Decentralized Release of Self-emerging Data using Smart Contracts

Chao Li and Balaji Palanisamy

School of Computing and Information, University of Pittsburgh, USA
{*chl205, bpalan*}*@pitt.edu*

***Abstract*—In the age of Big Data, releasing protected sensitive data at a future point in time is critical for various applications. Such self-emerging data release requires the data to be protected until a prescribed data release time and be automatically released to the recipient at the release time, even if the data sender goes offline. While straight-forward centralized approaches provide a basic solution to the problem, unfortunately they are limited to a single point of trust and involve a single point of control. This paper presents decentralized techniques for supporting self-emerging data using smart contracts in Ethereum blockchain networks. We design a credible and enforceable smart contract for supporting self-emerging data release. The smart contract employs a set of Ethereum peers to jointly follow the proposed timed-release service protocol allowing the participating peers to earn the remuneration paid by the service users. We model the problem as an extensive-form game with imperfect information to protect against possible adversarial attacks including some peers destroying the private data (*drop attack*) or secretly releasing the private data before the release time (*release-ahead attack*). We demonstrate the efficacy and attack-resilience of the proposed techniques through rigorous analysis and experimental evaluation. Our implementation and experimental evaluation on the Ethereum official test network demonstrate the low monetary cost and the low time overhead associated with the proposed approach and validate its guaranteed security properties.**

I. Introduction

In the age of Big Data, releasing protected sensitive data at a future point in time is critical for various applications. Such self-emerging data release requires the data to be protected until a prescribed data release time and be automatically released to the recipient at the release time, even if the data sender goes offline [23], [24], [29], [33]. Examples of applications requiring timed release of self-emerging data include secure auction systems (bidding information needs protection until all bids arrive), copyrights-aware data publishing (data is automatically released when the copyright expires) and secure voting mechanisms (votes are not allowed to be accessed until the end of the polling process).

Centralized systems such as cloud storage services [1], [6], [7] may provide a simple and straight-forward approach for implementing self-emerging data release. The service provider may simply keep the sensitive data until the prescribed release time and make it available at the release time. However, such a centralized approach limits the data protection to a single point of trust and a single point of control. Even in cases when the service providers are trustworthy, such centralized models lead to channels of attacks beyond the control of service providers for an adversary to compromise the security and privacy of the data. It includes insider attacks [15], [30], external attacks on the centralized data infrastructures, malware and large-scale denial-of-service attacks [2], [9]. In 2014, 28% of the respondents of the US State of Cybercrime Survey [15] reported being victims of insider attacks and 32% reported that insider attacks were more damaging than outsider attacks.

In this paper, we tackle the timed data release problem by developing a decentralized self-emerging data release mechanism over the Ethereum blockchain network [35] that does not involve a single point of trust. The choice of the Ethereum Blockchain as the underlying data infrastructure network is motivated by two factors. First, Blockchains are huge-scale massively distributed systems that make complete decentralization possible and they are inherently designed to be reliable and robust to failures. Second, Ethereum offers Turing-completeness that allows the protocol to be programmed and implemented as decentralized smart contracts. The goal of the proposed mechanism is to route the self-emerging data within the blockchain infrastructure and enable it to automatically appear at the release time while making it harder for an adversary to access it prior to the release time. To achieve this goal, the proposed mechanism, which is implemented using smart contracts, recruits a set of Ethereum peers to jointly follow the proposed timed-release service protocol allowing the participating peers to earn the remuneration paid by the service users. The protocol requires each recruited peer to pay a security deposit so that any detected misbehaviors can result in the deposit being confiscated. Specifically, we model the problem as an extensive-form game with imperfect information to protect against possible misbehaviors including some peers destroying the private data (*drop attack*) or secretly releasing the private data before the release time (*release-ahead attack*). Through a careful design of the smart contract based on game theory, we demonstrate that the best choice of any rational Ethereum peer in the proposed technique is to always honestly follow the correct protocol. We validate the efficacy and attack-resilience of the proposed techniques through rigorous analysis and experimental evaluation on the Ethereum official test network. The experiments demonstrate the low monetary cost and the low time overhead associated with the proposed approach and validate its guaranteed security properties.

II. Decentralized self-emerging data release

In this section, we present an overview of the proposed decentralized self-emerging data release mechanism and we introduce the key ideas behind the proposed protocol, named timed-release service protocol.

A. Key components

The proposed decentralized self-emerging data release mechanism involves four key components (Figure 1).

Data sender (S): At *setup time* t_s, a data sender encrypts the private data using a secret key, sends the encrypted data to a cloud and sends the encrypted secret key into the blockchain infrastructure for timed release at the expected *release time* t_r.

Data recipient (R): While the encrypted private data can be downloaded from the cloud at any time, the secret key from the blockchain infrastructure can be released to data recipients only at t_r determined by data senders.

Cloud: A cloud storage platform is used as a medium for data senders to transfer the encrypted private data to data recipients.

Blockchain infrastructure: The blockchain infrastructure forms the core component of the self-emerging data release mechanism. It implements the protocols necessary for offering timed-release services to data senders.

