Automated Penalization of Data Breaches using Crypto-augmented Smart Contracts

Easwar Vivek Mangipudi[1], Krutarth Rao[1], Jeremy Clark[2], and Aniket Kate[1]

[1] Purdue Universtiy, USA
[2] Concordia University, Canada

Abstract. This work studies the problem of automatically penalizing intentional or unintentional data breach (APDB) by a receiver/custodian receiving confidential data from a sender. We solve this problem by augmenting a blockchain *on-chain* smart contract between the sender and receiver with an *off-chain* cryptographic protocol, such that any significant data breach from the receiver is penalized through a monetary loss. Towards achieving the goal, we develop a natural extension of oblivious transfer called *doubly oblivious transfer* (DOT) which, when combined with robust watermarking and a claim-or-refund blockchain contract provides the necessary framework to realize the APDB protocol in a provably secure manner. In our APDB protocol, a public data breach by the receiver leads to her Bitcoin (or other blockchain) private signing key getting revealed to the sender, which allows him to penalize the receiver by claiming the deposit from the claim-or-refund contract. Interestingly, the protocol also ensures that the malicious sender cannot steal the deposit, even as he knows the original document or releases it in any form. We implement our APDB protocol, develop the required smart contract for Bitcoin and observe our system to be efficient and easy to deploy in practice. We analyze our DOT-based design against partial adversarial leakages and observe it to be robust against even small leakages of data.

1 Introduction

Data breach attacks on cloud hosts are increasing every year [1,2,3,4], the reasons for which vary from compromises of ill-maintained data servers to careless data custodians. Although it has been observed and reported that 90% of these data breaches can be avoided with good security practices on the custodian's infrastructure [5], there is no evident decrease in the number. In these cases, taking legal actions is not only expensive and time consuming but it is also difficult to establish the responsibility in today's geo-politically distributed data flows.

This work aims at raising the bar for the data receivers/custodians by introducing a complementary security mechanism that is inexpensive, automated, and is not restricted by the geo-political boundaries. In particular, our goal is to make the data custodians more accountable through automatically enforceable monetary penalties resulting in immediate loss of funds, and we call the associated contract the *automated penalization of data breach* (APDB) contract. Applicability scenarios for APDB contracts range from industrial data custodianship, leaking privately shared personal data of others on social media and even to non-disclosure agreements between mutually distrusting entities [6].

Example Scenario: Data Custodianship. Data Custodianship refers to the responsibility of safe storage and custody of the data [7]. A serious breach of the storage typically results in criminal litigation against the custodian. APDB can be useful when legal action is undesirable due to the uncertainty of recovering the payment (which increases if the winning party is owed court costs in addition to the actual remedy) [8]. We assume that the data owner/sender and the custodian/receiver agree on an amount of money that will be awarded to the owner should specified documents be demonstrably leaked by the custodian. Towards automatically ensuring that the owner will receive the funds, this amount could take the form of a surety bond that is held in trust by a Bitcoin or other permission-less/permissioned blockchain based cryptocurrency smart contract. Another simple example scenario can be in the case of a media download by users that should not be publicly shared. Here, the users make a deposit which will be forfeited by the media provider upon dishonest sharing of the content.

APDB does not preclude the use of the court system, it simply complements it, or shifts the responsibility of bringing legal action to the entity seeking to recover their bond. Allowing an escalation to court is important because some disclosures are in the public interest (whistle-blowing) [9]. In fact, in certain cases, a third party might pay the value of the bond for the information (newspaper, media, crowdfunding, etc.).

Contributions. In the form of APDB, we formalize the problem of automatically settling intentional or unintentional data breaches with a Bitcoin (or other blockchain) smart contract, eschewing the traditional recourse of costly legal action. Our APDB protocol is a crypto-augmented smart contract system to obtain an arbitrator-free settlement. It consists of four main components: a claim-or-refund smart contract, a robust watermarking scheme, a natural oblivious-transfer extension called Doubly Oblivious Transfer (DOT), and a non-interactive zero knowledge (NIZK) proof for mutually distrusting parties.

In our core protocol, the sender and receiver create a claim-or-refund transaction on Bitcoin [10,11,12] where an amount is deposited that can be spent at any time with a jointly signed transaction, or spent after a period of time by an sender-only signed transaction. The document provided to the receiver has the receiver's signing (private) key embedded in it with a robust binary watermarking scheme that cannot be removed (or retrieved) by anyone except the embedding party. The challenging aspects of the APDB protocol involve arranging for the signing key to be embedded such that (1) the sender does not learn the value of the key at the time of embedding, (2) the receiver does not learn the document contents until the key is embedded, and (3) the sender is convinced the embedded key is the receiver's correct signing key. Within these constraints, to perform the embedding the parties must jointly perform a two-party computation with their respective private inputs. Our novel DOT and committed receiver oblivious transfer ($CROT$) protocols, securely realize this two-party computation to ensure that the sender can retrieve the receiver's embedded key from the document if it leaks (widely enough to reach the sender) and spend the deposited cryptocurrency.

We have implemented the APDB system using the Relic library for the cryptographic primitives, a robust image watermarking scheme and claim-or-refund contract for Bitcoin. Given the prevalence of robust watermarking in the industry [13,14], we find our APDB system to be easy to deploy. Our single-threaded implementation takes on average 1.73 seconds when an 1.3MB image is used as data for the transfer when the 256-bit key is embedded once.

Given the *inherent* non-cryptographic robustness guarantees of the robust watermarking system, we also analyse partial data disclosures. In particular, even when the receiver decides to reveal the document partially, our proposed DOT protocol ensures that the embedding party or the sender can retrieve significantly more number of bits than when the standard oblivious transfer is used for the transfer. For example, when the receiver's 256-bit signing key is embedded 16 times in the document, even a 15% leakage of document blocks reveals roughly 235 bits of receiver's key to the sender with DOT as opposed to roughly only 50 bits that are revealed when oblivious transfer is employed.

2 An Overview of APDB

Problem Definition. We consider a setting where a sender wishes to disclose a document M to a receiver. The receiver is expected to hold a public key-secret key pair (pk, sk), where the key sk is a signing key of a (say) Bitcoin wallet corresponding to pk. Instead of the sender directly sending M to the receiver, we expect the sender and receiver to jointly compute a function $f((M, pk), sk)$ which should provide the receiver a version M_{sk} of M that has been tagged (or robustly watermarked) with the key sk. The protocol should abort (or not produce a meaningful M_{sk}) if sk from the receiver and pk from the sender are not a matching key pair. At the end of the protocol, the sender does not learn sk or M_{sk} and the receiver does not learn any further information about M. The receiver's Bitcoin wallet holds the escrow deposit for accountability.

We consider the problem in a mutually distrustful setting, and *either* the sender or the receiver can be malicious. A malicious sender can try to learn the signing key of the receiver so as to steal the deposit. When appropriate, he can also make the document public and try to accuse the receiver of dishonest disclosure. The malicious receiver, on the other hand, can try to remove/replace the watermark from the obtained document, and release the modified version to the public without revealing her key. In such an adversarial setting, we wish to satisfy the following privacy and integrity goals:
- *Sender Privacy*: Before the transfer completes, no information regarding the document is available to the receiver.

- *Receiver Privacy*: Before the disclosure of document by the receiver, no information regarding the receiver's signing key is available to the sender.
- *Sender Integrity*: In case of false accusation by the sender, no action is taken.
- *Receiver Integrity* (Revealing property): In case of disclosure of the document by the receiver, the signing key of the receiver is revealed to the sender.

We formalize these properties as an ideal functionality in Figure 7 in Section 5.

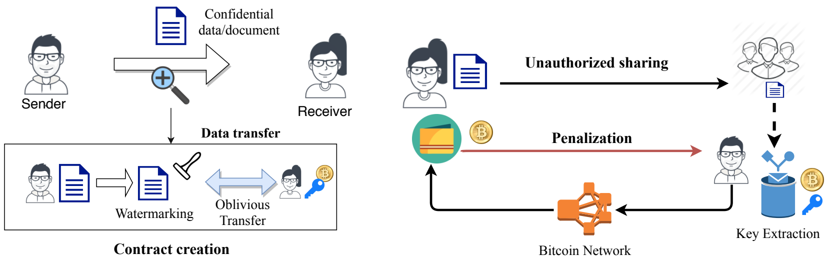


Fig. 1: APDB Protocol: High-level View

Solution Overview. We propose the APDB protocol, depicted in Figure 1 involving the two parties *Sender* and *Receiver*. The sender has the document M and the receiver has the signing key sk. The receiver initially makes a time-locked bitcoin deposit of an agreed value of funds that can be opened only if the signing-key of the receiver is available. The sender divides the document into several blocks and creates two watermarked versions (corresponding to 0 and 1) for *each block*. The parties run multiple *1-out-of-2* Oblivious Transfer (OT_1^2) protocol instances, one for transfer of each of the document blocks. The sender uses the watermarked blocks as inputs while the receiver uses each of the bits of his signing key as choice bits for the OT_1^2s and obtains one version of each block i.e., for a 256-bit signing key of the receiver, the sender (in the simplest case) divides the document into 256 blocks and creates two versions for each block using *robust watermarking*. The sender and receiver then perform 256 OT_1^2 s, where the choice bit for each OT_1^2 is each of the bits of 256-bit key of the receiver. The receiver also proves to the sender in zero knowledge that the signing key used for the deposit is indeed formed of the bits used for OT_1^2 s.

