Anonymously publishing liveness signals with plausible deniability

Michael Sonntag[1][0000−0002−2506−2350], René Mayrhofer[1][0000−0003−1566−4646], and Stefan Rass[2][0000−0003−2821−2489]

[1] Institute of Networks and Security, Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
{michael.sonntag,rene.mayrhofer}@ins.jku.at
[2] LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria stefan.rass@jku.at

Abstract. Sometimes entities have to prove to others that they are still alive at a certain point in time, but with the added requirements of anonymity and plausible deniability; examples for this are whistleblowers or persons in dangerous situations. We propose a system to achieve this via hash chains and publishing liveness signals on Tor onion services. Even if one participant is discovered and (made to) cooperate, others still enjoy plausible deniability. To support arbitrary numbers of provers on a potentially limited list of online storage services, an additional "key" distinguishes multiple provers. This key should neither be static nor predictable to third parties, and provide forward secrecy. We propose both a derivation from user-memorable passwords and an initial pairing step to transfer unique key material between prover and verifier. In addition to describing the protocol, we provide an open source App implementation and evaluate its performance.

Keywords: liveness · plausible deniability · hash chain

1 Introduction

For their personal safety, whistleblowers[3] need to prove that they are still alive to, e.g., prevent a "security package" stashed with a third party from being published. Of course, this only works well if the third party (or parties — there may be multiple) remains unknown to any potential threats for the whistleblowers themselves. Also, as soon as whistleblowers have been apprehended, their devices will be investigated and all data extracted. This should not allow to a) identify

[3] In this paper we use the term "whistleblower" as a placeholder for any person — or potentially some process — who may be in possession of confidential material, publicly, politically, militarily, or legally exposed for any reason, or otherwise threatened in relation to them potentially releasing such information to the public. There are too many threat scenarios to list exhaustively, which is why we use the commonly known term whistleblower when implying the general threat model and, synonymously, the term "prover" when implying the protocol aspects without loss of generality.

the third party/parties and b) if these are discovered despite all precautions, they should be able to plausibly deny that they are acting as such third parties. The same applies in reverse: if a third party is somehow discovered, it should remain impossible to discover the whistleblower or prove that someone suspected as that person was connected to this person in any way. While, depending on the circumstances, mere suspicion of being any party (whistleblower or trusted third party) might be enough for dire consequences [20, 24, 13], at least no technical proof should be possible. We note explicitly that hiding the fact that a party has been interacting with our proposed service (e.g., but not limited to, having installed the mobile client app) is outside the scope of this paper.

Such a scheme requires the whistleblower ("*prover*") to transmit or publish some data ("*signal*"), from which one/several third parties ("*verifiers*") can conclude the prover was "alive and well" at a certain point in time (e.g. when the signal was published). To prevent correlation attacks, signals must be published and retrieved asynchronously, i.e., stored on some publicly accessible location (typically a website) by the prover, while verifiers ideally access this location as part of their standard activities (e.g. "visiting some website and viewing content"). While a storage server for verifying liveness can also store the "security package" (to be published if the verifier is considered "dead"), it obviously must be encrypted and solely verifiers should know the decryption key. The storage server should not simultaneously be a verifier, as then direct and synchronous communication between them would take place; also trust in this context is typically something between two persons and not a person and some (potentially very large) organization. Both sending and verifying signals should look innocent, i.e., observing their traffic should not allow anyone to conclude that they are party to such a scheme. While a prover is typically a single person, one prover might have several verifiers. These could be e.g. journalists, friends, or representatives of trustworthy foreign institutions — someone the prover trusts[4]. At some previous point a brief secure communication with them is needed (e.g. in person), but not anymore during the scheme. If one verifier is discovered, not only the prover but any other verifier should enjoy anonymity and plausible deniability (see [3] for definitions and kinds) even under the assumption of collusion of *all* other participants. Plausible deniability is not a legal term; it should be seen as the existence of other explanations, which are at least as likely as the actual one, shifting the burden of proof to the other party. As a minimum, any existing suspicion may not be increased through any traces which might be found with full access to all devices and their forensic examination (i.e. including investigations touching national security and so enjoying almost unlimited ressources).

To solve this problem, we propose a system based on Tor onion services [5] combined with hash chains for publishing such signals. The signal itself is solely a binary flag "I am alive" and has no other content (for a related approach for signals alone without a verifier secret/considering storage see [21]), so here we focus solely on the publishing/verification and the storage aspect of the problem.

[4] Should this trust be misplaced, deniability gets very important — only the statement/data of that person should be available to attackers, but no other evidence.

2 Related Work

The potential issue of whistleblower protection has already been used as the motivation for very different technical approaches. Time-lock puzzles [18] seem to have been one of the first proposals suggesting the use of trusted agents to help with timed information release. More recently, public ledgers were proposed as a storage mechanism that could release information if heartbeats are not regularly received (e.g., in CALYPSO [12]). Other solutions to assist whisteblowers are confidential (in the sense of protecting sender anonymity) document submission systems such as SecureDrop[5] [22], which is in use by major newspapers [7], or anonymous messaging systems like Ricochet [2], Cwtch [17], or others (that no longer seem to be under active development). Group authentication in such anonymous messaging settings is even more challenging [4, 23].