B. Timed-release service protocol

The proposed timed-release service protocol recruits peers from the blockchain peer-to-peer network to store the data during $[t_s, t_r]$ and release the data to the recipients at t_r. The protocol allows any peer to join the mechanism at any time and declare any time period during which they are willing to provide services. In case no single peer can handle the entire $[t_s, t_r]$ time period, the protocol can split $[t_s, t_r]$ into a series of successive shorter time durations, each of which is handled by a different peer. In the example shown in Figure 1, the storage time duration $[t_s, t_r]$ is split into three fractions and the encrypted secret key is passed from sender S to recipient R through a routing path formed by P_1, P_2 and P_3. The proposed protocol enables such a routing scheme requiring the sender to first encrypt the secret key using the public key of the recipient and then iteratively form layers of encryption using the public keys of the selected peers on the routing path [16]. As a result, each peer on the routing path decrypts one layer of the encryption of the secret key using their private keys before forwarding it to the subsequent peers on the path until it reaches the recipient who decrypts the final layer of the encryption to obtain the key in plain text. The protocol incentivizes the participating peers by requiring the data senders to pay remunerations to the peers for obtaining the store and forward services from them to route the encrypted key along. Also, the protocol requires the participating peers to pay security deposits so that any detected misbehavior can result in their deposits being confiscated.

The timed-release service protocol satisfies two key requirements in order to be effective in practice. First, it ensures *credibility* so that senders, recipients and peers are guaranteed that they all see the same protocol when they participate in the timed-release service. We implement the timed-release service protocol using the Ethereum smart contract platform [35] which ensures that when smart contracts get deployed into the blockchain infrastructure, the protocol can be recorded in the blockchain and be available to the public and becomes nearly tamper-proof unless someone controls a majority of computation power of the distributed network [3]. Second, the protocol needs to be *enforceable* so that peers are guaranteed to receive remunerations for honestly performing the agreed services while being penalized for any misbehavior or failure to render the promised service. In our approach, the protocol forces the participants to pass the ownership of their money to the smart contract such that it ensures that the only way to receive payment from the smart contract is to trigger the contract with a satisfied condition dictated in the protocol.

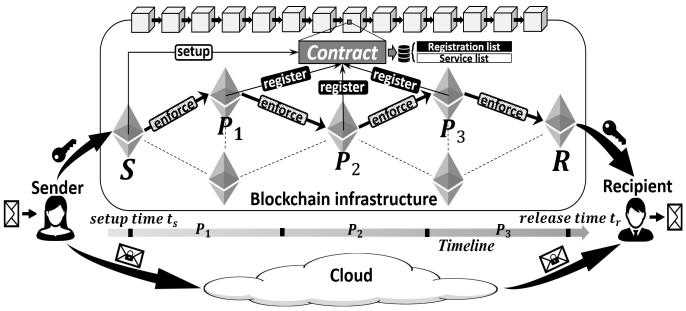

Fig. 1: Decentralized self-emerging data release

The proposed protocol consists of four key components which are briefly introduced here and we will present their detailed design in Section III. The implementation of the protocol is maintained and supported through a smart contract C.

Peer registration: At any point in time, a new peer P can register by paying a security deposit to the contract C to be added into the *registration list* maintained by C. This process makes the entire network learn that the peer has registered and can provide services during its prescribed working times. For example, in Figure 1, we find that P_1, P_2 and P_3 have been registered before the setup time t_s.

Service setup: At any point in time, a sender S can pay remunerations and submit peers selected from the registration list to the contract C and set up a timed-release service. This process makes the service to be recorded by a *service list* maintained by C. In Figure 1, we find that sender S requests a service at t_s with selected peers P_1, P_2 and P_3.

Service enforcement: After a service has been set up, the participants, namely sender S, recipient R and peers Ps should follow the protocol honestly in order to render the service successfully. Behaviors violating the protocol will lead to service failure and such misbehaviors are detected and penalized by the contract. In Figure 1, the process of routing the encrypted secret key from S to R through the path formed by the three peers is enforced by the contract C through paying remunerations for honest behaviors while confiscating deposits for misbehaviors detected by C.

Reporting mechanism: To effectively detect misbehaviors in the protocol implemented in the smart contract, the reporting mechanism incentivizes peers to report misbehaviors by announcing an award in the contract C.

C. Attack models

In our work, we model adversaries with rationality and consider two key attack models, namely *drop attack* and *release-ahead attack*.

Rational adversaries: Recently, modeling adversaries with rationality has been recommended in many attack scenarios [17], [19], [20], [32]. Informally, a semi-honest adversary follows the prescribed protocol but tries to glean more information from available intermediate results while a malicious adversary can take any action for launching attacks [21], [36]. A rational adversary lies in the middle of the two types. That is, they choose to violate protocols, such as colluding with other parties, only when doing so brings them a higher profit. In this paper, in order to design our mechanism with strong and practical security guarantees, we model all involved participants, namely S, R and P, to be rational adversaries without assuming any of them to be honest.

Adversarial attacks: The mechanism targets adversarial attacks launched after the data senders decide to release their private data. In many use cases, data senders, as the source of the private data and the initiator of the process, can determine whether to release the data and when the data should be released. For example, at a certain time point, Carol may decide to draw up a will before anyone else knows her plan. Then, by treating this time point as the registration deadline t_d and only selecting peers from the registration list that were registered before t_d, it can be guaranteed that all the selected peers were not intentionally registered for attacking just her data. In the rest of the paper, we assume that such a registration deadline t_d exists.

