Protecting Victims of Authorised Push Payment Fraud

1 Preliminaries

1.1 Notations and Assumptions

We use $\mathtt{Enc}(.)$ and $\mathtt{Dec}(.)$ to denote the encrypting and decrypting algorithms of a semantically secure symmetric key encryption scheme. In this work, we use a committee of honest arbiters $\mathcal{D} : \{\mathcal{D}_1, ..., \mathcal{D}_n\}$. Each arbiter, given a set of inputs, provides a binary verdict. We assume $\mathcal{D}_i$'s share a pair of keys $(pk_{\mathcal{D}}, sk_{\mathcal{D}})$ of an asymmetric key encryption. The encryption scheme has key generating $\tilde{\mathtt{keyGen}}(1^\lambda) \rightarrow (sk_{\mathcal{D}}, pk_{\mathcal{D}})$, encrypting $\tilde{\mathtt{Enc}}(pk_{\mathcal{D}}, .)$ and decrypting $\tilde{\mathtt{Dec}}(sk_{\mathcal{D}}, .)$ algorithms, where its public key is known to everyone. Also, we assume all arbiters have interacted with each other to agree on a secret key, $\bar{k}_0$. We use ϕ to denote a null value. Our proposed solution is built upon the existing online banking system. We assume the banking system has algorithm $\mathtt{pay}(.)$ that transfers money from the customer's account to a payee's account that is specific by the customer. We denote the inputs of this algorithm in_p. We assume the source code of the online banking system is static, and any change to the source code is transparent and can be detected, e.g., the bank uses a cryptographic commitment to commit to the source code. Moreover, we assume the online banking system is secure.

1.2 Digital Signature

A digital signature is a scheme for verifying the authenticity of digital messages or documents. Below, we restate its formal definition, taken from [8].

Definition 1. *A signature scheme involves three sub-processes,* $\mathtt{Signature} := (\mathtt{Sig.keyGen}, \mathtt{Sig.sign}, \mathtt{Sig.ver})$, *that are defined as follows.*

- $\mathtt{Sig.keyGen}(1^\lambda) \rightarrow (sk, pk)$. *A probabilistic algorithm run by a signer. It takes as input a security parameter. It outputs a key pair: (sk, pk), consisting of secret: sk, and public: pk keys.*
- $\mathtt{Sig.sign}(sk, pk, u) \rightarrow sig$. *An algorithm run the signer. It takes as input key pair: (sk, pk) and a file: u. It outputs a signature: sig.*
- $\mathtt{Sig.ver}(pk, u, sig) \rightarrow h \in \{0, 1\}$. *A deterministic algorithm run by a verifier. It takes as input public key: pk, file: u, and signature: sig. It checks the signature's validity. If the verification passes, then it outputs* 1; *otherwise, it outputs* 0.

A digital signature scheme should meet the following properties:

- *Correctness.* For every input u it holds that:

$$Pr\Big[\ \mathtt{Sig.ver}(pk, u, \mathtt{Sig.sign}(sk, pk, u)) = 1\ :\ \mathtt{Sig.keyGen}(1^\lambda) \rightarrow (sk, pk)\Big] = 1$$

- *Existential unforgeability under chosen message attacks.* A (PPT) adversary that obtains pk and has access to a signing oracle for messages of its choice, cannot create a valid pair (u^*, sig^*) for a new file u^* (that was never a query to the signing oracle), except with a small probability, σ. More formally:

$$\Pr\left[\begin{array}{l} u^* \notin Q \wedge \\ \mathtt{Sig.ver}(pk, u^*, sig^*) = 1 \end{array} : \begin{array}{l} \mathtt{Cer.keyGen}(1^\lambda) \rightarrow (sk, pk) \\ \mathcal{A}^{\mathtt{Sig.sign}(k,)}(pk) \rightarrow (u^*, sig^*) \end{array}\right] \leq \mu(\lambda)$$

where Q is the set of queries that $\mathcal{A}$ sent to the certificate generator oracle.

1.3 Smart Contract

Cryptocurrencies, such as Bitcoin [11] and Ethereum [14], beyond offering a decentralised currency, support computations on transactions. In this setting, often a certain computation logic is encoded in a computer program, called a *"smart contract"*. Although Bitcoin, the first decentralised cryptocurrency, supports smart contracts, the functionality of Bitcoin's smart contracts is very limited, due to the use of the underlying programming language that does not support arbitrary tasks. To address this limitation, Ethereum, as a generic smart contract platform, was designed. Thus far, Ethereum has been the most predominant cryptocurrency framework that lets users define arbitrary smart. This framework allows users to create an account with a unique account number or address. Such users are often called external account holders, which can send (or deploy) their contracts to the framework's blockchain. In this framework, a contract's code and its related data is held by every node in the blockchain's network. Ethereum smart contracts are often written in a high-level Turing-complete programming language called "Solidity".The program execution's correctness is guaranteed by the security of the underlying blockchain components. To prevent a denial of service attack, the framework requires a transaction creator to pay a fee, called *"gas"*, depending on the complexity of the contract running on it.

1.4 Commitment Scheme

A commitment scheme involves two parties, *sender* and *receiver*, and includes two phases: *commit* and *open*. In the commit phase, the sender commits to a message: x as $\texttt{Com}(x, r) = \texttt{Com}_x$, that involves a secret value: $r \stackrel{\$}{\leftarrow} \{0,1\}^\lambda$. In the end of the commit phase, the commitment $\texttt{Com}_x$ is sent to the receiver. In the open phase, the sender sends the opening $\ddot{x} := (x, r)$ to the receiver who verifies its correctness: $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy two properties: (a) *hiding*: it is infeasible for an adversary (i.e., the receiver) to learn any information about the committed message x, until the commitment $\texttt{Com}_x$ is opened, and (b) *binding*: it is infeasible for an adversary (i.e., the sender) to open a commitment $\texttt{Com}_x$ to different values $\ddot{x}' := (x', r')$ than that was used in the commit phase, i.e., infeasible to find $\ddot{x}'$, *s.t.* $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) = \texttt{Ver}(\texttt{Com}_x, \ddot{x}') = 1$, where $\ddot{x} \neq \ddot{x}'$. There exist efficient non-interactive commitment schemes both in (a) the standard model, e.g., Pedersen scheme [12], and (b) the random oracle model using the well-known hash-based scheme such that committing is : $\texttt{H}(x||r) = \texttt{Com}_x$ and $\texttt{Ver}(\texttt{Com}_x, \ddot{x})$ requires checking: $\texttt{H}(x||r) \stackrel{?}{=} \texttt{Com}_x$, where $\texttt{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a collision resistant hash function; i.e., the probability to find x and x' such that $\texttt{H}(x) = \texttt{H}(x')$ is negligible in the security parameter, λ.

1.5 Statement Agreement Protocol

Recently, a scheme called "Statement Agreement Protocol" (SAP) has been proposed in [3]. It lets two mutually distrusted parties, e.g., $\mathcal{B}$ and $\mathcal{C}$, efficiently agree on a private statement, π. Informally, the SAP satisfies the following four properties: (1) neither party can convince a third-party verifier that it has agreed with its counter-party on a different statement than the one both parties previously agreed on, (2) after they agree on a statement, an honest party can (almost) always prove to the verifier that it has the agreement, (3) the privacy of the statement is preserved (from the public), and (4) after both parties reach an agreement, neither can deny it. The SAP uses smart contract and commitment scheme. It assumes that each party has a blockchain public address, $adr_\mathcal{R}$ (where $\mathcal{R} \in \{\mathcal{B}, \mathcal{C}\}$). Below, we restate the SAP, taken from [3].

1. **Initiate**. $\texttt{SAP.init}(1^\lambda, adr_\mathcal{B}, adr_\mathcal{C}, \pi)$
   The following steps are taken by $\mathcal{B}$.
   (a) Deploys a smart contract that explicitly states both parties' addresses, $adr_\mathcal{B}$ and $adr_\mathcal{C}$. Let adr_{SAP} be the deployed contract's address.
   (b) Picks a random value r, and commits to the statement, $\texttt{Com}(\pi, r) = g_\mathcal{B}$.
   (c) Sends adr_{SAP} and $\ddot{\pi} := (\pi, r)$ to $\mathcal{C}$, and $g_\mathcal{B}$ to the contract.
2. **Agreement**. $\texttt{SAP.agree}(\pi, r, g_\mathcal{B}, adr_\mathcal{B}, adr_{\text{SAP}})$
   The following steps are taken by $\mathcal{C}$.

(a) Checks if $g_{\mathcal{B}}$ was sent from $adr_{\mathcal{B}}$, and checks locally $\texttt{Ver}(g_{\mathcal{B}}, \ddot{\pi}) = 1$.
(b) If the checks pass, it sets $b = 1$, computes locally $\texttt{Com}(\pi, r) = g_{\mathcal{C}}$, and sends $g_{\mathcal{C}}$ to the contract. Otherwise, it sets $b = 0$ and $g_{\mathcal{C}} = \perp$.

3. **Prove**. For either $\mathcal{B}$ or $\mathcal{C}$ to prove, it sends $\ddot{\pi} := (\pi, r)$ to the smart contract.
4. **Verify**. $\texttt{SAP.verify}(\ddot{\pi}, g_{\mathcal{B}}, g_{\mathcal{C}}, adr_{\mathcal{B}}, adr_{\mathcal{C}})$
The following steps are taken by the smart contract.
(a) Ensures $g_{\mathcal{B}}$ and $g_{\mathcal{C}}$ were sent from $adr_{\mathcal{B}}$ and $adr_{\mathcal{C}}$ respectively.
(b) Ensures $\texttt{Ver}(g_{\mathcal{B}}, \ddot{\pi}) = \texttt{Ver}(g_{\mathcal{C}}, \ddot{\pi}) = 1$.
(c) Outputs $s = 1$, if the checks, in steps 4a and 4b, pass. It outputs $s = 0$, otherwise.

1.6 Pseudorandom Function

Informally, a pseudorandom function is a deterministic function that takes a key of length Λ and an input; and outputs a value indistinguishable from that of a truly random function. In this paper, we use pseudorandom function: $\texttt{PRF} : \{0, 1\}^{\Lambda} \times \{0, 1\}^{*} \rightarrow \mathbb{F}_p$, where p is a large prime number, $|p| = \lambda$, and (Λ, λ) are the security parameters. In practice, a pseudorandom function can be obtained from an efficient block cipher [8].

1.7 Bloom Filter

A Bloom filter [4] is a compact data structure that allows us to efficiently check an element membership. It is an array of m bits (initially all set to zero), that represents n elements. It is accompanied with k independent hash functions. To insert an element, all the hash values of the element are computed and their corresponding bits in the filter are set to 1. To check an element membership, all its hash values are re-computed and checked whether all are set to 1 in the filter. If all the corresponding bits are 1, then the element is probably in the filter; otherwise, it is not. In Bloom filters it is possible that an element is not in the set, but the membership query indicates it is, i.e., false positives. In this work, we require that a Bloom filter uses *cryptographic* hash functions. Also, we ensure that the false positive probability is negligible, e.g., 2^{-40}. In Appendix A, we explain how the Bloom filter's parameters can be set.

2 An Overview of Payment with Dispute Resolution Scheme

In this section, we provide an overview of a Payment with Dispute Resolution (PwDR) scheme. Simply put, a PwDR scheme allows a customer to interact with its bank (via the online banking scheme) to transfer a certain amount of money from its account to another account in a transparent manner; meanwhile, it offers a distinct feature. Namely, when an APP fraud takes place, it lets an honest customer raise a dispute and *prove* to a third-party dispute resolver that it has acted honestly, so it can be reimbursed. It offers other features too. Specifically, an honest bank can also prove it has acted honestly. PwDR lets the parties prove their innocence without their counter-party's collaboration. It also ensures the message exchanged between a bank and customer remains confidential and even the party which resolves the dispute between the two learns as little information as possible. The PwDR scheme can be considered as an extension to the existing online banking system. To design such a scheme, we need to address several challenges. The rest of this section highlights these challenges.

2.1 Challenge 1: Lack of Transparent Logs

In the current online banking system, during a payment journey, the messages exchanged between customer and bank is usually logged by the bank and is not accessible to the customer without the bank's collaboration. Even if the bank provides access to the transaction logs, there is no guarantee that the logs have remained intact. Due to the lack of a transparent logging mechanism, a customer or bank can wrongly claim that (a) it has sent a certain message or warning to its counter-party or (b) it has never received a certain message, e.g., due to hardware or software failure. Thus, it would be hard for an honest party (especially a customer) to prove its innocence. To address this challenge, the PwDR protocol uses a standard smart contract (as a public bulletin board) to which each party needs to send (a copy of) all outgoing messages, e.g., payment requests, warnings, and confirmation of payments.

2.2 Challenge 2: Lack of Effective Warning's Accurate Definition in Banking

One of the determining factors in the process of allocating liability to the customers (after an APP fraud occurs) is paying attention to and following "warning(s)", according to the "Contingent Reimbursement Model" (CRM) code [10]. However, there exists no publicly available study on the effectiveness of every warning used and provided by the bank. Therefore, we cannot hold a customer accountable for becoming the fraud's victim, even if the related warnings are ignored. To address this challenge, we let a warning's effectiveness is determined on a case-by-case basis after the fraud takes place. In particular, the protocol provides an opportunity to a victim to challenge a certain warning whose effectiveness will be assessed by a committee, i.e., a small set of arbiters. In this setting, each member of the committee provides (an encoding of) its verdict to the smart contract, from which a dispute resolver retrieves all verdicts to find out the final verdict. The scheme ensures that the final verdict is in the customer's favour if at least threshold of committee members voted so.

2.3 Challenge 3: Linking Off-chain Payments with a Smart Contract

Recall that an APP fraud occurs when a payment is made. In the case where a bank sends a message (to the smart contract) to claim that it has transferred the money following the customer's request, it is not possible to automatically validate such a claim (e.g., for the smart contract doe the check) as the money transfer takes place outside of the blockchain network. To address this challenge, the protocol lets a customer raise a dispute and report it to the smart contract when it detects an inconsistency (e.g., the bank did not transfer the money but it wrongly declared it did so, or when it transferred the money but did not declare it). In this case, the above committee members investigate and provide their verdicts to the smart contract that allows the dispute resolver to extract the final verdict.