As the document is transferred through oblivious transfer, the sender can not gain any information about the signing-key of the receiver. However, if the document is revealed/disclosed before the time of expiry of the agreement, the sender learns the signing key of the receiver from the watermark of the revealed document. He can then proceed to penalize the malicious receiver by transferring the funds to himself. The multiple OT_1^2 s, one for each block, ensure that the watermark embedded in the document corresponds to the signing-key bits.

To transfer the funds out of the deposit, the sender needs both his and the receiver's signature which can not be obtained before the document is revealed to the public. Thus, he can penalize the receiver only if she is dishonest. If the receiver is honest, the agreement would expire after the agreed time and the funds will be transferred back to her. The transactional logic of the deposit is depicted as pseudo-code in Algorithm 1.

Algorithm 1 Claim-or-Refund contract logic

```
1: if Current time $t_{now} \geq t$ then
2:     Direct the locked funds back to the contract creator
3: else
4:     if Both the sender and receiver sign the transaction then
5:         Direct the funds to the mentioned recipient
6:     else
7:         Transaction is invalid
```

The receiver instead of full disclosure, can disclose the document partially to the public. She can reveal, say, half of the total 256 blocks received, so that only half the number of bits of her signing-key are revealed to the sender. However, for a 256-bit key of the receiver, the sender can in-fact divide the document into more numbers of blocks than just 256. This way, he can embed the key multiple times in the document, for example, the sender can divide it into 512 parts so that the key gets embedded twice. The sender can perform 512 OT_1^2 s with the receiver using her 256-bit key twice for the same. In such a scenario, the sender can extract more number of bits upon partial disclosure. Also, the information in the document may not be "uniform" throughout the document, so the sender can also try to embed the key multiple times in a document part where there is "more" information by dividing it into more number of parts at those document locations.

The receiver understands that one bit of her signing key is watermarked in each of document blocks received using that bit in OT_1^2. She also knows which particular bit is embedded in a particular document block, this is because, the watermark embedded in a block is same as the choice bit used for OT_1^2 in obtaining a document block. Leveraging this knowledge, the receiver can try to minimize the number of bits revealed to the sender. For example, with the sender dividing the document into 512 blocks and the receiver having a 256-bit signing-key, the receiver can reveal 100 blocks of the received document revealing only 50 bits to the sender. She can achieve this by revealing two blocks received with each bit for 50 bits. To prevent such an attack we propose a primitive called *Doubly Oblivious Transfer* (DOT). DOT prevents the receiver from learning which bit (index of the bit) of her key is watermarked into a certain block.

In DOT the sender has two messages m_0, m_1 and the receiver has two bits s_0, s_1 (refer Figure 2). The sender has an extra choice bit c using which he transfers m_{s_c} (associated with the bit s_c) to the receiver. At the end of DOT instance, the receiver cannot determine the value of c and m_{1-s_c} and the sender does not know the bit s_c that has been used in the transfer of m_{s_c}.[3]

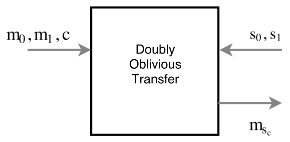


Fig. 2: Doubly Oblivious Transfer Primitive

For APDB, the sender can use DOT to transfer the document to the receiver such that she has no information about which of her bits is embedded in a certain document block. As we analyze in Section 6.1, this greatly improves the expected number of bits revealed to the sender in case of partial disclosure. For example, with the sender dividing the document into 512 blocks and 256-bit key at the receiver, upon disclosure of 100 blocks, the expected number of bits that the sender can extract is 90.3 instead of 50 while using just oblivious transfer.

Notice that our APDB protocol augments cryptographic primitive with a smart contract. Given the limited expressibility of Bitcoin contracts our (off-chain) cryptographic solution seems necessary but this may not be the case for turing-complete systems like Ethereum [15]. However, defining the complete solution as a smart contract will not be or may not remain inexpensive enough. Further comments regarding the contracts can be found in Section 8.1.

3 Functional Blocks

Robust Bit Watermarking. Once the dishonest receiver reveals the document, the sender learns the signing-key using watermark of the revealed document. For watermarking document blocks, we use a *robust* bit watermarking scheme with the property that the watermarked bit 0 or 1 cannot be removed without loss of significant information from the block. The actual watermarking scheme used can vary based on the type of the document being watermarked. We mostly follow the definition of robust watermarking by Adelsbach et al. [16].

Let $\mathcal{M}$ denote the set of all documents, $\mathcal{WM} \subseteq \{0, 1\}$ the set of two watermarks. $\mathcal{K}$ indicates the set of all keys and λ indicates the security parameter. The watermarking scheme is defined using three algorithms, one each for key generation, embedding and detection of the watermark. *Gen* (λ) is a probabilistic algorithm that outputs a key $k \in \mathcal{K}$ for the given λ. *Embed* (M, w, k) takes the document M, watermark $w \in \mathcal{WM}$ and key k as inputs and generates a

[3]For $s_0 = s_1 = b$, the receiver knows that she received m_b; however, that does not constitute any privacy leakage in our application as c and m_{1-s_c} remain private.

watermarked document M' while *Detect* (M', M, k, w) takes the watermarked document M', the original document M, the key k and the watermark w as input and outputs $\top$ if the watermark in M' matches w, else outputs $\perp$.

We require the watermarking scheme to satisfy the three properties of *Effectiveness, Robustness* and a weaker version of imperceptibility (as in [16]) called *Bit-imperceptibility. Effectiveness* indicates that a key k used to embed a watermark should also detect the watermark i.e., i.e., $\forall M \in \mathcal{M}, \forall k \in \mathcal{K}$ and $\forall w \in \mathcal{WM}$, if $Embed(M, w, k) \rightarrow M'$, then $Detect(M', M, k, w) = \top$. *Robustness* states that no probabilistic polynomial-time (PPT) adversary should be able to effectively change or remove the watermark in the watermarked document without leaving the document itself unusable. *Bit-imperceptibility* indicates that the knowledge of the watermarked document with some unknown watermark bit $w \in \mathcal{WM}$ should not reveal any additional information on the watermark bit that can be feasibly extracted. We discuss the robust watermarking algorithms in Section 7.

Oblivious Transfer. 1-out-of-2 oblivious transfer (OT_1^2) is a two-party (a sender and a receiver) computation mechanism, where the sender has two messages M_0 and M_1 and the receiver has a bit $b \in \{0, 1\}$. The goal is to transfer M_b to the receiver and at the end of the protocol, the receiver should not learn any information about M_{1-b} and the sender should not learn b. We consider the oblivious transfer protocol, called the Verified Simplest OT by Doerner *et al.* [17] which is an extended version of OT protocol by Chou et.al. [18], recalled in Appendix A along with Figure 9. The additional verification step forces the receiver to make oracle queries before receiving the encryptions from the sender there by making the protocol UC-Secure.

The other building block of our protocol would be a time-locked bitcoin deposit. The basics of bitcoin, the script for time locked deposit and the way two parties can create and verify a deposit can be found in Appendix B.

4 Doubly Oblivious Transfer — DOT

In our solution, the receiver receives the document blocks by running OT_1^2 multiple times with her signing key bits as the choice bits. However, while running OT_1^2, the receiver understands that each of the message that is received by using choice bit is indeed affected by the choice bit. To overcome this, we propose a primitive, in which the receiver, after giving multiple bits as input, receives several messages corresponding to the input bits, but the receiver does not have any information about which bit was used as choice bit for choosing a certain message. In the simplest case the sender has two messages along with a choice bit and the receiver has two bits. The sender *chooses* one of the indices of the bits of the receiver and the receiver receives the message corresponding to the bit of the chosen index. Here, the sender does not know which message has been received by the receiver and the receiver does not know which of her two bits is chosen as the choice bit to choose the messages. Hence we call it *Doubly Oblivious Transfer* (DOT) Protocol.