Many of these rely on onion routing [6] to hide sender and receiver IP addresses, particularly on *Tor Onion* services [26, 25], as we also propose here. We argue that the use of Tor onion services no longer constitutes a proof of suspicion towards potential whistleblowers, as even large-scale services like Facebook or Twitter now offer access via onion services to work around local network censorship/monitoring [27, 1]. Also, e.g., Cloudflare offers to provide access to websites hosted by them via onion services: when users connect via Tor to "some-domain.com" hosted by Cloudflare (and this feature is enabled), they will be redirected internally to an onion service of this domain [19]. From the security/anonymity point of view this changes little, but performance increases and no external exit-node is involved. Tor users will then use onion services even if they don't know the actual onion URL, increasing their utilization. A custom HTTP header was defined to "redirect" normal traffic to an onion address (which is much harder to remember) [11] too, for sites hosted in any other manner. When selecting anonymizing techniques, any method with a small number of users acts as a signal just through its network traffic patterns, independently of the content transported through it. Therefore, onion services seem to be the best compromise for a widely deployed and usable system with reasonable anonymity guarantees, even though onion service use can be measured at least statistically [10].

Recent research [8] describes how a deniable protocol can be subverted via remote attestation: through performing the local part of the protocol in a Trusted Execution Environment (TEE), a non-repudiable transcript can be generated in an undetectable way, so a party can later prove to third persons that the (then no longer undeniable) protocol was actually performed by them. However, this is not applicable to our proposal. We assume that an attacker can "convince" one (known) party/parties in some manner to cooperate, and therefore "trusts" them anyway. So moving the calculation of parts of our protocol into a TEE to attest that it actually took place as claimed has no influence on the deniability of the other party: the sender does not know whether the recipient even retrieved the signal and so could only prove that they sent a signal — providing no data on any potential verifier. On the other hand, the recipient can only prove that

[5] Available online at https://securedrop.org/

someone sent a valid signal and they received/verified it, but not who did this. For an attacker this would only make sense if he could modify the hardware of a party undetectedly, first having identified them. However, in that case software modification (trojanizing their device) would be sufficient and produce exactly the same information: confirmation that the local activity took place, but no additional information about the other side.

In this paper, we focus specifically on preventing de-anonymization of either a whistleblower (prover) or people communicating with them (verifiers) regarding their "liveness". That is, actual (confidential, integrity-assured, and potentially authenticated) document or message transmission is out of scope here. However, we focus on the deniability aspect of locally detectable signals under the assumption that end-user devices of provers or verifiers are captured, which in turn is out of scope of many of the anonymous messaging or document submission services mentioned. Our goal is to provide plausible deniability of involvement in a specific whistleblower process as a certain party (prover or verifier) even for a person whose device is forensically analysed. Therefore, our scope is especially on the sending and receiving of liveness signals as one component of a whistleblowing process and a tool for assisting in the protection of whistleblowers.

3 Possible Alternative Approaches

An alternative solution to the one proposed here could be "classified ads": on some public site (e.g. Twitter, Facebook) a code is posted by the prover, telling verifiers that she/he is still alive. But this approach has several shortcomings: you need to register to be able to post (which typically means a valid E-Mail address and some E-Mail communication for account verification; today often a phone number too) and logging in to send. Depending on the service logins may be required for checking the existence of a post. Finding them might mean deliberately searching for them (= entering keywords). States can easily block or (selectively) delay access to such services generally or for individual users. Removal by the website (e.g., "only text messages, no binary codes", complaints/takedown notices) are an increasing difficulty. Any "technical" signal like a hash value is obvious as such, easily discovered automatically, and directly tied to a sender, while "normal" text would either have to be repeated for every signal instance (=suspicious) or require a complex scheme/external storage for changing content. Also, the complete message history with start and end date is (usually publicly, but at least for the service provider) available and easily archived by third parties for later verification too. In contrast, the presence of a Tor browser, or e.g., Tails Linux (no installation needed), is much more open to interpretation and while potentially suspicious for certain regimes, typically not in itself illegal and not useful to identify past usage patterns or behavior in any case.

Signing a public message by any other person like a recent newspaper article (could even be randomly selected as long as the source is provided) requires either a real public key (i.e., identifying the prover) or agreeing with each prover on a separate keypair. Remembering these is difficult and storing them is also

potentially suspicious. Additionally, it immediately discloses the existence of the scheme and requires larger storage (for providing the source as well as a unique extract of the message beside the signature). This also doesn't solve the problem of the identical storage locations for each such message.

Such a scheme must be solely asynchronous communication or correlation attacks are possible, not only for verifying existing suspicions but with large-scale monitoring for identifying a — potentially small — subset of candidates for further investigation too. Therefore, approaches like double ratchet protocols [15] are unsuitable, as here no return channel exists. All logical communication must be solely one-directional from the prover to the verifier. Technically, the prover contacts the storage location to submit data, and at some later time verifiers individually contact the same storage location to retrieve it. Any return communication enhances the dangers of discovering an involved person.

4 Proposed Solutions

We propose two solutions, the main one described first and in detail, the possible alternative (albeit with certain shortcomings) briefly. The signal for this protocol is assumed to be short, like a single data block of 32–64 bytes, i.e., a hash value.

4.1 Main Solution

To prevent correlation attacks, signals are stored on third-party Tor onion services. This ensures transport encryption as well as meta data hiding (at least for address information if not for statistical traffic analysis) for prover and verifier when submitting/verifying a signal. If other activities, like web browsing via Tor (preferably on onion sites), are performed simultaneously, this effectively hides participation in such a protocol from on-path adversaries, as both submitting and checking a signal consist only of two (obtain&receive input for proof of work, submit result of challenge&submit/receive signal) short requests and responses.

