, receive $\mathcal{A}$'s $\{\mathsf{p}_{B_{j,i,0}}\}$. In the $\hat{\jmath}_B$th execution of Π_{OT}, play the role of the receiver and use the simulator $\mathcal{S}_{\mathsf{OT}}$ for Π_{OT} to extract $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,b}}\}_{i\in[n_B],b\in\{0,1\}}$. Decrypt $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,b}}\}_{i\in[n_B],b\in\{0,1\}}$ and obtain $\{B_{\hat{\jmath}_B,i,b}\}_{i\in[n_B],b\in\{0,1\}}$.
If $\hat{\jmath}_A = \perp$, generate ciphertexts for $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]$ as in the protocol. Then, play the role of the sender and execute Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from $\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}$ or $\mathsf{seed}^B_{j,\mathsf{OT}}$ as in the protocol. If $\hat{\jmath}_A \neq \perp$, generate ciphertexts for $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]\backslash\{\hat{\jmath}_A\}$ as in the protocol. For $[\lambda]\backslash\{\hat{\jmath}_A\}$, play the role of the sender and execute Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from $\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}$ or $\mathsf{seed}^B_{j,\mathsf{OT}}$ as in the protocol. For $j = \hat{\jmath}_A$, run the simulator $\mathcal{S}_{\mathsf{OT}}$ for Π_{OT} to extract the input x_A and return dummy ciphertexts $\{\mathsf{p}_{A_{\hat{\jmath}_A,i}}\}$ to $\mathcal{A}$.

7′. Generate $\{\sigma_j^B\}$ as an honest $\mathsf{P_B}$ in the protocol and send them to $\mathcal{A}$. Receive $\{\sigma_j^A\}$ from $\mathcal{A}$. If any of the signatures are invalid, then return to Step 1′.
Then simulate the computation of an honest $\mathsf{P_A}$ playing the role of the sender in the executions of Π_{OT} in Step 6 of the protocol and obtain $\hat{\mathcal{H}}_j^A$. If $\hat{\mathsf{c}}_j = 0$, the simulation is based on $\mathsf{seed}^A_{j,\mathsf{OT}}$ as the $\hat{\jmath}_A$th instance in the protocol. If $\hat{\mathsf{c}}_j = \mathsf{c}_j$, the simulation is based on $\mathsf{seed}'_j \oplus \mathsf{seed}^A_{j,\mathsf{OT}}$ as other instances in the protocol. Let J'_z be the set of indices that $\hat{\mathsf{c}}_j = 0$ and $\hat{\mathcal{H}}_j^A$ matches the simulation of Π_{OT} for the $\hat{\jmath}_A$th instance as described in the protocol. Let J'_s be the set of indices that $\hat{\mathsf{c}}_j = \mathsf{c}_j$ and $\hat{\mathcal{H}}_j^A$ matches the simulation of Π_{OT} for the checking instances.
Hence, there are following cases (let $\mathsf{Stupid}' = \mathsf{false}$ and $\mathsf{Type}' = \perp$ by default):
- If $|J'_s| < \lambda - 2$, then return to Step 1′.
- If $|J'_s| = \lambda - 2$ and $|J'_z| = 2$, set $\mathsf{Stupid}' = \mathsf{true}$. If $\hat{\jmath}_B \in J'_s$, set $\mathsf{caught}' = (\frac{\lambda-2}{\lambda}, \mathsf{true})$ and $\mathsf{Type}' = \mathsf{BadNum}$. Otherwise, set $\mathsf{caught}' = (\frac{\lambda-2}{\lambda}, \mathsf{false})$.
- When $|J'_s| = \lambda - 2$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{true}$, if $\hat{\jmath}_B \in J'_s$, set $\mathsf{caught}' = (\frac{\lambda-2}{\lambda-1}, \mathsf{true})$ and Type' according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$. Otherwise, set $\mathsf{caught}' = (\frac{\lambda-2}{\lambda-1}, \mathsf{false})$.
- When $|J'_s| = \lambda - 2$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{false}$, if $\hat{\jmath}_B \in J'_s$ or $\hat{\jmath}_B \in J'_z$, set $\mathsf{caught}' = (\frac{\lambda-1}{\lambda}, \mathsf{true})$ and Type' according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$. Otherwise, set $\mathsf{caught}' = (\frac{\lambda-1}{\lambda}, \mathsf{false})$.
- If $|J'_s| = \lambda - 2$ and $|J'_z| = 0$, then return to Step 1′.
- If $|J'_s| = \lambda - 1$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{true}$, set $\mathsf{caught}' = \perp$.
- When $|J'_s| = \lambda - 1$, $|J'_z| = 1$, and $\mathsf{GoodEq}' = \mathsf{false}$, if $\hat{\jmath} \in J_s$, then set $\mathsf{caught}' = \perp$. Otherwise, return to Step 1′.
- If $|J'_s| = \lambda - 1$ and $|J'_z| = 0$, then return to Step 1′.
- If $|J'_s| = \lambda$, then return to Step 1′.

8. If $\hat{\jmath}_A \neq \perp$, compute $z \leftarrow \mathcal{C}(x_A, x_B)$. Then regenerate the garbled circuit via

$$(\{A_{\hat{\jmath}_A,i}\}, \{B_{\hat{\jmath}_A,i}\}, \mathsf{GC}_{\hat{\jmath}_A}, \{Z_{\hat{\jmath}_A,i,b}\}) \leftarrow \mathcal{S}_{\mathsf{Gb}}(1^\kappa, \mathcal{C}, z)$$

and recompute the materials of the $\hat{\jmath}_A$th instance in Step 4b and 4c of the protocol. Since this garbled circuit is simulated, use/program dummy commitments/values in case of need, *e.g.*, for $\gamma^B_{\hat{\jmath}_A}$. Use the random oracle to program the opening of $\mathsf{c}_{\hat{\jmath}_A}$ and $\{\mathsf{p}_{A_{\hat{\jmath}_A,i}}\}$ with respect to the simulated garbled circuit. Then open the puzzle to $\mathcal{A}$ as in the protocol. $\mathcal{A}$ may also open her puzzle.