Ideal functionality $\mathcal{F}_{\mathcal{DOT}}$ interacts with sender S and receiver R.
- Upon receiving the input (s_0, s_1), $s_0, s_1 \in \{0,1\}$ from R, store both (s_0, s_1).
- Upon receiving the input (M_0, M_1) with $M_0, M_1 \in \{0,1\}^*$ and the choice bit $c \in \{0,1\}$ from S, store (M_0, M_1, c).
- After receiving all the inputs, check if $s_0, s_1, c \in \{0,1\}, M_0, M_1 \in \{0,1\}^*$, if yes, forward M_{s_c} to R, else abort.

Fig. 3: Ideal Functionality $\mathcal{F}_{DOT}$ of DOT

What is presented in Figure 3 is a simplified ideal functionality with the communication setup including public headers and private payloads along with session ids being assumed in the background as has been suggested in the work [19]. This simplified template is followed for all the ideal functionalities in the paper. Recall that DOT hides the index c and m_{1-s_c} from the receiver, but it need not essentially hide the value s_c itself. For $s_0 = s_1 = b$, the receiver knows the value b but not c.

Construction. We provide a construction which realizes the ideal functionality of DOT with two messages M_0, M_1 and a choice bit c at the sender and two bits s_0, s_1 at the receiver as given in the Figure 3. Both the parties possess public key-secret key pairs (refer Figure 4) and $pk = pk_S * pk_R$ where pk_S, pk_R are public keys of sender and receiver. The sender samples two elements from the group (can be points from the elliptic curve), encrypts the two messages using a symmetric encryption $E_{(.)}(.)$ with the keys obtained by hashing the elements. These encryptions are randomly permuted and forwarded to the receiver in the form of $\widehat{Enc}_i$. This is the first step in DOT. The sender then transfers the elements to the receiver such that the receiver can only decrypt M_{s_c}. The encryption and forwarding of messages prevents the need to map random message strings onto group elements for the ElGamal encryption in the next step.

The sender samples two more elements, populates $g_{i,j}$, $i, j \in \{0,1\}$ as shown in Figure 4 and encrypts all $g_{i,j}$ to the public key pk using $\mathcal{E}_{pk}(.)$ - a Re-randomizable encryption like ElGamal encryption to obtain $u_{i,j}$. Now two OT_1^2 instances are run, one for each i with $u_{i,j}$ as inputs. The receiver inputs s_i as the choice bit for the instance i of OT_1^2. The encryption of the elements to the key pk later helps the receiver to hide which keys have been obtained by her through OT_1^2 and helps the sender to hide the order in which the keys have been forwarded. Hiding the order implies hiding the mapping between bits s_i and elements obtained by the receiver through OT_1^2. The receiver after receiving the different u_{i,s_i} through OT_1^2 proceeds by applying $\mathcal{R}_{pk}(.)$, a re-randomization operation to obtain v_{i,s_i}. These re-randomized encryptions of obtained encrypted elements are now forwarded back to the sender. If there was no re-randomization step, the sender would know what elements have been obtained by the receiver and so will know what version of the message was taken by the receiver. Hence we use the re-randomization step to hide from the sender, information regarding which messages have been obtained by the receiver through OT_1^2. The sender

Sender		**Receiver**
Message blocks: M_0 and M_1, Choice bit: c		Bits: s_0, s_1
	Setup	
	Multiplicative (Public) Group $\mathbb{G}$, Generator g	
(pk_S, sk_S)		(pk_R, sk_R)
	$pk = pk_S * pk_R$	
	For all $i : 0 \leq i \leq 1$, execute the steps below	
	Symmetric encryption of Message Blocks	
$g_0, g_1 \leftarrow_R \mathbb{G}$		
$Enc_0 = E_{H(g_0)}(M_0), Enc_1 = E_{H(g_1)}(M_1)$		
$\widehat{Enc}_i = \pi_1(Enc_i)$, for permutation π_1	$\xrightarrow{\widehat{Enc}_i}$	
	El-Gamal encryption of group elements	
Set $g_{c,0} = g_0, g_{c,1} = g_1$		
$g_{1-c,0}, g_{1-c,1} \leftarrow_R \mathbb{G}$		
$u_{i,0} = \mathcal{E}_{pk}(g_{i,0})$, $u_{i,1} = \mathcal{E}_{pk}(g_{i,1})$		
	Oblivious Transfer [17]	
	Run OT_1^2 once for each i	
Input $u_{i,0}, u_{i,1}$		Input s_i
		Output u_{i,s_i}
	Re-randomization, Forwarding and Decryption	
		$v_{i,s_i} = \mathcal{R}_{pk}(u_{i,s_i})$
	$\xleftarrow{v_{i,s_i}}$	
$x_{i,s_i} = \mathcal{D}_{sk_S}(v_{i,s_i})$	$\xrightarrow{x_{c,s_c}}$	
		$g_{c,s_c} = \mathcal{D}_{sk_R}(x_{c,s_c})$
		Decrypt $\widehat{Enc}_{s_c}$ using $H(g_{c,s_c})$ appropriately

Fig. 4: Doubly Oblivious Transfer (DOT) Protocol

from v_{i,s_i}, decrypts his layer of ElGamal encryption using the decryption operation $\mathcal{D}_{sk_S}(.)$ to obtain x_{i,s_i}. He then drops x_{1-c,s_i} and forwards only the element x_{c,s_c} to the receiver. The element x_{c,s_c} (which at this point is only encrypted to the receiver's public key) is then decrypted by the receiver using her private key using $\mathcal{D}_{sk_R}(.)$ to obtain the element g_{c,s_c}. The key obtained as hash of g_{c,s_c} is used to decrypt the initially obtained random permutation of messages. Only one of them gets decrypted *correctly*. The receiver, while decrypting the encrypted messages, would not know which message is the correct encryption using the obtained key, she tries to decrypt each of the messages. For the receiver to be able to recognize the correct message for the key, we need a mechanism.

To achieve the decryption and identification of the correct message block by the receiver, the sender initially appends each of the messages with a string which is obtained as a certain public function $\widehat{f}(.)$ of key (like hash of the key) used to encrypt the message before the encryption process. After decrypting each block with the key, the receiver matches the appended string with the locally calculated string using $\widehat{f}(.)$ of the key. Whichever message has the correct match, is the correct message. Thus the receiver decrypts M_{s_c}.

Imagine the case when the initial encryptions are not permuted, then the receiver knows that the the encryptions received correspond to bit indices 0 and 1 in that order, so she can try to attack the system by setting one of the bits, say $s_0 = 0$ and the other $s_1 = 1$, then which ever encryption gets decrypted, will reveal which of the two s_is has been chosen by the sender. To prevent such a scenario, the initial permutation of the encryptions is necessary.

For a construction of the DOT protocol for a general case where the receiver has κ bits (of signing key) and the sender has 2κ messages, refer Appendix C.

Theorem 1. *The DOT protocol securely implements the functionality $\mathcal{F}_{DOT}$ under the following conditions:*
Corruption Model: *Static corruption (the sender or receiver is corrupted at the beginning of the protocol).*
Hybrid Functionalities: *H is modelled as a random oracle and secure channels between the parties are assumed.*
Computational Assumption: *The encryption scheme used in the initial step is symmetric, non-committing and robust [18]. Group used for OT_1^2 module G is a Gap-DH group.*

Proofs of all the theorems proving the security can be found in Section 6.

4.1 Committed Receiver Oblivious Transfer

Oblivious Transfer is used to transfer one message M_b where $b \in \{0,1\}$ of the two messages M_0 and M_1 from sender to the receiver with bit b. However in our APDB protocol which uses DOT (which in-turn uses OT_1^2) we further *require* the bit b to be a bit of the signing-key of the receiver. With a simple OT_1^2, the sender can not be sure if that is the case. To overcome this, we propose the committed receiver oblivious transfer ($CROT$) primitive. In $CROT$, the receiver forwards a non-interactive zero knowledge (NIZK) proof of knowledge to prove that the bit inputs from the receiver are in fact bits of the signing key. The functionality of the protocol $CROT$ is presented in the Figure 5. We depict the construction of the protocol in Figure 6 .

Ideal functionality $\mathcal{F}_{CROT}$ interacts with sender S and receiver R.
- R generates the key pair (sk, pk) and forwards the bits $s_i \in \{0,1\}, i \in [0, \cdots, \kappa-1]$ and the κ bit signing-key sk to $\mathcal{F}_{CROT}$ which stores them.
- S forwards the messages $M_{i,0}, M_{i,1}; i \in [0, \cdots, \kappa-1]$ to $\mathcal{F}_{CROT}$ which stores them.
- After receiving the inputs from both S and R, $\mathcal{F}_{CROT}$ verifies if the bits of sk are the bits s_i and (pk, sk) are a key pair. If the verification succeeds, it forwards the messages M_{i,s_i}, $i \in [0, \cdots, \kappa-1]$ to the receiver, else, aborts.

Fig. 5: Ideal Functionality $\mathcal{F}_{CROT}$ of $CROT$.