Because it might be possible to distinguish normal Tor traffic from onion service access [14], it is beneficial that signal storage servers at the same time offer non-related (e.g. web) services through the same endpoint. While other activities should be performed there, their exact nature and duration is irrelevant: it is solely important that accessing that onion service does not solely consist of sending/verifying a signal. Then the traffic will not be easily recognizable as traffic related to this protocol. If both sides are known (prover or verifier and storage server), correlation is possible (which works reliably even for Tor [16]), but then the additional activity again prevents identifying the actions performed there as part of a liveness scheme — only of accessing this specific onion service.

A storage server can be run by anyone as a public service; a signal itself consist of only few bytes, so very few resources are needed. Additionally, their actions are extremely simple: submitting a value for storage or retrieving a specific value. The only useful attacks are DoS (deleting/modifying signals, thereby rendering responses invalid) and helping in correlation attacks (disclosing exactly when

a signal is submitted/queried for or sending too large replies). If the person performs other activities via Tor on other sites simultaneously, the latter should not matter. Note that two signals/queries cannot be identified to come from the same party. As no payment is involved in the protocol, hosting such servers must be performed as a public service. However, because the demand on the server is likely minimal (see section 4.1), this can be easily added to other public onion services, simultaneously taking care of publicizing/obtaining such an onion URL.

To facilitate selection of individual signals, they are associated with a "key" changing on every signal, i.e. specifically not a static or unique "prover ID". The storage server requires some proof of work (PoW) for submitting or retrieving a signal, e.g. performing some calculation that requires several seconds on a fast computer. This reduces the danger of DoS attacks through submitting huge numbers of signals, continuously retrieving them, or attempting to traverse the key space. A relatively small PoW comparable to intentionally slow derivation functions like Argon2 (used for verifying passwords) is sufficient here to balance DoS attacks on signal servers with the load on clients (in addition to the cryptographic requirements for onion connections; when performing other activities on the storage server as suggested even a 30 second delay for very slow computers would be irrelevant). As this is a challenge/response protocol, even very strong computational ressources are of limited use: enforcing creation of a new Tor circuit for every interaction is possible and proves an additional hurdle solely for attackers. In section 4.1 we give an approximation of the load a single attacker can put on a storage server: very little, so even thousands of simultaneous attacks are of limited impact (e.g. 1000 attackers only produce 15 GB storage demand and negligible computational load - challenge generation/verification is trivial). In case a key is requested for which no data was stored, random data is generated by the server, stored (= same response on subsequent queries), and returned. Therefore, anyone querying cannot determine whether a signal had been submitted under that key or not. No identification is needed for submitting a signal, only completing the PoW. Signals are automatically deleted after an appropriate period, e.g., a day or a week (delay set by server for everyone to limit its storage load). Deleting them after the first query is not done, as an adversary might then discover whether a signal was queried for or not (and multiple verifiers for a single prover would not work).

As the signal itself is part of the hash chain, the verifier can be sure it was created by someone knowing the shared secret — i.e. the prover. Note that because of the direction of the chain, the verifier cannot calculate the "next" signal from the data they know, so impersonation by a (second) verifier is impossible.

Using multiple storage servers (= multiple onion URLs) enhances resilience. While a signal could be stored on each of them under the same key, it is advisable to use different data (and therefore different keys) on each server, as otherwise querying all of them for a non-existing key will produce different (random) values, while an existing key would return the same value, disclosing whether a key is valid or not. The number of storage servers required is determined by the trust in their continued provision of this service and the duration envisaged for the

liveness scheme (trustworthy servers + brief duration = one server). Note that servers cannot be added later except via an additional secure out-of-band data exchange. Spare servers, which are only used if the first one is unavailable for whatever reason, are possible: only their onion URL is required and no data needs to be shared with them (prover → server or verifier → server) or with the other parties (between prover and verifiers) regarding them. It is therefore possible to use servers from a public trusted list accessible to both; however, accessing the list is a potential sign of involvement.

The storage key is calculated based on a hash chain (see formal description below): the prover creates a start value and shares this and an additional secret with one or more trustworthy verifiers. This has to be done securely, but is a very brief and one-time-only activity before any part of the protocol takes places electronically and therefore potentially observable for attackers. If this shared secret is lost, there is no recovery mechanism — a new secret has to be agreed upon by prover and verifier. Concatenating the start value and the secret and hashing it twice (in different configuration) produces the next key. This prevents third parties, which do not know this shared secret, from determining the next location under which a signal from the same prover will be stored, respectively looked for, as well as replay attacks.

After successfully verifying a signal, locally stored data is overwritten by new values, preventing adversaries from obtaining information on previous signals. The new key data is the result after only a single hashing, so that knowing a signal key (or simply storing them all and later trying brute-force) does not allow "calculating" forward even if the shared secret is obtained by an adversary (to avoid recreating later elements in the chain, the shared secret is appended for the key data and prepended for the actual key). This approach allows a verifier to also calculate future keys, even if a (range of) signal(s) was missed. Verification should consider as many keys as could have been generated up to the current time (and lost intermediate values) - or until the prover is considered "dead".

The current key data both parties store is random data from the point of view of adversaries as it is produced as the output of a hash function[6]: all binary values are valid in-/outputs, so it can be easily encrypted with any local value too. For this we suggest a human-brain only secret (=password) converted via a Password-Based Key Derivation Function (PBKDF, e.g. Argon2) and XORed with the key data. In this way, both prover and verifier have to remember a single shared secret/password and an individual own secret/password not disclosed to anyone else at any time. Note that disclosing an incorrect password - and its derived data - cannot be distinguished from those based on the correct one.