9. If $\mathsf{caught}' = (\cdot, \mathsf{true})$, generate the corresponding cert and send it to $\mathcal{A}$ to complete the simulation.

10. Send the opening for c_B to $\mathcal{A}$ as in the protocol. Receives the opening for c_A from $\mathcal{A}$. If $\mathsf{Stupid}' = \mathsf{true}$, simulate the abortion of the honest party to complete the simulation. If the opening $(\hat{\jmath}_A, \{\mathsf{seed}'^B_j\}_{j\neq\hat{\jmath}_A}, \mathsf{witness}'^B_{\hat{\jmath}_A})$ is not correct, simulate the abortion of the honest party to complete the simulation.
11. Send the simulated garbled circuit $\mathsf{GC}_{\hat{\jmath}_A}$, together with corresponding label hash values, to $\mathcal{A}$. Receive the garbled circuit $\mathsf{GC}_{\hat{\jmath}_B}$ and related materials from $\mathcal{A}$. If they are invalid, simulate the abortion of the honest party to complete the simulation.
12. Send the simulated labels, $\gamma^B_{\hat{\jmath}_A}$, and corresponding decommitments to $\mathcal{A}$. Receives the labels, $\gamma^A_{\hat{\jmath}_B}$, and corresponding decommitments. If the received materials are invalid, simulate the abortion of the honest party to complete the simulation.
13. Use the simulator $\mathcal{S}_{\mathsf{Eq}}$ for Π_{Eq} to obtain β_A from $\mathcal{A}$. Then define the boolean function g as follows.
    (a) On input $x_B \in \{0,1\}^{n_B}$, select the corresponding input labels for x_B in the label tuples $\{(B_{\hat{\jmath}_B,i,0}, B_{\hat{\jmath}_B,i,1})\}_{i\in[n_B]}$.
    (b) Evaluate $\mathsf{GC}_{\hat{\jmath}_B}$ with the input-wire labels $\{A_{\hat{\jmath}_B,i,x_A[i]}\}_{i\in[n_A]}$ and $\{B_{\hat{\jmath}_B,i,x_B[i]}\}_{i\in[n_B]}$ as in the protocol to obtain $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$ and $z_{\hat{\jmath}_B}$. In particular, if some error occurs in the evaluation of garbled circuit, some hard-coded random values are used for $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$ and $z_{\hat{\jmath}_B}$.
    (c) Compute $\beta_B \leftarrow \mathsf{H}(\bigoplus_{i=1}^{n_O}(Z_{\hat{\jmath}_A,i,z_{\hat{\jmath}_B}[i]} \oplus Z_{\hat{\jmath}_B,i,z_{\hat{\jmath}_B}[i]}))$.
    (d) If $\beta_A = \beta_B$, return true. Otherwise, return false.
    Compute $e \leftarrow g(x_B)$ and give e to $\mathcal{A}$. If $\mathcal{A}$ aborts, then abort. If $e = \mathsf{false}$, then abort with output $\mathsf{cheating}$. Otherwise, output $z_{\hat{\jmath}_B}$.

$\mathbf{Expt}_1$ Different from $\mathbf{Expt}_0$, we choose $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]$ if $\mathsf{GoodEq} = \mathsf{false}$ in Step 2, and then we use $\hat{\jmath}_B$ in the subsequent steps at the outset of the experiment. We modify Step 7a of the simulation by using $\hat{\jmath}_B$ we have chosen. We then modify the following cases.
- For $|J_s| = \lambda - 2$ and $|J_z| = 2$, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{true})$ if $\hat{\jmath}_B \in J_s$. Otherwise, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{false})$.
- For $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{true}$, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda-1}, \mathsf{true})$ if $\hat{\jmath}_B \in J_s$. Otherwise, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{false})$.
- For $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, set $\mathsf{caught} = (\frac{\lambda-1}{\lambda}, \mathsf{true})$ if $\hat{\jmath}_B \in J_s$ or $\hat{\jmath}_B \in J_z$. Otherwise, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{false})$.
- For $|J_s| = \lambda - 1$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, set $\mathsf{caught} = \perp$ if $\hat{\jmath}_B \in J_s$.

The rest parts remain the same. It is easy to see that the outputs of $\mathbf{Expt}_1$ and $\mathbf{Expt}_0$ are identically distributed.

$\mathbf{Expt}_2$ In this experiment, let $c_{\mathsf{seed}^B_{\hat{\jmath}_B}}$, $c_{\mathsf{seed}'^B_{\hat{\jmath}_B}}$, and $c_{\mathsf{witness}'^B_{\hat{\jmath}_B}}$ be dummy commitments in Step 1 of the simulation. Use true randomness in the $\hat{\jmath}_B$th execution of Π_{OT} in Step 1 and 6 of the simulation, where $\mathcal{S}$ plays the role of the receiver. Since $\mathcal{S}$ will not generate cert with respect to the $\hat{\jmath}_B$th instance and the commitment scheme is hiding, it is straightforward to see that the output of $\mathbf{Expt}_2$ is computationally indistinguishable from the output of $\mathbf{Expt}_1$.

$\mathbf{Expt}_3$ In this experiment, use $\mathcal{S}_{\mathsf{OT}}$ for the $\hat{\jmath}_B$th instance of Π_{OT} in both Step 1 and 6 of the simulation, where $\mathcal{S}$ plays the role of the receiver, and extract all $\mathcal{A}$'s inputs. From the security of Π_{OT}, we know that the output of $\mathbf{Expt}_3$ is computationally indistinguishable from the output of $\mathbf{Expt}_2$.