Construction. The protocol construction for the ideal functionality $\mathcal{F}_{CROT}$ as given is the Figure 5 is presented here. The sender has messages $M_{i,j}$ for $0 \le i \le \kappa-1$ and $j \in \{0,1\}$. The receiver has a signing key sk (s_i for $0 \le i \le \kappa-1$ are the bits of sk). Given a multiplicative group G and its generator g, the sender initially chooses a random value $a \leftarrow \mathbb{Z}_q$ and forwards $\mathsf{h} = \mathsf{g}^a$ to the receiver. This would be the *Setup* phase. In the next *Commit and Prove* phase, the receiver chooses random $r_i \leftarrow_R Z_q$ and computes $\mathsf{c}_i = \mathsf{g}^{r_i}\mathsf{h}^{s_i}$ for $0 \le i \le \kappa - 1$. The c_i values are forwarded to the sender as commitments to the bits s_i. The receiver also forwards $r = \sum_{i=0}^{\kappa-1} 2^i r_i$ to the sender. Along with these, for $0 \le i \le \kappa - 1$, the receiver forwards non-interactive zero knowledge (NIZK) proofs of knowledge of exponents r_i and s_i such that $\mathsf{c}_i = \mathsf{g}^{r_i+as_i}$. Each of these NIZK proofs is realized using the standard Fiat-Shamir transformation [20] of an interactive sigma protocol for Pedersen commitments in the random oracle model. Following the formal symbolic notation introduced by Camenisch and Stadler [21], each proof is depicted as $\mathbf{PoK}\{(r_i, s_i)|\mathsf{g}^{r_i}\mathsf{h}^{s_i}\}$ in Figure 6. This phase is used by the receiver to prove that the bits s_i used for the transfer are indeed the bits of the signing key sk. The sender verifies if $\mathsf{c} = \mathsf{g}^r\mathsf{pk}^a$ for the computed $\mathsf{c} = \prod_{i=0}^{\kappa-1} c_i^{(2^i)}$. He also verifies the NIZK proof. If both the verifications succeed, he proceeds with the protocol, else, aborts. The verification would also fail if (pk, sk) are not a key pair.

After successful verification the sender computes the keys $k_{i,j} = H((\mathsf{c}_i{\cdot}h^{-j})^a)$ for each $0 \le i \le \kappa-1$ and $j \in \{0,1\}$. The sender verifies if the receiver computed the keys using the verification step similar to Verified Simplest OT [17]. He forwards the challenges $p_i = H(H(k_{i,0})) \oplus H(H(k_{i,1}))$ for each i and receives the responses in the form of p'_i and the sender verifies if $p'_i = H(H(k_{i,0}))$. The keys $k_{i,j}$ are used to encrypt messages $M_{i,j}$ respectively to obtain the cipher texts $C_{i,j}$. The cipher texts $C_{i,j}$ are forwarded to the receiver who attempts to decrypt the blocks C_{i,s_i} using the keys k_{i,s_i} finishing the *Transfer* phase. The receiver can not compute the keys $k_{i,1-s_i}$ (follows from Lemma 1 of [18]) and so can not decrypt $C_{i,1-s_i}$. One can observe that the protocol does not enforce the receiver to use "bits", if the receiver uses any other values other than bits in $CROT$, the receiver receives encryptions which can not be decrypted.

The model for $CROT$ includes static corruption of parties, modelling H as random oracle and group G being Gap-DH [22] while the encryption used is symmetric, non-committing and robust [18].

Theorem 2. *The $CROT$ protocol securely implements the ideal functionality $\mathcal{F}_{CROT}$ under the following assumptions:*
Corruption Model: *static corruption*
Hybrid Functionalities: *H is modeled as a random oracle and authenticated channels between users are assumed.*
Computational Assumptions: G *is Gap-DH. The symmetric encryption used is non-committing and robust.*

Sender		**Receiver**
	Multiplicative (Public) Group G, generator g	
$\mathsf{pk} = \mathsf{g}^{\mathsf{sk}}$		$\mathsf{sk} \in \{0,1\}^{\kappa}$
	For all $i : 0 \leq i \leq \kappa - 1$	
Message blocks: $M_{i,0}$ and $M_{i,1}$		Bit decomposition of sk: s_i
	Challenge	
$a \leftarrow_R \mathbb{Z}_q$	$\xrightarrow{\mathsf{h}=\mathsf{g}^a}$	
	Commit and Prove	
	For all $i : 0 \leq i \leq \kappa - 1$	
		$r_i \leftarrow_R \mathbb{Z}_q$
		$r = \sum_{i=0}^{\kappa-1} 2^i r_i$
		$\mathsf{c}_i = \mathsf{g}^{r_i}\mathsf{h}^{s_i}$
	$\xleftarrow{\mathsf{g}^r, \mathsf{c}_i, \mathbf{PoK}\{(r_i, s_i) \mid \mathsf{g}^{r_i}\mathsf{h}^{s_i}\}}$	
$\mathsf{c} = \prod_{i=0}^{\kappa-1} \mathsf{c}_i^{(2^i)}$		
Abort if $\mathsf{c} \neq (\mathsf{g}^r \mathsf{pk}^a)$ or		
if verfication of NIZK fails		
	Transfer	
	For all $i : 0 \leq i \leq \kappa - 1$ and $j \in \{0,1\}$	
$k_{i,j} = H((\mathsf{c}_i \cdot \mathsf{h}^{-j})^a)$		$k_{i,s_i} = H(\mathsf{h}^{r_i})$
$p_i = H(H(k_{i,0})) \oplus H(H(k_{i,1}))$	$\xrightarrow{p_i}$	$p'_i = H(k_{i,s_i}) \oplus p_i s_i$
Verify $p'_i = H(H(k_{i,0}))$	$\xleftarrow{p'_i}$	
$C_{i,j} = E_{k_{i,j}}(M_{i,j})$	$\xrightarrow{C_{i,j}}$	Decrypt C_{i,s_i} using k_{i,s_i}

Fig. 6: Committed Receiver Oblivious Transfer (CROT) Protocol

5 The APDB protocol

Here, we detail the steps of the APDB protocol which uses DOT with $CROT$. The watermarking and the DOT protocol are the *off-chain* cryptographic components while the smart-contract and the deposit are the *on-chain* parts.

1. **NetworkSetup**: The sender and receiver setup their Bitcoin identities by generating secret key-public key pairs; the sender has the document M.

2. **DepositSetup**$(sk, t, Value)$: A time-locked bitcoin deposit is created by the receiver with the signing key sk for a time t and for a amount of $Value$. The deposit is a 2-of-2 multisig deposit requiring the secret keys of both the sender and the receiver to transfer the funds.

3. **WaterMark**(M): The document M is broken into κ blocks $M_i, 0 \leq i \leq \kappa - 1$ for a κ-bit long sk and each block M_i is watermarked to generate two versions $M_{i,0}, M_{i,1}$. Any watermarking scheme which satisfies the previously mentioned properties (refer section 3) can be used.

4. ***DOT* with *CROT***$(M_{i,0}, M_{i,1}, sk)$: The Doubly Oblivious Transfer protocol, used to transfer the document, takes the watermarked blocks as input. In APDB, the DOT protocol instead of using OT_1^2, uses $CROT$. The protocol is same as the general case of DOT (as shown in Figure 10 of Appendix) but uses $CROT$ instead of OT_1^2. The sender watermarks the document blocks to obtain $M_{i,j}$, generates keys from sampled group elements and forwards the permuted symmetric encrypted versions of the blocks to the receiver. He then encrypts the group elements using El-Gamal encryption to the key $pk = pk_S * pk_R$ where pk_S, pk_R are the public keys of sender and receiver. The sender inputs encrypted elements in a permuted order to the $CROT$ protocol. The receiver after proving in zero knowledge that the input to the protocol is her signing key sk, receives a set of encrypted elements which She re-randomizes sends back. The sender, decrypts his layer of encryption, inverts the applied permutation to obtain the elements in their original order and forwards them to the receiver who will be able to decrypt them. The appropriately decrypted symmetrically encrypted blocks are then joined together to form the receiver's version of the document M_{sk}.

5. **Penalize**(M_{sk}, sk_S): Upon revelation of the document, the receiver's secret key sk is extracted from the document M_{sk} and is used with the sender's secret key sk_S to transfer the deposited funds to the sender to penalize the receiver.

Analysis. Figure 7 presents the ideal functionality $\mathcal{F}_{APDB}$ for APDB, while Theorem 3 proves its security. Here we show that the functionality achieves the desirable properties discussed in Section 2. The properties of sender and receiver privacy are trivially satisfied by the functionality as it does not reveal any information except transferring the corresponding watermarked blocks to the receiver. If the receiver discloses the document, the sender can extract the embedded watermark bits and hence the signing key of the receiver, thus satisfying the revealing property. If the sender tries to falsely accuse the receiver by revealing the document in any form, the receiver does not lose the deposit as the sender does not have the receiver's key without disclosure, this achieves the sender integrity property.