The (encrypted) key data stored by the verifier is the base data for the next key (note that this is not the key as such, as this will be hashed once before being used; see above), which is different at both parties exactly because of this encryption. The prover does not store this value; it is calculated only when

[6] We currently rely on standard, non-random-oracle hash functions for building the hash chains of location keys and signals, and therefore do not aim for proofs under the random oracle model for properties of these chains.

sending a signal. While the encryption key is the same length as the data, it includes the verifier's secret value so it changes on every signal too.

Because of these two elements we argue (without formal proof) that this fulfills IND-CCA2 as well as being indistinguishable from random noise as no other data encrypted with the same key is available to an adversary and XOR-encryption with a key used only once is similar to a one-time pad. The only difference is that the key in this scheme is the result of a hash function and therefore not truly random — but depending on the hash algorithm unpredictable enough without the input data. Any reasonably fast one-way derivation is suitable, especially cryptographic hash functions. As key and verification data are a single hash value each (e.g. 32 bytes), they can be explained as various other data because they look purely random and can so easily be incorporated in arbitrary steganographic schemes. Examples for this are e.g., internal checksums of (deliberately) damaged pictures (so no checksum verification is possible anyway), keys from other applications, or simply data left over on the disk from previous files. As the prover can designate any two similarly changing values as his (claimed as such) verification data (or randmly generate such), the data of both parties looks exactly the same (same count, length, properties). Therefore a prover can successfully claim to merely be a verifier.

Signal calculation algorithm

$\mathbf{Signal_0}$ "Random" data of length of hash function output. Random initialization vector or e.g. H(ProverSecret | SharedSecret) (Fig. 1: S_0; PS ProverSecret, SS SharedSecret)

$\mathbf{Signal_i}$ = H($\text{Signal}_{\text{i-1}}$ | SharedSecret) (Fig. 1: S_i)

Key calculation algorithm

$\mathbf{KeyData_N}$ "Random" data of length of hash function output. This is beyond the "end" of the chain of signals, i.e. a continuation (Fig. 1: $KD_N = S_N$)

$\mathbf{KeyData_i}$ = H($\text{KeyData}_{\text{i+1}}$ | SharedSecret) Data for deriving the next key (Fig. 1: KD_i)

$\mathbf{Key_i}$ = H(SharedSecret | $\text{KeyData}_{\text{i}}$) Key for publishing the signal (Fig. 1: K_i)

KeyData is used in reverse order, i.e. the first signal (value $\text{Signal}_{\text{N-1}}$) uses $\text{KeyData}_{\text{N-1}}$ resp. $\text{Key}_{\text{N-1}}$. The last possible signal uses Key_0 with value Signal_0.

Verification algorithm For its stored data, a Verifier takes S_N and SS (exchanged with Prover) and calculates these values by adding its own secret data:

$\mathbf{VerificationData_{N-1}}$ = H(Signal_{N} | VerifierSecret) (Fig. 2: V_i; VS VerifierSecret)

$\mathbf{KeyData_{N-1}}^{\mathbf{Enc}}$ = H(Signal_{N} | SharedSecret) $\oplus$ Argon2(VerifierSecret | $\text{VerificationData}_{\text{N-1}}$) (Fig. 2: $\text{KD}_{\text{N-1}}$)

Verifying signal i ([1...N] in temporally ascending sequence) works as follows:

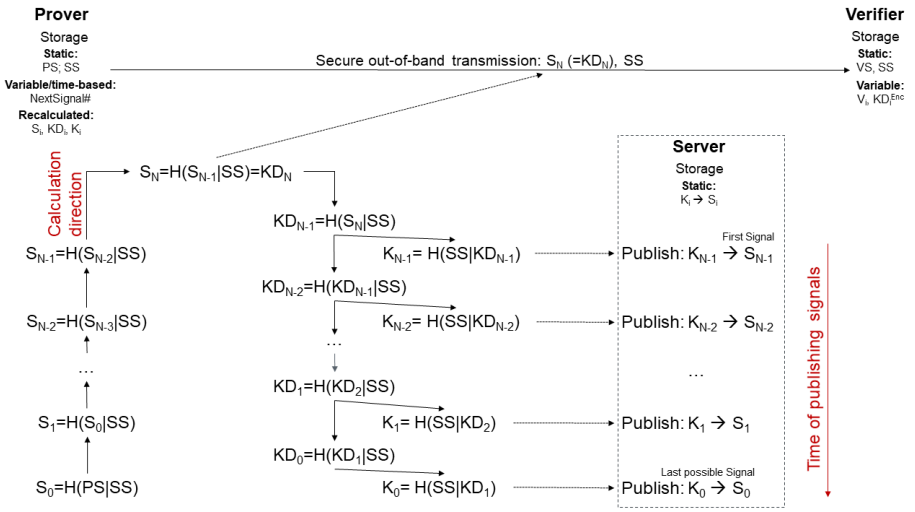

Fig. 1. Graphical description of the protocol - Prover part.

