Break-glass Encryption

Alessandra Scafuro(✉)

NCSU, Raleigh, USA
ascafur@ncsu.edu

Abstract. "Break-glass" is a term used in IT healthcare systems to denote an emergency access to private information without having the credentials to do so.

In this paper we introduce the concept of *break-glass encryption* for cloud storage, where the security of the ciphertexts – stored on a cloud – can be violated *exactly once*, for emergency circumstances, in a way that is *detectable* and without relying on a trusted party.

Detectability is the crucial property here: if a cloud breaks glass without permission from the legitimate user, the latter should detect it and have a proof of such violation. However, if the break-glass procedure is invoked by the *legitimate* user, then semantic security must still hold and the cloud will learn nothing. Distinguishing that a break-glass is requested by the legitimate party is also challenging in absence of secrets.

In this paper, we provide a formalization of break-glass encryption and a secure instantiation using hardware tokens. Our construction aims to be a feasibility result and is admittedly impractical. Whether hardware tokens are necessary to achieve this security notion and whether more practical solutions can be devised are interesting open questions.

1 Introduction

The purpose of an encryption scheme [GM84] is to protect data against any observer that is not the intended recipient of the data. Encryption has been historically used to protect messages in transmission over untrusted channels. Recently however, encryption is progressively being used in the context of cloud storage to protect the confidentiality of the data uploaded by the users to the cloud. In a cloud storage setting, the cloud is trusted to guarantee availability of the uploaded data at any time, but it is not necessarily trusted (or held accountable) for not leaking clients' data to third parties. Thus, the cloud can be seen as an untrusted but reliable channel that the client uses to communicate data to herself in the future.

The Need to Break. But what happens if the user loses the key? Or more generally, what if the user loses the ability to access to the secret key (e.g. because she lost her laptop, or simply because she is not alive anymore) but there is a need

A. Scafuro—Supported by NSF grant #1012798.

D. Lin and K. Sako (Eds.): PKC 2019, LNCS 11443, pp. 34–62, 2019.
https://doi.org/10.1007/978-3-030-17259-6_2

to retrieve the documents that she uploaded to the cloud? For this emergency condition, one would like to have a way to break the encryption *without knowing any cryptographic secret* associated to the user.

Break-glass Encryption. We introduce the concept of *break-glass*[1] encryption. This is an encryption scheme that guarantees semantic security – just like any traditional encryption scheme – but it additionally provides a new command called Break that allows one designated party (the cloud) to help an alleged user to break her ciphertexts. Each ciphertext can be broken *at most one time* in a way that is *detectable*. *Detectability is the crucial property.* If the cloud breaks the ciphertexts without having received any request from the user, then the user should be able to detect and publicly prove this violation. A bit more specifically, we consider a setting where a user uploads and updates a (potentially large) number of ciphertexts and we want two properties: (1) a *legitimate* break-glass procedure preserves semantic security, that is, an honest user should be able to use the cloud to break her ciphertexts in such a way that the cloud does *not* learn anything about the plaintexts; (2) an *illegitimate* break-glass procedure is detectable, that is, if the cloud breaks user's ciphertexts without any permission, this violation is detectable and can be proven to a third party. In other words, a legitimate break-glass procedure preserves the semantic security of the ciphertext, while an illegitimate break-glass procedure leaks data but provides a proof of the violation.

What Constitutes a Legitimate Break-glass Request? A peculiar aspect of a break-glass encryption is that the break-glass procedure should be requested *without knowing any secret.* This is indeed crucial since a user wants to break-glass exactly because he does not remember his secrets.

However, if no secret are required to request to break-glass, how do we distinguish a legitimate request – coming from the owner of the data – from an illegitimate one – coming from anyone else? What makes a request illegitimate?

This is a challenge unique for our setting. For any break-glass encryption, one has to first design a permission mechanism for creating legitimate permissions without any secret and identifying and/or denying illegitimate requests. To devise such a permission mechanism we leverage the following observation. If a user did *not* request a break-glass procedure, this means that she probably still possesses her secrets, and therefore she can use them to delegitimize the request.

More concretely, the high-level idea behind the permission mechanism is the following. Any user $\mathcal{U}$ has associated a (public) alert address (e.g., an email address, a Bitcoin account), which we call alert-info. When the cloud receives a break-glass request from a party on behalf of user $\mathcal{U}$, will first send an "alert" to user $\mathcal{U}$, by forwarding the break-glass request to the address alert-info. The

[1] The name break-glass encryption is inspired by the break-glass procedures used in access control of various systems (healthcare, computer systems, etc.). In a break-glass procedure the system administrator breaks into the account of a certain user without the legitimate credentials in order to retrieve his data.

cloud will then wait a certain interval of time $T_{\mathsf{WaitPermission}}$, this time could depend on the application and the permission mechanism. If the users knows the secret associated to alert-info, then she will be able to stop or endorse the permission by using her secrets. If not, the user will simply do nothing. After waiting $T_{\mathsf{WaitPermission}}$ steps, if no denying answer is received from $\mathcal{U}$, the silence is accepted as a proof that the user did indeed lose the key and a "silence" permission that $\mathcal{U}$ wishes to break the glass. Crucially, it is important that a cloud is not able to fabricate a "silence" permission; thus the silence response must be publicly verifiable. This is necessary for protecting the user against a malicious cloud that pretends that no answer was received; but also to protect the cloud in case a malicious user remains silent but then *later* accuses the cloud by fabricating a proof delegitimizing the request.

We abstract the properties of such verifiable permission mechanism in an ideal functionality $\mathcal{G}_{\mathsf{perm}}$ (see Fig. 2) and we discuss possible implementations using a blockchain or an email provider (see Sect. 4.3).

Detectability: Why Simple Solutions Do Not Work. At first sight, the break-glass property might seem trivial to achieve; after all we are adding a method to reveal something and not to conceal. Unfortunately, this is not the case, and the main reason is that for each breaking attempt we need to ensure detectability. To show this, we now discuss some trivial solutions that do not work.

A straightforward solution could be to upload the ciphertexts in one cloud, and give the secret key to another party, e.g., a friend, another cloud, a group of colleagues, etc. This approach fails in achieving detectability: if the cloud colludes with the party holding the key then ciphertexts can be decrypted at any time and without leaving any trace. Similarly, the approach of selecting a group of people that collectively holds the secret key suffers of the same problem: if the group comes together and decides to decrypt, there is no way for the user to ever notice. Furthermore, in this type of approach, it does not seem possible to guarantee semantic security in presence of legitimate break-glass procedure.

Another relatively straightforward approach is to use a one-time hardware token. Namely, the user prepares a token which has the secret key hardwired, and when queried, it will output the key and then stop responding. The user will then send to the cloud two things: the ciphertexts and the token, with the understanding that the token should be used only in case of emergency. To break the glass, the cloud simply queries the token and get the key. The user could detect if the break-glass procedure has been illegitimately performed by periodically pinging her token. This approach however does not achieve semantic security in presence of legitimate break-glass procedure. Indeed, since the cloud learns the key, will be able to decrypt everything even when following a legitimate request, and also trace the ciphertexts updates over time. Finally, this solution does not allow for any granularity in case of illegitimate break-glass procedure. Indeed, since the key is revealed, all ciphertexts are automatically broken. Instead, we would like a more fine-grained mechanism that tells the user exactly which ciphertexts have been compromised, or that it allows the user to

setup a leaking threshold (e.g., not more then 50% of the data should be ever decrypted).

When to Use Break-glass Encryption? Break-glass security is reminiscent of covert security [AL07], and it is meaningful in scenarios where the loss of reputation is a strong deterrence against cheating. In particular, our definition is stronger than covert security in that we explicitly require that, for any illegitimate breaking attempt, the client will get a proof that can be used to publicly accuse the cloud. Thus, we target the scenario of cloud storage, where the cloud is a functional and mostly credible company (e.g., Dropbox, iCloud Apple, Google drive). In this scenario the stake for reputation is very high, therefore it is very reasonable to assume that the benefit from breaking the security of a single client, are less appealing than losing the reputation and thus all the other clients. Clearly, break-glass encryption is not suitable for scenarios where the cloud storage is an unknown server, that has not accountability or credibility. In this case indeed, there is no reputation to maintain, thus not deterrence against cheating.

What Break-glass Encryption is Not. Break-glass encryption is different from a "trapdoored" encryption scheme, where one can put a trapdoor that allows a designed party (who knows the trapdoor) to decrypt. The crucial difference is that a trapdoor allows to decrypt undetectably, while we want to make sure that each break is detectable and it can be performed at most one-time.

1.1 Our Contribution and Our Techniques

In this paper we provide two main contributions:

- **Definition of break-glass encryption.** We introduce the new concept of *break-glass encryption*. This is an encryption scheme for the **cloud storage setting**, that allows a honest user to break her own ciphertexts when necessary, while preserving semantic security. We formally define break-glass encryption via an ideal functionality $\mathcal{F}_{\mathsf{break}}$. In this context, we also introduce a new ideal functionality, $\mathcal{G}_{\mathsf{perm}}$, for generating verifiable permissions for a user $\mathcal{U}$.
- **Construction of a break-glass encryption.** As a feasibility result, we show that break-glass encryption can be constructed using (stateful) hardware token [Kat07] in the $(\mathcal{G}_{\mathsf{perm}}, \mathcal{G}_{\mathsf{clock}})$-hybrid model, where $\mathcal{G}_{\mathsf{clock}}$ is the global clock functionality. We also suggest implementations of $\mathcal{G}_{\mathsf{perm}}$ using blockchain or email systems.

In the remaining part of this section we provide more details about the technical aspects of each contribution.

Definition of Break-glass Encryption. We consider a setting where there is a cloud $\mathcal{C}$ and a user $\mathcal{U}$, and the cloud is used for memory outsourcing. The

user can perform the following actions (1) upload/download ciphertexts; (2) update a ciphertext; (3) break-glass of one (or many) ciphertexts. Our ideal functionality $\mathcal{F}_{\mathsf{break}}$ should satisfy the following properties. If the cloud honestly performs a legitimate break-glass procedure on behalf of a user, then semantic security should still hold, namely, the cloud does not learn anything about the decryption. If the cloud performs an illegitimate break-glass command, then this action must be detectable by the user the very next time the user attempts to read *any* ciphertext, and the violation should be publicly verifiable.

Defining Permission Without Secret: $\mathcal{G}_{\mathsf{perm}}$ Functionality. We introduce the $\mathcal{G}_{\mathsf{perm}}$ functionality. This a functionality used by cloud $\mathcal{C}$ and user $\mathcal{U}$ to obtain and verify valid permissions from $\mathcal{U}$. In $\mathcal{G}_{\mathsf{perm}}$ each user $\mathcal{U}_i$ is associated to a public information $\mathsf{alert\text{-}info}_i$. We stress that this information is public and a user can retrieve it even if she loses all her secrets. This functionality provides the following interface: Register, Create Permission, and Verify Permission. Register is used by $\mathcal{U}$ to register the public information $\mathsf{alert\text{-}info}$. Create Permission is used by the cloud to obtain a permission π_{perm}, which is either a publicly verifiable endorsement of the request or a publicly verifiable silence proof from $\mathcal{G}_{\mathsf{perm}}$. This step uses timing information and invoke ideal functionality $\mathcal{G}_{\mathsf{clock}}$. This is the global clock functionality, previously used in [BMTZ17] in the context of defining the public ledger functionality and analysing the security of the bitcoin protocol. VerifyPermission is used by any party who wishes to check that $(\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}})$ is a valid permission granted by $\mathcal{U}$. We discuss realization of $\mathcal{G}_{\mathsf{perm}}$ based on blockchain or email in Sect. 4.3.