2.4 Challenge 4: Preserving Privacy

Although the use of a public transparent logging mechanism plays a vital role in resolving disputes, if no privacy-preserving mechanism is used, then it can violate parties' privacy, e.g., the customers' payment detail, bank's messages to the customer, or even each arbiter's verdict. To protect the privacy of the bank's and customers' messages against the public (and other customers), the PwDR scheme lets the customer and bank provably agree on encoding-decoding tokens that let them to encode their outgoing messages (sent to the smart contract). Later, either party can provide the token to a third party which can independently check the tokens' correctness, and decode the messages. To protect the privacy of the committee members' verdicts from the dispute resolver, the PwDR scheme must ensure that the dispute resolver can learn only the final verdict without being able to link a verdict to a specific member of the committee.

3 Definition of Payment with Dispute Resolution Scheme

In this section, we present a formal definition of a PwDR scheme. We first provide the scheme's syntax. Then, we formally define its correctness and security properties.

Definition 2. *A payment with dispute resolution scheme includes the following processes PwDR* := (`keyGen`, `bankInit`, `customerInit`, `genUpdateRequest`, `insertNewPayee`, `genWarning`, `genPaymentRequest`, `makePayment`, `genComplaint`, `verComplaint`, `resDispute`)*. It involves six types of entities; namely, bank $\mathcal{B}$, customer $\mathcal{C}$, smart contract $\mathcal{S}$, certificate generator $\mathcal{G}$, set of arbiters $\mathcal{D} : \{\mathcal{D}_1, ..., \mathcal{D}_n\}$, and a dispute resolver $\mathcal{DR}$. The processes of PwDR are defined below.*

- $\texttt{keyGen}(1^\lambda) \rightarrow (sk, pk)$. *It is a probabilistic algorithm run independently by $\mathcal{G}$ and one of the arbiters, $\mathcal{D}_j$. It takes as input a security parameter 1^λ. It outputs a pair of secret keys $sk := (sk_{\mathcal{G}}, sk_{\mathcal{D}})$ and public keys $pk := (pk_{\mathcal{G}}, pk_{\mathcal{D}})$. The public key pair, pk, is sent to all participants.*

- $\texttt{bankInit}(1^\lambda) \rightarrow (T, pp, \boldsymbol{l})$. *It is run by $\mathcal{B}$. It takes as input security parameter 1^λ. It allocates private parameters to $\ddot{\pi}_1$ and $\ddot{\pi}_2$. It generates an encoding-decoding token T, where $T := (T_1, T_2)$, each T_i contains a secret value $\ddot{\pi}_i$ and its public witness g_i. Given a value and its witness anyone can check if they match. It also generates a set of (additional) public parameters, pp, one of which is e that is a threshold parameter. It also generates an empty list, $\boldsymbol{l}$. It outputs T, pp and $\boldsymbol{l}$. $\mathcal{B}$ sends $(\ddot{\pi}_1, \ddot{\pi}_2)$ to $\mathcal{C}$ and sends $(g_1, g_2, pp, \boldsymbol{l})$ to $\mathcal{S}$.*

- $\texttt{customerInit}(1^\lambda, T, pp) \rightarrow a$. *It is a deterministic algorithm run by $\mathcal{C}$. It takes as input security parameter 1^λ, token T, and public parameters pp. It checks the correctness of the elements in T and pp. If the checks pass, it outputs 1. Otherwise, it outputs 0.*

- $\texttt{genUpdateRequest}(T, f, \boldsymbol{l}) \rightarrow \hat{m}_1^{(\mathcal{C})}$. *It is a deterministic process run by $\mathcal{C}$. It takes as input token T, new payee's detail f and empty payees' list $\boldsymbol{l}$. It generates $m_1^{(\mathcal{C})}$ which is an update request to the payees' list. It uses $T_1 \in T$ to encode $m_1^{(\mathcal{C})}$ which results in $\hat{m}_1^{(\mathcal{C})}$. It outputs $\hat{m}_1^{(\mathcal{C})}$. $\mathcal{C}$ sends the output to $\mathcal{S}$.*

- $\texttt{insertNewPayee}(\hat{m}_1^{(\mathcal{C})}, \boldsymbol{l}) \rightarrow \hat{\boldsymbol{l}}$. *It is a deterministic algorithm run by $\mathcal{S}$. It takes as input $\mathcal{C}$'s encoded update request $\hat{m}_1^{(\mathcal{C})}$, and $\mathcal{C}$'s payees' list $\boldsymbol{l}$. It inserts the new payee's detail into $\boldsymbol{l}$ and returns an updated list, $\hat{\boldsymbol{l}}$.*

- $\texttt{genWarning}(T, \hat{\boldsymbol{l}}, aux) \rightarrow \hat{m}_1^{(\mathcal{B})}$. *It is run by $\mathcal{B}$. It takes as input token T, $\mathcal{C}$'s encoded payees' list $\hat{\boldsymbol{l}}$ and auxiliary information: aux, e.g., a set of policies. Using $T_1 \in T$, it decodes and checks all elements of the list, e.g., whether they comply with the policies. If the check passes, then it sets $m_1^{(\mathcal{B})} =$ "pass"; otherwise, it sets $m_1^{(\mathcal{B})} = warning$, where the warning is a string containing a warning detail along with the string "warning". It uses T_1 to encode $m_1^{(\mathcal{B})}$ which yields $\hat{m}_1^{(\mathcal{B})}$. It outputs $\hat{m}_1^{(\mathcal{B})}$. $\mathcal{B}$ sends $\hat{m}_1^{(\mathcal{B})}$ to $\mathcal{S}$.*

- $\texttt{genPaymentRequest}(T, in_f, \hat{\boldsymbol{l}}, \hat{m}_1^{(\mathcal{B})}) \rightarrow \hat{m}_2^{(\mathcal{C})}$. *It is run by $\mathcal{C}$. It takes as input token T, a payment detail in_f, encoded payees' list $\hat{\boldsymbol{l}}$, and encoded warning message, $\hat{m}_1^{(\mathcal{B})}$. Using $T_1 \in T$, it decodes $\hat{\boldsymbol{l}}$ and $\hat{m}_1^{(\mathcal{B})}$ yielding $\boldsymbol{l}$ and $m_1^{(\mathcal{B})}$ respectively. It checks the warning. It sets $m_2^{(\mathcal{C})} = \phi$, if it does not want to proceed. Otherwise, it sets $m_2^{(\mathcal{C})}$ according to the content of in_f and $\boldsymbol{l}$ (e.g., the amount of payment and payee's detail). It uses T_1 to encode $m_2^{(\mathcal{C})}$ resulting in $\hat{m}_2^{(\mathcal{C})}$. It outputs $\hat{m}_2^{(\mathcal{C})}$. $\mathcal{C}$ sends $\hat{m}_2^{(\mathcal{C})}$ to $\mathcal{S}$.*

- $\texttt{makePayment}(T, \hat{m}_2^{(\mathcal{C})}) \rightarrow \hat{m}_2^{(\mathcal{B})}$. *It is a deterministic process run by $\mathcal{B}$. It takes as input token T, and encoded payment detail $\hat{m}_2^{(\mathcal{C})}$. Using $T_1 \in T$, it decodes $\hat{m}_2^{(\mathcal{C})}$ and checks the result's validity, e.g., ensures it is well-formed or $\mathcal{C}$ has enough credit. If the check passes, then it makes the payment and sets $m_2^{(\mathcal{B})} =$ "paid". Otherwise, it sets $m_2^{(\mathcal{B})} = \phi$. It uses T_1 to encode $m_2^{(\mathcal{B})}$ yielding $\hat{m}_2^{(\mathcal{B})}$. It outputs $\hat{m}_2^{(\mathcal{B})}$. $\mathcal{B}$ sends $\hat{m}_2^{(\mathcal{B})}$ to $\mathcal{S}$.*

- $\texttt{genComplaint}(\hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}, T, pk, aux_f) \rightarrow (\hat{z}, \hat{\ddot{\pi}})$. *It is run by $\mathcal{C}$. It takes as input the encoded warning message $\hat{m}_1^{(\mathcal{B})}$, encoded payment message $\hat{m}_2^{(\mathcal{B})}$, token T, public key pk, and auxiliary information aux_f. It initially sets fresh strings (z_1, z_2, z_3) to null. Using $T_1 \in T$, it decodes $\hat{m}_1^{(\mathcal{B})}$ and $\hat{m}_2^{(\mathcal{B})}$ and checks the results' content. If it wants to complain that (i) "pass" message should have been a warning or (ii) no message was provided, it sets z_1 to "challenge message". If its complaint is about the warning's effectiveness, it sets z_2 to a combination of an evidence $u \in aux_f$, the evidence's certificate $sig \in aux_f$, the certificate's public parameter, and "challenge warning", where the certificate is obtained from $\mathcal{G}$ via a query, Q. In certain cases, the certificate might be empty. If its complaint is about the payment, it sets z_3 to "challenge payment". It uses T_1 to encode $z := (z_1, z_2, z_3)$ and uses $pk_{\mathcal{D}}$ to encode $\ddot{\pi} := (\ddot{\pi}_1, \ddot{\pi}_2) \in T$. This results in $\hat{z}$ and $\hat{\ddot{\pi}}$ respectively. It outputs $(\hat{z}, \hat{\ddot{\pi}})$. $\mathcal{C}$ sends the pair to $\mathcal{S}$.*

- $\texttt{verComplaint}(\hat{z}, \hat{\ddot{\pi}}, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, j, sk_{\mathcal{D}}, aux, pp) \rightarrow \hat{\boldsymbol{w}}_j$. *It is run by every arbiter $\mathcal{D}_j$. It takes as input $\mathcal{C}$'s encoded complaint $\hat{z}$, encoded private parameters $\hat{\ddot{\pi}}$, the tokens' public parameters $g := (g_1, g_2)$, encoded messages $\hat{\boldsymbol{m}} = [\hat{m}_1^{(\mathcal{C})}, \hat{m}_2^{(\mathcal{C})}, \hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}]$, encoded payees' list $\hat{\boldsymbol{l}}$, the arbiter's index j, secret key $sk_{\mathcal{D}}$, auxiliary information aux, and public parameters pp. It uses $sk_{\mathcal{D}}$ to decode $\hat{\ddot{\pi}}$ that yields $\ddot{\pi} := (\ddot{\pi}_1, \ddot{\pi}_2)$. It uses $\ddot{\pi}_1$ to decode $\hat{z}, \hat{\boldsymbol{m}}$, and $\hat{\boldsymbol{l}}$ that results in $z := (z_1, z_2, z_3), \boldsymbol{m}$, and $\boldsymbol{l}$ respectively. It checks if $\ddot{\pi}_i$ matches g_i. If the check fails, it aborts; otherwise, it continues. It checks if $m_1^{(\mathcal{C})}$ and $m_2^{(\mathcal{C})}$ are non-empty; it aborts if the checks fail. It sets fresh parameters $(w_{1,j}, w_{2,j}, w_{3,j}, w_{4,j})$ to ϕ. If "challenge message" $\in z_1$, given $\boldsymbol{l}$, it checks whether "pass" message (in $m_1^{(\mathcal{B})}$) was given correctly or the missing message did not play any role in preventing the scam. If either checks passes, it sets $w_{1,j} = 0$; otherwise, it sets $w_{1,j} = 1$. If "challenge warning" $\in z_2$, it verifies the certificate in z_2. If it is invalid, it sets $w_{3,j} = 0$. If it is valid, it sets $w_{3,j} = 1$. It determines the effectiveness of the* *warning (in $m_1^{(\mathcal{B})}$), by running a process which determines that, i.e.,* $\texttt{checkWarning}(.) \in aux$*. If it is effective, i.e.,* $\texttt{checkWarning}(m_1^{(\mathcal{B})}) = 1$*, it sets its verdict to* 0*, i.e.,* $w_{2,j} = 0$*; otherwise, it sets* $w_{2,j} = 1$*. If "challenge payment"* $\in z_3$*, it checks if the payment was made (with the help of $m_2^{(\mathcal{B})}$). If the check passes, it sets* $w_{4,j} = 1$*; otherwise, it sets* $w_{4,j} = 0$*. It uses* $\ddot{\pi}_2$ *to encode* $\boldsymbol{w}_j = [w_{1,j}, w_{2,j}, w_{3,j}, w_{4,j}]$ *yielding* $\hat{\boldsymbol{w}}_j = [\hat{w}_{1,j}, \hat{w}_{2,j}, \hat{w}_{3,j}, \hat{w}_{4,j}]$*. It outputs* $\hat{\boldsymbol{w}}_j$*.* $\mathcal{D}_j$ *sends* $\hat{\boldsymbol{w}}_j$ *to* $\mathcal{S}$*.*

- $\texttt{resDispute}(T_2, \hat{\boldsymbol{w}}, pp) \rightarrow \boldsymbol{v}$*. It is a deterministic algorithm run by* $\mathcal{DR}$*. It takes as input token* T_2*, arbiters' encoded verdicts* $\hat{\boldsymbol{w}} = [\hat{\boldsymbol{w}}_1, ..., \hat{\boldsymbol{w}}_n]$*, and public parameters pp. It checks if the token's parameters match. If the check fails, it aborts; otherwise, it proceeds. It uses* $\ddot{\pi}_2 \in T_2$ *to decode* $\hat{\boldsymbol{w}}$ *and from the result it extracts final verdicts* $\boldsymbol{v} = [v_1, ..., v_4]$*. The extraction procedure ensures each* v_i *is set to* 1 *only if at least e arbiters' original verdicts (i.e.,* $w_{i,j,}$*) is* 1*, where* $e \in pp$*. It outputs* $\boldsymbol{v}$*. Note, if* $v_4 = 1$ *and (i) either* $v_1 = 1$ *(ii) or* $v_2 = 1$ *and* $v_3 = 1$*, then customer* $\mathcal{C}$ *must be reimbursed.*

Informally, a PwDR scheme has two properties; namely, *correctness* and *security*. Correctness requires that (in the absence of a fraudster) the payment journey is completed without the need for (i) the honest customer to complain and (ii) the honest bank to reimburse the customer. Below, we formally state it.