Drop attack: A drop attack happens when the encrypted secret key fails to reach the recipient R at release time t_r. For example, in Figure 1, after receiving the encrypted secret key from peer P_2, peer P_3 may decide to destroy it. In such attacks, due to the existence of the security deposit, a rational peer has no motivation to destroy the data. However, we notice that a drop attack can happen when an adversary intends to bribe the selected peer (say, P_3). Specifically, a drop attack can be successful when the rational adversary gets higher profit from it than the paid bribery and when the bribee receives higher bribery than the drop penalty. To break the win-win situation, we carefully design the detection mechanism in Section III-C to make drop attacks detectable and to allow the reporting mechanism in Section III-D to distinguish and penalize the adversaries. In addition, by modeling the protocol as an extensive-form game with imperfect information [25], we demonstrate that drop attack can be entirely prevented in our rational model.

Release-ahead attack: In release-ahead attacks, an adversary aims to obtain the secret key before the actual release time t_r and earn a profit by utilizing the data prior to the release time. In Figure 1, peer P_3 can launch a release-ahead attack by releasing its private key to recipient R before t_r. Similar to drop attacks, release-ahead attacks may happen through peer bribery. However, unlike drop attacks that can be detected, a release-ahead attack happens secretly as peers on the path can share stored data to any party without leaving a mark. Our proposed techniques handle this challenge by designing a reporting mechanism to model the release-ahead attack as an extensive-form game with imperfect information (Section III-D). It makes rational adversaries choose to never launch release-ahead attacks as the game ensures that the best choice of any rational Ethereum peer is to always honestly follow the correct protocol.

D. Assumptions

We make the following key assumptions in this paper:

- We assume that the monetary value v of the private data is known to the sender S. That is, the maximum profit made by an adversary from the two attack models, without considering the deposit penalty, is bounded by this value.
- We assume that the number of registered peers is adequate for providing the required service. We precisely assume that there are at least two different available registered peers at any moment for each service request.

III. Timed-release service protocol

We present the proposed timed-release service protocol organized along four subsections, each of which discusses a key component of the protocol.

A. Peer registration

In this subsection, we present the first part of the protocol, *peer registration*. It allows peers to submit their information to the public registration list maintained by the decentralized smart contract C and become publicly known as potential participants in the timed-release service protocol. The submitted information includes three components, namely when they are able to provide services (working windows), their public keys and security deposit.

Peer registration protocol

1. To be registered, each peer must submit a set of future working windows T^ws, a public key and a deposit to contract C.
2. Each peer can modify working windows T^ws and the unfrozen deposit at any time.

Peer working windows: As discussed in Section II-B, the proposed timed-release service protocol splits a long storage time duration, T^s into a series of successive shorter time durations, each of which is handled by a different peer during its working window, T^w, as the encrypted secret key gets routed on the blockchain network. Figure 2 shows an example representing T^w as horizontal segments in a coordinate frame with timeline and peer indexes as x and y axes respectively. Here, the segment at the bottom-left corner represents a working window $[t_1, t_2]$ belonging to P_i.

Deposit management mechanism: The proposed protocol uses deposits as a mechanism to penalize peer misbehaviors in

order to prevent drop and release-ahead attacks. Senders may want to pay more for getting a higher deposit from peers as guarantees of their behaviors to send private data with higher monetary value v. To support such requirements, we design a dynamic deposit management mechanism that incorporates deposit with two states: frozen and unfrozen. One can imagine that each peer has a deposit account in contract C. The deposit account is opened after registration and its balance is denoted as d^a. Initially, d^a is unfrozen. Later, data senders can calculate the amount of deposit they want from peers, denoted as d^s, based on the monetary value of the private data v. Then, during service setup, senders should only select peers from the registration list with at least d^s unfrozen deposit. The amount of d^s deposit, once being verified by contract C, will be frozen from accounts of selected peers until the end of their services. At any time, each peer can only manage its unfrozen deposit in account as the ownership of the frozen part has been temporarily transferred to contract C. In this way, the designed deposit management mechanism encourages peers with secure storage environment to keep a high deposit balance so that they can get jobs requiring a higher deposit d^s to earn more payments by taking higher risk.

B. Service setup

Next, we present the second part of the protocol, namely *service setup*, designed for allowing senders to select peers from the registration list based on their requirements and set up the service with contract C after paying remunerations.

Service setup protocol

1. Before setup time t_s, senders compute the remuneration $\widehat{r}$ and deposit d^s required by this service and then locally run the *peer selection algorithm* to select peers from the registration list satisfying their requirements.
2. At setup time t_s, senders submit service information including selected peers to contract C. Also, both sender S and recipient R should pay $p > d^s + \widehat{r}$ to contract C.
3. Upon receiving a setup request, contract C calculates remuneration $\widehat{r}$ and deposit d^s of this service, then:
   - 3.1. If $p > d^s + \widehat{r}$ and each selected peer has unfrozen deposit higher than d^s, C will approve the setup, freeze d^s of selected peers and refund $p - d^s - \widehat{r}$ to S, $p - d^s$ to R.
   - 3.2. Otherwise, C will reject the setup and refund p to S, R.