Defining Break-glass: $\mathcal{F}_{\mathsf{break}}$ Functionality. We capture the security properties of detectability, accountability and semantic security in presence of legitimate break-glass procedure in an ideal functionality $\mathcal{F}_{\mathsf{break}}$ (Fig. 1). $\mathcal{F}_{\mathsf{break}}$ interacts with two parties, a cloud $\mathcal{C}$ and a user $\mathcal{U}$. $\mathcal{F}_{\mathsf{break}}$ takes in input messages $m_1, \ldots, m_l$ from $\mathcal{U}$, who can then update and retrieve her messages many times (by invoking commands Update/Retrieve). $\mathcal{F}_{\mathsf{break}}$ provides a Break command that can be invoked by $\mathcal{C}$ only. It takes in input an index i (denoting the ciphertext that the party wishes to decrypt), a proof of permission $(\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}})$ or a proof of cheating (π_{cheat}). $\mathcal{F}_{\mathsf{break}}$ verifies the permission $(\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}})$ using $\mathcal{G}_{\mathsf{perm}}$, and then proceeds by sending m_i to the user $\mathcal{U}$ only. If the request is illegitimate, $\mathcal{F}_{\mathsf{break}}$ checks that π_{cheat} is a proof of cheating. If the check passes, $\mathcal{F}_{\mathsf{break}}$ sends m_i to the cloud, and records the cheating attempt.

For every operation requested by the user, $\mathcal{F}_{\mathsf{break}}$ proceeds only after receiving an ack from $\mathcal{C}$. This captures the real world fact that a cloud can always refuse to answer (note that this is true in any cloud system). In such case, our functionality give no explicit guarantees, since the user will just receive the message $(\mathsf{refuse}, \perp)$. In practice however, refusing the answer is a proof of misbehaviour and can be turned into a legal proof via court.

Construction. Our construction relies on hardware tokens. The token is the point of trust of the user. It is initialized with the secret key k used to encrypt the

data, a signing key $\mathsf{ssk}_{\mathcal{T}}$, and the verification key of the cloud $\mathsf{vpk}_{\mathcal{C}}$. The token is sent to the cloud $\mathcal{C}$ at the very beginning, and it stays with the cloud throughout the execution. We consider the case where the user can encrypt arbitrarily long files, but the size of the token is constant, that is, it must be independent on the number of blocks encrypted. This size constraint rules out any solution where we just keep all the ciphertexts inside the token or have the token record all the ciphertexts for which the cloud invoked the Break command.

The token performs a computation only when the inputs are authenticated wrt the cloud's public key. Authenticated inputs serve two purposes: first, it provides a proof in case a cloud operated the token illegitimately; second, it protects the cloud from false accusations about the operation of the token. Finally, the outputs of the token is also authenticated, in order to avoid that the cloud sends wrong information to the user.

Warm Up Solution Without Granularity. As warm up, we describe a solution that does not provide any granularity. Namely, a user cannot detect which ciphertexts have been violated and when. The first solution works as follows. The user sends her ciphertexts $C = (c_1, \ldots, c_n)$, encrypted under a secret key k to the cloud. Then she initializes a token $\mathcal{T}$ with the secret key k, the verification key of the cloud $\mathsf{vpk}_{\mathcal{C}}$ and the signature key $\mathsf{ssk}_{\mathcal{T}}$ used to authenticate $\mathcal{T}$'s outputs. The token $\mathcal{T}$ performs a very simple functionality: on input a permission perm and a fresh public key pk, it outputs the encryption of the secret key k and stops. Note that the token only checks that the input perm, pk is correctly signed by $\mathcal{C}$; but does not check if the permission (if any) given in input is valid. This check will be done later by the parties only in case of dispute. This solution is simple, but it leaves little control on the illegitimate queries. Indeed, with one such query, the cloud can immediately decrypt 100% of the ciphertexts. We would like a more fine-grained approach that allows the user to identify precisely which ciphertexts have been broken and potentially to setup a threshold on the total number of ciphertexts that can be broken.

A Fine-Grained Solution: Breaking Ciphertexts Selectively. To break the ciphertexts selectively, the token should not output the key. Instead, we need the token to decrypt selectively. The idea is to give in input to the token also a ciphertext c_i, so that the token will answer with m_i, i.e., the decryption of c_i, rather than the key. More precisely, the token will output an encryption of m_i under the public key pk, where pk is the public key chosen by the person who is requesting to break ciphertext c_i. Moreover, to make sure that c_i is marked as broken, the token will output a new version, $c_i' = \mathsf{Enc}(m_i||broken||\mathsf{perm})$ that must replace c_i, where perm is the permission used to invoke the break procedure (perm might be empty). Next time the user will download the i-th ciphertext, she will obtain c_i' and if she still has the key k, she will detect that c_i' was illegitimately broken; similarly, next time the cloud inputs c_i' to the token, the token will refuse to decrypt.

This solution is too naive. A malicious cloud can simply ignore the new marked ciphertext c_i' and send the old unbroken c_i to the user. Namely, the

cloud can always replay old ciphertexts, defeating the checks of the token/user. To overcome this problem, we propose a mechanism that makes valid ciphertext evolve over time, or in other words, *age*. We do so by simply adding bookkeeping information; namely, each encryption now will also contain a time t_i when it was last updated, the time T_0 when the first break occurred (if any). This means that by downloading *any* of the ciphertexts the user can determine if a break-glass has happened. Each ciphertext c_i needs to be refreshed every I timestamps (where I is a parameter that can vary with application). Since updating ciphertexts requires the use of the secret key, the cloud $\mathcal{C}$ will use the token to re-encrypt each ciphertext upon each interval I. Updating a ciphertex simply means to re-encrypt the message m_i concatenated with the *current* time, and the time of the first break-glass T_0 (if any). Now, when a user downloads the i-th ciphertext c_i, and tries to decrypt it, she expects to obtain the most updated time (within a window of I steps). If not, she will discard the ciphertext as stale, and consider this as a cheating attempt from the cloud.

Therefore, in this fine grained approach, the token performs two operations for the cloud: re-encryption and break. When the cloud inputs the command 're-encrypt', then the token expects in input a ciphertext c_i that needs to be re-encrypted with the current time. The token will accept to re-encrypt only if the time registered in c_i are at most I steps behind the current time.

Finally, there is a subtle issue that requires a careful tradeoff between the size of interval I and the size of the memory of the token. Consider the following attack. The cloud queries the token to re-encrypt c_i at time t obtaining c_i^t. The cloud then queries the token to break c_i^t, at time $t+1$ and obtains m_i as well as the new encryption c_i^{t+1} which is marked as broken. Then, the cloud completely discard c_i^{t+1} and instead queries the token to re-encrypt c_i^t at time $t+2$. If $t+I < t+2$ then the token accepts to re-encrypt c_i^t with the new time $t+2$, and output the new ciphertext c_i^{t+2} which is not marked as broken (however note that c_i^{t+2} will still have the field $T_0 \neq 0$ signaling that a break-glass took place). Thus, the cloud obtains a clean unmarked version of c_i which is updated to time $t+2$, even if c_i was broken at time $t+1$ and the user will not detect that this specific ciphertext was broken (however $\mathcal{U}$ will still know that a ciphertext was broken). This problem arises because we allow a interval I between re-encryptions and can be solved by simply remembering the indexes of the ciphertexts broken within a window of I steps. The size of this list depends on the size of I (and $logn$ where n is the number of ciphertexts).

How to Get Rid of Clocks in the Token. In the outlined solution, the token uses a clock to check the current time and identify stale ciphertexts. However, requiring a clock (even only loosely synchronized) in the token is a strong assumption (the token cannot simply connect to a public server to check the time). We remove this assumption by having the cloud $\mathcal{C}$ provide the current time as input to the token. Time is simply a monotonic function, and time is "correct" if it moves forward. Thus, instead of requiring the token to keep its own clock, the token could receive the time as input, store the last time it was queried, and accept a new "current" time only if it goes in the forward direction. Checking whether the time provided

by $\mathcal{C}$ is actually good will be done by the user when downloading the ciphertext. As long as the parties (i.e., the cloud and the user) agree on a common source for reliable time, then there will be no dispute of the current time. We stress that assuming that $\mathcal{C}$ and $\mathcal{U}$ agree on a common time is a natural assumption made by most real world systems that we use in everyday life. The Network Time Protocol (NTP) [MMBK,CHMV17] is one example of protocols used for synchronization of the communications over the internet. There has been a lot of work on attacks and defenses for the NTP protocols (see [MG16,MGV+17]), but this problem is orthogonal to the one discussed in this paper. Moreover, we stress that we only need $\mathcal{C}$ and $\mathcal{U}$ to be loosely synchronized, and the parameters of the encryption (i.e., the interval I and $T_{\mathsf{WaitPermission}}$) can be tailored accordingly.

On the Need of State, Obfuscation, Blockchain. We got rid of the clock for the token, by just assuming that the world (the cloud and the user) has a global clock. Can we get rid of the state too by assuming that the world share a global immutable state? If that was possible, we could use a stateless token, or even further, can we replace the token with Indistinguishability Obfuscation [ABG+13,GGH+13,GGHW17,BCP14]. Very recently blockchain technology provides the world with a common state that everyone seems to agree on, without trusting any party. Thus, a possible approach could be for the token to store its state as a transaction in the blockchain, and the cloud can query the token on input the transaction. However, this seems to be challenging since a token could not verify the validity of a transaction without having access to the entire blockchain. Recent work [LKW15,Jag15,KMG17,GG17] show how to construct one-time programs [GKR08] and time-lock encryption leveraging the blockchain (but they are based on witness encryption [GKP+13]). We do not rule out that an interesting solution can be developed using weaker cryptographic assumptions, we leave it as future work to explore this direction.

Other Considerations. For simplicity we assume that the token sent by the user runs the prescribed code (i.e., the user does not embed malicious code into the token). This is only for simplicity of exposition, since standard techniques using zero-knowledge proof could allow us to remove this requirement. We believe that this is a reasonable relaxation, especially for the envisioned application of break-glass encryption, and since this is the first attempt to achieve such security notion. We do not consider side-channel attacks on the token.

On Surveillance and Rational Adversaries. One can argue that this scheme has the undesired effect that it can be used by a government to break the privacy of its citizens (by subpoena the cloud). This is certainly true, but recall that the citizens would detect that their privacy is violated. Therefore, one can be in two cases. Case 1, one lives in a country where the state cares about citizens not being aware that they are monitored. In this case, the state would not use the break functionality to break encryption, but something more subtle. Case 2, one lives in a country where citizens are aware that they are watched. In this case, even if the state imposes the citizens to use a break-glass encryption scheme,

then the citizens can still break-glass encrypt a ciphertext (rather than their messages). In this way, even if a break is performed, the perpetrator will only learn more encryptions.

On Refusing to Provide the Service. Just like any client-server system, the cloud can always refuse to provide the service and ignore user's requests. In this case the user will *not* have a *cryptographic proof of cheating* as promised by the break-glass encryption scheme, however, the user can obtain a court order obligating the cloud to release ciphertexts and users' token.

2 Open Problems

The main goal of this work was to introduce the concept of break-glass encryption, and show that in principle is achievable. The proposed solution however is quite impractical and only provides a feasibility result. Several questions are left open: Are (stateful) hardware token necessary to achieve this notion of security? Can we devise a solution that achieves some granularity but it does not require the cloud to continuously update the ciphertexts by querying the token? What are other interesting implementations of $\mathcal{G}_{\mathsf{perm}}$ and can $\mathcal{G}_{\mathsf{perm}}$ have applications in other setting besides break-glass encryption?

3 Related Work

Concurrently and independently from our work, recently the concept of "disposable cryptography" has been introduced by Chung, Georgiou, Lai and Zikas in [CGLZ18]. While sharing some similarity with our work, the aims and the techniques are very different. The goal of this work is to provide an encryption scheme for cloud storage, that can be broken by *anyone* exactly once, in a detectable way. The motivation for break-glass is the case when the *legitimate user* wants to decrypt the data she uploaded to the cloud, but she lost all her secret keys. The goal of [CGLZ18] is to realize trapdoored cryptographic schemes that can be violated once, by a *designated entity* who possesses the trapdoor, which is *not* the legitimate user and without being detected. The motivation for dispensable backdoors is to allow law enforcement to break the scheme exactly once, the envisioned application is breaking into mobile phones undetectably. Somewhat related to the concept of break-glass cryptography is the idea of time-locked encryption [BN00, BGJ+16, BM09, BM17, LPS17]. In time-locked encryption some information is meant to be protected for a certain period time T, thus when the time expires, the cloud will be able to decrypt the information contained in the ciphertext. The difference between break-glass and time-locked encryption is in the fact that our cloud can always break the encryption if she wishes to do so, but at the price of being detected. Our adversarial model is very close in spirit to the covert model [BM09]. In this model the adversary is allowed to cheat and violate the privacy to the parties, but by doing so he will be caught and thus lose reputation.