$\text{KeyData}_{\text{N-i}}$ = $\text{KeyData}_{\text{N-i}}{}^{\text{Enc}} \oplus$ Argon2(VerifierSecret | $\text{VerificationData}_{\text{N-i}}$)

$\text{Key}_{\text{N-i}}$ = H(SharedSecret | $\text{KeyData}_{\text{N-i}}$)

Success? H(H($\text{Signal}_{\text{N-i}}$ | SharedSecret) | VerifierSecret) == $\text{VerificationData}_{\text{N-i}}$

If "Success": Calculate $\text{VerificationData}_{\text{N-i-1}}$=H($\text{Signal}_{\text{N-i}}$ | VerifierSecret) and store it. Conclude prover to be "alive". Calculate and store $\text{KeyData}_{\text{N-i-1}}{}^{\text{Enc}}$ = H($\text{KeyData}_{\text{N-i}}$ | SharedSecret) $\oplus$ Argon2(VerifierSecret | $\text{VerificationData}_{\text{N-i-1}}$)

If "Fail": Don't update $\text{KeyData}_{\text{N-i-1}}{}^{\text{Enc}}$. $\text{VerificationData}_{\text{N-i-1}}$ cannot be calculated anyway, as $\text{Signal}_{\text{N-i}}$ is unknown. Try again later with identical (retrieval problem) or on the next timeslot (incorrect result) updated data: $\text{KeyData}_{\text{N-i-1}}$=H($\text{KeyData}_{\text{N-i}}$ | SharedSecret) and $\text{Key}_{\text{N-i-1}}$ = H(SharedSecret | $\text{KeyData}_{\text{N-i-1}}$). Compare then H(H(H($\text{Signal}_{\text{N-i-1}}$ | SharedSecret) | SharedSecret) | VerifierSecret) to the previous $\text{VerificationData}_{\text{N-i}}$. If unsuccessful after a verifier-determined number of tries, conclude the prover to be "dead".

Data stored by prover

- Onion address(es) of storage server(s) (string or binary value of public key - hidden or encrypted similar to the current key generation data as described below; or as well-known, remembered, or bookmarked "harmless" service)
- Shared secret (human memory only). See Fig. 1: *SS*.
- Prover secret (human memory only) for signal/key generation. Fig. 1: *PS*.
- The number of the next signal (or some method of deriving it, e.g. through some starting point in time and the current date/time).

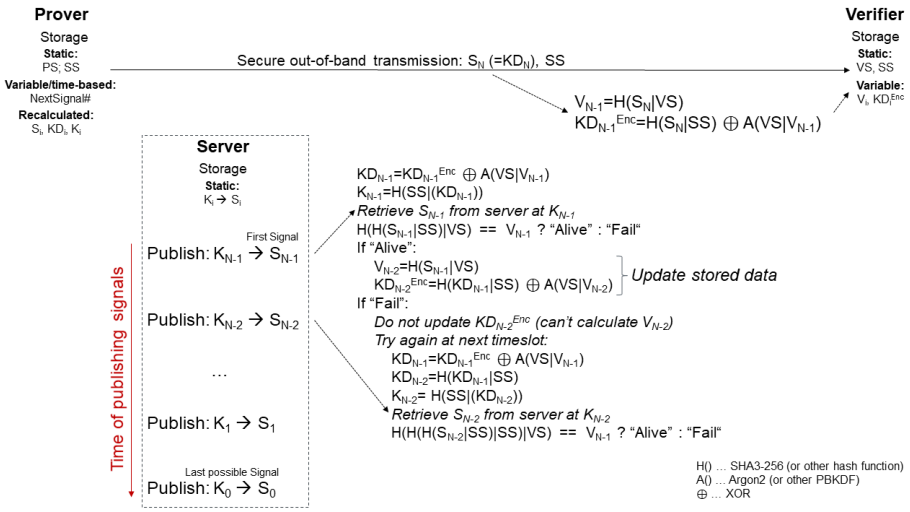

Fig. 2. Graphical description of the protocol - Verifier part.

Data stored by verifier

- Onion address(es) of storage server(s) (same as above)
- Shared secret (human memory only; same as above). See Fig. 2: SS.
- Verifier secret (human memory only; similar to above). See Fig. 2: VS.
- Current key generation data. Encrypted via XOR with data derived from verifier secret and verification data during storage and ratcheted forward after each sending. See Fig. 2: $\mathrm{KD_i}$ resp. $\mathrm{KD_i^{Enc}}$.
- Verification data for verifying the next signal value. Hash of current signal and verifier secret. See Fig. 2: V_i.

Data stored by onion service operator

- Map[key → signal]: Stored for a fixed duration, e.g., one day/week. For non-existing keys, random data is generated upon the first query and stored with the same retention period minus a random reduction to prevent detection of this fact.[7] Size per signal: $2 * 64$ Bytes + storage overhead. Assuming a 10-second PoW and one week retention period, a single computer solely sending random data can produce 60,480 signals; assuming 256 Byte for storage

[7] This combination of requirements on the storage server make it act as a random oracle with time-outs or, in another interpretation, a series of random oracles defined by overlapping epochs. Such a random oracle service might be helpful for other cryptographic protocols whose security properties depend on a random oracle instead of standard hash functions for some building blocks. By re-using our proposed storage service for such other purposes, plausible deniability in communicating with this service could be significantly improved.

(=100%/128 bytes overhead) this translates to 14.8 MB storage wasted by a single malicious host. The probability of randomly generating exactly the next key is extremely low (using 1,000 PCs: 60,480,000 tries of a possible 2^{256} for a 256 bit hash; approx. $1{:}5.22 * 10^{-70}$); the prover would then be unable to store the signal, but it is extremely unlikely this will affect the next key again. When tolerating at least one missing signal no problem occurs.
- Secret data for generating/validating proof-of-work challenges.
- Temporary session data for a duration slightly longer than the maximum delay for completing the proof-of-work.