Remuneration computation: The remuneration $\widehat{r}$ paid by sender S consists of two parts $\widehat{r_c}$ and $\widehat{r_s}$. The $\widehat{r_c}$ component is charged to compensate the cost of peers for invoking functions during the service, so $\widehat{r_c} = kr_c$ for k selected peers. The $\widehat{r_s}$ component is charged to reward peers for storing the secret key, so it should be higher for longer storage time $|T^s|$. Meanwhile, to encourage more peers to serve for long-term storage, S should be charged more for a later storage hour closer to release time t_r than an earlier one closer to t_s. Therefore, if we represent the charge of i^{th} storage hour as $\triangle r_s^i$ and set the first hour charge as $\triangle r_s^1$, by setting per hour increment of $\triangle r_s^i$ as α, we get $\triangle r_s^i = \triangle r_s^{i-1} + \alpha$, which further gives $\widehat{r_s} = \frac{|T^s|[2\triangle r_s^1 + (|T^s|-1)\alpha]}{2}$. Additionally, S should be charged more for a higher monetary value of private data v as an incentive to make peers maintain higher deposit balance, so we consider the above $\widehat{r_c}$ and $\widehat{r_s}$ as the charging standard when $v = \triangle v$ (e.g., $\triangle v = \$100$) and adjust the final $\widehat{r}$ based on that. To sum up, a sender should pay remuneration $\widehat{r} = (\lceil \frac{v}{\triangle v} \rceil)^\beta [kr_c + |T^s| \triangle r_s^1 + \frac{|T^s|(|T^s|-1)}{2}\alpha]$ in total and a peer serving for i^{th} to j^{th} hours in T^s should be paid $r = (\lceil \frac{v}{\triangle v} \rceil)^\beta [r_c + (j-i)\triangle r_s^1 + \frac{i+j-2}{2}\alpha]$, where $\alpha > 0$ and $\beta > 1$.

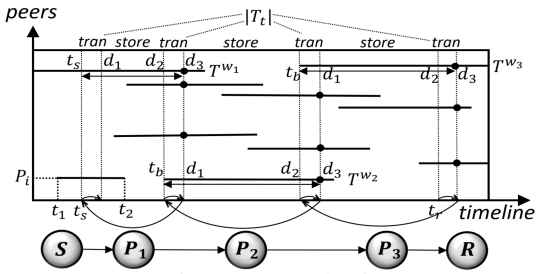

Fig. 2: Peer selection

Peer selection: The peer selection algorithm has two objectives, namely (i) minimizing remunerations paid by senders and (ii) maximizing the expected profit made by the peers. To realize the first objective, we note that the only way to reduce remuneration $\widehat{r}$ is to make k smaller, namely selecting fewer peers for a service, which does not impact the expected profit r earned by selected peers as $\widehat{r_s}$ is fixed. For achieving the second objective, we need the algorithm to always pick earlier hours in peer working windows T^ws first so that deposit d^s can be unfrozen as soon as possible. We design a greedy algorithm to achieve both of these objectives simultaneously. By decomposing the peer selection problem into a series of subproblems, we define each subproblem as 'given all peer working windows T^ws covering an input time point, output the T^w that makes the total number of selected peers minimum'. Once a T^w is selected, its beginning time t_b is then used as the input time point of the subsequent subproblem to select the next peer. Intuitively, in a subproblem, the greedy choice is to pick the T^w with earliest t_b. Next, we demonstrate the peer selection process with an example in Figure 2.

In the example, instead of release time t_r, the algorithm takes $t_r + |T_t|$ as the input time point of the first-round subproblem as we need to leave a buffer zone $|T_t|$ for data transfer between each pair of adjacent peers on path. In the first round, there are three available peer working windows T^ws covering $t_r + |T_t|$ and obviously T^{w_3}, due to its earliest begin time t_b among the three, is the greedy choice. As a result, we select P_3 as the last peer on path and set $T^{w_3}.t_b + |T_i|$ as input of the second-round subproblem. We then get T^{w_2} as second-round greedy choice, so we select P_2 and set $T^{w_2}.t_b + |T_i|$ as input of the third-round subproblem, which gives T^{w_1} as third-round greedy choice to pick P_1. This is the end of peer selection process as T^{w_1} has already covered setup time.

C. Service enforcement

The third component of the protocol deals with *service enforcement* that specifies the behaviors that should be followed by the sender S, recipient R and peers, Ps during the service process to render the service successfully. The protocol sets

deadlines for each behavior and treats any missing behavior as a drop attack to enable drop attacks behavior to be detectable. The deadlines are enforced as conditional statements in the smart contract code and any missed deadlines will result in the confiscation of security deposits. Next, we present the protocol with a discussion on the designed behaviors. We then model the protocol as an extensive-form game with imperfect information to prove that rational participating peers will always follow the protocol honestly.

Service enforcement protocol

1. Before time $t_s + |T_t|$, the sender must submit hashes of certificates and the encrypted whisper key to contract C. It must also encrypt the secret key using public keys of selected peers and transfer it to the first peer.
2. Each selected peer must decrypt one layer of the received encrypted secret key, submit the obtained certificate to contract C and verify the behavior of previous participants before its first deadline d_1. It must submit encrypted whisper key to contract C before its second deadline d_2 and transfer the secret key to the next peer before its third deadline d_3.
3. Before time $t_r + |T_t|$, the recipient must first decrypt the last layer of the encrypted secret key to submit the obtained certificate to contract C and then verify the behavior of both the previous participants and the recipient itself.
4. If any verification launched by a peer (or recipient) in term 2 (or 3) gives $False$, C should immediately terminate the service and judge the last participant on the path that fails to pass the verification to be guilty. Then, C should refund deposit d^s to all innocent participants, pay remuneration r to innocent peers and issue confiscated d^s and unused r to sender.
5. If a verification gives $Ture$, contract C should refund deposit d^s and pay remuneration r to all participants that have already honestly finished their job before their deadlines.