4 Definitions

4.1 Break the Glass Encryption Scheme

A break-glass encryption is a private-key encryption scheme designed for the cloud storage setting. It provides a procedure called Break which allows a user to decrypt her ciphertexts without knowing the secret key, exactly once. At high-level a break-glass encryption scheme must satisfy the following properties:

- *Completeness.* If the cloud and the user follow the protocol then the user is able to obtain the plaintexts that she encrypted originally, without knowing the key.
- *Confidentiality (Semantic-Security).* If no Break is performed, then the ciphertexts are semantically secure against any PPT malicious cloud.
- *Break-glass Confidentiality.* If break-glass is requested by a legitimate user, the cloud does not learn anything about the broken ciphertexts.
- *Break-glass Detectability.* If break-glass is performed by the cloud without user's permission, the cloud can decrypt each ciphertext exactly once, and each violation is detected by the user (unless the cloud refuses to respond).
- *Break-glass Accountability.* A user should be able to prove that the cloud performed an illegitimate break-glass request.

We provide a simulation-based definition [Gol04, HL10] and capture the above security requirements via an ideal functionality $\mathcal{F}_{\mathsf{break}}$ (Fig. 1). To capture break-glass accountability, $\mathcal{F}_{\mathsf{break}}$ is designed so that it will proceed with an illegitimate break requested by the cloud $\mathcal{C}$, only if $\mathcal{C}$ provides a proof of cheating, that we denote by cheat-proof. $\mathcal{F}_{\mathsf{break}}$ invokes ideal functionalities $\mathcal{G}_{\mathsf{perm}}$ and $\mathcal{G}_{\mathsf{clock}}$ (which are defined as global functionalities). This definitional approach was used in previous work in the (stronger) GUC setting by Badertscher et al. [BMTZ17]. Finally, $\mathcal{F}_{\mathsf{break}}$ captures the real world fact that a cloud can always refuse to provide a service. Thus, every operation on the outsourced messages is fulfilled by $\mathcal{F}_{\mathsf{break}}$ only if the cloud agrees on responding.

Definition 1 (Break-glass encryption scheme). *A scheme Π is a secure break-glass encryption scheme if it realizes the functionality $\mathcal{F}_{\mathsf{break}}$ in the sense of [HL10].*

4.2 The $\mathcal{G}_{\mathsf{perm}}$ Ideal Functionality

The ideal functionality $\mathcal{G}_{\mathsf{perm}}$ is described in Fig. 2 and is inspired by the signature ideal functionality of [Can04]. The purpose of this functionality is to alert the user $\mathcal{U}_i$, registered with alert address $\mathsf{alert\text{-}info}_i$, that a permission request was triggered by a party. The user $\mathcal{U}_i$ can then provide a proof to either legitimate or to invalidate the permission request. This proof is then sent to the cloud $\mathcal{C}_i$ associated to $\mathsf{alert\text{-}info}_i$. If the user fails to provide any proof within time $T_{\mathsf{WaitPermission}}$, then a proof of silence is generated and provided to $\mathcal{C}$.

FUNCTIONALITY $\mathcal{F}_{\mathsf{break}}$.

Participants: The cloud $\mathcal{C}$, a user $\mathcal{U}$, the adversary.
Variables: a boolean $flag$, when $flag = 1$ means that there has been an illegitimate break. A vector $\mathsf{Status} = \mathsf{Status}[1], \ldots, \mathsf{Status}[l]$, with $\mathsf{Status}[i] = b|nlegit$ where $b = 1$ means that the i-th ciphertext was broken; $nlegit = 1$ means that the break was not legitimate. A vector $\mathsf{Cheat}\Pi[1], \ldots, \mathsf{Cheat}\Pi_{\mathsf{cheat}}[l]$ collects proofs of illegitimate break-glass.
External Functionality: $\mathcal{G}_{\mathsf{perm}}$.
Algorithms: $\mathcal{F}_{\mathsf{break}}$ is parameterized by $\mathsf{VrfyCheatProof}$ to check the proofs of illegitimate access provided by a corrupted cloud.
Procedure:
▷ **Upload.** Upon receiving $(\mathsf{upload}, \mathsf{sid}, m_1, \ldots, m_l, \mathcal{U})$ from user $\mathcal{U}$, store the vector $M = m_1, \ldots, m_l$. (Ignore any other request of this type). Send $(\mathsf{uploaded}, \mathsf{sid}, l, \mathcal{U})$ to the cloud $\mathcal{C}$ and the adversary.
▷ **Update.** Upon receiving $(\mathsf{update}, \mathsf{sid}, i, m)$ from user $\mathcal{U}$, send $(\mathsf{update}, \mathsf{sid}, i)$ to $\mathcal{C}$. If $\mathcal{C}$ is corrupted, then wait for answer $(\mathsf{ack\text{-}updated}, \mathsf{sid}, \mathcal{U}, resp)$. If $resp = no$ send $(\mathtt{refuse}, \perp)$ to $\mathcal{U}$. Else, update $m_i := m$. Send $(\mathsf{updated}, \mathsf{sid}, i)$ to $\mathcal{U}, \mathcal{C}$.
▷ **Break.** Upon receiving $(\mathsf{break}, \mathsf{sid}, i, \mathsf{perm\text{-}proof}, \mathsf{cheat\text{-}proof})$ from $\mathcal{C}$.

1. Case 1: User's Request. Parse $\mathsf{perm\text{-}proof} = (\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}})$.
   (a) Validate permission: Send $(\mathsf{verify\text{-}permission}, \mathsf{alert\text{-}info}, \pi_{\mathsf{perm}})$ to $\mathcal{G}_{\mathsf{perm}}$. If the output is `granted`, proceed.
   (b) Send $(\mathsf{break\text{-}request}, \mathsf{sid}, i, \mathsf{alert\text{-}info}, \pi_{\mathsf{perm}})$ to $\mathcal{C}$. If $\mathcal{C}$ is corrupted, wait to receive $(\mathsf{ack\text{-}break}, \mathsf{sid}, \mathcal{U}, resp)$. If $resp = no$ send $(\mathtt{refuse}, \perp)$ to $\mathcal{U}$. Else proceed with the break procedure as follows:
   – (Never broken before) if $\mathsf{Status}[i] = 00$ then send $(m_i, flag)$ to $\mathcal{P}$.
   – (Already broken) if $\mathsf{Status}[i] = 1|nlegit$, send $(i$ is $\mathtt{broken})$ to $\mathcal{P}$.
2. Case 2. Illegitimate Request. If $\mathsf{VrfyCheatProof}(\mathsf{cheat\text{-}proof}) = 1$:
   * Set $flag = 1$. Set $\mathsf{Status}[i] = 11$ and send m_i to $\mathcal{C}$.
   * Register $\mathsf{Cheat}\Pi[i]$:=$\mathsf{cheat\text{-}proof}$.

▷ **Retrieve.** Upon receiving $(\mathsf{get}, \mathsf{sid}, i, \mathcal{U})$ from $\mathcal{U}$. Send $(\mathsf{retrieve\text{-}request}, \mathsf{sid}, i, \mathcal{U})$ to $\mathcal{C}$. If $\mathcal{C}$ is corrupted, then wait for the command $(\mathsf{ack\text{-}retrieve}, \mathsf{sid}, \mathcal{U}, resp)$; if $resp = no$ send $(\mathtt{refuse}, \perp)$ to $\mathcal{U}$. Else send $(m_i, flag, \mathsf{Status}[i])$ to $\mathcal{U}$.
▷ **Accuse with Proof**. Upon receiving $(\mathsf{accuse}, \mathsf{sid}, j)$ from a party $\mathcal{P}$. Send $(\mathsf{accused}, \mathsf{sid}, \mathcal{P})$ to $\mathcal{C}$ and $\mathsf{Cheat}\Pi[j]$ to $\mathcal{P}$.

Fig. 1. $\mathcal{F}_{\mathsf{break}}$ functionality

4.3 How to Implement $\mathcal{G}_{\mathsf{perm}}$

In this section we informally discuss two possible implementations of $\mathcal{G}_{\mathsf{perm}}$.

Implementation Using a Blockchain. Assuming the existence of a blockchain, $\mathcal{G}_{\mathsf{perm}}$ could be instantiated as follows. Procedure $(\mathsf{register}, \mathsf{alert\text{-}info}, \mathcal{U}_i)$ consists in having the user compute keys for a digital signature scheme and send the corresponding public key $\mathsf{vpk}_{\mathcal{U}_i}$ to the cloud. $\mathcal{C}$ will then set $\mathsf{alert\text{-}info} = \mathsf{vpk}_{\mathcal{U}_i}$.

FUNCTIONALITY $\mathcal{G}_{\mathsf{perm}}$.
$\mathcal{G}_{\mathsf{perm}}$ is parameterized by procedure InfoCheck used to check the validity of the credential provided by the user at registration phase.

Variables. $T_{\mathsf{WaitPermission}}$ is the time allowed to generate a valid permission or to deny a permission. $\mathcal{L}_{\mathcal{U}}$ is the list of registered users.

External Functionality. $\mathcal{G}_{\mathsf{clock}}$

▷ **Register.** Upon receiving $(\mathsf{register}, \mathsf{alert\text{-}info}, \mathcal{U}_i, \mathcal{C}_i)$ from user $\mathcal{U}_i$. If $\mathsf{InfoCheck}(\mathsf{alert\text{-}info}, \mathcal{U}_i) = 1$ then add $(\mathcal{U}_i, \mathsf{alert\text{-}info}, \mathcal{C}_i)$ to the list of registered users $\mathcal{L}_{\mathcal{U}}$ (where $\mathcal{C}_i$ is the party that obtains permissions from $\mathcal{U}_i$) and send it to the adversary.

▷ **Create Permission**. Upon receiving $(\mathsf{CreatePermission}, \mathsf{alert\text{-}info}, \mathcal{U}_i)$ from a party P. If $(\mathcal{U}_i, \mathsf{alert\text{-}info}, \mathcal{C}_i)$ is in $\mathcal{L}_{\mathcal{U}}$, then send $(\mathsf{CreatePermission}, \mathsf{alert\text{-}info})$ to $\mathcal{U}_i$.

- Upon receiving $(\mathsf{ack\text{-}check}, (\mathsf{alert\text{-}info}, \mathcal{U}_i), ans)$ from $\mathcal{U}_i$. Send $(\mathsf{GetProof}, \mathsf{alert\text{-}info}, \mathcal{U}_i, ans)$ to the adversary and obtain π.
- Else, if $T_{\mathsf{WaitPermission}}$ time has elapsed (use $\mathcal{G}_{\mathsf{clock}}$ for this), send $\mathsf{GenSilenceProof}(\mathcal{U}_i, \mathsf{alert\text{-}info}, \mathcal{P})$ to the adversary, and obtain π_ϵ. Set $\pi = \pi_\epsilon$.
- Check that no entry $(\mathsf{alert\text{-}info}, \mathcal{U}_i, \pi, 0)$ is recorded. If it is, output error message to $\mathcal{U}_i$. Else, record $(\mathsf{sid}, \mathsf{alert\text{-}info}, \mathcal{U}_i, \pi, ans, 1)$.
- Finally, send $(\mathsf{Permission}, \mathsf{alert\text{-}info}, \pi, ans)$ to $\mathcal{C}_i$.

▷ **Verify permission.** Upon receiving $(\mathsf{verify\text{-}permission}, \mathsf{alert\text{-}info}, \pi)$ from any party P_j, send $(\mathsf{VerifyPerm}, \mathsf{alert\text{-}info}, \pi, \Phi)$ to the adversary. Then,

1. If there is an entry $(\mathsf{alert\text{-}info}, \mathcal{U}_i, \pi, ans, 1)$ then
   – If $ans = YES$. Send $(\mathsf{alert\text{-}info}, \mathsf{verifiably-granted}, \pi)$ to P_j.
   – Else, if if $ans = NO$ Send $(\mathsf{alert\text{-}info}, \mathsf{verifiably-denied}, \pi)$ to P_j.
2. If there is no entry $(\mathsf{alert\text{-}info}, \mathcal{U}_i, \pi, ans, 1)$ recorded and $\mathcal{U}_i$ is not corrupted, then send $(\mathsf{alert\text{-}info}, \mathsf{notverified}, \pi)$ and record $(\mathsf{alert\text{-}info}, \mathcal{U}_i, \pi, ans, 0)$.
3. Else, if there is an entry $(\mathsf{alert\text{-}info}, \mathcal{U}_i, \pi, ans, 0)$ send $(\mathsf{alert\text{-}info}, \mathsf{notverified}, \pi)$ to P_j.
4. Else, set $(\mathsf{alert\text{-}info}, \mathcal{U}_i, \pi, ans, \Phi)$ and performs checks 1, 2, 3.