$\mathbf{Expt}_4$ In this experiment, we modify Step 3 of the simulation, such that the commitment c_B is computed as in the protocol and $\mathcal{S}$ extracts $\hat{\jmath}_A$ as in Step 3′. We also modify Step 6 of the simulation, such that the ciphertexts $\{\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}\}_{j\in[\lambda]}$ and $\{\mathsf{p}_{\mathsf{c}_j}\}_{j\in[\lambda]}$ are generated as in the protocol (note that $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ encrypts 0). Now no seed''_j's are needed. Then for the *finishing touches* procedure in Step 7, we do not need to program the decryption of $\mathsf{p}_{\mathsf{c}_j}$ and $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$. Since once an abortion is incurred, these dummy ciphertexts in $\mathbf{Expt}_3$ will be programmed to be the same as those in $\mathbf{Expt}_4$, we can easily see that the output of $\mathbf{Expt}_4$ is computationally indistinguishable from the output of $\mathbf{Expt}_3$.

$\mathbf{Expt}_5$ In this experiment, we modify Step 6 of the simulation as follows. If $\hat{\jmath}_A = \perp$, then $\mathcal{S}$ generates $\{\mathsf{p}_{A_{j,i,b}}\}_{j\in[\lambda]}$ as in the protocol and uses them as input in Π_{OT}. If $\hat{\jmath}_A \neq \perp$, then $\mathcal{S}$ generates $\{\mathsf{p}_{A_{j,i,b}}\}_{j\in[\lambda]\setminus\{\hat{\jmath}_A\}}$ as in the protocol and dummy $\{\mathsf{p}_{A_{j,i,b}}\}_{j=\hat{\jmath}_A}$, and then uses them as input in Π_{OT}. Then for the *finishing touches* procedure in Step 7, we do not need to program the decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]\setminus\{\hat{\jmath}_A\}$. Similarly, we can easily see that the output of $\mathbf{Expt}_5$ is computationally indistinguishable from the output of $\mathbf{Expt}_4$.

$\mathbf{Expt}_{5'}$ Because Step 1′ - 6′ in $\mathbf{Expt}_5$ are identical to Step 1 - 6, we can combine the rewinding and obtain this $\mathbf{Expt}_{5'}$ whose output is statistically indistinguishable from the output of $\mathbf{Expt}_5$

with the only difference is the probability of aborted rewinding. The description of $\mathbf{Expt}_{5'}$ is as follows.

