Cryptographic proof of custody for incentivized file-sharing

Pavel Kravchenko[1], Vlad Zamfir[2]

[1] Distributed Lab, pavel@distributedlab.com
[2] Coinculture, vlad@coinculture.info

Abstract. A cryptographic proof-of-custody is a zero-knowledge proof of simultaneous possession of a given data set and a private key. We will introduce proof-of-custody by explaining its application in incentivized file-sharing, and then we will formalize the conditions that it must satisfy. We then present a solution and provide calculations of its computational and bandwidth efficiency. Additionally, we will present 'initially promising solutions' and attack vectors against them, in the appendix.

1 Introduction

Rewarding uploaders and hosts in a file-sharing network in a non-exploitable manner is difficult because transmission of information is not a property of the transmitted information. This makes it fundamentally impossible to produce a cryptographic proof of having sent data across a *passive channel*; a channel that does not modify or sign the data it transmits. Instead of trying to prove that information was transmitted, one can ask the recipient of data for a proof that they have *custody* of that data.

A proof-of-custody of data is distinct from a traditional proof-of-knowledge of that data because it has the additional property that it could not have been produced by (or in collaboration with) someone who has the data but not the private piece of an asymmetric key pair. To illustrate, let us introduce our old friends Alice and Bob, and a third party, the auditing server. This time, Bob has an asymmetric key pair and Alice sends a file to Bob. The auditing server wants some form of cryptographic assurance that the file transfer took place.

A proof-of-Bob's-custody of the file cannot be produced by Alice unless Bob gives her his private key, and it cannot be produced by Bob unless he, in fact, has the entirety of the file in

question. We will see that if a proof of Bob's custody of the file is produced, the auditing server can be somewhat justified in assuming that the file transfer actually took place. More specifically, a working proof-of-custody scheme can serve as a replacement for proof-of-transmission when it is taken together with the assumption that the recipient will not reveal their private key in order to produce the proof.

Such a scheme can be used to securely fund file-transfers because it replaces a sender or recipient's report that the transfer took place with a cryptographic proof. Another very notable use case of provable custody of data in for file archives or dropboxes. There are likely other uses of proof-of-custody that have not yet come our attention.

2 Formal definition

We will now formalize what it means for it to be impossible for Bob to produce a proof-of-custody in cooperation with a party who has the file without also giving them access to his private signing key. This informal language will be replaced with the formal language of functional composition:

We define "the computation of $f(x,y)$ requires simultaneous access to both x and y" to mean that there does not exist a one-way function h such that $f(x,y) = g(h(x),y)$ or $f(x,y) = g(x,h(y))$, for some other function g. Informally, there is no way to compute $f(x,y)$ without being also able to recover both x and y.

Now, let D denote the data whose custody is in question, and let s refer to the private key corresponding to a public key P of an asymmetric key pair. A valid proof-of-custody scheme is a triple of functions (PoC, r, V) such that $V(PoC(s, D), r(D), P) = 1$ if and only if the computation of $PoC(s, D)$ requires simultaneous access to both s and D.

Specifically, $PoC(s, D)$ is the proof of custody, $r(D)$ is a "fingerprint" of D, and V is the "verification" function. Additionally we will require that r and PoC map their inputs to data structures that are much smaller than D, for large D, so that the proof may be said to be "concise". This is important because the bandwidth required to show that a file transfer took place should not be on the order of the size of the file. Additionally, we must note that $PoC(s, D)$ might be produced by an interactive cryptographic protocol, with commitment, challenge and response phases: Indeed, all of the promising proof-of-custody schemes that we have found take this form.

3 Limitations of proof-of-custody

Any server who asks for a proof needs to verify that the correct file's custody is being proven by the correct party. We therefore need to assume that the server has authenticated the public key of the party producing the proof. Additionally we must require that the server has access to a fingerprint of the file prior to requesting a proof (in our cases, this will be the root hash of its Merkle tree). Instead of treating these requirements, we will assume that the party who is paying for the file transfer provides the authenticated keys and fingerprint to the server. Finally, we cannot have the file itself be uploaded to the auditing server, so we must require that the proof be verifiable using only the data uploaded as part of the proof, along with the fingerprint and the key.