Fig. 2. $\mathcal{G}_{\mathsf{perm}}$ functionality

To make a break-glass request, $\mathcal{U}_i$, who potentially lost all the keys, will send a break-glass request to $\mathcal{C}$ (this request can be sent via a website form; to avoid denial of service attack one can enforce that to submit a request the user must pay some small amount of money). Upon receiving the request, $\mathcal{C}$ will look up the alert-info for $\mathcal{U}_i$ and proceed with the CreatePermission procedure.

Procedure $(\mathsf{CreatePermission}, \mathsf{alert\text{-}info}, \mathcal{U}_i)$ is implemented as follows. $\mathcal{C}$ prepares a permission request by posting a transaction $\mathsf{Tx}_{\mathsf{alert}}$ on the blockchain. Such transaction will contain a break-glass request in reference to the tuple $(\mathsf{alert\text{-}info}, \mathcal{C})$. After the transactions has been posted in a block of the blockchain, $\mathcal{C}$ waits $T_{\mathsf{WaitPermission}}$ time (this duration can be agreed on by the parties). Then, $\mathcal{C}$ downloads the blocks of the blockchain that appeared after the transaction $\mathsf{Tx}_{\mathsf{alert}}$ was posted and:

1. If there is no signed transaction that verifies under public key alert-info, then this sequence of $T_{\mathsf{WaitPermission}}$ blocks $(\mathbf{b}_1, \ldots, \mathbf{b}_{T_{\mathsf{WaitPermission}}})$ represents a proof of "silence" $\pi_\epsilon = (\mathbf{b}_1, \ldots, \mathbf{b}_{T_{\mathsf{WaitPermission}}})$ that $\mathcal{C}$ will use when querying the token.
2. Else, if within these blocks there is a transaction π_{ans} signed by alert-info denying $\tau_{\mathsf{alert\text{-}info}}$, this transaction will be the proof of denied permission $\pi = (\tau_{\mathsf{alert\text{-}info}}, \mathbf{b}_1, \ldots, \mathbf{b}_{T_{\mathsf{WaitPermission}}}, \pi_{\mathsf{ans}})$.
3. Else, if transaction π_{ans} is endorsing $\tau_{\mathsf{alert\text{-}info}}$, then such transaction alone will be the proof of permission $\pi = \pi_{\mathsf{ans}}$.

Note that the token is **not** connected to the blockchain, and it does **not** check any transaction. The blockchain transactions are checked only by the parties who will check the permission in case of a dispute. The advantage of a blockchain-based implementation is that it is decentralized, therefore the validity of the permission does not depend on any third party. The downside however is that the permission request must be posted on the blockchain, therefore revealing some information about the fact that a user of a certain cloud $\mathcal{C}$ lost her key.

Implementation with a (Trusted) Email Provider. $\mathcal{G}_{\mathsf{perm}}$ can also be implemented simply using an email system, and it requires the collaboration of the email service provider. In this case, the email provider is a trusted third party between the user and the cloud. Procedure $(\mathsf{register}, \mathsf{alert\text{-}info}, \mathcal{U}_i)$ consists in having the user register an email address alert-info that will be used for break-glass communications.

To make a break-glass request, $\mathcal{U}_i$, who potentially lost all the keys, and therefore also the password to access to the email address alert-info, will send to $\mathcal{C}$ a break-glass request (via a web-form, for example, as above).

Procedure $(\mathsf{CreatePermission}, \mathsf{alert\text{-}info}, \mathcal{U}_i)$ is implemented by having the cloud sending an email to the address alert-info with the detailed information about the break-glass request received for $\mathcal{U}_i$. If the cloud does not receive any reply after a period of $T_{\mathsf{WaitPermission}}$, it will proceed with the request. The proof π_ϵ for not having received a reply would require the intervention of the email providers of both user and cloud. A proof of valid permission is simply the email sent by address $\mathsf{alert\text{-}info}_i$ to the cloud, authorizing the procedure. Similarly, a proof of denied permission, is the email sent by address $\mathsf{alert\text{-}info}_i$ to the cloud, denying the permission.

5 Construction

A break-glass encryption scheme is defined by two procedures: the user's procedure, described in Figs. 3 and 4, and the cloud's procedure, described in Fig. 6. The cloud's procedure consists in interacting with the token $\mathcal{T}$, the token's algorithm is described in Fig. 5. We assume that the token behaves like the ideal token functionality $\mathcal{F}_{\mathsf{wrap}}$ [Kat07] (described in Fig. 10). However, for simplicity of notation we do not use the ideal functionality interface. Also, we assume that all communications are carried over authenticated channels.

In the following we describe user's procedures. $\mathcal{C}$'s procedure and $\mathcal{T}$'s procedure follow naturally.

5.1 User's Procedures

Procedure $\mathsf{Setup}(1^\lambda)$**.** $\mathcal{U}$'s procedure starts with a one-time initialization step when the token $\mathcal{T}$ is prepared. $\mathcal{U}$ generates a secret key k for the symmetric-key encryption scheme and keys for the signature scheme $(\mathsf{vk}_\mathcal{T}, \mathsf{ssk}_\mathcal{T})$. Key k is used to encrypt the data; the token uses this key to decrypt and re-encrypt the ciphertexts. Signing keys $(\mathsf{vk}_\mathcal{T}, \mathsf{ssk}_\mathcal{T})$ are used by the token to authenticate its outputs. Hence, the token is initialized with secret keys $(k, \mathsf{ssk}_\mathcal{T})$, the current time, and a parameter I denoting the window of time within which the ciphertext is considered valid. In this step, the user also register his alert address $\mathsf{alert\text{-}info}$ and the identity of the party she wants to authorize (i.e., $\mathcal{C}$) to the $\mathcal{G}_\mathsf{perm}$ functionality. Namely $\mathcal{U}$ sends $(\mathsf{register}, \mathsf{alert\text{-}info}, \mathcal{U}, \mathcal{C})$ to $\mathcal{G}_\mathsf{perm}$.

Procedure $\mathsf{Upload}()$**.** The second step for the user is to upload her data. We represent the data as a vector of l blocks (l can be very large). The user will encrypt each block, adding some bookkeeping information. The encryption of the i-th block will have the following format: $\mathsf{ctx}_i = \mathsf{Enc}_k(m_i||\mathsf{bookkeep}||\mathsf{perm})$ where:

- m_i is the message,
- $\mathsf{bookkeep} = [t_i, T_0, T_i]$ contains the bookkeeping information, keeping track of the time of last update, and time of break-glass operations. Specifically:
  - t_i is the time when ciphertext c_i was last updated. This time is used to defeat replay attacks.
  - T_0 is a global value (i.e., it is the same for all ciphertexts) and indicates the time when the first break-glass was performed. Adding this information allows the user to know that a break-glass has happened at least once (without needing to query the token).
  - T_i is the time when the i-th ciphertext was broken. This information allows fine-grained information about which ciphertexts have been compromised and when.
- $\mathsf{perm} = [\mathsf{alert\text{-}info}, \pi_\mathsf{perm}, pk, \sigma_\mathcal{C}]$ will contain the info about the break-glass permission (if any) generated by the cloud. This field is empty in normal circumstances. Specifically, $(\mathsf{alert\text{-}info}, \pi_\mathsf{perm})$ is the actual proof of permission obtained by the cloud – it can be empty if the cloud performs an illegitimate break-glass; pk is the public key used to encrypt the result of the decryption (when the break is legitimate, this ensures that only the client choosing pk will be able to decrypt the result of the decryption). Finally, $\sigma_\mathcal{C}$ is the signature computed by $\mathcal{C}$. This signature is necessary to hold the cloud accountable of invoking the break-glass procedure.

Procedure $\mathsf{Get}(i, k)$ is used to retrieve the i-th ciphertext. The cloud could refuse to send the ciphertext. If this happens, the user will consider this as a cheating behaviour and will accuse the cloud. The network data

can be used as evidence that the cloud received the request but did not fulfill it[2]. If the cloud replies with ciphertext c_i, $\mathcal{U}$ will decrypt it and obtain bookkeeping information: $\mathsf{bookkeep} = [t_i, T_0, T_i]$ and permission information $\mathsf{perm} = (\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}}, pk, \sigma_{\mathcal{C}})$.

$\mathcal{U}$ first checks the following:

1. Case 1. Stale Ciphertext. If $t_i < t - I$, this means that the ciphertext is not updated. Thus, the cloud replied with an older version of the ciphertext, perhaps to hide the fact that the updated ciphertext would have been marked with a information about an illegitimate break. A stale ciphertext triggers a red flag, and the user will use this communication and the network data as an evidence of cheating.
2. Case 2. Unauthorized break. If $T_0 \neq 0$ (recall that T_0 denotes the time the first break occurred) but the user never requested/approved a break-glass procedure then the user $\mathcal{U}$ will us the $\sigma_{\mathcal{C}}$ computed on a wrong or empty π_{perm} information, as a proof of cheating, and she invokes procedure $\mathsf{CloudCheating}(\mathsf{perm}, t)$. (Indeed, since the user did not approve any permission on $\mathcal{G}_{\mathsf{perm}}$ there exists no valid pair $(\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}})$ that could justify the break-glass action performed by $\mathcal{C}$).
3. Case 3. Unauthorized break of i-th ciphertext. If $T_i \neq 0$ (recall that T_i denotes the time when ciphertext c_i was broken) but the user never asked to break ciphertext c_i, $\mathcal{U}$ proceeds as in Step 2.

Else, if none of the conditions above is satisfied, there were no illegitimate breaks, and the user simply outputs the decrypted plaintext m_i.

Procedure $\mathsf{Break}(i, \mathsf{alert\text{-}info})$**.** This procedure is invoked by *any party* who would like to break ciphertext c_i. A break-glass procedure starts with a party sending a request to the cloud $\mathcal{C}$. The request has the following info: $(\mathsf{break}, i, \mathsf{alert\text{-}info})$ (recall that $\mathsf{alert\text{-}info}$ is the address used to alert user $\mathcal{U}$). On receiving such request, the cloud $\mathcal{C}$ will send a request to $\mathcal{G}_{\mathsf{perm}}$ to obtain a proof of permission. Namely, $\mathcal{C}$ sends $(\mathsf{CreatePermission}, \mathsf{alert\text{-}info}, \mathcal{U})$ to $\mathcal{G}_{\mathsf{perm}}$.

The functionality $\mathcal{G}_{\mathsf{perm}}$ will then send an alert to the actual user $\mathcal{U}$ by sending $(\mathsf{permission\text{-}request}, \mathcal{C})$ to $\mathcal{U}$. At this point the user can entire compute a proof π to endorse/deny the request by sending $(\mathsf{ack\text{-}check}, (\mathsf{alert\text{-}info}, \mathcal{U}_i), yes/no, \pi)$ to $\mathcal{G}_{\mathsf{perm}}$; or she can not respond at all, triggering the generation of a "proof of silence" π_ϵ. The cloud will then obtain $(\texttt{granted}, \pi_\epsilon)$ or $(\texttt{granted}, \pi)$ in case the permission is granted, or (denied, π) in case the permission is denied. If granted, the cloud will use proof π_ϵ or π as input to the token $\mathcal{T}$ in the break procedure.

Below we provide a table for the notation used in the procedures.

[2] We do not formally cover this cheating case, as it requires formalization of the network interface, which is outside the scope of this work.