0. Use KGen to generate a pair of key $(\mathsf{vk}_B, \mathsf{sigk}_B)$ and send vk_B to $\mathcal{A}$.
1. Go through the following steps with $\mathcal{A}$.
   (a) Receive $\{\mathsf{c}_{\mathsf{seed}_j^A}, \mathsf{c}_{\mathsf{seed}'^A_j}, \mathsf{c}_{\mathsf{witness}'^A_j}\}_{j\in[\lambda]}$ from $\mathcal{A}$. Pick $\hat{\jmath}_B \leftarrow\!\!\$\ [\lambda]$ and uniform κ-bit strings $\{\mathsf{seed}_j^B, \mathsf{seed}'^B_j, \mathsf{witness}'^B_j\}_{j\in[\lambda]}$. Compute $\mathsf{c}_{\mathsf{seed}_j^B}$, $\mathsf{c}_{\mathsf{seed}'^B_j}$, and $\mathsf{c}_{\mathsf{witness}'^B_j}$ for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ as in the protocol. Let $\mathsf{c}_{\mathsf{seed}^B_{\hat{\jmath}_B}}$, $\mathsf{c}_{\mathsf{seed}'^B_{\hat{\jmath}_B}}$, and $\mathsf{c}_{\mathsf{witness}'^B_{\hat{\jmath}_B}}$ be dummy commitments. Finally, send these commitments to $\mathcal{A}$ as in the protocol.
   (b) For all $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, as the receiver, use $b_j = 0$ to retrieve $\{\mathsf{seed}'^A_j\}_{i\in[\lambda]\backslash\{\hat{\jmath}_B\}}$ with randomness derived from $\{\mathsf{seed}_j^B\}$ in Π_{OT}. For the $\hat{\jmath}_B$th instance, run the simulator $\mathcal{S}_{\mathsf{OT}}$ for Π_{OT} and extract $(\mathsf{seed}'^A_{\hat{\jmath}_B}, \mathsf{witness}'^A_{\hat{\jmath}_B})$. As the sender, use as input $\{(\mathsf{seed}'^B_j, \mathsf{witness}'^B_j)\}_{j\in[\lambda]}$ with randomness derived from seed_j^B in Π_{OT}.
2. Compute $\mathsf{seed}'_j = \mathsf{seed}'^A_j \oplus \mathsf{seed}'^B_j$ for all $j \in [\lambda]$. Then let $h_B^j \leftarrow \mathsf{H}(\bigoplus_{[\lambda]\backslash j} \mathsf{seed}'_j)$ for all $j \in [\lambda]$. Use the simulator $\mathcal{S}_{\mathsf{Eq}}$ for Π_{Eq} to obtain h_A. If there exists h_B^j, such that $h_B^j = h_A$, set $\mathsf{GoodEq} = \mathsf{true}$. Then if $h_B^{\hat{\jmath}_B} = h_A$, return true for Π_{Eq} to $\mathcal{A}$ and restart the protocol, and otherwise return false to $\mathcal{A}$. If there is no $h_B^j = h_A$, set $\mathsf{GoodEq} = \mathsf{false}$ and returns false for Π_{Eq} to $\mathcal{A}$.
3. Compute and send commitment c_B to $\mathcal{A}$. Receive commitment c_A from $\mathcal{A}$. Then extract $(\hat{\jmath}_A, \{\mathsf{seed}'^B_j\}_{j\neq\hat{\jmath}_A}, \mathsf{witness}'^B_{\hat{\jmath}_A})$ from c_A. If this extracted message is correct, store $\hat{\jmath}_A$. Otherwise, let $\hat{\jmath}_A = \perp$.
4. For all $j \in [\lambda]$, prepare garbled circuits and input materials as in Step 4 of the protocol.
5. Compute and send dummy $\{\mathsf{c}_{\gamma_j^B}\}$ to $\mathcal{A}$. Receive $\{\mathsf{c}_{\gamma_j^A}\}$ from $\mathcal{A}$.
6. Generate the puzzle $\mathsf{p}_{s_j^B}$ for a random chosen master puzzle key s_j^B and ciphertexts $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ as in the protocol. Generate ciphertexts $\mathsf{p}_{\mathsf{c}_j}$ for computed c_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ and let the ciphertext $\mathsf{p}_{\mathsf{c}_{\hat{\jmath}_B}}$ encrypt 0. Then send these ciphertexts to $\mathcal{A}$.
   Receive puzzles $\mathsf{p}_{s_j^A}$, $\mathsf{p}_{\hat{\mathsf{c}}_j}$, and $\mathsf{p}_{\mathsf{seed}^A_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ from $\mathcal{A}$. Since time-lock puzzles can be solved in polynomial time, decrypt these puzzles and derive s_j^A, $\hat{\mathsf{c}}_j$, and $\mathsf{seed}^A_{j,\mathsf{OT}}$ for $j \in [\lambda]$.
   For all $j \neq \hat{\jmath}_B$, play the role of the receiver and run Π_{OT} with $\mathcal{A}$, using input 0^{n_B} and randomness derived from seed_j^B. In this way, receive $\mathcal{A}$'s $\{\mathsf{p}_{B_{j,i,0}}\}$. In the $\hat{\jmath}_B$th execution of Π_{OT}, play the role of the receiver and use the simulator $\mathcal{S}_{\mathsf{OT}}$ for Π_{OT} to extract $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,b}}\}_{i\in[n_B],b\in\{0,1\}}$. Decrypt $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,b}}\}_{i\in[n_B],b\in\{0,1\}}$ and obtain $\{B_{\hat{\jmath}_B,i,b}\}_{i\in[n_B],b\in\{0,1\}}$.
   If $\hat{\jmath}_A = \perp$, generate ciphertexts for $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]$ as in the protocol. Then play the role of the sender and execute Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from $\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}$ or $\mathsf{seed}^B_{j,\mathsf{OT}}$ as in the protocol. If $\hat{\jmath}_A \neq \perp$, generate ciphertexts for $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]\backslash\{\hat{\jmath}_A\}$ as in the protocol. Then for $j \in [\lambda]\backslash\{\hat{\jmath}_A\}$, play the role of the sender and execute Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from $\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}$ or $\mathsf{seed}^B_{j,\mathsf{OT}}$ as in the protocol. For $j = \hat{\jmath}_A$, run the simulator $\mathcal{S}_{\mathsf{OT}}$ for Π_{OT} to extract the input x_A and return dummy ciphertexts $\{\mathsf{p}_{A_{\hat{\jmath}_A,i}}\}$ to $\mathcal{A}$.
7. Generate $\{\sigma_j^B\}$ and send them to $\mathcal{A}$. Receive $\{\sigma_j^A\}$ from $\mathcal{A}$. If any of the signatures are invalid, execute the *finishing touches* procedure by programming random oracle to let decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol (if $\hat{\jmath}_A \neq \perp$), then simulate the abortion of the honest $\mathsf{P_B}$ to terminate the simulation.
   Simulate the computation of an honest $\mathsf{P_A}$ playing the role of the sender in the λ executions of Π_{OT} in Step 6 of the protocol based on $\mathsf{seed}^A_{j,\mathsf{OT}}$ and possibly seed'_j to obtain $\hat{\mathcal{H}}_j^A$. Let J_z be the set of indices that $\hat{\mathsf{c}}_j = 0$ and $\hat{\mathcal{H}}_j^A$ matches the simulation of Π_{OT} for the $\hat{\jmath}_A$th instance as described in the protocol. Let J_s be the set of indices that $\hat{\mathsf{c}}_j = \mathsf{c}_j$ and $\hat{\mathcal{H}}_j^A$ matches the simulation of Π_{OT} for the checking instances.
   Hence, there are following cases (let $\mathsf{Stupid} = \mathsf{false}$ and $\mathsf{Type} = \perp$ by default):
   - If $|J_s| < \lambda - 2$, execute the *finishing touches* procedure by programming random oracle to let decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol (if $\hat{\jmath}_A \neq \perp$), opening the puzzles as in Step 8, generating cert with respect to j and the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$ as in the protocol, and sending cert to $\mathcal{A}$ as Step 9 to complete the simulation.
   - If $|J_s| = \lambda - 2$ and $|J_z| = 2$, set $\mathsf{Stupid} = \mathsf{true}$. If $\hat{\jmath}_B \in J_s$, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{true})$ and $\mathsf{Type} = \mathsf{BadNum}$. Otherwise, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda}, \mathsf{false})$.
   - When $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{true}$, if $\hat{\jmath}_B \in J_s$, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda-1}, \mathsf{true})$ and Type according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$. Otherwise, set $\mathsf{caught} = (\frac{\lambda-2}{\lambda-1}, \mathsf{false})$.