Definition 3 (Correctness). *A PwDR scheme is correct if the key generation algorithm produces keys* $\texttt{keyGen}(1^\lambda) \rightarrow (sk, pk)$ *such that for any payee's detail f, payment's detail* in_f*, and auxiliary information* (aux, aux_f)*, if* $\texttt{bankInit}(1^\lambda) \rightarrow (T, pp, \boldsymbol{l})$*,* $\texttt{customerInit}(1^\lambda, T, pp) \rightarrow a$*,* $\texttt{genUpdateRequest}(T, f, \boldsymbol{l}) \rightarrow \hat{m}_1^{(\mathcal{C})}$*,* $\texttt{insertNewPayee}(\hat{m}_1^{(\mathcal{C})}, \boldsymbol{l}) \rightarrow \hat{\boldsymbol{l}}$*,* $\texttt{genWarning}(T, \hat{\boldsymbol{l}}, aux) \rightarrow \hat{m}_1^{(\mathcal{B})}$*,* $\texttt{genPaymentRequest}(T, in_f, \hat{\boldsymbol{l}}, \hat{m}_1^{(\mathcal{B})}) \rightarrow \hat{m}_2^{(\mathcal{C})}$*,* $\texttt{makePayment}(T, \hat{m}_2^{(\mathcal{C})}) \rightarrow \hat{m}_2^{(\mathcal{B})}$*,* $\texttt{genComplaint}(\hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}, T, pk, aux_f) \rightarrow (\hat{z}, \hat{\ddot{\pi}})$*,* $\forall j \in [n] :$ $\Big(\texttt{verComplaint}(\hat{z}, \hat{\ddot{\pi}}, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, j, sk_{\mathcal{D}}, aux, pp) \rightarrow \hat{\boldsymbol{w}}_j\Big)$*,* $\texttt{resDispute}(T_2, \hat{\boldsymbol{w}}, pp) \rightarrow \boldsymbol{v}$*, then* $(z_1 = z_2 = z_3 = \phi) \wedge (\boldsymbol{v} = 0)$*, where* $g := (g_1, g_2) \in T$*,* $\hat{\boldsymbol{m}} = [\hat{m}_1^{(\mathcal{C})}, \hat{m}_2^{(\mathcal{C})}, \hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}]$*,* $\hat{\boldsymbol{w}} = [\hat{\boldsymbol{w}}_1, ..., \hat{\boldsymbol{w}}_n]$*, and* $z := (z_1, z_2, z_3)$ *is the decoded* $\hat{z}$*.*

A PwDR scheme is secure if it satisfies three main properties; namely, (a) security against a malicious victim, (b) security against a malicious bank, and (c) privacy. Below, we formally define them. Intuitively, security against a malicious victim requires that the victim of the APP fraud which is not qualified for the reimbursement should not be reimbursed (despite it tries to be). More specifically, a corrupt victim cannot (a) make at least threshold committee members, $\mathcal{D}_j$s, conclude that $\mathcal{B}$ should have provided a warning, although $\mathcal{B}$ has done so, or (b) make $\mathcal{DR}$ conclude that the pass message was incorrectly given or a vital warning message was missing despite less than threshold $\mathcal{D}_j$s believe so, or (c) persuade at least threshold $\mathcal{D}_j$s to conclude that the warning was ineffective although it was effective, or (d) make $\mathcal{DR}$ believe that the warning message was ineffective although only less than threshold $\mathcal{D}_j$s do believe that, or (e) convince $\mathcal{D}_j$s to accept an invalid certificate, or (f) make $\mathcal{DR}$ believe that at least threshold $\mathcal{D}_j$s accepted the certificate although they did not, except for a negligible probability.

Definition 4 (Security against a malicious victim). *A PwDR scheme is secure against a malicious victim, if for any security parameter* λ*, auxiliary information aux, and probabilistic polynomial-time adversary* $\mathcal{A}$*, there exists a negligible function* $\mu(\cdot)$*, such that for an experiment* $\mathsf{Exp}_1^{\mathcal{A}}$*:*

$\mathsf{Exp}_1^{\mathcal{A}}(1^{\lambda},\, aux)$

$\texttt{keyGen}(1^{\lambda}) \rightarrow (sk, pk)$
$\texttt{bankInit}(1^{\lambda}) \rightarrow (T, pp, \boldsymbol{l})$
$\mathcal{A}(1^{\lambda}, T, pp, \boldsymbol{l}) \rightarrow \hat{m}_1^{(\mathcal{C})}$
$\texttt{insertNewPayee}(\hat{m}_1^{(\mathcal{C})}, \boldsymbol{l}) \rightarrow \hat{\boldsymbol{l}}$
$\texttt{genWarning}(T, \hat{\boldsymbol{l}}, aux) \rightarrow \hat{m}_1^{(\mathcal{B})}$
$\mathcal{A}(T, \hat{\boldsymbol{l}}, \hat{m}_1^{(\mathcal{B})}) \rightarrow \hat{m}_2^{(\mathcal{C})}$
$\texttt{makePayment}(T, \hat{m}_2^{(\mathcal{C})}) \rightarrow \hat{m}_2^{(\mathcal{B})}$
$\mathcal{A}(\hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}, T, pk) \rightarrow (\hat{z}, \hat{\ddot{\pi}})$
$\forall j, j \in [n]:$
$\Big(\texttt{verComplaint}(\hat{z}, \hat{\ddot{\pi}}, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, j, sk_{\mathcal{D}}, aux, pp) \rightarrow \hat{\boldsymbol{w}}_j = [\hat{w}_{1,j}, \hat{w}_{2,j}, \hat{w}_{3,j}, \hat{w}_{4,j}]\Big)$
$\texttt{resDispute}(T_2, \hat{\boldsymbol{w}}, pp) \rightarrow \boldsymbol{v} = [v_1, ..., v_4]$

it holds that:

$$\Pr\left[\begin{array}{l}\Big((m_1^{(\mathcal{B})} = warning) \wedge (\sum_{j=1}^{n} w_{1,j} \geq e)\Big) \\ \vee\Big((\sum_{j=1}^{n} w_{1,j} < e) \wedge (v_1 = 1)\Big) \\ \vee\Big((\texttt{checkWarning}(m_1^{(\mathcal{B})}) = 1) \wedge (\sum_{j=1}^{n} w_{2,j} \geq e)\Big) \\ \vee\Big((\sum_{j=1}^{n} w_{2,j} < e) \wedge (v_2 = 1)\Big) \\ \vee\Big(u \notin Q \wedge \texttt{Sig.ver}(pk, u, sig) = 1\Big) \\ \vee\Big((\sum_{j=1}^{n} w_{3,j} < e) \wedge (v_3 = 1)\Big)\end{array} : \mathsf{Exp}_1^{\mathcal{A}}(\mathsf{input})\right] \leq \mu(\lambda),$$

where $g := (g_1, g_2) \in T$, $\hat{\boldsymbol{m}} = [\hat{m}_1^{(\mathcal{C})}, \hat{m}_2^{(\mathcal{C})}, \hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}]$, $(w_{1,j}, .., w_{3,j})$ *are the result of decoding* $(\hat{w}_{1,j}, .., \hat{w}_{3,j}) \in \hat{\boldsymbol{w}}$, $\texttt{checkWarning}(.)$ *determines a warning's effectiveness,* $\mathsf{input} := (1^{\lambda}, aux)$, $(u, sig) \in x$, $sk_{\mathcal{D}} \in sk$, *and* n *is the total number of arbiters. The probability is taken over the uniform choice of* sk*, randomness used in the blockchain's primitives (e.g., in signatures), randomness used during the encoding, and the randomness of* $\mathcal{A}$.

Intuitively, security against a malicious bank requires that a malicious bank should not be able to disqualify an honest victim of the APP scam from being reimbursed. In particular, a corrupt bank cannot (a) make $\mathcal{DR}$ conclude that the "pass" message was correctly given or an important warning message was not missing despite at least threshold $\mathcal{D}_j$s do not believe so, or (b) convince $\mathcal{DR}$ that the warning message was effective although at least threshold $\mathcal{D}_j$s do not believe so, or (c) make $\mathcal{DR}$ believe that less than threshold $\mathcal{D}_j$s did not accept the certificate although at least threshold of them did that, or (d) make $\mathcal{DR}$ believe that no payment was made, although at least threshold $\mathcal{D}_j$s believe the opposite, except for a negligible probability.

Definition 5 (Security against a malicious bank). *A PwDR scheme is secure against a malicious bank, if for any security parameter* λ*, auxiliary information aux, and probabilistic polynomial-time adversary* $\mathcal{A}$*, there exists a negligible function* $\mu(\cdot)$*, such that for an experiment* $\mathsf{Exp}_2^{\mathcal{A}}$*:*

$\mathsf{Exp}_2^{\mathcal{A}}(1^\lambda,\ aux)$

$\texttt{keyGen}(1^\lambda) \rightarrow (sk, pk)$
$\mathcal{A}(1^\lambda) \rightarrow (T, pp, \boldsymbol{l}, f, in_f, aux_f)$
$\texttt{customerInit}(1^\lambda, T, pp) \rightarrow a$
$\texttt{genUpdateRequest}(T, f, \boldsymbol{l}) \rightarrow \hat{m}_1^{(\mathcal{C})}$
$\texttt{insertNewPayee}(\hat{m}_1^{(\mathcal{C})}, \boldsymbol{l}) \rightarrow \hat{\boldsymbol{l}}$
$\mathcal{A}(T, \hat{\boldsymbol{l}}, aux) \rightarrow \hat{m}_1^{(\mathcal{B})}$
$\texttt{genPaymentRequest}(T, in_f, \hat{\boldsymbol{l}}, \hat{m}_1^{(\mathcal{B})}) \rightarrow \hat{m}_2^{(\mathcal{C})}$
$\mathcal{A}(T, \hat{m}_2^{(\mathcal{C})}) \rightarrow \hat{m}_2^{(\mathcal{B})}$
$\texttt{genComplaint}(\hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}, T, pk, aux_f) \rightarrow (\hat{z}, \hat{\ddot{\pi}})$
$\forall j, j \in [n]:$
$\Big(\texttt{verComplaint}(\hat{z}, \hat{\ddot{\pi}}, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, j, sk_{\mathcal{D}}, aux, pp) \rightarrow \hat{\boldsymbol{w}}_j = [\hat{w}_{1,j}, \hat{w}_{2,j}, \hat{w}_{3,j}, \hat{w}_{4,j}]\Big)$
$\texttt{resDispute}(T_2, \hat{\boldsymbol{w}}, pp) \rightarrow \boldsymbol{v} = [v_1, ..., v_4]$

it holds that:

$$\Pr\left[\begin{array}{l}\Big((\sum_{j=1}^{n} w_{1,j} \geq e) \wedge (v_1 = 0)\Big) \\ \vee\Big((\sum_{j=1}^{n} w_{2,j} \geq e) \wedge (v_2 = 0)\Big) \\ \vee\Big((\sum_{j=1}^{n} w_{3,j} \geq e) \wedge (v_3 = 0)\Big) \\ \vee\Big((\sum_{j=1}^{n} w_{4,j} \geq e) \wedge (v_4 = 0)\Big)\end{array} : \mathsf{Exp}_2^{\mathcal{A}}(\mathsf{input})\right] \leq \mu(\lambda),$$

where $g := (g_1, g_2) \in T$, $\hat{\boldsymbol{m}} = [\hat{m}_1^{(\mathcal{C})}, \hat{m}_2^{(\mathcal{C})}, \hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}]$, $(w_{1,j}, .., w_{3,j})$ *are the result of decoding* $(\hat{w}_{1,j}, .., \hat{w}_{3,j}) \in \hat{\boldsymbol{w}}$, $\mathsf{input} := (1^\lambda, aux)$, $sk_{\mathcal{D}} \in sk$, *n is the total number of arbiters. The probability is taken over the uniform choice of sk, randomness used in the blockchain's primitives, randomness used during the encoding, and the randomness of* $\mathcal{A}$.

A careful reader may ask why the following two conditions (some forms of which where in the events for Definition 4) are not added to the above events list: (a) $\mathcal{B}$ makes at least threshold committee members conclude that it has provided a warning, although $\mathcal{B}$ has not (i.e., $m_1^{(\mathcal{B})} \neq warning \wedge \sum_{j=1}^{n} w_{1,j} < e$), and (b) $\mathcal{B}$ persuades at least threshold $\mathcal{D}_j$s to conclude that the warning was effective although it was not (i.e., $\texttt{checkWarning}(m_1^{(\mathcal{B})}) = 0 \wedge \sum_{j=1}^{n} w_{2,j} < e$). The reason is that $\mathcal{B}$ does not generate a complaint and interact directly with $\mathcal{D}_j$s; therefore, we do not need to add these two events to the above events' list. Now we move on to privacy. Informally, a PwDR is privacy-preserving if it protects the privacy of (1) the customers', bank's, and arbiters' messages (except public parameters) from non-participants of the protocol, including other customers, and (2) each arbiter's verdict from $\mathcal{DR}$ which sees the final verdict.

Definition 6 (Privacy). *A PwDR preserves privacy if the following two properties are satisfied.*

1. *For any probabilistic polynomial-time adversary* $\mathcal{A}_1$, *security parameter* λ, *and auxiliary information aux, there exists a negligible function* $\mu(\cdot)$, *such that for any experiment* $\mathsf{Exp}_3^{\mathcal{A}_1}$:

$$\boxed{\begin{array}{l}\textbf{Exp}_3^{\mathcal{A}_1}(1^\lambda,\ aux)\\[4pt]
\texttt{keyGen}(1^\lambda) \rightarrow (sk, pk)\\
\texttt{bankInit}(1^\lambda) \rightarrow (T, pp, \boldsymbol{l})\\
\texttt{customerInit}(1^\lambda, T, pp) \rightarrow a\\
\mathcal{A}_1(1^\lambda, pk, a, pp, g, \boldsymbol{l}) \rightarrow \Big((f_0, f_1), (in_{f_0}, in_{f_1}), (aux_{f_0}, aux_{f_1})\Big)\\
\gamma \stackrel{\$}{\leftarrow} \{0, 1\}\\
\texttt{genUpdateRequest}(T, f_\gamma, \boldsymbol{l}) \rightarrow \hat{m}_1^{(\mathcal{C})}\\
\texttt{insertNewPayee}(\hat{m}_1^{(\mathcal{C})}, \boldsymbol{l}) \rightarrow \hat{\boldsymbol{l}}\\
\texttt{genWarning}(T, \hat{\boldsymbol{l}}, aux) \rightarrow \hat{m}_1^{(\mathcal{B})}\\
\texttt{genPaymentRequest}(T, in_{f_\gamma}, \hat{\boldsymbol{l}}, \hat{m}_1^{(\mathcal{B})}) \rightarrow \hat{m}_2^{(\mathcal{C})}\\
\texttt{makePayment}(T, \hat{m}_2^{(\mathcal{C})}) \rightarrow \hat{m}_2^{(\mathcal{B})}\\
\texttt{genComplaint}(\hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}, T, pk, aux_{f_\gamma}) \rightarrow (\hat{z}, \hat{\ddot{\pi}})\\
\forall j, j \in [n]:\\
\Big(\texttt{verComplaint}(\hat{z}, \hat{\ddot{\pi}}, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, j, sk_{\mathcal{D}}, aux, pp) \rightarrow \hat{\boldsymbol{w}}_j\Big)\\
\texttt{resDispute}(T_2, \hat{\boldsymbol{w}}, pp) \rightarrow \boldsymbol{v}
\end{array}}$$

it holds that:

$$\Pr\big[\mathcal{A}_1(g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, \hat{z}, \hat{\ddot{\pi}}, \hat{\boldsymbol{w}}) \rightarrow \gamma \,:\, \mathsf{Exp}_3^{\mathcal{A}_1}(\mathsf{input})\big] \leq \frac{1}{2} + \mu(\lambda).$$