Whisper key submission: Our mechanism employs the Whisper protocol [12] to transfer secret keys between any two Ethereum peers by building private channels with symmetrical whisper keys. Specifically, the first peer should encrypt its whisper key with the public key of the second peer and submit it to contract C so that only the second peer can set up the channel.

Certificate: We design certificates for detecting drop attacks. For each peer and recipient, we need the sender to secretly generate a unique certificate and package it along with the corresponding layer of the encrypted secret key. Therefore, upon decrypting the received encrypted secret key with the private key, the peer (or recipient) will get the unique certificate. The peer (or recipient) then should submit the certificate to contract C. If the hash of the submitted certificate is same as the one submitted by sender, the correct reception of encrypted secret key can be proved. Otherwise, a drop attack is detected. However, with certificates, we can only detect that a drop attack has happened between two adjacent peers. It is hard to further figure out which of the two peers launched the attack as the channel between them is private. We will discuss how to handle such a dispute in Section III-D.

Verification: We design verification as a function of contract C for enforcing submission of whisper keys and certificates. A missing whisper key or certificate, both causing a drop attack, cannot be automatically detected by contract C. Here, we need the verification function to be triggered by Ethereum peers to check whether the submissions have been made on time. If all the submissions have been correctly made until the time of verification, the function returns a $True$. Otherwise, it returns a $False$. For each timed-release service, multiple verifications are required to detect a drop attack in a timely manner so that the service can be terminated on time and deposits of innocent peers can be unfrozen quickly. We carefully design the protocol as an extensive-form game with imperfect information to prove that any rational participant in this game will always choose to submit both whisper key and certificate on time.

The game induced by the protocol: We model the protocol as an *extensive-form game with imperfect information* [25], which can be represented as a game tree in Figure 3. For ease of explanation, the example only has one peer P between sender S and recipient R on path, but the services with more peers follow the same result. The game has three players $\{S, P, R\}$. Its basic actions are (whisper key and/or certificate) submission (s) and verification (v), so the action set is $\{s, v, \bar{s}, \bar{v}, sv, s\bar{v}, \bar{s}v, \bar{s}\bar{v}\}$, where $\bar{s}$ and $\bar{v}$ represent no submission and no verification respectively and $sv, s\bar{v}, \bar{s}v, \bar{s}\bar{v}$ stand for the combinations. The game tree consists of choice nodes $\{n_0, ..., n_6\}$ and terminal nodes $\{n_7, ..., n_{14}\}$. At the beginning of the game, sender S ($\{n_0\}$) can choose either to submit whisper key or not by taking one action from $\{s, \bar{s}\}$. Then, the game moves to peer P ($\{n_1, n_2\}$), who has no idea about the choice made by sender S (imperfect information). The peer P should choose one action from $\{sv, s\bar{v}, \bar{s}v, \bar{s}\bar{v}\}$, namely four combinations of doing submission and verification or not, but $s\bar{v}$ and $\bar{s}v$ can be omitted as they can be replaced by sv and $\bar{s}\bar{v}$. Finally, the game goes to the turn of recipient R ($\{n_3, n_4\}, \{n_5, n_6\}$), who has no idea of the action taken by sender S and peer P. Similar to P, recipient R should choose one action from $\{sv, \bar{s}\bar{v}\}$.

We now analyze the payoffs shown under the terminal nodes, where u_S, u_P and u_R represent payoff of sender S, peer P and recipient R respectively. The payoffs have uncertainty. Most peers on the path, by dropping the encrypted secret key, can only save a service cost c, but some peers can get an additional profit no more than the monetary value of the private data v. In this paper, for ease of explanation, we will only analyze the situation that both peer P and recipient R can get additional benefit v, denoted as $\bar{P}\bar{R}$, because all the other situations can reach the same Nash equilibrium [31]. In $\bar{P}\bar{R}$, we will show that if deposit $d^s > v$ is satisfied, then the best choice of each player is to do both submission and verification on time. We start from analyzing the choice of recipient R between sv and $\bar{s}\bar{v}$ at the last step of this game. At n_3, by choosing sv, R gets 0 at n_7, which is higher than $u_R = v - d^s$ at n_8 if $\bar{s}\bar{v}$ is chosen and $d^s > v$ is satisfied. By further checking n_4 to n_6, we can find sv always brings u_R no less than u_R from $\bar{s}\bar{v}$, which proves that sv dominates $\bar{s}\bar{v}$ and R should always choose sv no matter how the game has been played before. Following the same rule, peer P should always

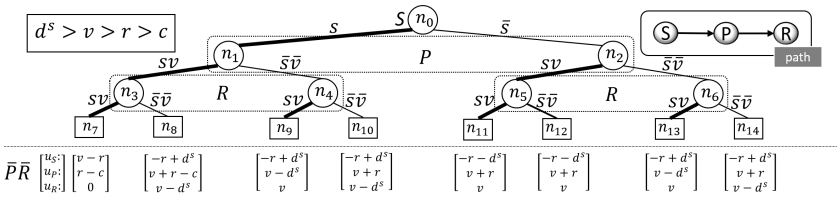

Fig. 3: Game tree induced by service enforcement protocol

choose sv at $\{n_1, n_2\}$ if $d^s > v-(r-c)$ is satisfied. Since we need $r > c$ to make Ps get positive profit from the service, $d^s > v-(r-c)$ can be automatically satisfied when $d^s > v$. Finally, with the same rule, sender S should always choose s at n_0. In game theory, if by taking a strategy, a player can make the expected payoff no less than that induced by taking any other strategy no matter what strategies are taken by other players, this strategy will become his or her best response. If all the players are taking their best responses, the game will reach a Nash equilibrium [31]. Nash equilibrium is the most important solution concept in game theory, which describes a situation that every player chooses the best response and no one can make payoff higher by changing strategy if no one else changes strategy. In this game, the Nash equilibrium is reached when all the players follow the bold edges, which results in all rational players, whether they are sender, recipient or peers, choosing to honestly obey the protocol.