Length of data to be remembered For both prover and verifier two data elements should be stored solely in memory. But how long do these need to be (as humans are notoriously bad at remembering long random data)? The algorithm doesn't require a specific length, as they either serve as (part of) input to a hash function or are (preferably) used with a PBKDF. Consequently, any format/length is fine, as long as it is hard to discover via brute force attacks. They are equivalent to passwords, with all their problems and advantages: a short uncommon string is sufficient, as is a longer phrase of common words.

4.2 Alternative Solution

An alternative solution is to not store a fixed onion URL and employ varying keys, but rather generate a new onion URL — and therefore onion service — for each signal to replace the key. This allows to trivially swap storage servers: every hoster of the previous scheme can easily provide arbitrary additional tiny onion services with reasonable overhead [9]. Hosting an onion service for others is in this case harmless: it solely provides a single short static value upon every possible request (no other content, no input parsing, etc). Using hierarchical encryption, from an initial private key and incorporating some secret data more private keys can be derived similar to a hash chain. Interestingly it is also possible with knowing solely the public key (and the secret data) to derive the matching public keys too. This technique is already used in onion services (v3) to mask the data sent to the hidden service directory (blinded keys [25]). However, each private key would have to be disclosed to the storage server as it is needed for operating the onion service. So the storage server cannot calculate the next keypair(s), the shared secret must be integrated in the key derivation (similar to deriving the keys from the keydata chain). This shared secret must be protected individually by each party (see above), as discovering it stored at both locations removes plausible deniability. As it is needed in cleartext for the calculations, hashing it for storage is impossible; but encryption works.

A deficiency of this approach is that, when performing hierarchical key derivation with Elliptic-Curve (EC) cryptography, the private key can be selected arbitrarily, but not the public key. So encrypting the private key by XOR with a random/secret value is possible and it still remains a "valid" private key. This is necessary for plausible deniability: disclosing an incorrect secret value still produces a valid private key (just without any other party). Because if private key

and shared secret are stored in clear or disclosed by the prover — and so obtained by an adversary — the matching public key (and all future derived keypairs) can trivially be computed. If such a public key is found by an adversary at a suspect, the verifier role can no longer be denied. Unfortunately, masking similar to the prover is impossible for the verifier, as unlike the private key the public key for an EC cryptosystem *can* be verified: not all possible values are valid public keys. So a transformation is needed taking a valid public key and encrypting it so that it can be decrypted with two different arbitrary data (e.g. two passwords) resulting in two valid public keys (the second used for deniability). This could be achieved via deniable encryption and an arbitrary other public key as decoy. However, this still requires the decoy to point to a working onion service or be immediately recognizable as such. So the storage location would have to know that decoy private key, too. To be indistinguishable from the real key, it must be changed according to an identical schedule, so the verifier would have to act as a fake prover in this regard and initiate the decoy onion service creation. Still, prover and verifier are not as symmetric as a prover now stores one private key, and a verifier two public ones: a prover cannot as easily claim to be a verifier, as the private key will typically not validate successfully as a public key either.[8]

Another issue is that, if the prover is discovered and cooperates, future onion URLs can be calculated and hosted by the adversary as opposed by third parties in the previous scheme. The adversary can then much more easily (knowing the onion URLs of the third-party storage servers in the first scheme still requires discovering their physical locations before monitoring is possible) check whether someone attempts to verify liveness ($\rightarrow$ did the prover lie?) and obtains control over the process ($\rightarrow$ correlation attacks against suspected verifiers). Discovering the verifier does not allow such attacks, as while the public key alone does allow to pre-calculate the onion addresses, these will not be active earlier and, because of lack of the private key, cannot be impersonated or located in advance.

5 Limitations and Discussion

5.1 Limitations

Some limitations of this approach exist. Namely, the presence of the software needed for the scheme is a sign someone participates in it. This could be resolved by including it in a "standard installation", i.e., lots of persons possessing this software, but only few actually using it. Dedicated secure messaging apps like Signal seem like an optimal avenue to include such additional functions by default for the benefit of endangered minorities. The code required is minimal, so usability would be the main concern. Forensic investigations may provide traces, whether some executable was run or not, so direct inclusion is preferable over adding it as "extension" or "module". Moving the software online is a bad idea, as this requires disclosing the secret (=human memory only) data to another party.

[8] So two public key decoys are needed, for which the prover must initiate onion creation too, and the verifier needs an arbitrary fake private key.

However, providing it on a webpage (→ no local software installation), e.g. as JavaScript, to be executed locally would work. This needs trust in the hoster and the transmission path that the downloaded program does not perform any other activities like sending data/keys to anyone. Securing the transmission is easy (→ TLS), leaving how to recognize and trust the site. Manual verification of the software against a well-known hash value is possible, but not user-friendly, as is verifying some custom added signature. Storing the data in browser local storage immediately discloses participation in the scheme, as then there is no reasonable alternative explanation. The obvious approach, hosting it on the onion service used to store signals, is not necessarily secure: as a hoster providing storage few attacks are "interesting" (see above) and little trust in it is needed. Potentially trusting them with the keys and all private information is a much larger question. But approaches like binary transparency for web server content may be a future mitigation. Local caching after verification is the equivalent of a local installation and is detrimental to deniability. So if no local installation is acceptable, some trusted website (to avoid raising suspicion an onion service, not the public Internet) for hosting the software is required.