I	Maximum time between two updates
$T_{\mathsf{WaitPermission}}$	Time waited before providing a silence proof
T_0	Time when the first Break has been received by $\mathcal{T}$
T_i	Time when c_i was broken
bookkeep	contains t_i, T_0, T_i
perm	contains alert-info, $\pi_{\mathsf{perm}}, pk, \sigma_C$
alert-info	Used to notify a user of a break-glass request
π_{perm}	Equal to either π_ϵ or $\pi_{\mathcal{U}_i}$ or $\perp$
π_ϵ	Proof of silence

6 Security Proof

Theorem 1. *Assume* (KeyGen, Enc, Dec) *is an INT-CTXT NM CPA-secure encryption scheme (Definition in Fig. 9 [BN08]), (*PKGen, PKEnc, PKDec) *is a CPA-secure public key encryption scheme,* (GenSignKey, Sign, Verify) *is a EUF-CMA secure signature scheme; assume that all communications are carried over authenticated channels. Then the scheme described in Figs. 3, 4, 5 and 6 securely realize the* $\mathcal{F}_{\mathsf{break}}$ *functionality in the* $(\mathcal{G}_{\mathsf{perm}}, \mathcal{G}_{\mathsf{clock}}, \mathcal{F}_{\mathsf{wrap}})$*-hybrid model.*

6.1 Case Malicious Cloud

The proof consists in showing a PPT simulator Sim that generates the view of a malicious cloud $\mathcal{C}^*$ while only having access to $\mathcal{F}_{\mathsf{break}}$ (Fig. 1), and an indistinguishability proof that the transcript generated by the simulator is indistinguishable from the output generated by the cloud in the real world execution.

Simulator. Sim has blackbox access to $\mathcal{C}^*$ and interacts with $\mathcal{F}_{\mathsf{break}}$ in the ideal world. The ideal functionality $\mathcal{G}_{\mathsf{clock}}$ is used by both the environment and Sim to get the current time, and $\mathcal{G}_{\mathsf{perm}}$ is used to get/validate a permission to break-glass. The simulator also simulates the $\mathcal{F}_{\mathsf{wrap}}$ functionality to $\mathcal{C}^*$. Sim is described in Fig. 7.

Informally, the goal of the simulator is to (1) simulate the ciphertexts without knowing the messages uploaded by the user, and (2) to correctly intercept the break-glass requests coming from the malicious cloud (Sim obtains the legitimate break-glass procedure requests from $\mathcal{G}_{\mathsf{perm}}$ via the command (Permission, alert-info, π)). The ciphertexts are simulated as encryptions of 0. Due to the INT-CTXT NM CPA security property of the underlying encryption scheme, and the tamper-proof property of hardware tokens (modeled as an ideal blackbox by $\mathcal{F}_{\mathsf{wrap}}$) this difference cannot be detected by the malicious cloud. $\mathcal{G}_{\mathsf{perm}}$ guarantees that a permission cannot be fabricated on behalf of $\mathcal{U}_i$ (if $\mathcal{U}_i$ is honest), thus an illegitimate break-glass procedure can be detected by observing the queries made to the token that have an invalid perm-proof field.

User Procedures I

Cryptographic Primitive Used.
$\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$: INT-CTXT NM-CPA secure encryption scheme.
$\Sigma = (\mathsf{GenSignKey}, \mathsf{Sign}, \mathsf{Verify})$: a EUF-CMA digital signature scheme.

Parameters.
– I: denotes the frequency with which the ciphertexts need to be updated.
– $\mathsf{vpk}_{\mathcal{C}}$ is the public key of the cloud $\mathcal{C}$.
– alert-info: **public** alert info used for requesting permission via $\mathcal{G}_{\mathsf{perm}}$.

External Functionalities: $\mathcal{G}_{\mathsf{clock}}$ **and** $\mathcal{G}_{\mathsf{perm}}$**.**

Procedure $\mathsf{Setup}(1^\lambda)$

- Generate key for encryption of the data: $k \leftarrow \Pi.\mathsf{KeyGen}(1^\lambda)$;
- Generate signature keys for token: $(\mathsf{vk}_{\mathcal{T}}, \mathsf{ssk}_{\mathcal{T}}) \leftarrow \Sigma.\mathsf{GenSignKey}(1^\lambda)$.
- Initialize token $\mathcal{T}$ with encryption key k, signature key $\mathsf{ssk}_{\mathcal{T}}$, cloud's public key $\mathsf{vpk}_{\mathcal{C}}$, interval I and $\mathsf{mytime} := t$, where t is the current time from $\mathcal{G}_{\mathsf{clock}}$. $\mathcal{T}$'s procedure is described in Figure 5.
- Send $\mathcal{T}$ to the cloud $\mathcal{C}$, publish verification key $\mathsf{vk}_{\mathcal{T}}$ to a public repository $\mathcal{D}$.
- Register with $\mathcal{G}_{\mathsf{perm}}$: send $(\mathsf{register}, \mathsf{alert\text{-}info}, \mathcal{U}, \mathcal{C})$ to $\mathcal{G}_{\mathsf{perm}}$.

Procedure $\mathsf{Upload}(M)$

- Parse $M = (m_1, \ldots, m_l)$.
- Encrypt each block m_j: $\mathsf{ctx}_j = \mathsf{Enc}_k(m_j||\mathsf{bookkeep}||\mathsf{perm})$ for $j \in \{1, \ldots, l\}$, where $\mathsf{bookkeep} := [t, 0, 0]$ and $\mathsf{perm} = [\perp, \perp, \perp, \perp]$.
- Send $(\mathsf{ctx}_j)_{j \in [l]}$ to $\mathcal{C}$.

Procedure $\mathsf{Get}(i, k)$
Get current time $time$ from $\mathcal{G}_{\mathsf{clock}}$.

- (Download ciphertext) Send command $\mathsf{Get}(i, time)$ to $\mathcal{C}$. If $\mathcal{C}$ does not respond, or responds with an invalid ciphertext then output $(\texttt{refuse}, time)$ and halt.
- Else, let c_i be ciphertext received from $\mathcal{C}$ and let $(m||\mathsf{bookkeep}||\mathsf{perm}) := \mathsf{Dec}(k, c_i)$. Parse $\mathsf{bookkeep} = [t_i||T_0||T_i]$ and $\mathsf{perm} = (v_1, v_2, pk, \sigma_{\mathcal{C}})$, and perform the following checks.
  1. BAD CASES:
     - (Stale ciphertext) If $t_i < time - I$. This means that the ciphertext was not updated, and considered as potential cheating attempt without immediate proof, hence output $(\texttt{refuse}, time)$.
     - (Unauthorized Break) . If $(T_0 \neq 0 \wedge \mathsf{Break}(\cdot, \mathsf{alert\text{-}info}))$ was never called before, OR if $(T_i \neq 0 \wedge \mathsf{Break}(i, \mathsf{alert\text{-}info}))$ was never called before, then:
       * If v_1, v_2 is not a valid permission, then set $x = (i, T_i, pk)$ and construct proof $\pi = (x, \sigma_{\mathcal{C}})$. Call procedure $\mathsf{CloudCheating}(\pi, time)$ (Described in Fig. 4).
       * If v_1, v_2 is a valid permission, then output "$\mathcal{G}_{\mathsf{perm}}$ `failure`".
  2. GOOD CASE. (No illegitimate break) Else if $t_i \in [time \pm I]$ output m_i.

Fig. 3. User procedures 1

User Procedures II

Cryptographic Primitive Used.

$\Pi = (\mathsf{PKGen}, \mathsf{PKEnc}, \mathsf{PKDec})$: CPA-secure Public Key encryption scheme.

Procedure $\mathsf{Break}(i, \mathsf{alert\text{-}info})$

Get current time $time$ from $\mathcal{G}_{\mathsf{clock}}$. Send $(\mathsf{CreatePermission}, \mathsf{alert\text{-}info}, \mathcal{U}_i)$ to $\mathcal{G}_{\mathsf{perm}}$. Set $\mathsf{T}_{\mathsf{break}} = time$. Then:

- Generate fresh keys $(pk', sk') \leftarrow \mathsf{PKGen}(1^\lambda)$.
- Send break-glass request. Send command $(\mathsf{break}, i, \mathcal{U}, \mathsf{alert\text{-}info}, pk')$ to $\mathcal{C}$. If $\mathcal{C}$ does not respond after more than $T_{\mathsf{WaitPermission}} + \delta$ steps then output $(\mathtt{refuse}, time)$.
- Check authenticity of the answer. Upon receiving $(c_{\mathsf{break}}, input, \sigma_{\mathcal{C}}, \sigma_i)$ from $\mathcal{C}$. For $input = [T_i, \mathsf{alert\text{-}info}, \pi_{\mathsf{perm}}, pk'])$, let $x = (c_{\mathsf{break}}, input, \sigma_{\mathcal{C}})$. Check that $\mathsf{Verify}_{\mathsf{vk}_{\mathcal{T}}}(x, \sigma_i) = 1$ and $T_i = time$. If not, output $(\mathtt{refuse}, time)$. Else, recover $(m_i || \ \mathsf{bookkeep} || \ \mathsf{VerifyPerm}) \leftarrow \mathsf{PKDec}(sk', c_{\mathsf{break}})$ and proceeds with the checks as in Procedure $\mathsf{Get}(\cdot, \cdot)$.

Procedure $\mathsf{Update}(i, m', k)$

Get current time $time$ from $\mathcal{G}_{\mathsf{clock}}$.

- Run $\mathsf{Get}(i, k)$. If the output is OK continue.
- Send the new ciphertext. Send $c_i' = \mathsf{Enc}_k(m' || \mathsf{bookkeep}' || \mathsf{perm}')$ to $\mathcal{C}$, where $\mathsf{bookkeep}' = (time || 0 || 0)$ and $\mathsf{perm}' = (\perp, \perp, \perp, \perp)$.

Procedure $\mathsf{CloudCheating}(\pi, time)$

Parse $\pi = (x, \sigma_{\mathcal{C}})$. If $\mathsf{Verify}(\mathsf{vpk}_{\mathcal{C}}, x, \sigma_{\mathcal{C}}) = 1$ Accuse $\mathcal{C}$ of cheating with proof $\pi, time$.

Interaction with $\mathcal{G}_{\mathsf{perm}}$

Upon receiving $(\mathsf{CreatePermission}, \mathsf{alert\text{-}info})$ from $\mathcal{G}_{\mathsf{perm}}$. Get current time $time$ from $\mathcal{G}_{\mathsf{clock}}$. Let δ a time interval depending on the implementation of $\mathcal{G}_{\mathsf{perm}}$.

- If $time = \mathsf{T}_{\mathsf{break}} \pm \delta$, then endorse request and send $(\mathsf{ack\text{-}check}, (\mathsf{alert\text{-}info}, \mathcal{U}_i), YES)$ to $\mathcal{G}_{\mathsf{perm}}$.
- $time = \mathsf{T}_{\mathsf{break}} \pm \delta$ but secrets are lost do nothing.
- Else, deny the request: send $(\mathsf{ack\text{-}check}, (\mathsf{alert\text{-}info}, \mathcal{U}_i), NO)$ to $\mathcal{G}_{\mathsf{perm}}$.
- If $time \neq \mathsf{T}_{\mathsf{break}} \pm \delta$ but secrets are lost, then output: `Failure to Stop Illegitimate Request.`

Fig. 4. User procedures 2

Indistinguishability Proof. *Overview.* We start by outlining the differences between the view of $\mathcal{C}^*$ in the ideal world and in the real world. The view of $\mathcal{C}^*$ consists in the initial set of ciphertexts $(\mathsf{ctx}_i^0)_{i \in [l]}$, and the output computed by the token $\mathcal{T}$. The crucial differences between the views in the two worlds are:

- Encryptions. In the real world $\mathcal{C}^*$ will observe correct encryptions of messages of the form $(m || \mathsf{bookkeep} || \mathsf{perm})$. Instead, in the ideal world, the ciphertexts are only encryptions of 0. The indistinguishability of the two set of encryptions intuitively follows from the CPA security of the underlying encryption scheme.

TOKEN PROCEDURE $\mathcal{T}$

Hardwired Values. Encryption key k, signing key $\mathsf{ssk}_{\mathcal{T}}$.

Variables.
List of the last κ broken ciphertexts $\mathcal{L}$.
Current time mytime. Set $\sigma^0_{\mathcal{C}} = \epsilon$

On input:$(\mathsf{CMD}, c, i, \mathsf{external\text{-}time}, \mathsf{perm}')$.