D. Reporting mechanism

In this subsection, we present the last part of the protocol, namely *reporting*, designed for handling both release-ahead attacks and the dispute of drop attacks that are hard to be detected by *service enforcement* protocol.

Reporting protocol

1. Any peer can report a release-ahead attack to contract C with evidence before t_r to earn an award a.
2. Any peer on the path can report a dispute of drop attack between a suspect (the peer before this reporter on path) and the reporter to contract C before deposit d^s of the suspect is unfrozen to earn an award a.

Release-ahead attack: As discussed in Section II-C, it is highly difficult to detect a secret release attack made by peers on the path. We design a reporting mechanism to enable a release-ahead attack to be reported with evidence by adversaries themselves. The evidence should include a message and the message signed by the private key of the disloyal peer, which has been released by the peer to the adversary. Then, contract C can verify the correctness of the private key with the public key of that peer. If the private key is proved to be the one of this peer, the adversary will get an award, a from contract C while the peer will lose its deposit d^s. This reporting mechanism is an effective way to prevent release-ahead attacks as long as both adversary and the peer are rational. In the game between them, the best response of the adversary is to always report the peer to earn the award a from contract C without any penalty. Based on this knowledge, the best response of any peer on the path is to never accept bribery. Therefore, the Nash equilibrium of this game makes such a release-ahead attack never happen.

Dispute of drop attack: As discussed in Section III-C, drop attacks cannot be solely prevented by verifications. After a drop attack is detected between two adjacent peers on the path when the second peer between the two fails to submit the correct certificate, it is hard to figure out which peer actually launched it. It can be either launched by the first peer by not sending the correct encrypted secret key to the second peer or by the second peer by maliciously denying the reception of the encrypted secret key. In addition, it can be launched by the sender S by submitting fake hashes of certificates to contract C at the very beginning. To solve it, we allow the second account to report the dispute. Upon receiving the report, contract C should confiscate deposit d^s of the three participants and send back an award a to the second peer. Again, this anti-intuitive reporting mechanism is an effective way to prevent drop attack dispute by making the three participants as a community of interests as long as these accounts are rational. In this game, when there is a drop attack, the second peer has the dominant action to always report the dispute because it will lose part of its deposit $d^s - a$ by reporting it but lose the entire deposit d^s due to the missing certificate by not reporting it. With this knowledge, the best response of the first peer and sender is to never launch a drop attack because otherwise they will lose the entire deposit $d^s > v$ due to the report. Finally, given the best response of the first peer and sender, if $d^s > v + a$ is satisfied, the best response of the second peer is also to never launch a drop attack because otherwise it will lose $d^s - a > v$ due to the report. As a result, the Nash equilibrium is reached when all of them choose to never launch a drop attack.

IV. Implementation

In this section, we present the implementation of the proposed self-emerging data release smart contract and experimentally evaluate its performance and security.

A. Implementation

We programmed the smart contract in the contract-oriented programming language *Solidity* [10], deployed it to the Ethereum official test network *rinkeby* [8] and tested it with Ethereum official Go implementation *Geth* [5]. We used the *SolRsaVerify* contract [11] to verify signatures in the release reporting mechanism. We ran our experiments on an Intel Core i7 2.70GHz PC with 16GB RAM.

We design the smart contract to include fifteen main functions for supporting the four components of the protocol presented in Section III. The functions are shown in Table I. Specifically, any peer can invoke *newPeer()* to be registered and then manage its data through the other three functions in *register*. During *setup*, a sender should sign the contract through *senderSign()* and then invoke *setup()* to complete

Components	Functions	Gas cost	Monetary cost
Register	newPeer	278672	\$0.87
	updateBalance	31785	\$0.10
	updateWindow	42805	\$0.13
	updatePubKey	49866	\$0.16
Setup	senderSign	235687	\$0.73
	recipientSign	45605	\$0.14
	setup	1142774	\$3.56
Enforce	setCert	45932	\$0.14
	verifyCert	33121	\$0.10
	setWhisperKey	38787	\$0.12
	verification	87435	\$0.27
Report	releaseReport	101552	\$0.32
	releaseAward	66723	\$0.21
	dropReport	70065	\$0.22
	dropAward	75565	\$0.24

TABLE I: Summary of functions in the smart contract

service setup. At the beginning of a service, sender S should invoke *setCert()* and *setWhisperKey()* to submit hashes of certificates and encrypted whisper key to C. Then, during the service process, *verifyCert()* is invoked by peers Ps and recipient R to submit certifications, *setWhisperKey()* is invoked by Ps to submit encrypted whisper key and *verification()* is invoked by Ps and R to do verification. Finally, any Ethereum peer can invoke *releaseReport()* to report a release-ahead attack and get award through *releaseAward()*. Similarly, P and R on path can report a drop attack through *dropReport()* and get award through *dropAward()*.