Another limitation is that all parties need to store the onion URL(s) of the storage server(s) (identical for each pair of prover/verifier), and because of the length/format this can hardly be done in human memory. If an onion URL solely provides a liveness service as described here, discovering the URL abolishes plausible deniability of participating in such a scheme. If an onion service provides other widely-used services, e.g., a normal webserver with legal and widely interesting content, the presence is much less of an evidence and could then even be stored as a bookmark. Candidates for including liveness storage services are e.g. secure drop sites that are already available as onion services at major news outlets (potentially problematic: also "undesirable" sites for adversaries interested in unmasking provers) or even public services like news outlets or social networks.

5.2 Plausible Deniability Achieved?

We now look at various scenarios to determine whether plausible deniability actually exists under the proposed scheme. We assume one party is fully cooperative (because of bribes, torture etc) so the adversary obtains access to all their locally stored data, and either provides correct secrets from memory — or lies. Can such a lie be detected, and what does this imply for the other party/parties?

- Prover is "cooperative" and provides all secrets correctly, including memorized ones: The adversary obtains access to the initialization data and can calculate all keys for the future with the shared secret. But none of this data is found at a suspected verifier, as there the key data is encrypted with a secret solely known to this verifier, and the shared secret is stored solely in human memory. The last key data of the suspected verifier can therefore be anything else or relate to a different prover. Lying about the verifier key or the shared secret produces keys indistinguishable from the ones used in the

past and that can also be retrieved (as they are considered "valid" by the respective server(s)), but which cannot be successfully verified as a signal. So either the (fake) prover is no longer active, the verifier lied, or the data found is not related to such a scheme at all. Which case it is cannot be decided based on available data and all options are at least possible. As old key generation values cannot be generated (preimage attacks on the hash are assumed to be impractical) and are not stored, it cannot be proven that no previous values were valid keys or validated successfully. Even if all (globally observed) old keys and signals are known (e.g., stored for future validation; storage server cooperates/is the adversary), these cannot be used to generate later keys, preventing the adversary from proving no "old" but valid signals ever existed. As future keys are known through the prover, correlation attacks might be possible regarding suspected verifiers, but as the communication is asynchronous this requires locating the onion server, as without access to it, it is impossible to determine whether this server is accessed.

- Prover is "cooperative" but lies about the shared and/or prover secret: The adversary can calculate all (wrong) keys for the future, but nobody is going to check them — but this cannot be detected by an adversary. If this can be verified (the adversary controls the storage service or it cooperates), the prover can no longer deny that they are lying, but only if done quickly; otherwise, verifiers might already have concluded the prover as no longer active and do not check any longer.
- Verifier is "cooperative" and provides all secrets correctly: This is symmetric to the prover. Future keys can be calculated, but this data is not found at a suspected prover. However, working together with the storage service — after it being located — would allow correlation attacks (but nothing else).
- Verifier is "cooperative" but lies about the shared or verifier secret: The adversary can calculate (incorrect) future keys and query for them, but none can be verified successfully. Whether the prover is no longer active, stopped publishing signals for other reasons, or the verifier lied cannot be discerned.
- If a prover is found and all physically stored data is available, they can still claim to merely be a verifier: any arbitrary data can be designated as the variable verification and key data a verifier stores. The only limitation of plausible deniability is that no future "signal" can be successfully validated, so the (imaginary) "prover" is considered no longer active. Note that old keys/signals cannot be generated from this data in any case, so any past data stored by an adversary cannot be used for validation.
- If a verifier is found and all physically stored data is available, they can claim to be a prover (if desirable): they can designate the current key generation data as the initial data, producing a correct but unused hash chain. That there is no verifier validating these is again impossible to determine without timely collusion of the storage server. The only limitation is that the verification data may not be discernible as such and can be explained as unrelated — which should be possible for all the data.

5.3 Performance Evaluation

Using a prototype implementation of the protocol in Java, we measure the practical runtime overhead of creating and verifying signals on the client respectively server sides. Both were executed on a (today comparatively slow) PC with Core i5-2400 CPU (3.1 GHz) running Windows 10. We ignore the webserver here, as that would require a PoW for the client, skewing the results enormously.

We measure execution speed after warm-up of the Java runtime by executing at least 10 similar operations (calculating the first signal with 1,000,000 potential signals as well as verifying) in the same instance before starting the measurements, which are executed 100 times for averaging measurement noise on the test systems. Calculating the initialization data (i.e., calculating the full signal chain up plus a single additional hash to obtain the initialization data to be shared with the verifier) takes on average 1.4 seconds. Verifying a signal requires approx. 9 ms (most of this probably test overhead, as very few hashes need to be calculated). Producing the last signal takes a bit longer, as the hash chain must be calculated up to the first signal, and then down again to obtain the last key data: 2.6 seconds. This seems reasonable, especially when considering that few application will require a million potential signals. Reducing this to 10,000 signals, the duration drops to 13/27 ms for generation and 8 ms for validation (=unchanged; independent of potential signal count), which should not be problematic even for slow computers.

5.4 Prototype App Implementation

A prototypical implementation of the whole end-to-end user interaction process in the form of an Android app accessing the fully implemented prototype webserver is available as open source at https://github.com/rmayr/livenesssignal-android. The app implements both prover and verifier roles and only stores data as described in the protocol above permanently on-device, protected with the local app password. Passwords need to be entered on every invocation of the app (sending or checking) and are not stored permanently.