2. *For any probabilistic polynomial-time adversaries $\mathcal{A}_2$ and $\mathcal{A}_3$, security parameter λ, and auxiliary information aux, there exists a negligible function $\mu(\cdot)$, such that for any experiment $\mathsf{Exp}_4^{\mathcal{A}_2}$:*

$$\boxed{\begin{array}{l}\textbf{Exp}_4^{\mathcal{A}_2}(1^\lambda,\ aux)\\[4pt]
\texttt{keyGen}(1^\lambda) \rightarrow (sk, pk)\\
\texttt{bankInit}(1^\lambda) \rightarrow (T, pp, \boldsymbol{l})\\
\texttt{customerInit}(1^\lambda, T, pp) \rightarrow a\\
\mathcal{A}_2(1^\lambda, pk, a, pp, \boldsymbol{l}) \rightarrow (f, in_f, aux_f)\\
\texttt{genUpdateRequest}(T, f, \boldsymbol{l}) \rightarrow \hat{m}_1^{(\mathcal{C})}\\
\texttt{insertNewPayee}(\hat{m}_1^{(\mathcal{C})}, \boldsymbol{l}) \rightarrow \hat{\boldsymbol{l}}\\
\mathcal{A}_2(T, \hat{\boldsymbol{l}}, aux) \rightarrow \hat{m}_1^{(\mathcal{B})}\\
\texttt{genPaymentRequest}(T, in_f, \hat{\boldsymbol{l}}, \hat{m}_1^{(\mathcal{B})}) \rightarrow \hat{m}_2^{(\mathcal{C})}\\
\mathcal{A}_2(T, pk, aux_f, \hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{C})}) \rightarrow (\hat{m}_2^{(\mathcal{B})}, \hat{z}, \hat{\ddot{\pi}})\\
\forall j, j \in [n]:\\
\Big(\texttt{verComplaint}(\hat{z}, \hat{\ddot{\pi}}, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, j, sk_{\mathcal{D}}, aux, pp) \rightarrow \hat{\boldsymbol{w}}_j\Big)\\
\texttt{resDispute}(T_2, \hat{\boldsymbol{w}}, pp) \rightarrow \boldsymbol{v}
\end{array}}$$

it holds that:

$$\Pr\big[\mathcal{A}_3(T_2, pk, pp, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, \hat{z}, \hat{\ddot{\pi}}, \hat{\boldsymbol{w}}, \boldsymbol{v}) \rightarrow w_j \,:\, \mathsf{Exp}_4^{\mathcal{A}_2}(\mathsf{input})\big] \leq Pr' + \mu(\lambda),$$

where $g := (g_1, g_2) \in T$, $\hat{\boldsymbol{m}} = [\hat{m}_1^{(\mathcal{C})}, \hat{m}_2^{(\mathcal{C})}, \hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}], \hat{\boldsymbol{w}} = [\hat{\boldsymbol{w}}_1, .., \hat{\boldsymbol{w}}_n]$, $\mathsf{input} := (1^\lambda, aux)$, $sk_{\mathcal{D}} \in sk$, n is the total number of arbiters. Moreover, Pr' is defined as follows. Let arbiter $\mathcal{D}_i$ output 0 and 1 with probabilities $Pr_{i,0}$ and $Pr_{i,1}$ respectively. Then, Pr' is defined as $Max\{Pr_{1,0}, Pr_{1,1}, ..., Pr_{n,0}, Pr_{n,1}\}$. In the above privacy definition, the probability is taken over the uniform choice of sk, the probability that *each $\mathcal{D}_j$ outputs* 0 *or* 1*, the randomness used in the blockchain's primitives, the randomness used during the encoding, and the randomness of $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Definition 7 (Security). *A PwDR is secure if it meets security against a malicious victim, security against a malicious bank, and preserves privacy with respect to definitions 4, 5, and 6 respectively.*

4 Payment with Dispute Resolution Protocol

In this section, first we provide an outline of the PwDR protocol. Then, we present a few subroutines that will be used in this protocol. After that, we describe the PwDR protocol in detail.

4.1 An Overview of the PwDR Protocol

In this section, we provide an overview of the PwDR protocol. For the sake of simplicity, we assume $\mathcal{C}$ wants to transfer a certain amount of money to a new payee. Initially, only for once, customer $\mathcal{C}$ and bank $\mathcal{B}$ agree on a smart contract $\mathcal{S}$. They also use the SAP to provably agree on two private statements that include two secret keys. The keys will be used to encrypt messages sent to $\mathcal{S}$ and will be used by $\mathcal{D}_j$s and $\mathcal{DR}$ to decrypted related messages. When $\mathcal{C}$ wants to transfer money to a new payee, it signs into the standard online banking system. Then, it generates an update request that specifies the new payee's detail, encrypts the request, and sends the result to $\mathcal{S}$. After that, $\mathcal{B}$ decrypts and checks the request's validity, e.g., whether it meets its internal policy or the requirements of the "Confirmation of Payee" [6]. Depending on the request's content, $\mathcal{B}$ generates a pass or warning message. It encrypts the message and sends the result to $\mathcal{S}$. Then, $\mathcal{C}$ checks $\mathcal{B}$'s message and depending on the content of this message, it decides whether it wants to proceed to made payment. If it decides to do so, then it sends an encrypted payment detail to $\mathcal{S}$. This allows $\mathcal{B}$ to decrypt the message and locally transfer the amount of money specified in $\mathcal{C}$'s message. Once the money is transferred, $\mathcal{B}$ sends an encrypted "paid" message to $\mathcal{S}$.

Once $\mathcal{C}$ realises that it has fallen victim to an APP fraud, it raises a dispute. In particular, it generates an encrypted complaint that could challenge the effectiveness of the warning and/or any payment inconsistency (as explained in Section 2.3). It can also include in the complaint an evidence/certificate, e.g., to claim that it falls into the vulnerable customer category as defined in [10]. $\mathcal{C}$ encrypts the complaint. It also encrypts the secret key (under arbiters' public key) that it uses to encrypt the messages. It sends to $\mathcal{S}$ the ciphertexts along with a proof asserting the secret key's correctness. Upon receiving $\mathcal{C}$'s complaint, each committee member verifies the proof. If the verification passes, it decrypts and compiles $\mathcal{C}$'s complaint to generate a (set of) verdict. Then, each committee member encodes its verdict and sends the encryption of the encoded verdict to $\mathcal{S}$. To resolve a dispute between $\mathcal{C}$ and $\mathcal{B}$, either of them can invoke $\mathcal{DR}$. To do so, they directly send to it one of the above secret keys and a proof asserting that key was generated correctly. $\mathcal{DR}$ verifies the proof. If the verification passes it locally decrypts the encrypted encoded verdicts (after retrieving them from $\mathcal{S}$) and then combines the result to find out the final verdict. If the final verdict indicates the legitimacy of $\mathcal{C}$'s complaint, then $\mathcal{C}$ must be reimbursed. Note, the verdicts are encoded in such a way that even after decrypting them, the dispute resolver cannot link a verdict to a committee member or even figure out how many 1 or 0 verdicts have been provided (except when all verdicts are 0). However, it can find out whether at least threshold committee members voted in faviour of $\mathcal{C}$. Shortly, we present novel verdict encoding-decoding protocols that offer the above features.

4.2 A Subroutine for Determining Bank's Message Status

As we stated earlier, in the payment journey the customer may receive a "pass" message or even nothing at all, e.g., due to a system failure. In such cases, a victim of an APP fraud may complain that if the pass or missing message was a warning message, then it would have prevented the victim from falling to the APP fraud. To assist the committee members to deal with such complaints deterministically, we propose a process, called `verStat`. It is run locally by each committee member. It outputs 0 if a pass message was given correctly or the missing message could not prevent the scam, and outputs 1 otherwise. The process is presented in figure 1.

$\texttt{verStat}(add_{\mathcal{S}}, m^{(\mathcal{B})}, \boldsymbol{l}, \Delta, aux) \rightarrow w_1$

- *Input.* $add_{\mathcal{S}}$: the address of smart contract $\mathcal{S}$, $m^{(\mathcal{B})}$: $\mathcal{B}$'s warning message, $\boldsymbol{l}$: customer's payees' list, Δ: a time parameter, and aux: auxiliary information, e.g., bank's policy.
- *Output.* $w_1 = 0$: if the "pass" message had been given correctly or the missing message did not play any role in preventing the scam; $w_1 = 1$: otherwise.

1. reads the content of $\mathcal{S}$. It checks if $m^{(\mathcal{B})} =$"pass" or the encrypted warning message was not sent on time (i.e., never sent or sent after $t_0 + \Delta$). If one of the checks passes, it proceeds to the next step. Otherwise, it aborts.
2. checks the validity of customer's most recent payees' list $\boldsymbol{l}$, with the help of the auxiliary information, aux.
   - if $\boldsymbol{l}$ contains an invalid element, it sets $w_1 = 1$.
   - otherwise, it sets $w_1 = 0$.
3. returns w_1.

Fig. 1: Process to Determine a Bank's Message Status

4.3 A Subroutine for Checking Warning's Effectiveness

To help the committee members deterministically and accurately compile a victim's complaint about the effectiveness of a warning (that bank provides during the payment journey) we propose a process, called `checkWarning`. This process is run locally by each committee member. It also allows the victims to provide a certificate/evidence as part of their complaints. The process outputs a pair (w_2, w_3). It sets $w_2 = 0$ if the given warning message is effective, and sets $w_2 = 1$, if it is not. It sets $w_3 = 1$ if the certificate that the victim provided is valid (or empty) and sets $w_3 = 0$ if it is invalid. This process is presented in figure 2.

4.4 Subroutines for Encoding-Decoding Verdicts

In this section, we present verdict encoding and decoding protocols. They let a third party $\mathcal{I}$, e.g., $\mathcal{DR}$, find out whether threshold arbiters voted 1, while satisfying the following requirements. The protocols should (1) generate unlinkable verdicts, (2) not require arbiters to interact with each other for each customer, and (3) be efficient. Since the second and third requirements are self-explanatory, we only explain the first one. Informally, the first property requires that the protocols should generate encoded verdicts and final verdict in a way that $\mathcal{I}$, given the encoded verdicts and final verdict, should not be able to (a) link a verdict to an arbiter (except when all arbiters' verdicts are 0), and (b) find out the total number of 1 or 0 verdicts when they provide different verdicts. In this section, we present two variants of verdict encoding and decoding protocol. The first variant is highly efficient and suitable for the case where the threshold is 1. The second variant is generic and works for any threshold. The latter variant is slightly less efficient than the former one. These two variants might be of independent interest. Below, we explain each variant.

Variant 1: Highly Efficient Verdict Encoding-Decoding Protocol. Below, we present efficient verdict encoding and decoding protocols; namely, Private Verdict Encoding (PVE) and Final Verdict Decoding (FVD). They let $\mathcal{I}$ find out whether at least one arbiter voted 1, while satisfying the above requirements. This variant mainly relies on our observation that if a set of random values and 0s are XORed, then the result reveals nothing, e.g., about the number of non-zero and zero values. Below, we formally state it. We refer readers to Appendix B for the proof.

$\texttt{checkWarning}(add_{\mathcal{S}}, z, m^{(\mathcal{B})}, aux') \rightarrow (w_2, w_3)$

- *Input.* $add_{\mathcal{S}}$: the address of smart contract $\mathcal{S}$, z: $\mathcal{C}$'s complaint, $m^{(\mathcal{B})}$: $\mathcal{B}$'s warning message, and aux': auxiliary information, e.g., guideline on warnings' effectiveness.
- *Output.* $w_2 = 0$: if the given warning message is effective; $w_2 = 1$: if the warning message is ineffective. Also, $w_3 = 1$: if the certificate in z is valid or no certificate is provided; $w_3 = 0$: if the certificate is invalid.

1. parse $z = m||sig||pk||$"challenge warning". If the certificate sig is empty, then it sets $w_3 = 0$ and proceeds to step 2. Otherwise, it:
   (a) verifies the certificate: $\texttt{Sig.ver}(pk, m, sig) \rightarrow h$.
   (b) if the certificate is rejected (i.e., $h = 0$), it sets $w_3 = 0$. It goes to step 4.
   (c) otherwise (i.e., $h = 1$), it sets $w_3 = 1$ and moves onto the next step.
2. checks if "warning" $\in m^{(\mathcal{B})}$. If the check is passed, it proceeds to the next step. Otherwise, it aborts.
3. checks the warning's effectiveness, with the assistance of the evidence m and auxiliary information aux'.
   - if it is effective, it sets $w_2 = 0$.
   - otherwise, it sets $w_2 = 1$.
4. returns (w_2, w_3).

Fig. 2: Process to Check Warning's Effectiveness

Theorem 1. *Let set $S = \{s_1, ..., s_m\}$ be the union of two disjoint subsets S' and S'', where S' contains non-zero random values pick uniformly from a finite field $\mathbb{F}_p$, S'' contains zeros, $|S'| \geq c' = 1$, $|S''| \geq c'' = 0$, and pair (c', c'') is public information. Then, $r = \bigoplus_{i=1}^{m} s_i$ reveals nothing beyond the public information.*

At a high level, PVE and FVD work as follows. The arbiters only once for all customers agree on a secret key of a pseudorandom function. This key will let each of them, in PVE, generate a pseudorandom masking value such that if all masking values are XORed, they would cancel out each other and result in 0.[1] In PVE, each arbiter encodes its verdict by (i) representing it as a parameter which is set to 0 if the verdict is 0, or to a random value if the verdict is 1, and then (ii) masking this parameter by the above pseudorandom value. It sends the result to $\mathcal{I}$. In FVD, to decode the final verdict and find out whether any arbiter voted 1, $\mathcal{I}$ XORs all encoded verdicts. This removes the masks and XORs are verdicts' representations. If the result is 0, then it concludes that all arbiters must have voted 0; therefore, the final verdict is 0. However, if the result is not 0 (i.e., a random value), then it knows that at least one of the arbiters voted 1, so the final verdict is 1.