- When $|J_s| = \lambda - 2$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, if $\hat{\jmath}_B \in J_s$ or $\hat{\jmath}_B \in J_z$, set $\mathsf{caught} = (\frac{\lambda-1}{\lambda}, \mathsf{true})$ and Type according to the possible certificate with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$. Otherwise, set $\mathsf{caught} = (\frac{\lambda-1}{\lambda}, \mathsf{false})$.
- If $|J_s| = \lambda - 2$ and $|J_z| = 0$, execute the *finishing touches* procedure by programming random oracle to let decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol (if $\hat{\jmath}_A \neq \perp$), opening the puzzle as in Step 8, generating cert with respect to j and the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$ as in the protocol, and sending cert to $\mathcal{A}$ as in Step 9 to complete the simulation.
- If $|J_s| = \lambda - 1$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{true}$, set $\mathsf{caught} = \perp$.
- When $|J_s| = \lambda - 1$, $|J_z| = 1$, and $\mathsf{GoodEq} = \mathsf{false}$, if $\hat{\jmath} \in J_s$, then set $\mathsf{caught} = \perp$. Otherwise, execute the *finishing touches* procedure by programming random oracle to let decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol (if $\hat{\jmath}_A \neq \perp$), opening the puzzle as in Step 8, and simulating the abortion to complete the simulation.
- If $|J_s| = \lambda - 1$ and $|J_z| = 0$, set $\mathsf{Stupid} = \mathsf{true}$. If $\hat{\jmath} \in J_s$, execute the *finishing touches* procedure by programming random oracle to let decryption of $\{\mathsf{p}_{A_{j,i,b}}\}$ be the correct values as those in the protocol (if $\hat{\jmath}_A \neq \perp$), opening the puzzle as in Step 8, generating cert with respect to the inconsistent $\hat{\mathsf{c}}_j$ or $\hat{\mathcal{H}}_j^A$ as in the protocol, and sending cert to $\mathcal{A}$ as in Step 9 to complete the simulation.
  If $\hat{\jmath} \notin J_s$, open the puzzle as in Step 8 of the protocol and simulate the abortion of the honest $\mathsf{P_B}$ to terminate the simulation.
- If $|J_s| = \lambda$, execute the *finishing touches* procedure by opening the puzzle as in Step 8 and simulating the abortion to complete the simulation.

8. If $\hat{\jmath}_A \neq \perp$, compute $z \leftarrow \mathcal{C}(x_A, x_B)$. Then regenerate the (simulated) garbled circuit via
$$(\{A_{\hat{\jmath}_A,i}\}, \{B_{\hat{\jmath}_A,i}\}, \mathsf{GC}_{\hat{\jmath}_A}, \{Z_{\hat{\jmath}_A,i,b}\}) \leftarrow \mathcal{S}_{\mathsf{Gb}}(1^\kappa, \mathcal{C}, z)$$
and recompute the materials of the $\hat{\jmath}_A$th instance in Step 4b and 4c of the protocol. Since this garbled circuit is simulated, use/program dummy commitments/values in case of need, *e.g.*, for $\gamma^B_{\hat{\jmath}_A}$. Use the random oracle to program the opening of $\mathsf{c}_{\hat{\jmath}_A}$ and $\{\mathsf{p}_{A_{\hat{\jmath}_A,i}}\}$ with respect to the simulated garbled circuit. Then open the puzzle to $\mathcal{A}$ as in the protocol. $\mathcal{A}$ may also open her puzzle.
9. If $\mathsf{caught} = (\cdot, \mathsf{true})$, generate the corresponding cert and send it to $\mathcal{A}$ to complete the simulation.
10. Send the opening for c_B to $\mathcal{A}$ as in the protocol. Receive the opening for c_A from $\mathcal{A}$. If $\mathsf{Stupid} = \mathsf{true}$, simulate the abortion of the honest party to complete the simulation. If the opening $(\hat{\jmath}_A, \{\mathsf{seed}'^B_j\}_{j \neq \hat{\jmath}_A}, \mathsf{witness}'^B_{\hat{\jmath}_A})$ is not correct, simulate the abortion of the honest party to complete the simulation.
11. Send the simulated garbled circuit $\mathsf{GC}_{\hat{\jmath}_A}$, together with corresponding label hash values, to $\mathcal{A}$. Receive the garbled circuit $\mathsf{GC}_{\hat{\jmath}_B}$ and related materials from $\mathcal{A}$. If they are invalid, simulate the abortion of the honest party to complete the simulation.
12. Send the simulated labels, $\gamma^B_{\hat{\jmath}_A}$, and corresponding decommitments to $\mathcal{A}$. Receive the labels, $\gamma^A_{\hat{\jmath}_B}$, and corresponding decommitments. If the received materials are invalid, simulate the abortion of the honest party to complete the simulation.
13. Use the simulator $\mathcal{S}_{\mathsf{Eq}}$ for Π_{Eq} to obtain β_A from $\mathcal{A}$. Then define the boolean function g as follows.
    (a) On input $x_B \in \{0,1\}^{n_B}$, select the input-wire labels for x_B in $\{(B_{\hat{\jmath}_B,i,0}, B_{\hat{\jmath}_B,i,1})\}_{i \in [n_B]}$.
    (b) Evaluate $\mathsf{GC}_{\hat{\jmath}_B}$ with input-wire labels $\{A_{\hat{\jmath}_B,i,x_A[i]}\}_{i \in [n_A]}$ and $\{B_{\hat{\jmath}_B,i,x_B[i]}\}_{i \in [n_B]}$ as in the protocol to obtain $\{Z_{\hat{\jmath}_B,i}\}_{i \in [n_O]}$ and $z_{\hat{\jmath}_B}$. In particular, if some error occurs in the evaluation of garbled circuit, some specified hard-coded random values are used for $\{Z_{\hat{\jmath}_B,i}\}_{i \in [n_O]}$ and $z_{\hat{\jmath}_B}$.
    (c) Compute $\beta_B \leftarrow \mathsf{H}(\bigoplus_{i=1}^{n_O}(Z_{\hat{\jmath}_A,i,z_{\hat{\jmath}_B}[i]} \oplus Z_{\hat{\jmath}_B,i,z_{\hat{\jmath}_B}[i]}))$.
    (d) If $\beta_A = \beta_B$, return true. Otherwise, return false.
    Compute $e \leftarrow g(x_B)$ and give e to $\mathcal{A}$. If $\mathcal{A}$ aborts, then abort. If $e = \mathsf{false}$, then abort with output $\mathsf{cheating}$. Otherwise, output $z_{\hat{\jmath}_B}$.