0. Check that i is not in the breaking list: If $i \in \mathcal{L}$ do nothing.

1. Check and Update time.
If external-time $\leq$ mytime output $\perp$ (The cloud is querying with a time that is too far in the past). Else, update current time: set mytime := external-time.

2. Decrypt.
Decrypt c using key k and obtain $(m||\mathsf{bookkeep}||\mathsf{perm})$. If decryption fails, do nothing. Let $\mathsf{bookkeep} = (t_i||T_0||T_i)$.

- (Already Broken) If $T_i \neq 0$ then halt and output $\perp$. (This ciphertext is already broken. No re-encryption required)
- (Stale) If $t_i <$ mytime $- I$ then halt and output $\perp$.
- (Error) If $t_i >$ mytime$+I$ then halt and output output "Error, someone encrypted under my key?" and stop.

3. Execute Command CMD.

Break If CMD =`break`.
0. Parse $\mathsf{perm}' = (\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}}, pk, \sigma_{\mathcal{C}})$.
1. Checks: If $\mathsf{Verify}_{\mathsf{vpk}_{\mathcal{C}}}(i||\mathsf{external\text{-}time}||\mathsf{perm}', \sigma_{\mathcal{C}},) = 1$ continues. (Add i to the list of recently broken ciphertext). Unqueue $\mathcal{L}$, then add i to $\mathcal{L}$. Else, ignore the request.
2. Case: First Break. If $T_0 = 0$ then set $T_0 =$ mytime, set $\sigma^0_{\mathcal{C}} = \sigma_{\mathcal{C}}$.
3. Re-encrypt using the fresh key pk. $c_{\mathsf{break}} \leftarrow \mathsf{PKEnc}(pk, m_i, \mathsf{bookkeep})$.
4. Authenticate the break-info: $\sigma_{\mathcal{T}} = \mathsf{Sign}_{\mathsf{ssk}_{\mathcal{T}}}(c_{\mathsf{break}}, \mathsf{perm\text{-}proof}, \sigma_{\mathcal{C}})$.
5. Mark the i-th ciphertext as broken.
   (a) Update $\mathsf{bookkeep} = (\mathsf{mytime}, T_0, \mathsf{mytime})$.
   (b) Compute $c'_i = \mathsf{Enc}_k(c_{\mathsf{break}}, \mathsf{bookkeep}, \mathsf{perm}')$. (Note. This ciphertext will never be re-encrypted again).
6. Output c_{break}, perm-proof, $\sigma_{\mathcal{C}}, \sigma_{\mathcal{T}}, c'_i$.

Re-encryption. If CMD = `Reencrypt`.
Set $\mathsf{bookkeep} = (\mathsf{mytime}, T_0, 0)$. Set $\mathsf{perm} = (\perp, \perp, \perp, \sigma^0_{\mathcal{C}})$.
Output $c'_i = \mathsf{Enc}_k(m||\mathsf{bookkeep}||\mathsf{perm})$.

Fig. 5. The token procedure

- Token's functionality. In the real world, the token will accept any valid encryption provided in input. Namely, on input a ciphertext c, the token will first try to decrypt with its secret key, and if the decryption is successful will proceed with the necessary steps. Instead, in the ideal word, the simulated

CLOUD PROCEDURES

Parameters.
I: denotes the frequency with which the ciphertexts need to be updated.

Private Input. Signing key: $\mathsf{ssk}_{\mathcal{C}}$. **External Functionalities:** $\mathcal{G}_{\mathsf{clock}}$ **and** $\mathcal{G}_{\mathsf{perm}}$.

Setup for user $\mathcal{U}_i$.
Upon receiving $\mathcal{T}_{\mathcal{U}}, \mathsf{vk}_{\mathcal{T}}, (\mathsf{ctx}_i)_{i\in[l]}$ from user $\mathcal{U}$:

- Store ciphertexts $(\mathsf{ctx}_i)_{i\in[l]}$ and user's verification key $\mathsf{vk}_{\mathcal{T}}$.
- Activate Maintenance procedure for $\mathcal{U}$.

Procedure $\mathsf{Maintenance}(\mathsf{ctx}_i, \mathcal{T}_{\mathcal{U}}, \mathsf{vk}_{\mathcal{T}})$
Get $time$ from $\mathcal{G}_{\mathsf{clock}}$.
Every I steps: query $\mathcal{T}(c_i, i, \mathtt{Reencrypt}, time, \perp)$ and obtain c_i^{new}. Replace $c_i := c_i^{new}$, $\forall i \in [l]$.

Answering User's requests.

- Get. Upon receiving $\mathsf{Get}(i, t)$. Get current time: $time \leftarrow \mathcal{G}_{\mathsf{clock}}(\mathsf{clockread})$. If $t \in [time \pm \delta]$ then reply with c_i.
- Break. Upon receiving $(\mathsf{break}, i, \mathcal{U}, \mathsf{alert\text{-}info}, pk')$.
  1. Send $(\mathsf{GetPermission}, \mathsf{alert\text{-}info}, \mathcal{U}, \mathcal{C})$ to $\mathcal{G}_{\mathsf{perm}}$. If $\mathcal{G}_{\mathsf{perm}}$ outputs $(\mathtt{granted}, \pi_{\mathsf{perm}})$. compute $\sigma_{\mathcal{C}} = \mathsf{Sign}_{\mathsf{ssk}_{\mathcal{C}}}(i, time, \mathsf{alert\text{-}info}, \pi_{\mathsf{perm}}, pk')$ (else, do nothing).
  2. Query token $\mathcal{T}(\mathsf{Break}, i, c_i, time, \mathsf{perm})$, where $\mathsf{perm}=(\mathsf{alert\text{-}info}, \pi_{\mathsf{perm}}, pk', \sigma_{\mathcal{C}})$ and forward the answer to $\mathcal{U}$.

Fig. 6. Cloud procedure

token accepts only encryptions that were computed by the simulator itself. In other words, if the cloud is able to compute a ciphertext that is valid in the real world and accepted by the real token, this ciphertext will not be accepted by the simulated token. Similarly, in the Get functionality, a real user would accept any valid ciphertext that $\mathcal{C}^*$ provides, instead the simulated user would abort if a valid ciphertext was not computed by the simulated token. The indistinguishability between the two worlds follows from the integrity ciphertext property INT-CTXT NM CPA security defined by Bellare and Namprempre in [BN08], which we report in Fig. 9.

– Break invocation. Recall, there are two types of break requests. The ones generated by the user, and the ones generated by the cloud. The simulator obtains the user requests directly from $\mathcal{F}_{\mathsf{break}}$, and will forward them to the adversary $\mathcal{C}^*$. The main task of the simulator however is to identify the break requests that are *initiated by the cloud*. Since the cloud must interact with the token in order to successfully decrypt a ciphertext[3] the simulator will

[3] To see why, note that, besides the access to the token, a cloud only has a list of ciphertexts. The output of the token is either a ciphertext, or a message m, but no other information about the secret key is given in output. Thus, if a cloud is able to decrypt a ciphertext, without calling the break command, this cloud is violating the CPA-security of the ciphertext.

Simulator Sim

Upload and Initialization. Upon receiving request $(\mathsf{uploaded}, \mathsf{sid}, l, \mathcal{U})$ from $\mathcal{F}_{\mathsf{break}}$ do:

- Generate key for encryption $k \leftarrow \Pi.\mathsf{KeyGen}(1^\lambda)$ and prepare ciphertexts: $[\mathsf{ctx}_1^0, \ldots, \mathsf{ctx}_l^0]$ where $\mathsf{ctx}_i^0 = \mathsf{Enc}_k(0^{p(\lambda)})$.
- Generate signature keys for token: $(\mathsf{vk}_{\mathcal{T}}, \mathsf{ssk}_{\mathcal{T}}) \leftarrow \Sigma.\mathsf{GenSignKey}(1^\lambda)$.
- Get the initial time from $\mathcal{G}_{\mathsf{clock}}$ and store it in variable $\mathsf{tkntime}$.
- Initialize matrices $L, \mathcal{L}_{\mathsf{sign}}, \mathcal{B}$. L stores the ciphertexts computed by Sim, $\mathcal{L}_{\mathsf{sign}}$ stores the signatures computed by $\mathsf{Sim}_{\mathcal{T}}$, $\mathcal{B}$ stores the ciphertexts that have been broken. We denote by $L[i] = (c_0^i, t_0), (c_1^i, t_1), \ldots$ the list of ciphertexts generated for the i-th element. At the beginning, $L[i] := [\mathsf{ctx}_i^0, \mathsf{tkntime}]$
  $\mathcal{L}_{\mathsf{sign}}$ contains the signatures computed by the token.
- Send $\mathsf{vk}_{\mathcal{T}}, [\mathsf{ctx}_1^0, \ldots, \mathsf{ctx}_l^0]$ to $\mathcal{C}^*$.

Update. Upon receiving $(\mathsf{update}, \mathsf{sid}, i)$ from $\mathcal{F}_{\mathsf{break}}$. Get current time: $time \leftarrow \mathcal{G}_{\mathsf{clock}}(\mathsf{clockread})$.
First, send $\mathsf{Get}(i, time)$ to $\mathcal{C}^*$. If no response is received then send $(\mathsf{ack\text{-}updated}, \mathsf{sid}, \mathcal{U}, NO)$ to $\mathcal{F}_{\mathsf{break}}$. Else, let c_i' be the ciphertext received from $\mathcal{C}^*$. Analyse c_i' as follows:

- Bad Cases.
  1. (Case: Broken ciphertext) If $c_i' \in \mathcal{B}$ then do nothing.
  2. (Case: Wrong ciphertext) If $c_i' \notin L$ and decryption fails, then send $(\mathsf{ack\text{-}updated}, \mathsf{sid}, \mathcal{U}, NO)$.
  3. (Case: Stale ciphertext) If there exists a pair $(c_i', t') \in L[i, t']$ but $t' < time - I$ then send $(\mathsf{ack\text{-}updated}, \mathsf{sid}, \mathcal{U}, NO)$.
  4. (Failure Case: Good ciphertext not provided by the simulated token) If $c_i' \notin L$ and $\mathsf{Dec}(k, c_i') \neq \perp$ then output `Integrity Encryption Failure` and stop.
- Good cases. If there exists pair $(c_i', t) \in L$ s.t. $t' \in [time - I]$ then send $(\mathsf{ack\text{-}updated}, \mathsf{sid}, \mathcal{U}, yes)$ to $\mathcal{F}_{\mathsf{break}}$. Then compute $c'' \leftarrow \mathsf{Enc}_k(0)$, add $(c'', time)$ to $L[i]$ and finally send the updated ciphertext c'' to $\mathcal{C}^*$.

User's Initiated Break Upon receiving $(\mathsf{Permission}, \mathsf{sid}, \mathsf{alert\text{-}info}, \pi, ans)$ from $\mathcal{G}_{\mathsf{perm}}$. If $ans = no$, record π in a list of denied permissions $\mathsf{DeniedList}$. Else, if $ans = 1$ continue with the break-glass procedure as an honest user.

1. $(\mathsf{break}, \mathsf{sid}, i, \mathsf{perm\text{-}proof} = \pi, \mathsf{cheat\text{-}proof} = \perp)$
2. Get current time: $time \leftarrow \mathcal{G}_{\mathsf{clock}}(\mathsf{clockread})$. Store $(\mathsf{user\text{-}break}, time)$.
3. Generate fresh keys $(pk', sk') \leftarrow \mathsf{PKGen}(1^\lambda)$.
4. Send $(\mathsf{break}, i, \mathsf{alert\text{-}info}, pk')$ to $\mathcal{C}^*$.
5. Upon receiving response ans from $\mathcal{C}^*$ do.
   - $\mathcal{C}^*$ **refuses to collaborate** If $ans = \perp$ then send $(\mathsf{ack\text{-}break}, \mathsf{sid}, \mathcal{U}, NO)$ to $\mathcal{F}_{\mathsf{break}}$.
   - $\mathcal{C}^*$ **gives** (x, σ)**.** Parse $x = (c_{\mathsf{break}}, c_i, \mathsf{external\text{-}time}, \mathsf{alert\text{-}info}, \pi, pk, \sigma_{\mathcal{C}})$.
     (a) Good signature. If $(x, \sigma) \in \mathcal{L}_{\mathsf{sign}}[\mathsf{external\text{-}time}]$ then send $(\mathsf{ack\text{-}break}, \mathsf{sid}, \mathcal{U}, yes)$ to $\mathcal{F}_{\mathsf{break}}$
     (b) Forgery. If σ verifies on x, but $(x, \sigma) \notin \mathcal{L}_{\mathsf{sign}}[\mathsf{external\text{-}time}]$ then output `Forgery Failure` and halt.