For testing purpose, we generated 100 Ethereum accounts as registered peers. The 100 working windows are distributed in the future 1200 hours. Their start times follow an exponential distribution with a mean of 300 hours while their lengths follow a normal distribution with a mean of 15 hours and a standard deviation of 5 hours. In addition, we design an input parameter $Time$ to simulate the time during testing.

B. Experimental evaluation

We use the presented test setting to experimentally evaluate the performance and security of the smart contract. In one test instance, to send the private data to 1000 hours in the future, the peer selection algorithm selected five peers to form the path. Due to space limitations, we omit details about the five shorter time durations generated through partitioning the 1000-hour time duration. Next, we analyze the monetary cost to invoke the functions in this situation and then test the contract in different conditions including drop attack and release-ahead attack scenarios.

Monetary cost: The monetary costs of functions are shown in Table I. In Ethereum, each function call will cost some gases if it changes the state of contract. Therefore, the raw data measured here is the gas cost of each function, which is then transferred to cost in \$ based on 1 gas = $1.0371979124 \times 10^{-8}$ ETH and 1 ETH = \$300 as of date, 10/29/2017 [4]. As can be seen, most functions cost very little. Specifically, among the fifteen functions, seven cost lower than \$0.2 and twelve cost lower than \$0.32. The remaining three functions are *newPeer()* (\$0.87), *senderSign()* (\$0.73) and *setup()* (\$3.56). They cost higher as data is stored into the registration list and service list in C through the three functions. However, since each P only calls *newPeer()* for once during registration and each S only calls *senderSign()* and *setup()* once during service setup, these costs are quite acceptable in practice. Thus, in case of sending data to 1000 hours later, a timed-release service costs \$7.51 in total.

Cond	S	P1	P2	P3	P4	P5	R
1.	5	5	5	5	5	5	5
2.	7.872	5.010	5.017	5.026	5.035	5.040	5
3.	8.489	5.010	4.4	5.026	5.035	5.040	5
4.	8.212	5.310	5.017	5.026	5.035	4.4	5
5.	1.347	5.010	5.017	5.026	4.4	1.7	5

TABLE II: Security evaluation

Security evaluation: We then evaluate the security protection by testing the results of a timed-release service with five selected peers in different conditions when the S, R and Ps engage in suspicious behaviors, shown in Table II. The remuneration parameters are set as $\alpha = 0.000012$ ETH, $\beta = 1.1$, $\triangle r_s^1 = 0.000001$ ETH, $\triangle v = 1$ ETH, $r_c = 0.002$ ETH, $d^s = 1.2v$, $a = 0.1v$ respectively.

- Condition 1: Before the service, S, R and the five Ps all hold 5 available ETH. Then, S wants to send a secret key with its monetary value $v = 3$ ETH.
- Condition 2: If all the participants follow the protocol honestly, S can earn 2.872 ETH from the 3 ETH v after paying 0.128 ETH to Ps . Each P can earn its remuneration based on the length of its service time as well as the distance of its service from the setup time t_r.
- Condition 3: If P_2 does not submit its whisper key or certificate on time, its confiscated deposit $d_s = 3.6$ ETH will make its final payoff to be $5 - 3.6 + 3 = 4.4$ ETH.
- Condition 4: If P_5 releases its private key to P_1, P_1 can report it to earn the 0.3 ETH award, which will make P_5 get $5 - 3.6 + 3 = 4.4$ ETH payoff.
- Condition 5: If P_4 does not send the secret key to P_5 through the private channel, P_5 can report this drop dispute, which will make P_4 get 4.4 ETH payoff. Without reporting it to earn the 0.3 ETH award, P_5 can only get $5 - 3.6 = 1.4$ ETH payoff due to the failure of certificate submission.

As can be seen, in conditions 3 to 5, adversaries with misbehavior only get 4.4 ETH payoff, which makes them lose 0.6 ETH. Therefore, a rational Ethereum peer should always choose to honestly follow the protocol resulting in condition 2.

V. Related work

The problem of revealing private data only after a certain time in future was first described by May as timed-release cryptography in 1992 [29] and has intrigued many researchers in the field of cryptography since then. There are four sets of representative solutions in the literature. The first category of solutions was designed to make data recipients solve a mathematical puzzle, called time-lock puzzle, before reading the messages [13], [14], [33]. The time-lock puzzle can only be solved with sequential operations, thus making multiple computers no better than a single computer. This solution suffers

from two key drawbacks. First, the time taken to solve a puzzle may be different on different computers. Second, the puzzle computation is associated with a significant computation cost, which does not lead to a scalable cost-effective solution. The second group of solutions relies on a third party, also known as a time server, to release the protected information at the release time in future. The information, sometimes called time trapdoors, can be used by recipients to decrypt the encrypted message [23], [24], [33]. However, the time server in this model has to be trusted to not collude with recipients so that encrypted messages cannot be entered before release time. This restriction makes this set of solutions involve a single point of trust. The third set of approaches use blockchains as a reference time clock correctness guaranteed by the distributed network [22], [28]. By combining witness encryption [18] with blockchain, one can leverage the computation power of PoW in blockchain to decrypt a message after a certain number of new blocks have been generated. However, the current implementation of witness encryption is far from practical, which requires an astronomical decryption time estimated to be 2^{100} seconds [28]. Recent work has studied the problem in the context of Distributed Hash Table (DHT) networks [26], [27]. The idea behind these techniques is to leverage the scalability and distributed features of DHT P2P networks to make message securely hidden before release time. In contrast to such DHT-based solutions that do not offer guaranteed resilience to potential misbehaviors, the decentralized self-emerging data release techniques presented in this paper employs a blockchain infrastructure that offers more robust and attractive features including higher protocol enforceability by using incentives and security deposits.