$\mathbf{Expt}_6$ We modify the previous experiment when $\hat{\jmath}_A \neq \perp$. In Step 8, if $\hat{\jmath}_A \neq \perp$, regenerate the garbled circuit via
$$(\{A_{\hat{\jmath}_A,i,b}\}, \{B_{\hat{\jmath}_A,i,b}\}, \mathsf{GC}_{\hat{\jmath}_A}, \{Z_{\hat{\jmath}_A,i,b}\}) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C})$$
and program the random oracle to let the labels $\{A_{\hat{\jmath}_A,i,x_A[i]}\}$ be the decrypted value of $\{\mathsf{p}_{A_{\hat{\jmath}_A,i}}\}$. We also program the random oracle to let labels $\{A_{\hat{\jmath}_A,i,b}\}$ and $\{B_{\hat{\jmath}_A,i,b}\}$ be the labels used for the generation of $\mathsf{c}_{\hat{\jmath}_A}$ and $\gamma^B_{\hat{\jmath}_A}$. Then in Step 12 of the simulation, send $\{B_{\hat{\jmath}_A,i,x_B[i]}\}$ instead of simulated $\{B_{\hat{\jmath}_A,i}\}$ to $\mathcal{A}$. The materials programmed with respect to random oracle are adjusted according to the garbled circuit. Based on the security of the garbling scheme and hash function,

and the hiding property of the commitment, we know that the output of $\mathbf{Expt}_6$ is computationally indistinguishable from the output of $\mathbf{Expt}_5$.

$\mathbf{Expt}_7$ In this experiment, if $\hat{\jmath}_A \neq \perp$, we move the garbled circuit regeneration to Step 4. Hence, materials generated in Step 4 and 6 do not need to be dummy and be programmed later. The commitment $\mathsf{c}_{\gamma^B_{\hat{\jmath}_A}}$ is generated based on the labels and x_B. Now $\mathcal{S}$ does not need to extract the committed value from c_A. It is straightforward to see that the output of $\mathbf{Expt}_7$ is computationally indistinguishable from the output of $\mathbf{Expt}_6$.

$\mathbf{Expt}_8$ In this experiment, generate $\mathsf{c}_{\mathsf{seed}^B_{\hat{\jmath}_B}}$, $\mathsf{c}_{\mathsf{seed}'^B_{\hat{\jmath}_B}}$, and $\mathsf{c}_{\mathsf{witness}'^B_{\hat{\jmath}_B}}$ as in the protocol. Since these commitments will not be opened, according to the security of the commitment, the output of $\mathbf{Expt}_8$ is computationally indistinguishable from the output of $\mathbf{Expt}_7$.

$\mathbf{Expt}_9$ In this experiment, solving the time-lock puzzle (in Step 6 and 7) and simulating the computation of $\mathsf{P_A}$ playing the role of the sender in Π_{OT} (in Step 7) are moved to Step 8 and 9. It is easy to see that the output of $\mathbf{Expt}_9$ is identical to the output of $\mathbf{Expt}_8$.

$\mathbf{Expt}_{10}$ In this experiment, we modify Step 10 of the simulation, such that we now do not need to simulate the abortion of the protocol if $\mathsf{Stupid} = \mathsf{true}$. Alternatively, the simulator simulate the abortion if the opening of c_A from $\mathcal{A}$ is incorrect. In the case that $\mathsf{Stupid} = \mathsf{true}$ and the protocol has proceeded to this step, we have $|J_s| = \lambda - 2$ and $|J_z| = 2$. Hence, $\mathcal{A}$ cannot provide the correct opening of c_A except for a negligible probability based on the security of Π_{OT} and commitment scheme. Therefore, the output of $\mathbf{Expt}_{10}$ is computationally indistinguishable from the output of $\mathbf{Expt}_9$.

$\mathbf{Expt}_{11}$ In this experiment, we modify Step 2 of the simulation, such that we compute seed'_j only for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$. Then we modify the simulation for the computation of $\mathsf{P_A}$ playing the role of the sender for sending wire labels in Π_{OT}. The simulations are based on $\mathsf{seed}^A_{j,\mathsf{OT}}$ and possibly seed'_j for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$. In this way, we obtain $\hat{\mathcal{H}}^A_j$ for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$.

Let $\hat{J}_z$ be the set of indices that $\hat{\mathsf{c}}_j = 0$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the $\hat{\jmath}_A$th instance as described in the protocol. Let $\hat{J}_s$ be the set of indices that $\hat{\mathsf{c}}_j = \mathsf{c}_j$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the checking instances.

- If $|\hat{J}_s| < \lambda - 2$, generate cert as in the protocol and send cert to $\mathcal{A}$ to complete the simulation.
- If $|\hat{J}_s| = \lambda - 2$ and $|\hat{J}_z| = 1$, continue the execution of the protocol.
- If $|\hat{J}_s| = \lambda - 2$, $|\hat{J}_z| = 0$, generate cert as in the protocol and send cert to $\mathcal{A}$ to complete the simulation.
- If $|\hat{J}_s| = \lambda - 1$, abort and terminate the simulation.

It is easy to see that the output of $\mathbf{Expt}_{11}$ is identical to the output of $\mathbf{Expt}_{10}$.