4 Requirements for proof-of-custody

1. To produce a valid proof, both a private key *s* and file *D* must be known.
2. It is impossible to compute a valid proof based on a smaller image of the file, some parts of the file, or using any process that requires transferring less data from Alice to Bob than the size of the file.
3. It is impossible to produce a valid proof based on modification of the private key (from a blinded key, for example).
4. The size of the proof is much smaller than the size of the file (ideally it is sublinear in the size of the file).
5. It should be possible for the auditor to specify the security level of the proof.
6. The server, or anyone else who sees communication between the Server and Bob, must not be able to recover the private key of Bob from any of the submitted data. That is, *PoC(s, D)* must be a one-way function.

5 Attack vectors on proof-of-custody schemes

1. Bob predicts which parts of the file will be audited by the server and requests only those parts from Alice.
2. Bob creates a modified private key that Alice can use to produce a proof on their behalf (without any file transfer).
3. Bob creates a random proof so that during the verification process he can download the needed parts of the file and pick-up parameters that will produce a valid proof.

We will share examples of these attacks against 'weak' proof-of-custody schemes in the Appendix.

6 Solution overview

All of the candidate solutions that we will present have the following form:

1. Bob splits the file D into smaller pieces, $\{D_i\}$.
2. Bob prepares proofs for each piece and sends a commitment for each to the server
3. The auditing server produces a random challenge; a sample of the commitments. Note that the sample size determines the security level of the proof.
4. Bob responds to the challenge by providing proofs of custody for the chosen chunks.
5. The server verifies the proofs.

6.1 Required background: zero-knowledge proofs

A cryptographic zero-knowledge proof or protocol is a method by which one party can prove to another that a given statement is true without conveying any information apart from the fact that the statement is true indeed. If producing a proof requires knowledge of secret information on the part of the producer, the verifier will not be able to create a new zero-knowledge proof of the statement, since the verifier does not possess the secret information. This technique was first introduced in 1985 by Shafi Goldwasser, Silvio Micali, and Charles Rackoff [1].

6.2 A modified Schnorr identification protocol.

This protocol will prepare the reader for understanding the working proof-of-custody protocol. It is the simplest zero-knowledge protocol:

An elliptic curve $E(F_q)$, and a point P $E(F_q)$ of order n are publicly known parameters of the protocol, where F_q is a finite field of order q. Bob's public key is a point on the curve, Q $E(F_q)$, which is also of order n. Bob's secret key s is an integer $1 < s < n$ such that $Q = sP$. Bob's goal is to provide a zero-knowledge-proof to the server that he knows s such that $Q = sP$.

The following steps will be repeated m times:

1. Commitment: Bob chooses a secret random number k $\{1,...,n\}$ and sends $R = kP$ to the sever.
2. Challenge: The server chooses a random "challenge" e $\{0,1\}$ and sends e to Bob.
3. Response: Bob computes $a = (k + se) \bmod n$, and sends a to the server.
4. Verification: The server accepts Bob's proof if $aP = R + eQ$.

The secret random parameter k is used to "blind" the important secret, s. If Bob ever uses the same k twice (as might occur if k is nonrandom), s can be computed by the server from his commitment of $a = k + s$. If e is received from the interval $[0, n]$ then this protocol turns into the Schnorr identification protocol.

The protocol is *complete*. That is, if Bob does have access to s such that $Q = sP$ and he conforms to the protocol's expectations, the server will always accept his proof.

If Bob does not know s, then he has to cheat by guessing the random challenge bit e, *before* sending out the commitment. In the cheating "proof," the commitment will be as follows: $R_0 = kP$ if $e = 0$, and $R_1 = kP - Q$ if $e = 1$. Then in both cases Bob sends $a = k$ on a response stage. If he guessed e correctly, then the verification step does not lead to rejection of the proof:

1) $aP = kP = R_0 = R_0 + eQ$ if $e = 0$, and
2) $aP = kP = R_1 + Q = R_1 + eQ$ if $e = 1$.

In this case Bob will have 50% chance of having his proof accepted in each iteration of the interaction, as he has a 50% chance of guessing e before his commitment. So we have a *soundness error rate* of 50%. If none of m iterations result in rejection, then the probability of Bob's having cheated successfully falls to 2^{-m}. Therefore, the server will be sufficiently confident that Bob cannot cheat the zero-knowledge-proof if m is sufficiently large (because 2^{-m} will be sufficiently small). In practice, $m = 10$ will often provide enough confidence.