It is evident that the protocols meet properties (2) and (3). The primary reason they also meet property (1) is that each masked verdict reveals nothing about the verdict (and its representation) and given the final verdict, $\mathcal{I}$ cannot distinguish between the case where there is exactly one arbiter that voted 1 and the case where multiple arbiters voted 1, as in both cases $\mathcal{I}$ extracts only a single random value, which reveals nothing about the number of arbiters which voted 0 or 1 (due to Theorem 1). Note, the protocols' correctness holds with an overwhelming probability, i.e., $1 - \frac{1}{2^\lambda}$. Specifically, if two arbiters represent their verdict by an identical random value, then when they are XORed they would cancel out each other which can affect the result's correctness. The same holds if the XOR of multiple verdicts' representations results in a value that can cancel out another verdict's representation. Nevertheless, the probability that such an event occurs is negligible in the security parameter $|p| = \lambda$, i.e., at most $\frac{1}{2^\lambda}$.

[1] This is similar to the idea used in the XOR-based secret sharing [13].

Variant 2: Generic Verdict Encoding-Decoding Protocol. Now, we present efficient *generic* verdict encoding-decoding protocol, denoted by GPVE and GFVD. They let $\mathcal{I}$ find out whether at least e arbiters voted 1, where e can be any integer in the range $[1, n]$. This variant is built upon the previous one; however, it also uses a novel combination of Bloom filter and combinatorics. It relies on our observation that a Bloom filter encoding a set of random values reveals nothing about the set's elements, except with a negligible probability. We formally state it in Theorem 2 whose proof is given in Appendix C.

Theorem 2. *Let set $S = \{s_1, ..., s_m\}$ be a set of random values picked uniformly from $\mathbb{F}_p$, where the cardinality of S is public information. Let* `BF` *be a Bloom filter encoding all elements of S. Then,* `BF` *reveals nothing about any element of S, beyond the public information, except with a negligible probability in the security parameter.*

The arbiters (similar to Variant 1) agree on a secret key of a pseudorandom function. In GPVE, as before, each arbiter will use this key to generate a pseudorandom masking value such that if all arbiters' masking values are XORed, they would cancel out each other. Then, each arbiter represents its verdict by a parameter. In particular, if its verdict is 0, then it sets the parameter to 0. However, if its verdict is 1, it sets the parameter to a fresh *pseudorandom* value α_j (instead of a random value used in Variant 1), where this pseudorandom value is also derived from the above key. Therefore, there would be a set $A = \{\alpha_1, ..., \alpha_n\}$ from which $\mathcal{D}_j$ would pick α_j to represent its verdict if its verdict is 1. Next, each arbiter masks its verdict representation by its masking value. It sends the result to $\mathcal{I}$. Arbiter $\mathcal{D}_n$ also generates a new set W that contains all combinations of verdict 1's representations that satisfy the threshold, e. More specifically, for every integer i in the range $[e, n]$, it computes the combinations (without repetition) of i elements from set $A = \{\alpha_1, ..., \alpha_n\}$. In the case where multiple elements are taken at a time (i.e., $i > 1$), the elements are XORed with each other. Note, $\mathcal{D}_n$ generates set B regardless of whether a certain arbiter's vote is 0 or 1. Let $W = \{(\alpha_1 \oplus ... \oplus \alpha_e), (\alpha_2 \oplus ... \oplus \alpha_{e+1}), ..., (\alpha_1 \oplus ... \oplus \alpha_n)\}$ be the result. For instance, if the total number of arbiters, n, is 3 and the threshold, e, is 2, then $W = \{(\alpha_1 \oplus \alpha_2), (\alpha_2 \oplus \alpha_3), (\alpha_1 \oplus \alpha_3), (\alpha_1 \oplus \alpha_2 \oplus \alpha_n)\}$. After that, it generates an empty Bloom filter and inserts all elements of W into this Bloom filter. Let `BF` be the Bloom filter that encodes W's elements. It sends `BF` to $\mathcal{I}$. Note that inserting the combinations into `BF` ensures that the privacy of each vote's representation (e.g., α_j) is protected from $\mathcal{I}$.

In GFVD, to decode and extract the final verdict, as in Variant 1, party $\mathcal{I}$ XORs all masked verdict representations which removes the masking values and XORs all verdicts' representations. Let c be the result. If $c = 0$, then $\mathcal{I}$ concludes that all arbiters must have voted 0 (with a high probability); so, it sets the final verdict to 0. However, if c is a non-zero value, then it checks whether c is in the Bloom filter. If it is, then it concludes that at least threshold arbiters voted 1, so it sets the final vector to 1. Otherwise (if c is not in the Bloom filter), it concludes that less than threshold arbiters voted 1; therefore, it sets the final verdict to 0. Figures 6 and 7, in Appendix D, present the GPVE and GFVD protocols in detail. Note that since `BF` contains all possible combinations of verdict 1's representations that meet the threshold, $\mathcal{DR}$ can always find out if c meets the threshold by just checking if c is in the `BF`. The total number of the combinations, i.e., the cardinality of W, is relatively small when the number of arbiters is not very high. In general, due to the binomial theorem, the cardinality of W is determined as follows:

$$|W| = \sum_{i=e}^{n} \frac{n!}{i!(n-i)!}$$

For instance, when $n = 10$ and $e = 6$, then W's cardinality is only 386. In the above scheme, instead of inserting the combinations into `BF` we could simply hash the combinations and give the hash values to $\mathcal{I}$. However, using a Bloom filter allows us to save considerable communication cost. For instance, when $n = 10$, $e = 6$, and SHA-256 is used, then $\mathcal{D}_n$ needs to send $98816 = 386 \times 256$ bits to $\mathcal{I}$, whereas if they are inserted to a Bloom filter, then it only needs to send 22276 bits to $\mathcal{I}$. Thus, by using a Bloom filter, it can save communication costs by at least a factor of 4. We refer readers to Appendix E for further discussion on the verdict encoding-decoding protocols.

$\underline{\texttt{PVE}(\bar{k}_0, \text{ID}, w_j, o, n, j) \rightarrow \bar{w}_j}$

- *Input.* $\bar{k}_0$: a key of pseudorandom function $\texttt{PRF}(.)$, ID: a unique identifier, w_j: a verdict, o: an offset, n: the total number of arbiters, and j: an arbiter's index.
- *Output.* $\bar{w}_j$: an encoded verdict.

Arbiter $\mathcal{D}_j$ takes the following steps.

1. computes a pseudorandom value, as follows.
   - if $j < n : r_j = \texttt{PRF}(\bar{k}_0, o||j||\text{ID})$.
   - if $j = n : r_j = \bigoplus_{i=1}^{n-1} r_i$.
2. sets a fresh parameter, w'_j, as below.

$$w'_j = \begin{cases} 0, & \text{if } w_j = 0 \\ \alpha_j \stackrel{\$}{\leftarrow} \mathbb{F}_p, & \text{if } w_j = 1 \end{cases}$$

3. encodes w'_j as follows. $\bar{w}_j = w'_j \oplus r_j$.
4. outputs $\bar{w}_j$.

Fig. 3: Private Verdict Encoding (PVE) Protocol

$\underline{\texttt{FVD}(n, \bar{\boldsymbol{w}}) \rightarrow v}$

- *Input.* n: the total number of arbiters, and $\bar{\boldsymbol{w}} = [\bar{w}_1, ..., \bar{w}_n]$: a vector of all arbiters' encodes verdicts.
- *Output.* v: final verdict.

A third-party $\mathcal{I}$ takes the following steps.

1. combines all arbiters' encoded verdicts, $\bar{w}_j \in \bar{\boldsymbol{w}}$, as follows. $c = \bigoplus_{j=1}^{n} \bar{w}_j$
2. sets the final verdict v depending on the content of c. Specifically,

$$v = \begin{cases} 0, & \text{if } c = 0 \\ 1, & \text{otherwise} \end{cases}$$

3. outputs v.

Fig. 4: Final Verdict Decoding (FVD) Protocol

4.5 The PwDR Protocol

In this section, we present the PwDR protocol in detail.

1. *Generating Certificate Parameters*. $\texttt{keyGen}(1^\lambda) \rightarrow (sk, pk)$
   The certificate generator takes step 1a and an arbiter takes step 1b below.
   (a) calls $\texttt{Sig.keyGen}(1^\lambda) \rightarrow (sk_\mathcal{G}, pk_\mathcal{G})$ to generate signing secret key $sk_\mathcal{G}$ and verfiying public key $pk_\mathcal{G}$. It publishes the public key, $pk_\mathcal{G}$.
   (b) calls $\widetilde{\texttt{keyGen}}(1^\lambda) \rightarrow (sk_\mathcal{D}, pk_\mathcal{D})$ to generate decrypting secret key $sk_\mathcal{D}$ and encrypting public key $pk_\mathcal{D}$. It publishes the public key $pk_\mathcal{D}$ and sends $sk_\mathcal{D}$ to the rest of arbiters.

Let $sk := (sk_{\mathcal{G}}, sk_{\mathcal{D}})$ and $pk := (pk_{\mathcal{G}}, pk_{\mathcal{D}})$. Note, this phase takes place only once for all customers.

2. Bank-side Initiation. $\texttt{bankInit}(1^{\lambda}) \rightarrow (T, pp, \boldsymbol{l})$
   $\mathcal{B}$ takes the following steps.
   (a) picks secret keys $\bar{k}_1$ and $\bar{k}_2$ for symmetric key encryption scheme and pseudorandom function $\texttt{PRF}$ respectively. It sets two private statements as $\pi_1 = \bar{k}_1$ and $\pi_2 = \bar{k}_2$.
   (b) calls $\texttt{SAP.init}(1^{\lambda}, adr_{\mathcal{B}}, adr_{\mathcal{C}}, \pi_i) \rightarrow (r_i, g_i, adr_{\text{SAP}})$ to initiate agreements on statements $\pi_i \in \{\pi_1, \pi_2\}$ with customer $\mathcal{C}$. Let $T_i := (\ddot{\pi}_i, g_i)$ and $T := (T_1, T_2)$, where $\ddot{\pi}_i := (\pi_i, r_i)$ is the opening of g_i. It also sets parameter Δ as a time window between two specific time points, i.e., $\Delta = t_i - t_{i-1}$. Briefly, it is used to impose an upper bound on a message delay.
   (c) sends $\ddot{\pi} := (\ddot{\pi}_1, \ddot{\pi}_2)$ to $\mathcal{C}$ and sends public parameter $pp := (adr_{\text{SAP}}, \Delta)$ to smart contract $\mathcal{S}$.

3. Customer-side Initiation. $\texttt{customerInit}(1^{\lambda}, T, pp) \rightarrow a$
   $\mathcal{C}$ takes the following steps.
   (a) calls $\texttt{SAP.agree}(\pi_i, r_i, g_i, adr_{\mathcal{B}}, adr_{\text{SAP}}) \rightarrow (g'_i, b_i)$, to check the correctness of parameters in $T_i \in T$ and (if accepted) to agree on these parameters, where $(\pi_i, r_i) \in \ddot{\pi}_i \in T_i$ and $1 \leq i \leq 2$. Note, if both $\mathcal{B}$ and $\mathcal{C}$ are honest, then $g_i = g'_i$. It also checks Δ in $\mathcal{S}$, e.g., to see whether it is sufficiently large.
   (b) if the above checks fail, it sets $a = 0$. Otherwise, it sets $a = 1$. It sends a to $\mathcal{S}$.

4. Generating Update Request. $\texttt{genUpdateRequest}(T, f, \boldsymbol{l}) \rightarrow \hat{m}_1^{(\mathcal{C})}$
   $\mathcal{C}$ takes the following steps.
   (a) sets its request parameter $m_1^{(\mathcal{C})}$ as below.
      - if it wants to setup a new payee, then it sets $m_1^{(\mathcal{C})} := (\phi, f)$, where f is new payee's detail.
      - if it wants to amend existing payee's detail, then it sets $m_1^{(\mathcal{C})} := (i, f)$, where i is the index of the element in $\boldsymbol{l}$ that should be changed to f.
   (b) at time t_0, sends to $\mathcal{S}$ the encryption of $m_1^{(\mathcal{C})}$, i.e., $\hat{m}_1^{(\mathcal{C})} = \texttt{Enc}(\bar{k}_1, m_1^{(\mathcal{C})})$.

5. Inserting New Payee. $\texttt{insertNewPayee}(\hat{m}_1^{(\mathcal{C})}, \boldsymbol{l}) \rightarrow \hat{\boldsymbol{l}}$
   $\mathcal{S}$ takes the following steps.
   - if $\hat{m}_1^{(\mathcal{C})}$ is not empty, it appends $\hat{m}_1^{(\mathcal{C})}$ to the payee list $\hat{\boldsymbol{l}}$, resulting an updated list, $\hat{\boldsymbol{l}}$.
   - if $\hat{m}_1^{(\mathcal{C})}$ is empty, it does nothing.

6. Generating Warning. $\texttt{genWarning}(T, \hat{\boldsymbol{l}}, aux) \rightarrow \hat{m}_1^{(\mathcal{B})}$
   $\mathcal{B}$ takes the following steps.
   (a) checks if the most recent list $\hat{\boldsymbol{l}}$ is not empty. If it is empty, it halts. Otherwise, it proceeds to the next step.
   (b) decrypts each element of $\hat{\boldsymbol{l}}$ and checks their correctness, e.g., checks whether each element meets its internal policy or CoP requirements stated in aux. If the check passes, it sets $m_1^{(\mathcal{B})}$ = "pass". Otherwise, it sets $m_1^{(\mathcal{B})}$ = warning, where warning is a string that contains a warning's detail concatenated with the string "warning".
   (c) at time t_1, sends to $\mathcal{S}$ the encryption of $m_1^{(\mathcal{B})}$, i.e., $\hat{m}_1^{(\mathcal{B})} = \texttt{Enc}(\bar{k}_1, m_1^{(\mathcal{B})})$.

7. Generating Payment Request. $\texttt{genPaymentRequest}(T, in_f, \hat{\boldsymbol{l}}, \hat{m}_1^{(\mathcal{B})}) \rightarrow \hat{m}_2^{(\mathcal{C})}$
   $\mathcal{C}$ takes the following steps.
   (a) at time t_2, decrypts the content of $\hat{\boldsymbol{l}}$ and $\hat{m}_1^{(\mathcal{B})}$. It sets a payment request $m_2^{(\mathcal{C})}$ to ϕ or in_f, where in_f contains the payment's detail, e.g., the payee's detail in $\boldsymbol{l}$ and the amount it wants to transfer.
   (b) at time t_3, sends to $\mathcal{S}$ the encryption of $m_2^{(\mathcal{C})}$, i.e., $\hat{m}_2^{(\mathcal{C})} = \texttt{Enc}(\bar{k}_1, m_2^{(\mathcal{C})})$.