VI. Conclusion

In this paper, we develop decentralized techniques for supporting self-emerging data using smart contracts in Ethereum blockchain networks. Our proposed timed release service protocol implemented as a smart contract is nearly immutable in the Ethereum blockchain. The credibility and enforceability of the protocol are guaranteed through a careful design based on extensive-form games with imperfect information to prevent possible misbehaviors including drop attacks and release-ahead attacks. We developed the smart contract using *Solidity* and implemented the mechanism on the Ethereum official test network. Our rigorous theoretical analysis and extensive experiments demonstrate the low monetary cost and the low time overhead associated with the proposed approach and validate its security properties. In future work, we plan to deal with the situation that powerful adversaries can make their controlled peers register even before the registration deadline selected by the data owners. Potential solutions include establishing a reputation system to make it harder for malicious peers to be selected or adopting secret share scheme [34] to transmit shares of a secret key through multiple paths to make it harder for adversaries to restore the secret key.

Reference

[1] Amazon simple storage service (s3):. https://aws.amazon.com/s3/.

[2] Aws best practices for ddos resiliency:. https://d0.awsstatic.com/whitepapers/DDoS_White_Paper_June2015.pdf.

[3] Ethernodes: The ethereum node explorer. https://www.ethernodes.org/network/1.

[4] Etherscan: gas price. https://etherscan.io/chart/gasprice.

[5] Geth: Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum.

[6] Google cloud storage:. https://cloud.google.com/storage/.

[7] Microsoft azure storage:. https://azure.microsoft.com/en-us/services/storage/.

[8] Rinkeby: Ethereum official testnet. https://www.rinkeby.io/#stats.

[9] Semantec report: The continued rise of ddos attacks:. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-continued-rise-of-ddos-attacks.pdf.

[10] The solidity contract-oriented programming language. https://github.com/ethereum/solidity.

[11] Solrsaverify: Verification of rsa sha256 pkcs1.5 signatues. https://github.com/adriamb/SolRsaVerify.

[12] Whisper. https://github.com/ethereum/wiki/wiki/Whisper.

[13] Nir Bitansky et al. Time-lock puzzles from randomized encodings. In *ITCS*, pages 345–356. ACM, 2016.

[14] Dan Boneh and Moni Naor. Timed commitments. In *Advances in CryptologyCrypto 2000*, pages 236–254. Springer, 2000.

[15] CERT Insider Threat Center. Us state of cybercrime survey (2014), 2014.

[16] Roger Dingledine et al. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[17] Changyu Dong et al. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. *ACM CCS*, 2017.

[18] Sanjam Garg et al. Witness encryption and its applications. In *STOC*, pages 467–476. ACM, 2013.

[19] Adam Groce and Jonathan Katz. Fair computation with rational players. In *EUROCRYPT*, pages 81–98. Springer, 2012.

[20] Siyao Guo et al. Rational sumchecks. In *Theory of Cryptography Conference*, pages 319–351. Springer, 2016.

[21] Carmit Hazay and Yehuda Lindell. A note on the relation between the definitions of security for semi-honest and malicious adversaries. *IACR Cryptology ePrint Archive*, 2010:551, 2010.

[22] Tibor Jager. How to build time-lock encryption. *IACR Cryptology ePrint Archive*, 2015:478, 2015.

[23] Kohei Kasamatsu et al. Time-specific encryption from forward-secure encryption. In *SCN*, pages 184–204. Springer, 2012.

[24] Ryo Kikuchi et al. Strong security notions for timed-release public-key encryption revisited. In *ICISC*, pages 88–108. Springer, 2011.

[25] Kevin Leyton-Brown and Yoav Shoham. Essentials of game theory: A concise multidisciplinary introduction. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2(1):1–88, 2008.

[26] Chao Li and Balaji Palanisamy. Emerge: Self-emerging data release using cloud data storage. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 26–33. IEEE, 2017.

[27] Chao Li and Balaji Palanisamy. Timed-release of self-emerging data using distributed hash tables. In *ICDCS*, pages 2344–2351. IEEE, 2017.

[28] Jia Liu, Flavio Garcia, and Mark Ryan. Time-release protocol from bitcoin and witness encryption for sat. *IACR Cryptology ePrint Archive*, 2015:482, 2015.

[29] Timothy May. Timed-release crypto. *http://www. hks. net. cpunks/cpunks-0/1560. html*, 1992.

[30] Andrew P Moore et al. The big picture of insider it sabotage across us critical infrastructures. In *Insider Attack and Cyber Security*, pages 17–52. Springer, 2008.

[31] John F Nash et al. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950.

[32] Thanh Hong Nguyen et al. Analyzing the effectiveness of adversary modeling in security games. In *AAAI*, 2013.

[33] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[34] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[35] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[36] Wei-Wei Zhang and Ke-Jia Zhang. Cryptanalysis and improvement of the quantum private comparison protocol with semi-honest third party. *Quantum information processing*, 12(5):1981–1990, 2013.