The security of this scheme relies on the difficulty of the discrete logarithm problem (DLP), which for elliptic curve cryptography can be defined as:

Given a finite cyclic group G of order n, a generator P of G, and an element Q, find the integer s, $1 < s <$ n*, such that $Q = sP \pmod n$.*

7 Solution to the custody problem

As in the modified Schnorr protocol we will assume that there is a publicly known elliptic curve $E(F_q)$ and an agreed-upon point $P \in E(F_q)$ of order n. Bob's public key is a point $Q \in E(F_q)$ of order n. Bob's secret key s is an integer $1 < s < n$ such that $Q = sP$. Bob's goal is to prove that he knows s *and* $\{D_i\}$, a set of t chunks that make up the file D. The server must be able to verify with arbitrarily high probability that Bob indeed possesses s *and* $\{D_i\}$ simultaneously.

Commitment phase. For each $i \in [1,t]$ Bob chooses a random $k_i \in [1,n]$ and sends $R_i = k_iP$ *and* $H_i = H(k_i + D_i)$ to the server. Note that H is a secure one-way hashing function.

Interactive verification phase. The following steps are repeated m times:

1. Selection: The server chooses a random $i \in [1,t]$**.**
2. Challenge: The server chooses a random "challenge" $e_i \in [0,1]$ and sends e and i to Bob.
3. Response: Bob computes $a_i = k_i + se_i \pmod n$ and sends a_i to the server. If $e_i = 0$, Bob also sends D_i to the server as part of a merkle proof that D_i is a member of D.
4. Verification: The server accepts the proof if:
   - for $e_i = 0$: $a_iP = R_i$, $H(a_i + D_i) = H_i$, and the merkle proof of D_i is valid
   - for $e_i = 1$: $a_iP = R_i + Q$

How it works:

1. If server selects $e = 0$, Bob's response will be $a = k$, so the server will be able to check that $R = kP$ and $H(k + D_i)$ constructed correctly, so proof is given for file chunk D_i.
2. If server selects $e = 1$, the response is $a = k + s$ so the server can check that user knows their key s. Very notably, if k is revealed to the server (or to some party with D_i) they will be able to calculate s from Bob's response, if it is valid.

Successfully cheating this protocol, as in the previous one, requires that Bob is able to predict the server's choice of e_i, so that he can provide k_i to a party with D_i, so that they can calculate $H_i = H(k_i + D_i)$ for Bob, without Bob risking that they will be able to recover his private key. Probability of cheating successfully for each round is 50%, as before, so m can be chosen to be large enough for the server to have confidence that Bob indeed has custody of D.

The important question is whether the production of commitments $\{H_i = H(k_i + D_i)\}$ can be done by Alice instead of Bob. If $|H(k_i + D_i)| << |D_i|$ (as we would like to require for all i) this would be an effective attack vector against our proof-of-custody scheme. But, if some k_i is given to Alice and the server chooses $e_i = 1$ to challenge Bob, he will have to respond with $a_i = k_i + s$ to produce a valid proof. In this case, Alice will be able to calculate Bob's private key, s, as she knows k_i and $a_i = k_i + s$. Therefore, if Bob reveals k_i to Alice but cannot predict the auditor's

challenge, then she will be able to recover *s* with 50% probability, which is something we have assumed that Bob cannot accept. On the other hand, Alice might simply attempt to bruteforce *k* from $H(k + D_i)$, because she knows D_i, so she can recover s. Thus, the hash function must be cryptographically secure, and should perhaps be memory-hard, using for example *scrypt* [3], to make brute-forcing *k* more expensive.

This protocol can also be made *non-interactive* if the Fiat-Shamir digital signature protocol [4] is used. It is worth noting that *m* in this case should be bigger, because Bob can try to find a particular result of the hash function that "asks" him for proof only for particular parts of the file that he already knows.

8 Efficiency considerations

Bandwidth calculations and number of math operations for different sizes of D_i:

File size/ Piece size, *m=10*	Complexity of proof	Size of proof	Approximate size of verification	Complexity of verification (client)	Complexity of verification (server)
	(1 mult over EC, 1 hash) $\|D\|/\|D_i\|$*	*65B* * $\|D\|/\|D_i\|$	*(32B +* $\|D_i\|$*)*m*	*(1 mult, 1 addition)*m*	*(1 EC multi, 1 hash)* m*
100KB/1KB	*100 mult, 100 hash*	*6.5 KB*	*10.3 KB*	*10 mult, 10 addition*	*10 mult, 10 hash*
10MB/1KB	*10000 mult, 10000 hash*	*650 KB*	*10.3 KB*	*10 mult, 10 addition*	*10 mult, 10 hash*
1GB/1KB	*1M mult, 1M hash*	*65MB*	*10.3 KB*	*10 mult, 10 addition*	*10 mult, 10 hash*
100KB/10KB	*10 mult, 10 hash*	*650 B*	*100 KB*	*10 mult, 10 addition*	*10 mult, 10 hash*
10MB/10KB	*1000 mult, 1000 hash*	*65 KB*	*100 KB*	*10 mult, 10 addition*	*10 mult, 10 hash*
1GB/10KB	*100K mult, 1000K hash*	*6.5 MB*	*100 KB*	*10 mult, 10 addition*	*10 mult, 10 hash*

Note: $|D|$ means size of file *D*. Note also that although the server must verify D_i*'s* membership of *D* using merkle proofs, the efficiency of the production and verification of these proofs is not included in the above table.