$\mathbf{Expt}_{12}$ In this experiment, we modify Step 13 of the simulation as follows. Uses the simulator $\mathcal{S}_{\mathsf{Eq}}$ for Π_{Eq} to obtain β_A from $\mathcal{A}$. Then on input $x_B \in \{0,1\}^{n_B}$, evaluate the garbled circuit $\mathsf{GC}_{\hat{\jmath}_B}$ with input-wire labels $\{A_{\hat{\jmath}_B,i,x_A[i]}\}_{i\in[n_A]}$ and $\{B_{\hat{\jmath}_B,i,x_B[i]}\}_{i\in[n_B]}$ as in the protocol to obtain $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$ and $z_{\hat{\jmath}_B}$. If any decoded bits is $\perp$, let $z_{\hat{\jmath}_B} = \perp$.

If $z_{\hat{\jmath}_B} = \perp$, let $\beta_B \leftarrow\!\!\$\, \{0,1\}^\kappa$. Otherwise, we let $\beta_B \leftarrow \mathsf{H}(\bigoplus_{i=1}^{n_O}(Z_{\hat{\jmath}_A,i,z_{\hat{\jmath}_B}[i]} \oplus Z_{\hat{\jmath}_B,i,z_{\hat{\jmath}_B}[i]}))$. Check whether $\beta_A = \beta_B$ and send the result to $\mathcal{A}$. If $\mathcal{A}$ aborts, then abort. If $\beta_A \neq \beta_B$, then abort with output $\mathsf{cheating}$. Otherwise, output $z_{\hat{\jmath}_B}$.

It is easy to see that the output of $\mathbf{Expt}_{12}$ is computationally indistinguishable from the output of $\mathbf{Expt}_{11}$.

$\mathbf{Expt}_{13}$ In this experiment, run the $\hat{\jmath}_B$th execution of Π_{OT} in Step 1b and 6 honestly using randomness derived from $\mathsf{seed}^B_{\hat{\jmath}_B}$. Meanwhile, execute Π_{Eq} as an honest $\mathsf{P_B}$. According to the security of Π_{OT} and Π_{Eq}, the output of $\mathbf{Expt}_{13}$ is computationally indistinguishable from the output of $\mathbf{Expt}_{12}$.

$\mathbf{Expt}_{14}$ In this experiment, move the evaluation of the garbled circuit from Step 13 to Step 12 of the simulation. Since in Step 12, the simulator has already received the garbled circuit and wire label, this modification does not change the simulation. Hence, the output of $\mathbf{Expt}_{14}$ is identical to the output of $\mathbf{Expt}_{13}$. We present $\mathbf{Expt}_{14}$ as follows.

0. Use KGen to generate a pair of key $(\mathsf{vk}_B, \mathsf{sigk}_B)$ and send vk_B to $\mathcal{A}$.
1. Go through the following steps with $\mathcal{A}$.
   (a) Receive $\{\mathsf{c}_{\mathsf{seed}^A_j}, \mathsf{c}_{\mathsf{seed}'^A_j}, \mathsf{c}_{\mathsf{witness}'^A_j}\}_{j\in[\lambda]}$ from $\mathcal{A}$. Meanwhile, pick $\hat{\jmath}_B \leftarrow\!\!\$\, [\lambda]$ and uniform κ-bit strings $\{\mathsf{seed}^B_j, \mathsf{seed}'^B_j, \mathsf{witness}'^B_j\}_{j\in[\lambda]}$. Compute $\mathsf{c}_{\mathsf{seed}^B_j}$, $\mathsf{c}_{\mathsf{seed}'^B_j}$, and $\mathsf{c}_{\mathsf{witness}'^B_j}$ for $j \in [\lambda]$ as in the protocol. Finally, send these commitments to $\mathcal{A}$ as in the protocol.

(b) For all $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$, as the receiver, use $b_j = 0$ to retrieve $\{\mathsf{seed}'^A_j\}_{i\in[\lambda]\backslash\{\hat{\jmath}_B\}}$ with randomness derived from $\{\mathsf{seed}^B_j\}$ in Π_{OT}. For the $\hat{\jmath}_B$th instance, use $b_{\hat{\jmath}_B} = 1$ with randomness derived from $\mathsf{seed}^B_{\hat{\jmath}_B}$ to retrieve $\mathsf{witness}'^A_{\hat{\jmath}_B}$. As the sender, use as input $\{(\mathsf{seed}'^B_j, \mathsf{witness}'^B_j)\}_{j\in[\lambda]}$ with randomness derived from seed^B_j in Π_{OT}.