Ideal functionality $\mathcal{F}_{APDB}$ interacts with sender S and receiver R.
- The receiver R generates the key pair (sk, pk), forwards the bits $s_i \in \{0,1\}, i \in [0, \cdots, \kappa - 1]$ of the κ bit signing-key sk and sk to the functionality which stores the received input.
- The sender S forwards the *watermarked* document blocks $M_{i,0}, M_{i,1} \in \{0,1\}^*, i \in [0, \cdots, \gamma - 1]$ to the functionality which stores the received input.
- The functionality verifies if the bits of sk are the bits s_i, if the verification succeeds, it forwards the message blocks $M_{i,l}$ where $l = s_{\pi(i)}$ to R, else, it aborts.

Fig. 7: Ideal Functionality $\mathcal{F}_{APDB}$ of APDB

Theorem 3. *The APDB protocol securely implements the ideal functionality $\mathcal{F}_{APDB}$ under the following assumptions:*
Corruption Model: *static corruption*
Hybrid Functionalities: *H is modeled as a random oracle and authenticated channels between users are assumed.*
Computational Assumptions: *CDH and DDH are assumed to be hard in $\mathbb{G}$, G is Gap-DH. The symmetric encryption used is non-committing and robust.*

6 Proofs of security of DOT, APDB and CROT

We prove the security of the *DOT* protocol by constructing a simulator which generates an indistinguishable view in the real world - ideal world paradigm for the adversary.

Corrupted Sender. At the beginning of the protocol, the simulator answers all oracle queries randomly and stores the query and reply pairs in the form of (q_k, r_k). At a later point of time, it receives the encrypted messages $\widehat{Enc_i}$, $i \in \{0,1\}$ and participates in oblivious transfer for the next step. The simulator sets bits s_i, $i \in \{0,1\}$ randomly with values from $\{0,1\}$ as choice bits before participating in the protocol. For OT_1^2 part of the protocol, the simulator invokes the simulator of the Verified Simplest OT [17] (which is an extended version of the oblivious transfer protocol and its simulator developed by Chou *et al.* [18,22]) for the corrupted sender case (we call it, $\mathcal{S}_{OT}$). Through OT_1^2 the simulator $\mathcal{S}_{DOT}$ receives the elements u_{i,s_i} which it tries to decrypt (its layer of encryption, the sender is expected to encrypt the messages with $\mathcal{E}_{pk}(.)$). If any of the received elements results in an error during decryption, it aborts. Else, it re-randomizes the encryption using $\mathcal{R}_{pk}(.)$ to obtain v_{i,s_i} and forwards them back to the sender. It then receives an encrypted group element as x_{c,s_c} which it tries to decrypt and hashes it to obtain the decryption key. It decrypts one of the received messages with the obtained key. If it results in an error, it aborts. The simulator decrypts the initial $\widehat{Enc_i}$ as follows: for each i, k, from the initially stored pairs (q_k, r_k), it does $Dec_{r_k}(\widehat{Enc_i})$. The first value that gets decrypted meaningfully is set as M_i for any i. If no key r_k decrypts meaningfully, it sets $M_i = \perp$. Then the simulator obtains the choice bit c of the sender as follows:

during the OT_1^2 protocol, the simulator $\mathcal{S}_{OT}$ finds out message inputs from the sender side [18] and forwards them to $\mathcal{S}_{DOT}$. For each OT_1^2 instance i, $\mathcal{S}_{DOT}$ receives two messages $g_{i,0}, g_{i,1}$ from $\mathcal{S}_{OT}$, the simulator $\mathcal{S}_{DOT}$ stores all the elements in the form of $g_{i,j}$. For each i, the simulator checks which of the elements $g_{i,j}$, $j \in \{0,1\}$, matches with the decrypted element (obtained from sender in the last step of the protocol). Whenever a match is seen, c is set to i. The simulator $\mathcal{S}_{DOT}$ forwards the messages M_i, $i \in \{0,1\}$ and choice bit c to the ideal functionality.

The adversary can not distinguish between a real world view and simulated view owing to the following facts: the simulator $\mathcal{S}_{OT}$ is UC-Secure [17] and reports it to be secure against static corruption under Gap-DH assumption); ElGamal encryption offers semantic security when DDH is hard; the real world honest receiver's output will be different only if the simulator decrypts the encryptions received to a different value apart from the ones used by the sender, but this happens with a negligible probability owing to the robustness of the encryption scheme.

Corrupted Receiver. The simulator initially generates two strings $C_1 \leftarrow \mathcal{A}_1(1^\lambda)$ and $C_2 \leftarrow \mathcal{A}_1(1^\lambda)$ and forwards to the receiver. It then samples four group elements $g_{i,j}$ for $i,j \in \{0,1\}$ and encrypts them using ElGamal encryption $\mathcal{E}_{pk}(.)$ to obtain $u_{i,j}$. It performs two instances of OT_1^2 and uses $u_{i,j}$ as inputs for instance i of OT_1^2. The receiver inputs s_i to the OT_1^2 instance i. For the OT_1^2 protocol, the simulator invokes the simulator of Verified Simplest Oblivious Transfer by Doerner *et al.* [17] for the corrupted receiver case (call it $\mathcal{S}_{OT}$). At a later point of time, it obtains an re-randomized elements v_{i,s_i}, decrypts its layer of encryption using $\mathcal{D}_{sk_S}()$ to obtain x_{i,s_i} and forwards x_{c,s_c} for a randomly chosen bit c. It then answers all oracle queries randomly except at the points $g_{i,j}$. When queried on any of the points $g_{i,j}$, the simulator sends the bits j, j to the functionality and obtains a message m'. It then replies to the query with a key $k \leftarrow \mathcal{A}_2(C_p, m)$ where p is uniformly picked from $\{1,2\}$ for every instance of the simulation.

The receiver can not distinguish the real and simulated view. This is because: ElGamal encryption offers semantic security when DDH is hard, OT_1^2 used is UC-secure [17] and the fact that when the simulator does not abort, the indistinguishability holds from non-committing property of the encryption scheme.

Proof of CROT. The security of the protocol directly follows from the fact that the OT_1^2 protocol [17] is UC-secure under Gap-DH and ZK proof of knowledge of exponent forwarded by the receiver does not leak any information regarding s_i to the sender. Hence $CROT$ is UC-secure. Let the simulator which simulates the indistinguishable view for adversary in the real world - ideal world paradigm for $CROT$ be $\mathcal{S}_{CROT}$.

Proof of APDB. The simulator $\mathcal{S}_{APDB}$ would simply invoke the simulator $\mathcal{S}_{DOT}$ while interacting with corrupt sender and receiver. The protocol is secure as the simulator $\mathcal{S}_{DOT}$ which internally invokes $\mathcal{S}_{CROT}$ (instead of $\mathcal{S}_{OT}$), produces an indistinguishable view for the adversary in the real world-ideal world paradigm.

6.1 Illustration

We illustrate the utility of APDB with DOT using $CROT$ with an example. The sender can break the document down into more than κ blocks, say 2κ, to perform $CROT$ twice, there by embedding the receiver's key two times. The finer he breaks the document, the more number of times he will be able to embed the receiver's key and so can extract more number of bits upon partial disclosure. For a receiver with 256 bit key, the sender for embedding the key twice divides the document into 512 blocks and creates two watermarked versions for each of the 512 blocks and wishes to transfer 512 messages.

The receiver wishes to selectively reveal parts of the document to the public while not revealing too much of her key bits to the sender. It is understood that the receiver reveals at least enough number of blocks (not too few) to carry useful/sufficient information. Let us assume she wishes to reveal 100 document blocks. We wish to compare how many bits she will actually reveal to the sender when she reveals 100 document blocks when APDB with DOT is used, to a scenario where just OT_1^2 is used to transfer the messages instead of DOT.

If the sender uses just OT_1^2 for the message transfer, he inputs one pair of messages for each OT_1^2 and performs 512 such OT_1^2 instances to transfer the 512 messages. In this case, the receiver knows which document block has been obtained using a particular key bit and so knows which two blocks have a certain key bit embedded in them. As she knows which two blocks have the same bit embedded in them, she will reveal 50 such pairs (with the same key bit) to the public so that the sender can learn only 50 of her signing key bits .

However, if the sender uses DOT with $CROT$ to transfer the document and the receiver decides to reveal 100 document blocks, as she does not know which key bit is embedded in a certain document block, she randomly picks 100 document blocks and reveals to the public. The expected number of key bits revealed to the sender in such a scenario would be 90.3 for 100 blocks as opposed to 50 bits with just OT_1^2. Following [23,24], the expected number of bits revealed to the sender when m blocks of the document are released with κ -bit key being watermarked over ℓ times in the document is $\kappa\left[1-\left[\binom{(\kappa-1)*\ell}{m}/\binom{\kappa*\ell}{m}\right]\right]$.