8. Making Payment. $\texttt{makePayment}(T, \hat{m}_2^{(\mathcal{C})}) \rightarrow \hat{m}_2^{(\mathcal{B})}$
   $\mathcal{B}$ takes the following steps.
   (a) at time t_4, decrypts the content of $\hat{m}_2^{(\mathcal{C})}$, i.e., $m_2^{(\mathcal{C})} = \texttt{Dec}(\bar{k}_1, \hat{m}_2^{(\mathcal{C})})$.
   (b) at time t_5, checks the content of $m_2^{(\mathcal{C})}$. If $m_2^{(\mathcal{C})}$ is non-empty, i.e., $m_2^{(\mathcal{C})} = in_f$, it checks if the payee's detail in in_f has already been checked and the payment's amount does not exceed the customer's credit. If the checks pass, it runs the off-chain payment algorithm, $\texttt{pay}(in_f)$. In this case, it sets $m_2^{(\mathcal{B})}$="paid". Otherwise (i.e., if $m_2^{(\mathcal{C})} = \phi$ or neither checks pass), it sets $m_2^{(\mathcal{B})} = \phi$. It sends to $\mathcal{S}$ the encryption of $m_2^{(\mathcal{B})}$, i.e., $\hat{m}_2^{(\mathcal{B})} = \texttt{Enc}(\bar{k}_1, m_2^{(\mathcal{B})})$.

9. *Generating Complaint*. $\texttt{genComplaint}(\hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}, T, pk, aux_f) \rightarrow (\hat{z}, \hat{\ddot{\pi}})$
   $\mathcal{C}$ takes the following steps.
   (a) decrypts $\hat{m}_1^{(\mathcal{B})}$ and $\hat{m}_2^{(\mathcal{B})}$; this results in $m_1^{(\mathcal{B})}$ and $m_2^{(\mathcal{B})}$ respectively. Depending on the content of the decrypted values, it sets its complaint's parameters $z := (z_1, z_2, z_3)$ as follows.
      - if $\mathcal{C}$ want to make one of the two below statements, it sets $z_1 =$"challenge message".
         (i) the "pass" message (in $m_1^{(\mathcal{B})}$) should have been a warning.
         (ii) $\mathcal{B}$ has not provided any message (i.e., neither pass nor warning) and if $\mathcal{B}$ provided a warning then the fraud would have been prevented.
      - if $\mathcal{C}$ wants to challenge the effectiveness of the warning (in $m_1^{(\mathcal{B})}$), it sets $z_2 = m||sig||pk_{\mathcal{G}}||$ "challenge warning", where m is an evidence, $sig \in aux_f$ is the evidence's certificate (obtained from the certificate generator $\mathcal{G}$), and $pk_{\mathcal{G}} \in pk$.
      - if $\mathcal{C}$ wants to complain about the inconsistency of the payment (in $m_2^{(\mathcal{B})}$), then it sets $z_3 =$ "challenge payment". Otherwise, it sets $z_3 = \phi$.
   (b) at time t_6, sends to $\mathcal{S}$ the following values:
      - the encryption of complaint z, i.e., $\hat{z} = \texttt{Enc}(\bar{k}_1, z)$.
      - the encryption of $\ddot{\pi} := (\ddot{\pi}_1, \ddot{\pi}_2)$, i.e., $\hat{\ddot{\pi}} = \tilde{\texttt{Enc}}(pk_{\mathcal{D}}, \ddot{\pi})$. Note, $\ddot{\pi}$ contains the openings of the private statements' commitments (i.e., g_1, g_2), and is encrypted under each $\mathcal{D}_j$'s public key.

10. *Verifying Complaint*. $\texttt{verComplaint}(\hat{z}, \hat{\ddot{\pi}}, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, j, sk_{\mathcal{D}}, aux, pp) \rightarrow \hat{\boldsymbol{w}}_j$
    Every Arbiter, $\mathcal{D}_j \in \{\mathcal{D}_1, ..., \mathcal{D}_n\}$, takes the following steps.
    (a) at time t_7, decrypts $\hat{\ddot{\pi}}$, i.e., $\ddot{\pi} = \tilde{\texttt{Dec}}(sk_{\mathcal{D}}, \hat{\ddot{\pi}})$.
    (b) checks the validity of $(\ddot{\pi}_1, \ddot{\pi}_2)$ in $\ddot{\pi}$ by locally running the SAP's verification, i.e., $\texttt{SAP.verify}(.)$, for each $\ddot{\pi}_i$. It returns s. If $s = 0$, it halts. If $s = 1$ for both $\ddot{\pi}_1$ and $\ddot{\pi}_2$, it proceeds to the next step.
    (c) decrypts $\hat{\boldsymbol{m}} = [\hat{m}_1^{(\mathcal{C})}, \hat{m}_2^{(\mathcal{C})}, \hat{m}_1^{(\mathcal{B})}, \hat{m}_2^{(\mathcal{B})}]$ using $\texttt{Dec}(\bar{k}_1, .)$, where $\bar{k}_1 \in \ddot{\pi}_1$. Let $[m_1^{(\mathcal{C})}, m_2^{(\mathcal{C})}, m_1^{(\mathcal{B})}, m_2^{(\mathcal{B})}]$ be the result.
    (d) checks whether $\mathcal{C}$ made an update request to its payee's list. To do so, it checks if $m_1^{(\mathcal{C})}$ is non-empty and (its encryption) was registered by $\mathcal{C}$ in $\mathcal{S}$. Also, it checks whether $\mathcal{C}$ made a payment request, by checking if $m_2^{(\mathcal{C})}$ is non-empty and (its encryption) was registered by $\mathcal{C}$ in $\mathcal{S}$ at time t_3. If either check fails, it halts.
    (e) decrypts $\hat{z}$ and $\hat{\boldsymbol{l}}$ using $\texttt{Dec}(\bar{k}_1, .)$, where $\bar{k}_1 \in \ddot{\pi}_1$. Let $z := (z_1, z_2, z_3)$ and $\boldsymbol{l}$ be the result.
    (f) sets its verdicts according to the content of $z := (z_1, z_2, z_3)$, as follows.
       - if "challenge message" $\notin z_1$, it sets $w_{1,j} = 0$. Otherwise, it runs $\texttt{verStat}(add_{\mathcal{S}}, m_1^{(\mathcal{B})}, \boldsymbol{l}, \Delta, aux) \rightarrow w_{1,j}$, to determine if a warning (in $m_1^{(\mathcal{B})}$) should have been given (instead of the "pass" or no message).
       - if "challenge warning" $\notin z_2$, it sets $w_{2,j} = w_{3,j} = 0$. Otherwise, it runs $\texttt{checkWarning}(add_{\mathcal{S}}, z_2, m_1^{(\mathcal{B})}, aux') \rightarrow (w_{2,j}, w_{3,j})$, to determine the effectiveness of the warning (in $m_1^{(\mathcal{B})}$).
       - if "challenge payment" $\in z_3$, it checks whether the payment has been made. If the check passes, it sets $w_{4,j} = 1$. If the check fails, it sets $w_{4,j} = 0$. If "challenge payment" $\notin z_3$, it checks if "paid" is in $m_2^{(\mathcal{C})}$. If the check passes, it sets $w_{4,j} = 1$. Otherwise, it sets $w_{4,j} = 0$.
    (g) encodes its verdicts $(w_{1,j}, w_{2,j}, w_{3,j}, w_{4,j})$ as follows.
       i. locally maintains a counter, $o_{adr_{\mathcal{C}}}$, for each $\mathcal{C}$. It sets its initial value to 0.
       ii. calls $\texttt{PVE}(.)$ to encode each verdict. In particular, it performs as follows. $\forall i, 1 \leq i \leq 4:$
          - calls $\texttt{PVE}(\bar{k}_0, adr_{\mathcal{C}}, w_{i,j}, o_{adr_{\mathcal{C}}}, n, j) \rightarrow \bar{w}_{i,j}$
          - $o_{adr_{\mathcal{C}}} = o_{adr_{\mathcal{C}}} + 1$.

          By the end of this step, a vector $\bar{\boldsymbol{w}}_j$ of four encoded verdicts is computed, i.e., $\bar{\boldsymbol{w}}_j = [\bar{w}_{1,j}, ..., \bar{w}_{4,j}]$.
       iii. uses $\bar{k}_2 \in \ddot{\pi}_2$ to further encode/encrypt $\texttt{PVE}(.)$'s outputs as follows. $\hat{\boldsymbol{w}}_j = \texttt{Enc}(\bar{k}_2, \bar{\boldsymbol{w}}_j)$.
    (h) at time t_8, sends to $\mathcal{S}$ the encrypted vector, $\hat{\boldsymbol{w}}_j$.

11. *Resolving Dispute*. $\texttt{resDispute}(T_2, \hat{\boldsymbol{w}}, pp) \rightarrow \boldsymbol{v}$
    $\mathcal{DR}$ takes the below steps at time t_9, when it is invoked by $\mathcal{C}$ or $\mathcal{S}$ which sends $\ddot{\pi}_2 \in T_2$ to it.
    (a) checks the validity of $\ddot{\pi}_2$ by locally running the SAP's verification, i.e., $\texttt{SAP.verify}(.)$, that returns s. If $s = 0$, it halts. Otherwise, it proceeds to the next step.

(b) computes the final verdicts, as below.

i. uses $\bar{k}_2 \in \ddot{\pi}_2$ to decrypt the arbiters' encoded verdicts, as follows. $\forall j, 1 \leq j \leq n$: $\bar{\boldsymbol{w}}_j = \texttt{Dec}(\bar{k}_2, \hat{\boldsymbol{w}}_j)$, where $\hat{\boldsymbol{w}}_j \in \hat{\boldsymbol{w}}$.

ii. constructs four vectors, $[\boldsymbol{u}_1, ..., \boldsymbol{u}_4]$, and sets each vector $\boldsymbol{u}_i$ as follows. $\forall i, 1 \leq i \leq 4$: $\boldsymbol{u}_i = [\bar{w}_{i,1}, ..., \bar{w}_{i,n}]$, where $\bar{w}_{i,j} \in \bar{\boldsymbol{w}}_j$.

iii. calls $\texttt{FVD}(.)$ to extract each final verdict, as follows. $\forall i, 1 \leq i \leq 4$: calls $\texttt{FVD}(n, \boldsymbol{u}_i) \rightarrow v_i$.

(c) outputs $\boldsymbol{v} = [v_1, ..., v_4]$.

Customer $\mathcal{C}$ must be reimbursed if the final verdict is that (i) the "pass" message or missing message should have been a warning or (ii) the warning was ineffective and the provided evidence was not invalid, and (iii) the payment has been made. Stated formally, the following relation must hold:

$$\Big(\underbrace{(v_1 = 1)}_{\text{(i)}} \vee \underbrace{(v_2 = 1 \ \wedge \ v_3 = 1)}_{\text{(ii)}}\Big) \wedge \Big(\underbrace{v_4 = 1}_{\text{(iii)}}\Big).$$

Note that in the above PwDR protocol, even $\mathcal{C}$ and $\mathcal{B}$ that know the decryption secret keys, $(\bar{k}_1, \bar{k}_2)$, cannot link a certain verdict to an arbiter, for two main reasons; namely, (a) they do not know the masking random values used by arbiters to mask each verdict and (b) the final verdicts $(v_1, ..., v_4)$ reveal nothing about the number of 1 or 0 verdicts, except when all arbiters vote 0. We highlight that we used PVE and FVD in the PwDR protocol only because they are highly efficient. However, it is easy to replace them with GPVE and GFVD, e.g., when the required threshold is greater than one.

Theorem 3. *The above PwDR scheme is secure, with regard to definition 7, if the digital signature is existentially unforgeable under chosen message attacks, the blockchain, SAP, and pseudorandom function* $\texttt{PRF}(.)$ *are secure, and the encryption schemes are semantically secure.*

5 Security Analysis of the PwDR Protocol

To prove the main theorem (i.e., Theorem 3), we show that the PwDR scheme satisfies all security properties defined in Section 3. We first prove that it meets security against a malicious victim.

Lemma 1. *If the digital signature is existentially unforgeable under chosen message attacks, and the SAP and blockchain are secure, then the PwDR scheme is secure against a malicious victim, with regard to Definition 4.*

Proof. First, we focus on event I : $\Big((m_1^{(\mathcal{B})} = \mathit{warning}) \wedge (\sum\limits_{j=1}^{n} w_{1,j} \geq e)\Big)$ which considers the case where $\mathcal{B}$ has provided a warning message but $\mathcal{C}$ manages to convince at least threshold arbiters to set their verdicts to 1, that ultimately results in $\sum\limits_{j=1}^{n} w_{1,j} \geq e$. We argue that the adversary's success probability in this event is negligible in the security parameter. In particular, due to the security of SAP, $\mathcal{C}$ cannot convince an arbiter to accept a different decryption key, e.g., $k' \in \ddot{\pi}'$, that will be used to decrypt $\mathcal{B}$'s encrypted message $\hat{m}_1^{(\mathcal{B})}$, other than what was agreed between $\mathcal{C}$ and $\mathcal{B}$ in the initiation phase, i.e., $\bar{k}_1 \in \ddot{\pi}_1$. To be more precise, it cannot persuade an arbiter to accept a statement $\ddot{\pi}'$, where $\ddot{\pi}' \neq \ddot{\pi}_1$ except with a negligible probability, $\mu(\lambda)$. This ensures that honest $\mathcal{B}$'s original message (and accordingly the warning) is accessed by every arbiter with a high probability. Next, we consider event II : $\Big((\sum\limits_{j=1}^{n} w_{1,j} < e) \wedge (v_1 = 1)\Big)$ that captures the case where only less than threshold arbiters approved that the pass message was given incorrectly or the missing message could prevent the APP fraud, but the final verdict that $\mathcal{DR}$ extracts implies that at least threshold arbiters approved that. We argue that the probability that this event occurs is negligible in the security parameter. Specifically, due to the security of the SAP, $\mathcal{C}$ cannot persuade (a) an arbiter to accept a different encryption key and (b) $\mathcal{DR}$ to accept a different decryption key other than what was agreed between $\mathcal{C}$ and $\mathcal{B}$ in the initiation phase. Specifically, it cannot persuade them to accept a statement $\ddot{\pi}'$, where $\ddot{\pi}' \neq \ddot{\pi}_2$ except with a negligible probability, $\mu(\lambda)$.