Retrieve Upon receiving $(\mathsf{retrieve\text{-}request}, \mathsf{sid}, i, \mathcal{U})$ at time $time$, send $(\mathsf{Get}, i, time, \sigma_{\mathcal{U}})$ to $\mathcal{C}^*$.

- If $\mathcal{C}^*$ sends $\perp$ then send $(\mathsf{ack\text{-}retrieve}, \mathsf{sid}, i, \mathcal{U}, no)$.
- Else, let c^* be the ciphertext sent by the cloud. Let $t \in [time - \delta, time + \delta]$.
  - If there exists $(c^*, t) \in L[i]$ then send $(\mathsf{ack} - \mathsf{inquire}, \mathsf{sid}, \mathcal{U}, yes)$.
  - Else, send $(\mathsf{ack} - \mathsf{retrieve}, \mathsf{sid}, \mathcal{U}, no)$ to $\mathcal{F}_{\mathsf{break}}$.

Fig. 7. Simulator

Token simulation $\mathsf{Sim}_{\mathcal{T}}$.
$\mathcal{B}$ stores the ciphertexts that have been broken.
On input $(\mathsf{CMD}, c, i, \mathsf{alert\text{-}info}, \mathsf{external\text{-}time}, \pi, \sigma_{\mathcal{C}})$:
0. Check broken list If $c \in \mathcal{B}|_{|I|}$ then do nothing.
1. Check time and Ciphertext Validity. If $\mathsf{mytime} < \mathsf{external\text{-}time}$ then update $\mathsf{mytime} = \mathsf{external\text{-}time}$

1. (Stale Ciphertext) If $(c, j) \in L[i]$ but $j \notin [\mathsf{mytime} \pm I]$ then do nothing.
2. (Invalid iphertext c) If there is no $(c, j) \in L[i]$ do nothing.
3. (Forged Ciphertext) If there is no $(c, j) \in L[i]$ but $\mathsf{Dec}_k(c) \neq \perp$ then output `Integrity Encryption Failure` and halts.

Re-encryption If CMD =`Reencrypt`. Compute $c \leftarrow \mathsf{Enc}_k(0)$ and add $L[i] := [c, \mathsf{tkntime}]$. Output c.
Break CMD =`Break`. Verify signature $\sigma_{\mathcal{C}}$ on input $x = (\mathsf{alert\text{-}info}, \pi, break, i, pk, \mathsf{mytime})$. If check passes do:

1. Detect illegitimate request. If $\pi \neq \perp$ check if it is a legitimate permission by sending $(\mathsf{verify\text{-}permission}, \mathsf{alert\text{-}info}, \pi)$ to $\mathcal{G}_{\mathsf{perm}}$. If $\mathcal{G}_{\mathsf{perm}}$ sends $(\mathsf{sid}, \mathsf{alert\text{-}info}, \mathsf{verifiably-denied}, \pi)$ or $(\mathsf{sid}, \mathsf{alert\text{-}info}, \mathsf{notverified}, \pi)$ then this is a marked as an illegitimate request.
2. Send illegitimate request to $\mathcal{F}_{\mathsf{break}}$. First, set $\mathsf{cheat\text{-}proof} = (\xi, \sigma_{\mathcal{C}})$ and send $(\mathsf{break}, \mathsf{sid}, i, \perp, \mathsf{cheat\text{-}proof})$ and receive m_i.
   (a) Add i to the list of broken ciphertexts: $\mathcal{B} \leftarrow \mathcal{B} \cup i$.
   (b) Set break time: Record $T_i = \mathsf{tkntime}$; (if $T_0 = 0$) Record $T_0 = \mathsf{tkntime}$, record $\sigma_{\mathcal{C}}^0 = \sigma_{\mathcal{C}}$.
   (c) Compute encryption. Set $c_i = \mathsf{Enc}_k(0^{p(n)})$, add $L[i] = (c_i, \mathsf{tkntime})$.
   (d) Compute token's signature. Set σ_i on input $(m_i||c_i||\mathsf{tkntime}||auth||\pi||\sigma_{\mathcal{C}})$ add $\sigma_i \mathcal{L}_{\mathsf{sign}}$.
   (e) Return (m_i, c_i) to $\mathcal{C}^*$
3. (Initiated by User.) Else, send $(\mathsf{ack\text{-}break}, \mathsf{sid}, \mathcal{U}, yes)$ to $\mathcal{F}_{\mathsf{break}}$. Do steps as above, but instead of outputting m_i, output a dummy encryption $c^* = \mathsf{PKEnc}(pk, 0)$.

Fig. 8. Token simulator

use the simulated token to intercept requests that do not have a valid proof of permission and send them to the ideal functionality. Note that at this step, we are using security of $\mathcal{G}_{\mathsf{perm}}$. Namely, we are assuming that a cloud cannot fabricate a valid permission without the help of the user. If this was not the case the simulator could not use the absence of permission to detect illegitimate break-glass requests.

We will prove the above intuition via a sequence of hybrid games.

Hybrid Arguments Overview. We show the following sequence of hybrid experiments. Hybrid H_0 denotes the real world, in hybrid H_1 all ciphertexts generated by the user and the token are collected in a table, and the user's procedure and the token's procedure will accept only ciphertexts in this table (i.e., valid

ciphertexts that are not part of this table are not accepted). Indistinguishability between H_0 and H_1 follows from the INT-CTXT NM CPA Security of the symmetric key encryption scheme. In H_2 and $\bar{H}_2$ we remove the semantic from all the encryptions and simply compute encryptions of 0. Indistinguishability between H_1 and H_2 follows from the CPA security of the underlying symmetric-key encryption scheme. Finally, in H_3 the user accepts only signatures generated by the simulated token, instead of accepting any valid signature. Indistinguishability between H_2 and H_3 follows from the unforgeability of the underlying signature scheme. We assume that all communications between cloud and token are authenticated.

Hybrid H_0. This is the real world experiment. Sim honestly follows the user procedure Figs. 3 and 4, and $\mathcal{T}$'s procedure (Fig. 5).

Hybrid H_1 (Integrity and Non-malleability). This experiment is as H_0 with the only difference that Sim stores the encryptions computed by the user and the token in a matrix L, and token and user accept only encryptions that are in L. If they receive any other encryption that is valid but it is not in L, then the simulated user/token will abort and output `Integrity Encryption Failure`. Note that H_0 and H_1 are different only in the case where $\mathcal{C}^*$ is able to find at a ciphertext c^* that is a valid encryption under secret key k, but it was not computed by the token/user.

In the following lemma we show that probability that $\mathcal{C}^*$ generates such a valid ciphertext is negligible, therefore H_0 and H_1 are computationally indistinguishable.

Lemma 1 (Ciphertext Integrity). *If* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *achieves integrity of ciphertext property (INT-CTX, Fig. 9) then event* `Integrity Encryption Failure` *happens with negligible probability.*

Towards a contradiction, assume that there exists a $\mathcal{C}^*$ such that `Integrity Encryption Failure` happens with non-negligible probability $p(\lambda)$. This means that $\mathcal{C}^*$ queried $\mathsf{Sim}_{\mathcal{T}}$ with a valid ciphertext c^* (i.e., a ciphertext that can be correctly decrypted but it was not compute neither by $\mathsf{Sim}_{\mathcal{T}}$ nor by the user). If this is the case, then we can construct an adversary $\mathcal{A}$ that wins the INT-CTXT game with the same probability, as follows.

Reduction INT-CTX Security. $\mathcal{A}$ playing in experiment $\mathsf{Exp}^{\mathsf{INT-CTX}}$ (Fig. 9), has access to encryption oracle and black-box access to $\mathcal{C}^*$. $\mathcal{A}$ simulates real world experiment to $\mathcal{C}^*$:

- (0) $\mathcal{A}$ plays as the honest user and therefore knows all the plaintexts $m_1, \ldots, m_l$.
- (1) Encryption. To generated ciphertexts on behalf of the token and the user, $\mathcal{A}$ uses its oracle access to Enc, provided by the experiment $\mathsf{Exp}^{\mathsf{INT-CTX}}$. $\mathcal{A}$ collects all the ciphertext generated, together with the plaintext used, in a matrix L'. (This matrix is different from the matrix used by the simulator in that the simulator does not need to remember the correspondent plaintexts).

– (2) Decryption. To decrypt a ciphertexts c provided by the cloud, $\mathcal{A}$ will first check if the ciphertexts are contained in the matrix L'. If $c \notin L'$ then $\mathcal{A}$ will call $\mathsf{VF}(c)$ in $\mathsf{Exp}^{\mathsf{INT-CTX}}$ and obtain answer m. If $m \neq \perp$ then $\mathcal{A}$ wins the game and halts. Else, if $m = \perp$, $\mathcal{A}$ simply continues the reduction, following the honest user and token procedure.

Analysis. Note that $\mathcal{A}$ follows the honest user's procedure and honest token's procedure just like in the H_0. $\mathcal{A}$ will interrupt the reduction and deviate from H_0, only if the cloud provides a ciphertext c that is accepted $VF(c)$ in which case $\mathcal{A}$ simply halts, just like the simulator in H_1. Thus the probability that $\mathcal{A}$ wins the game and halts the reduction, it is closely related to the probability that there is a difference between H_0 and H_1. Since the underlying encryption scheme is assumed to be INT-CTX secure, the probability of $\mathcal{A}$ winning is negligible, consequently, the distributions of transcripts in H_0 and H_1 are distinguishable with negligible probability.

Due to Lemma 1, it follows that probability that $\mathcal{C}^*$ generates such a valid ciphertext is negligible, therefore H_0 and H_1 are computationally indistinguishable.

Hybrid H_2^j $j = 1, \ldots$ (CPA-security). In this sequence of hybrid experiments we change the value encrypted in the j-th ciphertext. Instead of encrypting the actual information $(m||\mathsf{bookkeep}||\mathsf{perm})$ we will encrypt to 0 (but for the sake of the simulation we will still keep record of the plaintexts that should be instead encrypted.) The difference between H_2^j and H_1^{j-1} is that in H_2^j one more ciphertext is computed as encryption of 0. Assume that there is a distinguisher between the two experiments, we will construct an adversary for CPA-security.

Hybrid $\bar{H}_2^j$ for $j = 1, \ldots$ (PK CPA-security). In this sequence of hybrid we replace the encryptions output by the token after a user-triggered break-glass encryption (i.e., c_{break}). Instead of encrypting the actual message m_i, it will encrypt 0. This sequence of hybrid is indistinguishable to the CPA-security of the public key encryption scheme.

Hybrid H_3 (Unforgeability of Token's signature). In this hybrid, the procedure of the simulated user is modified as follows. The simulator (playing as user) accepts only signatures that are in $\mathcal{L}_{\mathsf{sign}}$. When a signature (x^*, σ^*) verifies under $\mathsf{vk}_{\mathcal{T}}$ but $\sigma^* \notin \mathcal{L}_{\mathsf{sign}}$ then the simulated user will output `Forgery Failure` and abort. Therefore, the difference between H_2 and H_3 is that in H_2 a user would accept any signature σ^* that verifies under $\mathsf{vk}_{\mathcal{T}}$ (i.e., $\mathsf{Verify}(\mathsf{vk}_{\mathcal{T}}, x^*, \sigma^*)$), instead in H_3, when a valid signature $\sigma^* \notin \mathcal{L}_{\mathsf{sign}}$ is presented by $\mathcal{C}^*$, the user will abort.

The following lemma shows that the probability that $\mathcal{C}^*$ can compute such a signature is negligible due to the unforgeability of the underlying signature scheme.

Lemma 2. *If* $(\mathsf{GenSignKey}, \mathsf{Sign}, \mathsf{Verify})$ *is a EUF-CMA digital signature scheme, then event* `Forgery Failure` *happens with negligible probability.*

Assume, towards a contradiction, that there exists an adversary $\mathcal{C}^*$ that is able to generate a signature valid σ^* that was not generated by $\mathsf{Sim}_{\mathcal{T}}$ with probability $p(\lambda)$. Thus, we can construct an adversary $\mathcal{A}$ that computes a forgery with the same probability as follows.