Figure 8 indicates the number of bits revealed to the sender against the percentage of blocks revealed to the public when the signing-key is watermarked ℓ times with $\ell \in \{2,4,8,16\}$. When the key is embedded 8 times, a leakage of 20% of the document/file can leak up to 211 bits of the key whereas, when it is embedded 16 times, even a 15% leak reveals as many as 235 bits. This scenario is particularly useful with larger files like video files, where the key can be embedded many number of times such that even a minor clip of the video can reveal close to the whole of the signing key. The plot in the Figure 8 compares the number of signing-key bits revealed to the sender when APDB uses DOT and OT. It clearly indicates that higher the number of times the key is embedded, higher are the number of bits revealed to the sender upon leakage. However, one has to note that the maximum number of times a key can be embedded by dividing the document depends on the document and its entropy.

Computation and Communication Overhead. For the transfer protocol, the number of exponentiations at the sender and receiver is linear in ℓ. When DOT uses CROT, the number of exponentiations performed by the sender would be $11\ell\kappa + \ell$ and by the receiver would be $7\ell\kappa$. The communication in the DOT protocol involves forwarding two versions of AES encrypted blocks, messages of CROT and forwarding of κ ElGamal encrypted points by the receiver and the sender. In CROT, the sender forwards 2κ ElGamal encrypted elements while the receiver forwards 3κ elements including the proof of knowledge messages.

	Watermarking	Full protocol
$\ell = 1$	0.357 ± 0.009	1.737 ± 0.226
$\ell = 4$	1.346 ± 0.213	16.067 ± 0.638
$\ell = 16$	1.643 ± 0.283	83.101 ± 1.623

Table 1: Time (mean ± standard deviation) taken (in seconds) for steps of the protocol when signing key is embedded for $\ell = 1, 4$ and 16

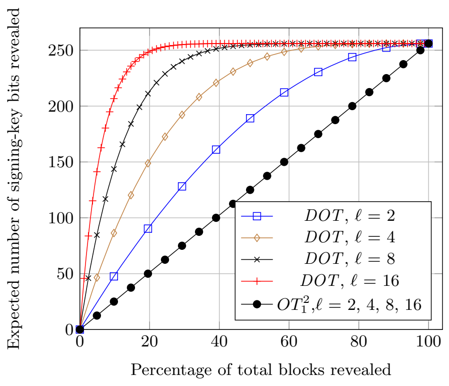


Fig. 8: Number of bits revealed to sender upon dishonest disclosure by receiver when APDB is employed with OT_1^2 and DOT with 256-bit signing key

7 Implementation and Analysis

We have implemented the APDB protocol as a single-threaded program and analyzed its performance on a MacOS machine with 3.1 GHz Intel Core i7 and 16 GB RAM. Our implementation involves the DOT protocol with robust watermarked images and a claim-or-refund contract as a Bitcoin script and as a Hyperledger chain-code. An execution run involves the transfer of an image to the receiver, and we examine the execution times for the different involved modules. The receiver's key is 256-bit long and the sender breaks the document into blocks before proceeding with the protocol.

Watermarking. The sender, after creating the document blocks, watermarks each block with 0 and 1 to generate two versions. We employ the watermarking system by Meerwald [25] which implements the Cox algorithm [26] of robust watermarking for the image blocks. The Cox algorithm is well-studied and benchmarked against several attacks [27]. In our scheme, we watermark the image document by embedding the key multiple times, Table 1 indicates the watermarking time taken where the 256-bit is embedded for $\ell = 1, 4$ and 16 indicating embedding once, 4 and 16 times. For $\ell = 1, 4$ the document in divided into 256 and 1024 blocks respectively which are transferred using the DOT protocol to the receiver who reconstructs the image from the received blocks. For demonstrative purposes, the original image before watermarking and the image reconstructed at the receiver for $\ell = 1$ are available in Figure 11 in Appendix. While we use the Cox algorithm, we reiterate that depending on the data type and application, any robust watermarking scheme can be used in our protocol for that specific application. Works such as [28], [29], [30] present different audio watermarking schemes while works like [31], [32] deal with robust video watermarking. For software watermarking, schemes suggested in [33], [34] can be considered.

Cryptographic Module - *DOT*. For the cryptographic part, we used the RELIC library [35]. The receiver's key is 256-bit long. The sender breaks the document into blocks, encrypts each of the watermarked document and forwards the blocks to the receiver in the first step of DOT protocol. The encryption used to for this step is AES in the counter mode. The sender generates group elements while participating in the DOT protocol to transfer the blocks which are ElGamal encrypted, which are later re-randomized by the receiver. The receiver decrypts the AES encrypted document blocks with the keys obtained through the ElGamal encryption and oblivious transfer.

Table 1 provides the computation timing details for the complete protocol i.e., the time including breaking the document into blocks to the point where the receiver reconstructs the document from received watermarked blocks. It presents the statistics of execution times taken over 100 runs of the experiment. Notice that the timing values reported are when the process is running in a single-thread. With multi-threading and pre-processing ElGamal encryption exponentiation, we expect significant improvement in performance and reduction in timing. To simulate the dishonest breach and eventual procurement of the leaked document by the receiver, the reconstructed image is sent to the sender of the document. The sender runs the key-extraction algorithm on the obtained image and extracts the receiver's key to perform the penalization.

8 Related work and Discussion

We are unaware of any academic research into cryptographically-enforceable automated penalization of data breach. A closely related subject, one that is well-studied, is traitor tracing [36,37]. In a traitor tracing scheme, decryption boxes with unique private keys (for a common public key) are distributed to a number of subscribers. If a device is reverse engineered and the key is leaked, the device it came from can be determined by the service provider. A recent proposal by Kiayias and Tang [38] adds a Bitcoin smart contract to hold a bond that is recoverable. This body of work has limited applicability to our APDB problem for three main reasons: (1) we want to detect leaked documents that have been meaningfully written, not keys which are arbitrary, random values; (2) we want the entity distributing the values to not learn the value until it is leaked; and (3) unlike in the smart contract variant [38], we cannot have the provider provision the signing key for use by both parties. For these reasons, we do not build our solution from traitor tracing schemes.

Using bitcoin contracts for collatorizing the fair and correct execution of cryptographic protocols has been explored earlier [10,11,39]. Our bitcoin contract is a standard claim-or-refund transaction common in this literature. The main difference is that one party must prove that the singing key used in this transaction is consistent with the one taken as input to a private computation.

8.1 Discussion

Multiple Receivers. In a scenario involving multiple receivers of the same document, the sender can embed the signing key of a each receiver multiple times into each receiver's version of the document. He can do so by dividing the document into higher number of parts compared to the receiver's key length. This ensures that, in case of collusion and each receiver contributing a small portion of his document while colluding, the sender can still extract considerable amounts of signing keys from the revealed document.

Contracts. In Section B, we developed a penalization smart contract for the Bitcoin scripting system, which intentionally has a limited set of instructions. Systems like Ethereum [15] expand this set of instructions into a fully-featured programming language allowing it to perform much elaborate tasks where it is easily possible to write our claim-or-refund contract. However, despite the much better expressivity, it does not seem to be possible to create an elaborate contact that can *efficiently* substitute the required DOT protocol and robust watermarking scheme.We implemented the penalizing claim-or-refund smart contract as a Bitcoin smart contract as well as a Hyperledger chaincode, as they allow the systems to be executed in a permissionless as well as permissioned blockchain setting. In the future, it would be interesting to create similar solutions using Solidity over the Ethereum network that can at least partially reduce the required cryptographic tools.

Fairness. The receiver deposits the bitcoins before the commencement of the protocol and so, if the document transfer does not go through, his funds will be locked till the end of the deposit time period. This is not 'fair' for the receiver. However, in a more realistic setting, in such a scenario the parties would just re-run the protocol and transfer the document.

Miner. The receiver can indeed be a miner in a Bitcoin system. He can try to pre-mine transactions to escape penalty incase of disclosure. This scenario can be prevented by the approach taken in [12, Sec. 6]. In case the sender has the knowledge only of the breach without having access to the revealed document, he can choose to make the watermarking algorithm's private-key public to make the receiver lose her deposit.

9 Conclusion

In this work, we have devised and implemented a crypto-augmented smart contract that disincentives intentional or unintentional data breach by automated penalization. Our aim here is to raise the bar for the data receivers/custodians by introducing a complementary security mechanism that is inexpensive, automated, and is not restricted by the geo-political boundaries.