Now, we move on to event III : $\Big((\texttt{checkWarning}(m_1^{(\mathcal{B})}) = 1) \wedge (\sum\limits_{j=1}^{n} w_{2,j} \geq e)\Big)$. It captures the case where $\mathcal{B}$ has provided an effective warning message but $\mathcal{C}$ manages to make at threshold arbiters set their verdicts to 1, that ultimately results in $\sum\limits_{j=1}^{n} w_{2,j} \geq e$. The same argument provided to event I is applicable to this even too. Briefly, due to the security of SAP, $\mathcal{C}$ cannot persuade an arbiter to accept a different decryption key other than what was agreed between $\mathcal{C}$ and $\mathcal{B}$ in the initiation phase. Therefore, all arbiters will receive the original message of $\mathcal{B}$, including the effective warning message, except a negligible probability, $\mu(\lambda)$. Now, we consider event IV : $\Big((\sum\limits_{j=1}^{n} w_{2,j} < e) \wedge (v_2 = 1)\Big)$, which captures the case where at least threshold arbiters approved that the warning message was effective but the final verdict that $\mathcal{DR}$ extracts implies that they approved the opposite. The security argument of event II applies to this event as well. In short, due to the security of the SAP, $\mathcal{C}$ cannot persuade an arbiter to accept a different encryption key, and cannot convince $\mathcal{DR}$ to accept a different decryption key other than what was initially agreed between $\mathcal{C}$ and $\mathcal{B}$, except a negligible probability, $\mu(\lambda)$. Now, we analyse event V : $\Big(u \notin Q \wedge \texttt{Sig.ver}(pk, u, sig) = 1\Big)$. This even captures the case where the malicious victim comes up with a valid signature/certificate on a message that has never been queried to the signing oracle. Nevertheless, due to the existential unforgeability of the digital signature scheme, the probability that such an event occurs is negligible, $\mu(\lambda)$. Next, we focus on event VI : $\Big((\sum\limits_{j=1}^{n} w_{3,j} < e) \wedge (v_3 = 1)\Big)$ that considers the case where less than threshold arbiters indicated that the signature (in $\mathcal{C}$'s complaint) is valid, but the final verdict that $\mathcal{DR}$ extracts implies that at least threshold arbiters approved the signature. This means the adversary has managed to switch the verdicts of those arbiters which voted 0 to 1. However, the probability that this even occurs is negligible as well. Because, due to the SAP's security, $\mathcal{C}$ cannot convince an arbiter and $\mathcal{DR}$ to accept different encryption and decryption keys other than what was initially agreed between $\mathcal{C}$ and $\mathcal{B}$, except a negligible probability, $\mu(\lambda)$. Therefore, with a negligible probability the adversary can switch a verdict for 0 to the verdict for 1.

Moreover, a malicious $\mathcal{C}$ cannot frame an honest $\mathcal{B}$ for providing an invalid message by manipulating the smart contract's content, e.g., by replacing an effective warning with an ineffective one in $\mathcal{S}$, or excluding a warning from $\mathcal{S}$. In particular, to do that, it has to either forge the honest party's signature, so it can send an invalid message on its behalf, or fork the blockchain so the chain comprising a valid message is discarded. In the former case, the adversary's probability of success is negligible as long as the signature is secure. The adversary has the same success probability in the latter case, because it has to generate a long enough chain that excludes the valid message which has a negligible success probability, under the assumption that the hash power of the adversary is lower than those of honest miners and due to the blockchain's liveness property an honestly generated transaction will eventually appear on an honest miner's chain [7]. $\square$

Now, we first present a lemma formally stating that the PwDR scheme is secure against a malicious bank and then prove this lemma.

Lemma 2. *If the SAP and blockchain are secure, and the correctness of verdict encoding-decoding protocols (i.e., PVE and FVD) holds, then the PwDR scheme is secure against a malicious bank, with regard to Definition 5.*

Proof. We first focus on event I : $\Big((\sum\limits_{j=1}^{n} w_{1,j} \geq e) \wedge (v_1 = 0)\Big)$ which captures the case where $\mathcal{DR}$ is convinced that the pass message was correctly given or an important warning message was not missing, despite at least threshold arbiters do not believe so. We argue that the probability that this event takes place is negligible in the security parameter. Because, $\mathcal{B}$ cannot persuade $\mathcal{DR}$ to accept a different decryption key, e.g., $k' \in \ddot{\pi}'$, other than what was agreed between $\mathcal{C}$ and $\mathcal{B}$ in the initiation phase, i.e., $\bar{k}_2 \in \ddot{\pi}_2$, except with a negligible probability. Specifically, it cannot persuade $\mathcal{DR}$ to accept a statement $\ddot{\pi}'$, where $\ddot{\pi}' \neq \ddot{\pi}_2$ except with probability $\mu(\lambda)$. Furthermore, as discussed in Section 4.4, due to the correctness of the verdict encoding-decoding protocols, i.e., PVE and FVD, the probability that multiple representations of verdict 1 cancel out each other is negligible too, $\frac{1}{2^{\lambda}}$. Thus, event I occurs only with a negligible probability, $\mu(\lambda)$. To assert that events II : $\Big((\sum_{j=1}^{n} w_{2,j} \geq e) \wedge (v_2 = 0)\Big)$, III : $\Big((\sum_{j=1}^{n} w_{3,j} \geq e) \wedge (v_3 = 0)\Big)$, and IV : $\Big((\sum_{j=1}^{n} w_{4,j} \geq e) \wedge (v_4 = 0)\Big)$ occur only with a negligible probability, we can directly use the above argument provided for event I. To avoid repetition, we do not restate them in this proof. Moreover, a malicious $\mathcal{B}$ cannot frame an honest $\mathcal{C}$ for providing an invalid message by manipulating the smart contract's content, e.g., by replacing its valid signature with an invalid one or sending a message on its behalf, due to the security of the blockchain. □

Next, we prove the PwDR protocol's privacy. As before, we first formally state the related lemma and then prove it.

Lemma 3. *If the encryption schemes are semantically secure, and the SAP and encoding-decoding schemes (i.e., PVE and FVD) are secure, then the PwDR scheme is privacy-preserving with regard to Definition 6.*

Proof. We first focus on property 1, i.e., the privacy of the parties' messages from the public. Due to the privacy-preserving property of the SAP, that relies on the hiding property of the commitment scheme, given the public commitments, $g := (g_1, g_2)$, the adversary learns no information about the committed values, $(\bar{k}_1, \bar{k}_2)$, except with a negligible probability, $\mu(\lambda)$. Thus, it cannot find the encryption-decryption keys used to generate ciphertext $\hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, \hat{z}$, and $\hat{\boldsymbol{w}}$. Moreover, due to the semantically security of the symmetric key and asymmetric key encryption schemes, given ciphertext $(\hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, \hat{z}, \hat{\ddot{\pi}}, \hat{\boldsymbol{w}})$ the adversary cannot learn anything about the related plaintext, except with a negligible probability, $\mu(\lambda)$. Thus, in experiment $\mathsf{Exp}_3^{\mathcal{A}_1}$, adversary $\mathcal{A}_1$ cannot tell the value of $\gamma \in \{0, 1\}$ significantly better than just guessing it, i.e., its success probability is at most $\frac{1}{2} + \mu(\lambda)$. Now we move on to property 2, i.e., the privacy of each verdict from $\mathcal{DR}$. Due to the privacy-preserving property of the SAP, given $g_1 \in g$, a corrupt $\mathcal{DR}$ cannot learn $\bar{k}_1$. So, it cannot find the encryption-decryption key used to generate ciphertext $\hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}$, and $\hat{z}$. Also, public parameters (pk, pp) and token T_2 are independent of $\mathcal{C}$'s and $\mathcal{B}$'s exchanged messages (e.g., payment requests or warning messages) and $\mathcal{D}_j$s verdicts. Furthermore, due to the semantical security of the symmetric key and asymmetric key encryption schemes, given ciphertext $(\hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, \hat{z}, \hat{\ddot{\pi}})$ the adversary cannot learn anything about the related plaintext, except with a negligible probability, $\mu(\lambda)$. Also, due to the security of the PVE and FVD protocols, the adversary cannot link a verdict to a specific arbiter with a probability significantly better than the maximum probability, Pr', that an arbiter sets its verdict to a certain value, i.e., its success probability is at most $Pr' + \mu(\lambda)$, even if it is given the final verdicts, except when all arbiters' verdicts are 0. We conclude that, excluding the case where the all verdicts are 0, given $(T_2, pk, pp, g, \hat{\boldsymbol{m}}, \hat{\boldsymbol{l}}, \hat{z}, \hat{\ddot{\pi}}, \hat{\boldsymbol{w}}, \boldsymbol{v})$, adversary $\mathcal{A}_3$'s success probability in experiment $\mathsf{Exp}_4^{\mathcal{A}_2}$ to link a verdict to an arbiter is at most $Pr' + \mu(\lambda)$.

□

Theorem 4. *The PwDR is secure according to Definition 7.*

Proof. Due to Lemma 1, the PwDR scheme is secure against a malicious victim. Also, due to lemmas 2 and 3 it is secure against a malicious bank and is privacy preserving, respectively. Thus, it satisfies all the properties of Definition 7, meaning that the PwDR is indeed secure according to this definition. □

6 Evaluation

In this section, we provide the PwDR protocol's asymptotic computation and communication cost analysis. Table 1 summarises the result.

6.1 Asymptotic Cost Analysis

Computation Cost. In this section, we analyse the computation cost of the protocol. We first analyse $\mathcal{C}$'s cost. In Phase 3, $\mathcal{C}$ invokes a hash function twice to check the correctness of the private statements'

Table 1: The PwDR protocol's asymptotic cost. In the table, n is the number of arbiters and e is the threshold.

Party	**Setting**		**Computation Cost**	**Communication Cost**
	$e = 1$	$e > 1$		
Customer	✓	✓	$O(1)$	$O(1)$
Bank	✓	✓	$O(1)$	$O(1)$
Arbiter $\mathcal{D}_1, ..., \mathcal{D}_{n-1}$	✓	✓	$O(1)$	$O(1)$
Arbiter $\mathcal{D}_n$	✓		$O(n)$	$O(1)$
		✓	$O(\sum\limits_{i=e}^{n} \frac{n!}{i!(n-i)!})$	$O(\sum\limits_{i=e}^{n} \frac{n!}{i!(n-i)!})$
Dispute resolver	✓	✓	$O(n)$	$O(1)$

parameters. In Phase 4, it invokes the symmetric encryption once to encrypt its update request. In Phase 7, it invokes the symmetric encryption twice to decrypt $\mathcal{B}$'s warning message and to encrypt its payment request. In Phase 9, it runs the symmetric encryption three times to decrypt $\mathcal{B}$'s warning and payment messages and to encrypt its complaint. In the same phase, it invokes the asymmetric encryption once to encrypt the private statements' opening. Therefore, $\mathcal{C}$'s complexity is $O(1)$. Next, we analyse $\mathcal{B}$'s cost. In Phase 2, it invokes the hash function twice to commit to two statements. In Phase 6, it calls the symmetric key encryption once to encrypt its outgoing warning message. In Phase 8, it also invokes the symmetric key encryption once to encrypt the outgoing payment message. Thus, $\mathcal{B}$'s complexity is $O(1)$ too. Next, we analyse each arbiter's cost. In Phase 10, each $\mathcal{D}_j$ invokes the asymmetric key encryption once to decrypt the private statements' openings. It also invokes the hash function twice to verify the openings. It invokes the symmetric key encryption six times to decrypt $\mathcal{C}$'s and $\mathcal{B}$'s messages that were posted on $\mathcal{S}$ (this includes $\mathcal{C}$'s complaint). Recall, in the same phase, each arbiter encodes its verdict using a verdict encoding protocol. Now, we evaluate the verdict encoding complexity of each arbiter for two cases: (a) $e = 1$ and (b) $e \in (1, n]$. Note, in the former case the PVE is invoked while in the latter GPVE is invoked. In case (a), every arbiter $\mathcal{D}_j$, except $\mathcal{D}_n$, invokes the pseudorandom function once to encode its verdict. However, arbiter $\mathcal{D}_n$ invokes the pseudorandom function $n - 1$ times and XORs the function's outputs with each other. Thus, in case (a), arbiter $\mathcal{D}_n$'s complexity is $O(n)$ while the ret of arbiters' complexity is $O(1)$. In case (b), every arbiter $\mathcal{D}_j$, except $\mathcal{D}_n$, invokes the pseudorandom function twice to encode its verdict. But, arbiter $\mathcal{D}_n$ invokes the pseudorandom function $n - 1$ times and XORs the function's outputs with each other. It also invokes the pseudorandom function n times to generate all arbiters' representations of verdict 1. It computes all $y = \sum\limits_{i=e}^{n} \frac{n!}{i!(n-i)!}$ combinations of the representations that meet the threshold which involves $O(y)$ XORs. It also inserts y elements into a Bloom filter that requires $O(y)$ hash function evaluations. So, in case (b), arbiter $\mathcal{D}_n$'s complexity is $O(y)$ while the ret of the arbiters' complexity is $O(1)$. To conclude, in Phase 10, arbiter $\mathcal{D}_n$'s complexity is either $O(n)$ or $O(y)$, while the rest of arbiters' complexity is $O(1)$. Now, we analyse $\mathcal{DR}$'s cost in Phase 11. It invokes the hash function once to check the private statement's correctness. It also performs $O(n)$ symmetric key decryption to decrypt arbiters' encoded verdicts. Now, we evaluate the verdict decoding complexity of $\mathcal{DR}$ for two cases: (a) $e = 1$ and (b) $e \in (1, n]$. In the former case (in which FVD is invoked), it performs $O(n)$ XOR to combine all verdicts. Its complexity is also $O(n)$ in the latter case (in which GFVD is invoked), with a small difference that it also invokes the Bloom filter's hash functions, to make a membership query to the Bloom filter. Thus, $\mathcal{DR}$'s complexity is $O(n)$.

Communication Cost. In this section, we analyse the communication cost of the PwDR protocol. Briefly, $\mathcal{C}$'s complexity is $O(1)$ as in total it sends only six messages to other parties. Similarly, $\mathcal{B}$'s complexity is $O(1)$ as its total number of outgoing messages is only nine. Each arbiter $\mathcal{D}_j$ sends only four messages to the smart contract, so its complexity is $O(1)$. However, if GFVD is invoked, then arbiter $\mathcal{D}_n$ needs to send also a Bloom filter that costs it $O(y)$. Moreover, $\mathcal{DR}$'s complexity is $O(1)$, as its outgoing messages include only four binary values.