Reduction EUF-CMA Security

$\mathcal{A}$ playing in experiment $\mathsf{Exp}^{\mathsf{forge}}$, has oracle access to $\mathcal{C}^*$ and simulates experiment H_2 to $\mathcal{C}^*$ with the following difference:

1. Token Signatures. When the token is required to compute a signature on a message x, $\mathcal{A}$ will forward x to $\mathsf{Exp}^{\mathsf{forge}}$ and obtain signature σ. Add σ to the list $\mathcal{L}_{\mathsf{sign}}$ and set it as the output of the token.
2. Decision. Upon receiving a signature (x^*, σ^*) from $\mathcal{C}^*$, such that $\sigma^* \notin \mathcal{L}_{\mathsf{sign}}$. If (x^*, σ^*) verifies then send σ^* to $\mathsf{Exp}^{\mathsf{forge}}$ and output win.

Analysis. $\mathcal{A}$ wins the forgery game $\mathsf{Exp}^{\mathsf{forge}}$ with the same probability that $\mathcal{C}^*$ computes a valid σ^* and trigger event `Forgery Failure`. Since by assumption the underlying signature scheme is EUF-CMA secure, then probability that $\mathcal{A}$ trigger the above event is negligible.

6.2 Exculpability in Presence of a Malicious User

In the ideal functionality a user obtains a proof to accuse a cloud only if the cloud actually invoked a break command without permission granted from $\mathcal{G}_{\mathsf{perm}}$. In the ideal world there is nothing that the user can do to trigger an accusation against an honest cloud (without violating $\mathcal{G}_{\mathsf{perm}}$).

Instead in the real world, there are several ways the user could accuse an honest cloud. We divide them in four categories: network attack, permission attack, token attack and forgery attack, which we describe below. We show that three of them can be quickly ruled out by definition, while the implausibility of the fourth one can be ruled out by unforgeability property of the underlying signature scheme.

1. Network Attack. A malicious user could accuse the cloud of not responding. This accusation can be challenged by the cloud by having access to logs on the network traffic that guarantees that a correct answer was correctly and timely delivered to the user.
2. Permission Attack. A malicious user could trigger a break-glass procedure, and then accuse the cloud of having fabricated such permission. Since our protocol works in the $\mathcal{G}_{\mathsf{perm}}$-hybrid model, we assume that the procedure for granting permission cannot be counterfeit by anyone.
3. Token Attack. A malicious user could accuse the cloud of not correctly updating the ciphertext. We note however that accusation is not possible since we assume that the token is trusted and will follow the honest procedure. Thus, the cloud will be able to show updated ciphertexts as a proof of honest behaviour.

4. Forgery Attack. A user could accuse an honest cloud by fabricating a valid signature σ that verifies under $\mathsf{vpk}_{\mathcal{C}}$, on a message that contains the word `break` but does not contain any valid authorization received by $\mathcal{G}_{\mathsf{perm}}$. Let us call this event `Sign Forgery Accusation`. We show in Lemma 3 that this events happen with negligible probability.

Permission attack and Token attack are ruled out, since we are assuming to work in the $\mathcal{G}_{\mathsf{perm}}$-hybrid model, and we assume that the token is trusted. For network attacks, we also implicitly assume that there is a way for the cloud to prove that the messages were timely delivered to the user.

Lemma 3. *If* $(\mathsf{GenSignKey}, \mathsf{Sign}, \mathsf{Verify})$ *is a EUF-CMA digital signature scheme, then event* `Sign Forgery Accusation` *happens with negligible probability.*

Assume, towards a contradiction, that there exists a malicious user $\mathcal{U}^*$ that is able to accuse $\mathcal{C}$ by generating a valid signature σ^* that was not generated by $\mathcal{C}$ with probability $p(\lambda)$. Thus, we can construct an adversary $\mathcal{A}$ that computes a forgery with the same probability as follows.

Reduction EUF-CMA Security

$\mathcal{A}$ playing in experiment $\mathsf{Exp}^{\mathsf{forge}}$, has oracle access to $\mathcal{U}^*$ and simulate the cloud to $\mathcal{U}^*$.

1. Protocol Execution. $\mathcal{A}$ receives the token from $\mathcal{U}^*$ and fulfills all the requests received by $\mathcal{U}^*$ by simply following the honest cloud procedure and using the Signature oracle provided by $\mathsf{Exp}^{\mathsf{forge}}$.
2. Accuse. When $\mathcal{U}^*$ sends an accusation on input $\pi = (x, \sigma^*)$, if $\mathsf{Verify}(\mathsf{vpk}_{\mathcal{C}}, x, \sigma^*) = 1$ send π to $\mathsf{Exp}^{\mathsf{forge}}$ and output 1.

Analysis. Since by assumption the underlying signature scheme is EUF-CMA secure, probability of event `Sign Forgery Accusation` is negligible.

Acknowledgments. We thank Laurie Williams for the initial discussion on break-glass encryption, as well as many other insightful conversations. We also thank the anonymous reviewers for their useful comments.

A Additional Security Definitions

Ciphertext Integrity INT-CTX [BN08]. The definition of Cipher Integrity INT-CTX, introduced by Bellare et al. in [BN08] is described in Fig. 9.
Ideal Functionality $\mathcal{F}_{\mathsf{wrap}}$. For completeness we report the ideal $\mathcal{F}_{\mathsf{wrap}}$ functionality in Fig. 10.

INT-CTX NM Experiment

Proc Initialize
$K \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$, $S \leftarrow \emptyset$.
Proc Enc (M)
$C \xleftarrow{\$} \mathsf{Enc}_K(M)$. $S \leftarrow S \cup \{C\}$.
Proc VF(C)
$M \leftarrow \mathsf{Dec}_K(C)$.
If $M \neq \perp$ and $C \notin S$ $\mathsf{win} \leftarrow \mathsf{true}$.
Return $M \neq \perp$.
Proc Finalize
Return win

Fig. 9. INT-CTX game [BN08]

Ideal Functionality $\mathcal{F}_{\mathsf{wrap}}$.
The functionality is parameterized by a polynomial $p(\cdot)$ and an implicit security parameter λ.

Create: Upon receiving an input (create, sid, C, U, M) from a party C (i.e., the token creator), where U is another party (i.e., the token user) and M is an interactive Turing machine, do: If there is no tuple of the form $\langle C, U, \star, \star, \star\rangle$ stored, store $\langle C, U, M, 0, \emptyset, \rangle$. Send $(create, \langle sid, C, U\rangle)$ to the adversary.

Deliver: Upon receiving (READY, $\langle sid, C, U\rangle$) from the adversary, send (READY, $\langle \mathsf{sid}, C, U\rangle$) to U.

Execute: Upon receiving an input (RUN, $\langle \mathsf{sid}, C, U\rangle, msg$) from U, find the unique stored tuple $\langle C, U, M, i, \mathsf{state}\rangle$. If no such tuple exists, do nothing. Otherwise, do: If M has never been used yet (i.e.,$i = 0$), then choose uniform $\omega \in \{0,1\}^{p(\lambda)}$ and set $\mathsf{state} := \omega$. Run $(\mathsf{out}, \mathsf{state}') := M(msg; \mathsf{state})$ for at most $p(\lambda)$ steps where out is the response and state' is the new state of M (set $\mathsf{out} := \perp$ and $\mathsf{state}' := \mathsf{state}$ if M does not respond in the allotted time). Send (RESPONSE,$\langle \mathsf{sid}, C, U\rangle, \mathsf{out}$) to U. Erase $\langle C, U, M, i, \mathsf{state}\rangle$ and store $\langle C, U, M, i+1, \mathsf{state}'\rangle$.

Fig. 10. $\mathcal{F}_{\mathsf{wrap}}$ functionality [Kat07]

References

[ABG+13] Ananth, P., Boneh, D., Garg, S., Sahai, A., Zhandry, M.: Differing-inputs obfuscation and applications. IACR Cryptology ePrint Archive 2013, p. 689 (2013)

[AL07] Aumann, Y., Lindell, Y.: Security against covert adversaries: efficient protocols for realistic adversaries. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 137–156. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_8

[BCP14] Boyle, E., Chung, K.-M., Pass, R.: On extractability obfuscation. In: Lindell, Y. (ed.) TCC 2014. LNCS, vol. 8349, pp. 52–73. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54242-8_3

[BGJ+16] Bitansky, N., Goldwasser, S., Jain, A., Paneth, O., Vaikuntanathan, V., Waters, B.: Time-lock puzzles from randomized encodings. In: Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, 14–16 January 2016, pp. 345–356 (2016)

[BM09] Barak, B., Mahmoody-Ghidary, M.: Merkle puzzles are optimal—an $O(n^2)$-query attack on any key exchange from a random oracle. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 374–390. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03356-8_22

[BM17] Barak, B., Mahmoody-Ghidary, M.: Merkle's key agreement protocol is optimal: an o(n^2) attack on any key agreement from random oracles. J. Cryptol. **30**(3), 699–734 (2017)

[BMTZ17] Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: a composable treatment. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 324–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_11

[BN00] Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 236–254. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44598-6_15

[BN08] Bellare, M., Namprempre, C.: Authenticated encryption: relations among notions and analysis of the generic composition paradigm. J. Cryptol. **21**(4), 469–491 (2008)

[Can04] Canetti, R.: Universally composable signature, certification, and authentication. In: 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004), Pacific Grove, CA, USA, 28–30 June 2004, p. 219 (2004)

[CGLZ18] Chung, K.-M., Georgiou, M., Lai, C.-Y., Zikas, V.: Cryptography with dispensable backdoors. IACR Cryptology ePrint Archive 2018, p. 352 (2018)

[CHMV17] Canetti, R., Hogan, K., Malhotra, A., Varia, M.: A universally composable treatment of network time. In: 30th IEEE Computer Security Foundations Symposium, CSF 2017, pp. 360–375 (2017)

[GG17] Goyal, R., Goyal, V.: Overcoming cryptographic impossibility results using blockchains. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10677, pp. 529–561. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_18

[GGH+13] Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, Berkeley, CA, USA, 26–29 October, pp. 40–49 (2013)

[GGHW17] Garg, S., Gentry, C., Halevi, S., Wichs, D.: On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. Algorithmica **79**(4), 1353–1373 (2017)

[GKP+13] Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: How to run turing machines on encrypted data. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 536–553. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_30

[GKR08] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-time programs. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 39–56. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85174-5_3

[GM84] Goldwasser, S., Micali, S.: Probabilistic encryption. J. Comput. Syst. Sci. **28**(2), 270–299 (1984)

[Gol04] Goldreich, O.: The Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge (2004)

[HL10] Hazay, C., Lindell, Y.: Efficient Secure Two-Party Protocols: Techniques and Constructions. ISC. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14303-8

[Jag15] Jager, T.: How to build time-lock encryption. IACR Cryptology ePrint Archive 2015, p. 478 (2015)

[Kat07] Katz, J.: Universally composable multi-party computation using tamper-proof hardware. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 115–128. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72540-4_7

[KMG17] Kaptchuk, G., Miers, I., Green, M.: Managing secrets with consensus networks: fairness, ransomware and access control. IACR Cryptology ePrint Archive 2017, p. 201 (2017)

[LKW15] Liu, J., Kakvi, S.A., Warinschi, B.: Extractable witness encryption and timed-release encryption from bitcoin. IACR Cryptology ePrint Archive 2015, p. 482 (2015)

[LPS17] Lin, H., Pass, R., Soni, P.: Two-round concurrent non-malleable commitment from time-lock puzzles. IACR Cryptology ePrint Archive 2017, p. 273 (2017)

[MG16] Malhotra, A., Goldberg, S.: Attacking NTP's authenticated broadcast mode. Comput. Commun. Rev. **46**(2), 12–17 (2016)

[MGV+17] Malhotra, A., Van Gundy, M., Varia, M., Kennedy, H., Gardner, J., Goldberg, S.: The security of NTP's datagram protocol. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 405–423. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_23

[MMBK] Mills, D., Martin, J., Burbank, J., Kasch, W.: RFC 5905: network time protocol version 4: protocol and algorithms specification. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc5905