To realize our protocol, we have employed robust watermarking and a claim-of-refund smart contract, and proposed a new primitive called Doubly Oblivious Transfer (DOT). DOT along with committed receiver oblivious transfer ($CROT$) not only ensures that the signing key used by the receiver for the deposit is same as the one used to obtain the document, but also provides no information to the receiver about which of her signing key bits has been embedded in a certain document part. We have implemented the complete smart-contract protocol, and observed it to be practical and easy to deploy.

References

1. C. Hourihan and B. Cline, "A look back: U.s. healthcare data breach trends"," https://hitrustalliance.net/content/uploads/2014/05/HITRUST-Report-U.S. -Healthcare-Data-Breach-Trends.pdf, 2008.
2. "Man in the cloud (mitc) attacks," https://www.imperva.com/docs/HII_Man_In_The_Cloud_Attacks.pdf, 2015.
3. Y. Rahulamathavan, M. Rajarajan, O. F. Rana, M. S. Awan, P. Burnap, and S. K. Das, "Assessing data breach risk in cloud systems," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2015, pp. 363–370.
4. T. Floyd, M. Grieco, and E. F. Reid, "Mining hospital data breach records: Cyber threats to u.s. hospitals," in *2016 IEEE Conference on Intelligence and Security Informatics (ISI)*, Sept 2016, pp. 43–48.
5. "Data protection and breach," https://otalliance.org/system/files/files/resource/documents/dpd_2015_guide.pdf, 2015.
6. "Data breaches," https://www.privacyrights.org/data-breaches?title=&breach_type%5B%5D=267.
7. "Nsw data and information custodianship policy," https://www.finance.nsw.gov.au/ict/sites/default/files/NSW%20Data%20and%20Information%20Custodianship%20Polic%20v1-0.pdf.
8. T. J. Cunningham, B. Huffman, and C. M. Salmon, "Settlement trends in data breach litigation," https://www.financierworldwide.com/settlement-trends-in-data-breach-litigation, 2014.
9. C. M. Bast, "At what price silence: are confidentiality agreements enforceable?" *William Mitchell Law Review*, vol. 25, no. 2, 1999.
10. I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols," in *ICC*, 2014.
11. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin," in *IEEE Symposium on Security and Privacy*, 2014.
12. T. Ruffing, A. Kate, and D. Schröder, "Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins," in *ACM CCS*, 2015.
13. "Friendmts," https://www.friendmts.com/nab-2017-showcase/.
14. "Digital watermarking alliance," http://digitalwatermarkingalliance.org/.
15. "Ethereum Website," https://www.ethereum.org/.
16. A. Adelsbach, S. Katzenbeisser, and A. Sadeghi, "A computational model for watermark robustness," in *8th International Workshop Information Hiding (IH)*, 2006, pp. 145–160.
17. J. Doerner, Y. Kondi, E. Lee, and a. shelat, "Secure two-party threshold ecdsa from ecdsa assumptions," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 595–612. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.00036
18. T. Chou and C. Orlandi, "The simplest protocol for oblivious transfer," in *LATINCRYPT*, 2015.
19. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, "Universally composable two-party and multi-party secure computation," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: ACM, 2002, pp. 494–503. [Online]. Available: http://doi.acm.org/10.1145/509907.509980

20. A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Proceedings on Advances in cryptology—CRYPTO '86*, 1987, pp. 186–194.
21. J. Camenisch and M. Stadler, "Proof systems for general statements about discrete logarithms," *Technical report/Dept. of Computer Science, ETH Zürich*, vol. 260, 1997.
22. Z. A. Genc, V. Iovino, and A. Rial, "The simplest protocol for oblivious transfer revisited." [Online]. Available: https://eprint.iacr.org/2017/370.pdf
23. T. Härder and A. Bühmann, "Database caching – towards a cost model for populating cache groups," in *Advances in Databases and Information Systems*, A. Benczúr, J. Demetrovics, and G. Gottlob, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 215–229.
24. S. B. Yao, "An attribute based model for database access cost analysis," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 45–67, Mar. 1977. [Online]. Available: http://doi.acm.org/10.1145/320521.320535
25. P. Meerwald, "Watermarking source code," Online, 2005. [Online]. Available: http://www.cosy.sbg.ac.at/~pmeerw/Watermarking
26. I. J. Cox, J. Kilian, F. T. Leighton, and T. Shamoon, "Secure spread spectrum watermarking for multimedia," *IEEE TIP*, vol. 6, no. 12, pp. 1673–1687, 1997.
27. I. Amer, T. Sheha, W. Badawy, and G. Jullien, "A tool for robustness evaluation of image watermarking algorithms," in *Advanced Techniques in Computing Sciences and Software Engineering*, K. Elleithy, Ed. Dordrecht: Springer Netherlands, 2010, pp. 59–63.
28. B. Y. Lei, I. Y. Soon, and Z. Li, "Blind and robust audio watermarking scheme based on svd–dct," *Signal Processing*, vol. 91, no. 8, pp. 1973 – 1984, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165168411000764
29. W.-N. Lie and L.-C. Chang, "Robust and high-quality time-domain audio watermarking based on low-frequency amplitude modification," *IEEE Transactions on Multimedia*, vol. 8, no. 1, pp. 46–59, Feb 2006.
30. Y. Erfani and S. Siahpoush, "Robust audio watermarking using improved ts echo hiding," *Digital Signal Processing*, vol. 19, no. 5, pp. 809 – 814, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1051200409000426
31. R. Lancini, F. Mapelli, and S. Tubaro, "A robust video watermarking technique in the spatial domain," in *International Symposium on VIPromCom Video/Image Processing and Multimedia Communications*, 2002, pp. 251–256.
32. J. Zhang, A. T. S. Ho, G. Qiu, and P. Marziliano, "Robust video watermarking of h.264/avc," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 2, pp. 205–209, Feb 2007.
33. R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *Information Hiding*, I. S. Moskowitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 157–168.
34. S. Kim and D. J. Wu, "Watermarking cryptographic functionalities from standard lattice assumptions," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 503–536.
35. "Relic: Efficient library for cryptography," https://github.com/relic-toolkit.
36. B. Chor, A. Fiat, and M. Naor, "Tracing traitors," in *CRYPTO*, 1994.
37. D. Boneh and M. Franklin, "An efficient public key traitor tracing scheme," in *CRYPTO*, 1999.
38. A. Kiayias and Q. Tang, "Traitor deterring schemes: Using bitcoin as collateral for digital content," in *ACM CCS*, 2015.

39. A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts,” in *IEEE Symposium on Security and Privacy*, 2016.

40. S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

A 1-out-of-2 Verified Simplest Oblivious Transfer:

Sender		**Receiver**
Messages M^0, M^1		choice bit b
$a \leftarrow_R \mathbb{Z}_q$	$\xrightarrow{h=g^a}$	$r \leftarrow_R \mathbb{Z}_q$
	$\xleftarrow{c=g^r h^b}$	
$k_0 = H(c^a)$		
$k_1 = H((c/h)^a)$		
$p = H(H(k_0)) \oplus H(H(k_1)$	$\xrightarrow{p}$	$k_b = H(h^r)$
Verify $p' = H(H(k_0))$	$\xleftarrow{p'}$	$p' = H(k_b) \oplus pb$
$C_0 = E_{k_0}(M^0)$		
$C_1 = E_{k_1}(M^1)$	$\xrightarrow{C_0,C_1}$	
		Decrypt C_b

Fig. 9: 1-out-of-2 Oblivious Transfer [17]

In this protocol, by Doerner et.al. [17] (which is an augmented version of Simplest Oblivious Transfer by Chou *et al.* [18]), given a multiplicative group $\mathbb{G}$ and its generator g, the sender initially chooses a random value $a \leftarrow_R \mathbb{Z}_q$ and the receiver chooses a random value $r \leftarrow_R \mathbb{Z}_q$. The sender transmits $h = g^a$ to the receiver who computes $c = g^{ab+r}$ and transmits to the sender. The sender then computes two keys k_0 and k_1 as $k_0 = H(c^a)$ and $k_1 = H(ch^{-1})^a$ and computes a challenge $p = H(H(k_0)) \oplus H(H(k_1)$ and forwards it to the receiver. The receiver computes the key $k_b = H(h^r)$ and returns $p' = H(k_b) \oplus pb$. After verifying if $p' = H(H(k_{)}))$, the sender encrypts M_0 and M_1 using these two keys generating C_0 and C_1 which are then forwarded to the receiver. The receiver decrypts the message M_b using the key $k_c = h^r$. Depending on b, only one of k_0 and k_1 would be equal to g^{ar} computed by the receiver. The other key g^{ar-r^2} can not be computed by the receiver and hence learns no information about M_{b-1}. As the sender just encrypts and forwards the two messages, learns no information about the bit b. Figure 9 provides the depiction of the protocol. The advantage of adding the verification step is that it forces the receiver to compute the keys before receiving the encryptions and makes the protocol (UC)secure in the real-world ideal paradigm.