6.2 Concrete Performance Analysis

As we saw in the previous section, the customer's and bank's complexity is very low; however, one of the arbiters, i.e., arbiter $\mathcal{D}_n$, and the dispute resolver have non-constant complexities. These non-constant costs were imposed by the verdict inducing-decoding protocols. Therefore, to study these parties' runtime in the PwDR, we implemented both variants of the verdict encoding-decoding protocols (that were presented in Section 4.4). They were implemented in C++. The source of the variant 1 and 2 is available in [1] and [2] respectively. To conduct the experiment, we used a MacBook Pro laptop with quad-core Intel core $i5$, 2 GHz CPU and 16 GB RAM. We ran the experiment on average 10 times. The prototype implementation utilises the "Cryptopp" library[2] for cryptographic primitives, the "GMP" library[3] for arbitrary precision arithmetics, and the "Bloom Filter" library [4]. In the experiment, we set the false-positive rate in a Bloom filter to 2^{-40} and the finite field size to 128 bits. Table 2 provides the runtime of the three types of parties for various numbers of arbiters in two cases; namely, when the threshold is 1 and when it is greater than 1. In the former case, we used the PVE and FVD protocols. In the latter case, we used the GPVE and GFVD ones.

Table 2: PwDR runtime (in ms). Broken-down by parties. In the table, n is the number of arbiters and e is the threshold.

Party	$n=6$		$n=8$		$n=10$		$n=12$	
	$e=1$	$e=4$	$e=1$	$e=5$	$e=1$	$e=6$	$e=1$	$e=7$
Arbiter $\mathcal{D}_1,...,\mathcal{D}_{n-1}$	0.014	0.030	0.014	0.030	0.014	0.030	0.014	0.030
Arbiter $\mathcal{D}_n$	0.019	0.220	0.033	0.661	0.035	2.87	0.052	10.15
Dispute resolver $\mathcal{DR}$	0.001	0.015	0.001	0.016	0.001	0.069	0.003	0.09

As the above table depicts, the runtime of $\mathcal{D}_n$ and $\mathcal{DR}$ increases gradually when the number of arbiters grows, while $\mathcal{D}_1,...\mathcal{D}_{n-1}$ having a constant runtime. Nevertheless, the overall costs is very low. In particular, the highest runtime is only about 10 milliseconds which belongs to $\mathcal{D}_n$ when $n=12$ and $e=7$. It is also evident that the parties' runtime in the PVE and FVD protocols is much lower than their runtime in the GPVE and GFVD ones. To compare the parties' runtime, we also fixed the threshold to 6 and ran the experiment for different values of n. Figure 5 summarises the result. As this figure indicates, the runtime of $\mathcal{D}_n$ and $\mathcal{DR}$ almost linearly grows when the number of arbiters increases; moreover, among these three types of parties, $\mathcal{D}_n$ has the highest runtime.

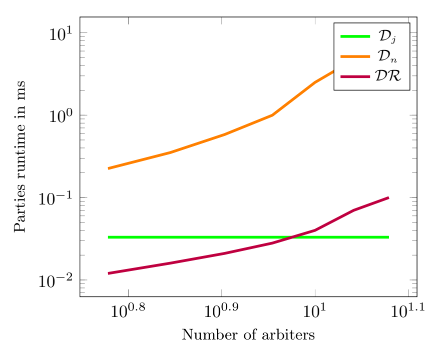

Fig. 5: Parties' runtime in the PwDR.

[2] https://www.cryptopp.com

[3] https://gmplib.org

[4] http://www.partow.net/programming/bloomfilter/index.html

References

1. Abadi, A.: Variant 1: Highly efficient verdict encoding-decoding protocol (2021), https://github.com/AydinAbadi/PwDR/blob/main/PwDR-code/encoding-decoding.cpp
2. Abadi, A.: Variant 2: Generic verdict encoding-decoding protocol (2021), https://github.com/AydinAbadi/PwDR/blob/main/PwDR-code/generic-encoding-decoding.cpp
3. Abadi, A., Murdoch, S.J., Zacharias, T.: Recurring contingent payment for proofs of retrievability. Cryptology ePrint Archive, Report 2021/1145 (2021), https://ia.cr/2021/1145
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. (1970)
5. Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M.H.M., Tang, Y.: On the false-positive rate of bloom filters. Inf. Process. Lett. (2008)
6. Confirmation of Payee Team: Confirmation of payee: Response to consultation cp20/1 and decision on varying specific direction 10 (2020), https://www.psr.org.uk/media/qrbc03jm/psr-ps20-1-variation-of-specific-direction-10-february-2020.pdf
7. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: EUROCRYPT (2015)
8. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (2014), https://www.crcpress.com/Introduction-to-Modern-Cryptography-Second-Edition/Katz-Lindell/p/book/9781466570269
9. Küsters, R., Liedtke, J., Müller, J., Rausch, D., Vogt, A.: Ordinos: A verifiable tally-hiding e-voting system. In: IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020. IEEE (2020)
10. Lending Standards Board: Contingent reimbursement model code for authorised push payment scams (2021), https://www.lendingstandardsboard.org.uk/wp-content/uploads/2021/04/CRM-Code-LSB-Final-April-2021.pdf
11. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2019)
12. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO (1991)
13. Schneier, B.: Applied cryptography - protocols, algorithms, and source code in C, 2nd Edition. Wiley (1996)
14. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014)

A Bloom Filter

In this work, we use Bloom filters to let parties (in Feather) identify real set elements from errors. A Bloom filter [4] is a compact data structure for probabilistic efficient elements' membership checking. A Bloom filter is an array of m bits that are initially all set to zero. It represents n elements. A Bloom filter comes along with k independent hash functions. To insert an element, all the hash values of the element are computed and their corresponding bits in the filter are set to 1. To check an element's membership, all its hash values are re-computed and checked whether all are set to one in the filter. If all the corresponding bits are one, then the element is probably in the filter; otherwise, it is not. In Bloom filters false positives are possible, i.e. it is possible that an element is not in the set, but the membership query shows that it is. According to [5], the upper bound of the false positive probability is: $q = p^k(1 + O(\frac{k}{p}\sqrt{\frac{\ln m - k \ln p}{m}}))$, where p is the probability that a particular bit in the filter is set to 1 and calculated as: $p = 1 - (1 - \frac{1}{m})^{kn}$. The efficiency of a Bloom filter depends on m and k. The lower bound of m is $n \log_2 e \cdot \log_2 \frac{1}{q}$, where e is the base of natural logarithms, while the optimal number of hash functions is $\log_2 \frac{1}{q}$, when m is optimal. In this paper, we only use optimal k and m. In practice, we would like to have a predefined acceptable upper bound on false positive probability, e.g. $q = 2^{-40}$. Thus, given q and n, we can determine the rest of the parameters.

B Variant 1 Encoding-Decoding Protocol's Main Theorem and Proof

Theorem 1. *Let set $S = \{s_1, ..., s_m\}$ be the union of two disjoint subsets S' and S'', where S' contains non-zero random values pick uniformly from a finite field $\mathbb{F}_p$, S'' contains zeros, $|S'| \geq c' = 1$, $|S''| \geq c'' = 0$, and pair (c', c'') is public information. Then, $r = \bigoplus_{i=1}^{m} s_i$ reveals nothing beyond the public information.*

Proof. Let s_1 and s, be two random values picked uniformly at random from $\mathbb{F}_p$. Let $\bar{s} = s_1 \oplus \underbrace{0 \oplus ... \oplus 0}_{|S''|}$. Since $\bar{s} = s_1$, two values $\bar{s}$ and s have identical distribution. Thus, $\bar{s}$ reveals nothing in this case. Next, let $\tilde{s} = \underbrace{s_1 \oplus s_2 \oplus ... \oplus s_j}_{|S'|}$, where $s_i \in S'$. Since each s_i is a uniformly random value, the XOR of them is a uniformly random value too. That means values $\tilde{s}$ and s have identical distribution. Thus, $\tilde{s}$ reveals nothing in this case as well. Also, it is not hard to see that the combination of the above two cases reveals nothing too, i.e., $\bar{s} \oplus \tilde{s}$ and s have identical distribution. □

C Variant 2 Encoding-Decoding Protocol's Main Theorem and Proof

Theorem 2. *Let set $S = \{s_1, ..., s_m\}$ be a set of random values picked uniformly from $\mathbb{F}_p$, where the cardinality of S is public information. Let* `BF` *be a Bloom filter encoding all elements of S. Then,* `BF` *reveals nothing about any element of S, beyond the public information, except with a negligible probability in the security parameter, i.e., with a probability at most $\frac{|S|}{2^\lambda}$.*

Proof. First, we consider the simplest case where only a single element of S is encoded in `BF`. In this case, due to the pre-image resistance of the Bloom filter's hash functions and the fact that the set's element was picked uniformly at random from $\mathbb{F}_p$, the probability that `BF` reveals anything about the original element is at most $\frac{1}{2^\lambda}$. Now, we move on to the case where all elements of S are encoded in `BF`. In this case, the probability that `BF` reveals anything about at least an element of the set is $\frac{|S|}{2^\lambda}$, due to the pre-image resistance of the hash functions, the fact that all elements were selected uniformly at random from the finite field, and the union bound. Nevertheless, when a `BF`'s size is set appropriately to avoid false-positive without wasting storage, this reveals the number of elements encoded in it, which is public information. Thus, the only information `BF` reveals is the public one. □

D Generic Verdict Encoding-Decoding Protocol

Figures 6 and 7 present the generic verdict encoding-decoding protocols (i.e., GPVE and GFVD), that let a semi-honest third party $\mathcal{I}$ find out if at least e arbiters voted 1, where e can be any integer in the range $[1, n]$.

E Further Discussion on the Verdict Encoding-decoding Protocol

Recall that each variant of our verdict encoding-decoding protocol is a voting mechanism. It lets a third party, $\mathcal{I}$, find out if threshold arbiters voted 1, while (i) generating unlinkable verdicts, (ii) not requiring arbiters to interact with each other for each customer, (iii) hiding the number of 0 or 1 verdicts from $\mathcal{I}$, and (iv) being efficient. Therefore, it is natural to ask:

Is there any e-voting protocol that can simultaneously satisfy all the above requirements?

The short answer is no. Recently, a provably secure e-voting protocol that can hide the number of 1 and 0 votes has been proposed by Kusters *et al.* [9]. Although this scheme can satisfy the above security requirements, it imposes a high computation cost, as it involves computationally expensive primitives such as zero-knowledge proofs, threshold public-key encryption scheme, and generic multi-party computation. In contrast, our verdict encoding-decoding protocols rely on much more lightweight operations such as XOR and hash function evaluations. We also highlight that our verdict encoding-decoding protocols are in a different setting than the one in which most of the e-voting protocols are. Because the former protocols are in the setting where there exists a small number of arbiters (or voters) which are trusted and can interact with each other once; whereas, the latter (e-voting) protocols are in a more generic setting where there is a large number of voters, some of which might be malicious, and they are not required to interact with each other.

$\texttt{GPVE}(\bar{k}_0, \text{ID}, w_j, o, e, n, j) \rightarrow (\bar{w}_j, \texttt{BF})$

– *Input.* $\bar{k}_0$: a key of pseudorandom function $\texttt{PRF}(.)$, ID: a unique identifier, w_j: a verdict, o: an offset, e: a threshold, n: the total number of arbiters, and j: an arbiter's index.
– *Output.* $\bar{w}_j$: an encoded verdict.

Arbiter $\mathcal{D}_j$ takes the following steps.

1. computes a pseudorandom value, as follows.
   - if $j < n : r_j = \texttt{PRF}(\bar{k}_0, 1||o||j||\text{ID})$.
   - if $j = n : r_j = \bigoplus_{i=1}^{n-1} r_i$.

   Note, the above second step is taken only by $\mathcal{D}_n$.
2. sets a fresh parameter, w'_j, that represents a verdict, as below.
$$w'_j = \begin{cases} 0, & \text{if } w_j = 0 \\ \alpha_j = \texttt{PRF}(\bar{k}_0, 2||o||j||\text{ID}), & \text{if } w_j = 1 \end{cases}$$
3. masks w'_j as follows. $\bar{w}_j = w'_j \oplus r_j$.
4. if $j = n$, computes a Bloom filter that encodes the combinations of verdict representations (i.e., w'_j) for verdict 1. In particular, it takes the following steps.
   - for every integer i in the range $[e, n]$, computes the combinations (without repetition) of i elements from set $\{\alpha_1, ..., \alpha_n\}$. In the case where multiple elements are taken at a time (i.e., $i > 1$), the elements are XORed with each other. Let $W = \{(\alpha_1 \oplus ... \oplus \alpha_e), (\alpha_2 \oplus ... \oplus \alpha_{e+1}), ..., (\alpha_1 \oplus ... \oplus \alpha_n)\}$ be the result.
   - constructs an empty Bloom filter. Then, it inserts all elements of W into this Bloom filter. Let $\texttt{BF}$ be the Bloom filter encoding W's elements.
5. outputs $(\bar{w}_j, \texttt{BF})$.

Fig. 6: Generic Private Verdict Encoding (GPVE) Protocol

Note that each variant of our verdict encoding-decoding protocol requires every arbiter to provide an encoded vote in order for $\mathcal{I}$ to extract the final verdict. To let each variant terminate and $\mathcal{I}$ find out the final verdict in the case where a set of arbiters do not provide their vote, we can integrate the following idea into each variant. We define a manager arbiter, say $\mathcal{D}_n$, which is always responsive and keeps track of missing votes. After the voting time elapses and $\mathcal{D}_n$ realises a certain number of arbiters did not provide their encoded vote, it provides 0 votes on their behalf and masks them using the arbiters' masking values.

$\texttt{GFVD}(n, \bar{\boldsymbol{w}}, \texttt{BF}) \rightarrow v$

- *Input.* n: the total number of arbiters, and $\bar{\boldsymbol{w}} = [\bar{w}_1, ..., \bar{w}_n]$: a vector of all arbiters' encodes verdicts.
- *Output.* v: final verdict.

A third-party $\mathcal{I}$ takes the following steps.

1. combines all arbiters' encoded verdicts, $\bar{w}_j \in \bar{\boldsymbol{w}}$, as follows. $c = \bigoplus_{j=1}^{n} \bar{w}_j$
2. checks if c is in the Bloom filter, $\texttt{BF}$.
3. sets the final verdict v depending on the content of c. Specifically,

$$v = \begin{cases} 0, & \text{if } c = 0 \text{ or } c \notin \texttt{BF} \\ 1, & \text{if } c \in \texttt{BF} \end{cases}$$

4. outputs v.

Fig. 7: Generic Final Verdict Decoding (GFVD) Protocol