To test the app and allow other researchers easy experimentation, a first instance of our prototype webserver for hosting signals is available at http://fng5mhuck2n7l4we2egjnbp6l4cofw46wjyvi7t6s37uhwbcanmylyqd.onion/liveness This instance can be used by the general public, but may be subject to future restrictions depending on resource considerations. However, as described above, in the absence of explicit DoS attacks, typical use of such a service should only result in insignificant system load in the sense of computation, storage, and data transfer requirements.

6 Conclusion

We provide a system for publishing liveness signals with plausible deniability: even if some participant of such a scheme is discovered, others can still plausibly deny their involvement or claim the opposite role. Two drawbacks are that

possession of the software necessary for the scheme might provide a hint to participation, and that a third party is needed to host/publish signals. These are issues of our prototype implementation and could be overcome by a) integrating our liveness verification protocol into well-known apps or as a web page and b) hosting of storage servers by well-known web hosts. The system is easy to implement, fast, and fulfills all our requirements for anonymity and security.

Acknowledgments This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH, and Österreichische Staatsdruckerei GmbH.

References

1. A. Muffett: On behalf of @Twitter, I am delighted to announce their new @TorProject onion service, at:. https://twitter.com/AlecMuffett/status/1501282223009542151
2. Brooks, J.: Ricochet. https://ricochet.im/
3. Celi, S., Symeonidis, I.: The current state of denial. Privacy Enhancing Technologies Symposium (2020), https://petsymposium.org/2020/files/hotpets/The_current_state_of_denial.pdf
4. Corrigan-Gibbs, H., Ford, B.: Dissent: Accountable anonymous group messaging. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. p. 340–350. CCS '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1866307.1866346, https://doi.org/10.1145/1866307.1866346
5. Dingledine, R.: Next Generation Tor Onion Services. DEF CON 25 (2017)
6. Goldschlag, D., Reed, M., Syverson, P.: Onion routing. Communications of the ACM **42**(2), 39–41 (1999)
7. Guardian, T.: The guardian securedrop. https://www.theguardian.com/securedrop
8. Gunn, L.J., Parra, R.V., Asokan, N.: Circumventing cryptographic deniability with remote attestation. Proceedings on Privacy Enhancing Technologies **2019**(3), 350–369 (2019). https://doi.org/doi:10.2478/popets-2019-0051, https://doi.org/10.2478/popets-2019-0051
9. Höller, T., Raab, T., Roland, M., Mayrhofer, R.: On the feasibility of short-lived dynamic onion services. In: 2021 IEEE Security and Privacy Workshops (SPW). pp. 25–30. IEEE (May 2021). https://doi.org/10.1109/SPW53761.2021.0001
10. Höller, T., Roland, M., Mayrhofer, R.: On the state of V3 onion services. In: Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet (FOCI '21). pp. 50–56. ACM (Aug 2021). https://doi.org/10.1145/3473604.3474565
11. Kadianakis, G.: Onion-location. https://gitweb.torproject.org/tor-browser-spec.git/tree/proposals/100-onion-location-header.txt

12. Kokoris-Kogias, E., Alp, E.C., Gasser, L., Jovanovic, P., Syta, E., Ford, B.: Calypso: Private data management for decentralized ledgers. Cryptology ePrint Archive, Report 2018/209 (2018), https://ia.cr/2018/209
13. Kumagai, J.: The whistle-blower's dilemma. IEEE Spectrum (Apr 2004), https://spectrum.ieee.org/the-whistleblowers-dilemma
14. Kwon, A., AlSabah, M., Lazar, D., Dacier, M., Devadas, S.: Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 287–302. USENIX Association, Washington, D.C. (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/kwon
15. Marlinspike, M.: The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/
16. Nasr, M., Bahramali, A., Houmansadr, A.: DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM (jan 2018). https://doi.org/10.1145/3243734.3243824, https://doi.org/10.1145.3243734.3243824
17. Open Privacy Research Society: cwtch. https://cwtch.im/
18. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep. (1996)
19. Sayrafi, M.: Introducing the cloudflare onion service. https://blog.cloudflare.com/cloudflare-onion-service/ (09 2018)
20. Snowden, E.: Permanent Record. Pan Macmillan (Sept 2019)
21. Sonntag, M.: Anonymous proof of liveness. In: Proc. IDIMT-2021. Trauner Verlag (2021)
22. Swartz, A.: Securedrop. https://github.com/freedomofpress/securedrop
23. Syta, E., Peterson, B., Wolinsky, D.I., Fischer, M., Ford, B.: Deniable anonymous group authentication. Tech. Rep. YALEU/DCS/TR-1486, Yale University (February 2014)
24. Tate, J.: Bradley Manning sentenced to 35 years in WikiLeaks case. Washington Post, online archived at https://web.archive.org/web/20130825043050/http://articles.washingtonpost.com/2013-08-21/world/41431547_1_bradley-manning-david-coombs-pretrial-confinement (Aug 2013)
25. The Tor Project: Tor Rendezvous Specification - Version 3. https://github.com/torproject/torspec/blob/master/rend-spec-v3.txt
26. Tor Project, I.: The Tor project. https://www.torproject.org/ (2021)
27. W. Hoffman: Facebook's Dark Web .Onion Site Reaches 1 Million Monthly Tor Users. https://www.inverse.com/article/14672-facebook-s-dark-web-onion-site-reaches-1-million-monthly-tor-users