B Bitcoin Claim-or-Refund Contract

Bitcoin [40] is a peer-to-peer decentralized network where participants are represented by a public and private key pair. The hash of the public key serves as the user's address and the private key is used to sign and authorize transactions. *Script* in Bitcoin is a stack-based language simulating a Push Down Automata and is used to write a smart contract. Spending funds typically involves executing/running two scripts on the spender's machine. The first is scriptPubKey which is embedded in the input transaction under the script field. It entails the conditions that must be met to spend the unspent transaction outputs (UTXO). The second one is *scriptSig* which is an unlocking script provided by the user who wants to spend the UTXO. When *scriptSig* and *scriptPubKey* are executed in sequence, the user gets to know if the transaction is valid. Bitcoin offers both sender and receiver of the funds an aspect of privacy until the funds in the deposit are directed to a recipient i.e., in our case, after the documents become public and the key gets revealed to the sender. Such privacy is not observable in any other non-blockchain financial system.

Time-Locked Compensation Deposits. We construct scriptPubKey with two prominent Bitcoin scripting language operators:
OP_CHECKLOCKTIMEVERIFY and OP_CHECKMULTISIGVERIFY.

OP_CHECKLOCKTIMEVERIFY allows users to create transactions whose outputs can only be spent in the future. OP_MULTISIGVERIFY allows the creation of transactions which need multiple signatures. In our case, the receiver creates a deposit which is locked till a future time t. The funds of the deposit can be transferred only if both the signatures of sender and the receiver are submitted before the time t. After time t, the unspent funds are transferred back to the receiver. Embedding such instructions into the funds is commonly referred to as a smart contract. Our smart contract automates the claim-or-refund functionality. The funds are transferred either when the time of the agreement expires or when the signatures of both sender and receiver are available. The scriptPubKey that receiver uses in the contract is

```
IF
OP_CHECKLOCKTIMEVERIFY OP_DROP
pk_R OP_CHECKSIGVERIFY
ELSE
OP_2 pk_R pk_S OP_2 OP_CHECKMULTISIGVERIFY
ENDIF
```

Utilizing Bitcoin. Before the protocol begins, after the two parties agree on the APDB process, the sender shares his/her public key pk_S with the receiver to create a deposit. The sender will assert that the receiver creates a transaction TX that is valid for a mutually agreed upon time t, and can be redeemed by the sender instantly with the signing keys of the sender (sk_S) and the receiver(sk_R). Here, the deposit should hold the funds equal to an agreed upon value $Value$. *VerifyDeposit(TX)* at the sender verifies the above mentioned criterion. This algorithm receives the hash of the transaction as an input and verifies that the transaction meets the above mentioned criteria, i.e. it is a valid deposit that directs $Value$ to the sender if the sender has both the signing/private keys. Earlier versions of Bitcoin allowed senders to broadcast time locked transactions and these transactions would be in the unverified transactions pool until the time lock expired or an unlocking *scriptSig* was provided by the spender of TX. However, current (as of April 2018) Bitcoin transaction does not permit nodes to propagate transactions that have an active time lock. Therefore, the receiver sends TX over any secure communication channel so that the sender can verify and sign the transaction. Once the document becomes public, we are assured from the watermarking scheme that the leaked copy of the document will have the receiver's signing key. Using the extraction algorithm *Extract(*M*,* M_{sk}*)* the sender can reconstruct the signing key sk. Once the sender has sk, he can sign the transaction TX with the *Sign(*TX*,* sk*)* and broadcast the signed transaction directing the funds in TX to his Bitcoin address.

Sender **Receiver**

Setup

Multiplicative (Public Group) $\mathbb{G}$, Generator g

(pk_S, sk_S) (pk_R, sk_R)

$pk = pk_S * pk_R$

For all $i : 0 \leq i \leq \kappa - 1$, $j \in \{0, 1\}$ execute the following steps

Message blocks: $M_{i,j}$ Bits: s_k for $0 \leq k \leq \kappa - 1$

$g_{i,j} \leftarrow_R \mathbb{G}$

$Enc_{i,j} = E_{H(g_{i,j})}(M_{i,j})$

$\widehat{Enc}_{i,j} = \pi_1(Enc_{i,j})$ $\xrightarrow{\widehat{Enc}_{i,j}}$

for permutation π_1

El-Gamal encryption of group elements

For each j, $\widehat{g}_{i,j} = \pi(g_{i,j})$ for Permutation π

$u_{i,j} = \mathcal{E}_{pk}(\widehat{g}_{i,j})$

Oblivious Transfer[17]

(Run OT_1^2 once for each i)

Input $u_{i,j}$ Input s_i

Output u_{i,s_i}

Re-randomization, Forwarding and Decryption

$\xleftarrow{v_{i,s_i}}$ $v_{i,s_i} = \mathcal{R}_{pk}(u_{i,s_i})$

$x_{i,s_i} = \mathcal{D}_{sk_S}(v_{i,s_i})$

$w_{i,s_i} = \pi^{-1}(x_{i,s_i})$ $\xrightarrow{w_{i,s_i}}$ $\widehat{g}_{i,s_i} = \mathcal{D}_{sk_R}(w_{i,s_i})$

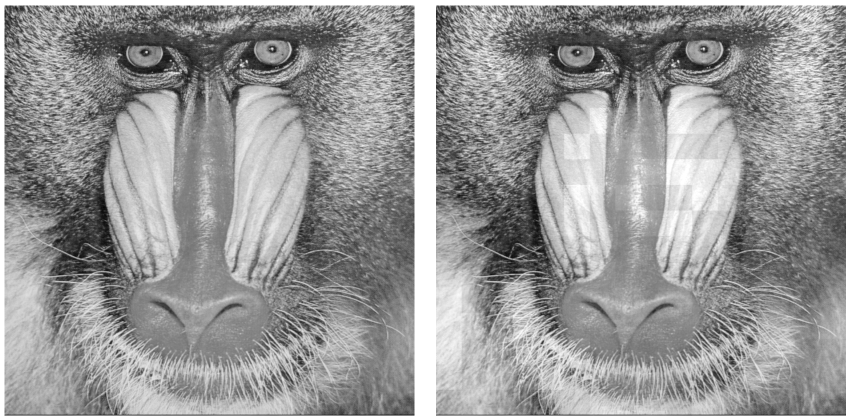
Decrypt $\widehat{Enc}_{i,s_i}$ using $H(\widehat{g}_{i,s_i})$ appropriately

Fig. 10: Doubly Oblivious Transfer Protocol (General Case)

C Generalization of DOT protocol

The DOT protocol can be easily extended to work with multiple messages at the sender and κ-bit signing key of the receiver as shown in Figure 10. In the general case, the sender has a total of 2κ messages $M_{i,j}$, for $0 \leq i \leq \kappa - 1, j \in \{0,1\}$ and the receiver has bits $s_n, 0 \leq n \leq \kappa - 1$. After participating in the protocol, the receiver receives $M_{i,l}, l = s_{\pi(i)}$ for a permutation π of set of indices i chosen at the sender. The permutation of indices is the general case equivalent of the choice bit c of the two bit case.

Forwarding a random permuted order of encrypted messages remains similar for the general case. When the elements are sampled in the general case, sampling extra elements is not *necessary.* The sender performs a permutation π on the rows i of the elements $g_{i,j}$ to obtain $\widehat{g}_{i,j}$ which are encrypted using $\mathcal{E}_{pk}(.)$ as before. Now, $\widehat{g}_{i,j}$ are input to i instances of OT_1^2 to which the receiver inputs s_i as the choice bits for each instance i. The receiver obtains u_{i,s_i}, re-randomizes the encryption using $\mathcal{R}_{pk}(.)$ and sends back v_{i,s_i}. After receiving v_{i,s_i}, the sender reverses the permutation order to obtain $w_{i,s_i} = \pi^{-1}(v_{i,s_i})$. He then decrypts his layer of encryption using $\mathcal{D}_{sk_S}(.)$ and forwards x_{i,s_i} to the receiver who decrypts her layer of decryption to obtain $\widehat{g}_{i,s_i}$. These $\widehat{g}_{i,s_i}$ are hashed to obtain the final keys which are then used to decrypt the $\widehat{Enc_i}$ received in the first step. Note that if the number of messages is not a multiple of 2κ, the sender can sample extra elements and encrypt them to input them in OT_1^2. After receiving the encrypted elements from the receiver, he can discard the elements at the indices where the extra elements have been placed in the OT_1^2 step. Also, if the receiver tries to attack the protocol by manipulating the cipher texts after the re-randomization step, she will not be able to receive meaningful keys for the correct decryption, she can gain no information regarding the sender's messages or permutation applied on encrypted messages.



(a) Original image before watermarking at the sender

(b) Watermarked image reconstructed by the receiver

Fig. 11: Original and reconstructed images