2. Compute $\mathsf{seed}'_j = \mathsf{seed}'^A_j \oplus \mathsf{seed}'^B_j$ for all $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$. Let $h_B \leftarrow \mathsf{H}(\bigoplus_{[\lambda]\backslash\hat{\jmath}_B} \mathsf{seed}'_j)$. Use Π_{Eq} to check whether $h_B = h_A$. If it holds, restart the protocol. Otherwise, continue the execution.
3. Compute and send commitment c_B to $\mathcal{A}$. Receive commitment c_A from $\mathcal{A}$.
4. For all $j \in [\lambda]$, prepare garbled circuits and input materials as in Step 4 of the protocol.
5. Compute and send $\{\mathsf{c}_{\gamma^B_j}\}$ to $\mathcal{A}$. Receive $\{\mathsf{c}_{\gamma^A_j}\}$ from $\mathcal{A}$.
6. Generate the puzzle $\mathsf{p}_{s^B_j}$ for a random chosen master puzzle key s^B_j and ciphertexts $\mathsf{p}_{\mathsf{seed}^B_{j,\mathsf{OT}}}$ and $\mathsf{p}_{\mathsf{c}_j}$ for $j \in [\lambda]$ as in the protocol. Then send them to $\mathcal{A}$.
   Receive puzzles $\mathsf{p}_{s^A_j}$, $\mathsf{p}_{\hat{\mathsf{c}}_j}$ and $\mathsf{p}_{\mathsf{seed}^A_{j,\mathsf{OT}}}$ for $j \in [\lambda]$ from $\mathcal{A}$.
   For all $j \neq \hat{\jmath}_B$, play the role of the receiver and run Π_{OT} with $\mathcal{A}$, using input 0^{n_B} and randomness derived from seed^B_j. In the $\hat{\jmath}_B$th execution of Π_{OT}, play the role of the receiver with input x_B and receive $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,x_B[i]}}\}_{i\in[n_B]}$.
   Generate ciphertexts for $\{\mathsf{p}_{A_{j,i,b}}\}$ for $j \in [\lambda]$ as in the protocol. Then play the role of the sender and execute Π_{OT} with $\mathcal{A}$ using $\{(\mathsf{p}_{A_{j,i,0}}, \mathsf{p}_{A_{j,i,1}})\}_{i\in[n_A]}$ as input with randomness derived from $\mathsf{seed}'_j \oplus \mathsf{seed}^B_{j,\mathsf{OT}}$ or $\mathsf{seed}^B_{j,\mathsf{OT}}$ as in the protocol.
7. Generate $\{\sigma^B_j\}$ and send them to $\mathcal{A}$. Receive $\{\sigma^A_j\}$ from $\mathcal{A}$. If any of the signatures are invalid, output $\perp$ to terminate.
8. Open the puzzle to $\mathcal{A}$ as in the protocol. $\mathcal{A}$ may also open her puzzle.
9. Decrypt the received puzzles/ciphertexts to derive s^A_j, $\hat{\mathsf{c}}_j$, and $\mathsf{seed}^A_{j,\mathsf{OT}}$ for $j \in [\lambda]$. In addition, decrypt $\{\mathsf{p}_{B_{\hat{\jmath}_B,i,x_B[i]}}\}_{i\in[n_B]}$ to obtain $\{B_{\hat{\jmath}_B,i,x_B[i]}\}_{i\in[n_B]}$.
   Simulate the computation of an honest $\mathsf{P_A}$ playing the role of the sender in the executions of Π_{OT} in Step 6 of the protocol for $j \in [\lambda]\backslash\{\hat{\jmath}_B\}$ based on $\mathsf{seed}^A_{j,\mathsf{OT}}$ and possibly seed'_j to obtain $\hat{\mathcal{H}}^A_j$. Let $\hat{J}_z$ be the set of indices that $\hat{\mathsf{c}}_j = 0$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the $\hat{\jmath}_A$th instance as described in the protocol. Let $\hat{J}_s$ be the set of indices that $\hat{\mathsf{c}}_j = \mathsf{c}_j$ and $\hat{\mathcal{H}}^A_j$ matches the simulation of Π_{OT} for the checking instances.
   - If $|\hat{J}_s| < \lambda-2$, generate cert as in the protocol and send cert to $\mathcal{A}$ to complete the simulation.
   - If $|\hat{J}_s| = \lambda - 2$ and $|\hat{J}_z| = 1$, continue the execution of the protocol.
   - If $|\hat{J}_s| = \lambda - 2$, $|\hat{J}_z| = 0$, generate cert as in the protocol and send cert to $\mathcal{A}$ to complete the simulation.
   - If $|\hat{J}_s| = \lambda - 1$, abort with output $\perp$.
10. Send the opening for c_B to $\mathcal{A}$ as in the protocol. Receive the opening for c_A from $\mathcal{A}$. If the opening $(\hat{\jmath}_A, \{\mathsf{seed}'^B_j\}_{j\neq\hat{\jmath}_A}, \mathsf{witness}'^B_{\hat{\jmath}_A})$ for c_A is incorrect, abort with output $\perp$.
11. Send the garbled circuit $\mathsf{GC}_{\hat{\jmath}_A}$, together with corresponding label hash values, to $\mathcal{A}$. Receive the garbled circuit $\mathsf{GC}_{\hat{\jmath}_B}$ and related materials from $\mathcal{A}$. If they are invalid, abort with output $\perp$.
12. Send the labels $\{B_{\hat{\jmath}_A,i,x_B[i]}\}$, $\gamma^B_{\hat{\jmath}_A}$, and corresponding decommitments to $\mathcal{A}$. Receive the labels, $\gamma^A_{\hat{\jmath}_B}$, and corresponding decommitments. If the received materials are invalid, abort with output $\perp$.
13. As in the protocol, evaluate the garbled circuit $\mathsf{GC}_{\hat{\jmath}_B}$ with input-wire labels $\{A_{\hat{\jmath}_B,i,x_A[i]}\}_{i\in[n_A]}$ and $\{B_{\hat{\jmath}_B,i,x_B[i]}\}_{i\in[n_B]}$ to obtain $\{Z_{\hat{\jmath}_B,i}\}_{i\in[n_O]}$ and $z_{\hat{\jmath}_B}$. If any decoded bits is $\perp$, let $z_{\hat{\jmath}_B} = \perp$.
   If $z_{\hat{\jmath}_B} = \perp$, let $\beta_B \leftarrow_{\$} \{0,1\}^\kappa$. Otherwise, we let $\beta_B \leftarrow \mathsf{H}(\bigoplus_{i=1}^{n_O}(Z_{\hat{\jmath}_A,i,z_{\hat{\jmath}_B}[i]} \oplus Z_{\hat{\jmath}_B,i,z_{\hat{\jmath}_B}[i]}))$. Use Π_{Eq} to check whether $\beta_A = \beta_B$. If $\mathcal{A}$ aborts, then abort. If $\beta_A \neq \beta_B$, then abort with output $\mathsf{cheating}$. Otherwise, output $z_{\hat{\jmath}_B}$.

We can easily check that $\mathbf{Expt}_{14}$ indeed is the execution of the real protocol $\Pi_{\mathsf{RobustPVC}}$ between $\mathsf{P_B}$ and $\mathcal{A}$. Hence, the protocol $\Pi_{\mathsf{RobustPVC}}$ achieves simulatability. □