1. Security level (probability with which malicious user will be detected) growths as $1-2^{-m}$. Only size of data that should be sent to the server during verification phase is influenced from it.

9 Conclusion and future work

We have presented a scheme for proof-of-custody that allows for the production of a cryptographic proof that a given user possesses a particular file and private key at the same time. This scheme can provide the basis of file-sharing that is paid for by a third party, without requiring trust in the downloading and uploading parties. We believe that this can be the basis for the creation of important cryptoeconomic tools. Some future work will therefore involve writing a software library for producing and verifying proofs-of-custody. However, it is not clear that the proof-of-custody scheme given here is optimal. There may be other methods which require fewer commitments, or which require less computation to produce or verify. Future work will focus on developing more concise and efficient methods of producing proofs-of-custody.

Acknowledgements

We thank Ethan Buchman for countless conversations about the custody problem from the day it was initially formulated, and for his help editing this manuscript; Alex Shevchuk, for useful discussions regarding this work, and Marco Kiewe for providing perfect conditions in which this problem was solved. The term “proof-of-custody” was coined by Charles Hoskinson on April 10, 2014.

References

[1] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In Proceedings of 17th Ann. ACM Symp. on Theory of Computing, pages 291–304, 1985. A journal version under the same title appears in: SIAM Journal of Computing vol. 18, pp. 186–208, 1989.

[2] C.P. Schnorr. Efficient identification and signature for smart cards. In G. Brassard, editor, Advances in Cryptology — Proceedings of CRYPTO'89, Lecture Notes in Computer Science 435, pages 239–252. Springer-Verlag, 1990.

[3] C. Percival. Stronger key derivation via sequential memory-hard functions. Presented at BSDCan’09, May 2009.

[4] A. Fiat and A. Shamir. How to prove yourself: practical solutions of identification and signature problems. In A.M. Odlyzko, editor, Advances in Cryptology — Proceedings of CRYPTO'86, Lecture Notes in Computer Science 263, pages 186–194. Springer-Verlag, 1987.

Appendix

Here we will show other 'candidate' solutions to the custody problem that were found to be vulnerable. We include these examples to show attacks on proof-of-custody schemes, and to share solutions that 'almost work' and may be useful in slightly different contexts.

1. Short hash scheme

To provide a proof, Bob:

1. Splits data into small parts $\{D_i\} = \{D_1, D_2 \dots D_n\}$.
2. Signs every D_i with private key, producing $\{S_i\}$.
3. Hash each S_i to a small result value H_i (some modified hash that produces a t bit result out of a 256 bit input, t could be from 4 to 16), yielding the set $\{H_i\}$.
4. Uploads $\{H_i\}$ to the server.

To verify the PoC, the auditing server:

1. Randomly selects m different parts of file.
2. Requests selected parts from the user (or Alice or anyone).
3. Requests the corresponding $\{S_i\}$ from Bob.

After Bob provides the requested data, server:

1. Verifies signatures $\{S_i\}$ on $\{D_i\}$.
2. Hashes received $\{S_i\}$ and compares with committed values $\{H_i\}$. Accepts proof if all $\{S_i\}$ are correct.

Problem: Because ECDSA signatures are non-deterministic (their creation is a function of a random component, r) an adversary can always try to find an r such that the signature derived from it will match the small hash that was uploaded earlier. Therefore one can create arbitrary proofs (composed of small hashes) and find valid signatures that give the same small hashes after the server chooses what hashes it will verify. This attack is effective only for small hashes, because the attacker should perform approximately 2^t signing and hashing operations for each of the requested proofs to find an appropriate r. For the proof to be concise, however, we must require that t be small.

2. Short signature scheme

To create a proof, Bob:

1. Splits data into small parts $\{D_i\} = \{D_1, D_2 \dots D_n\}$.

2. Uses a modified signature algorithm that produces smaller signatures than the key size (for example key size is 256 bit, signature 16 bit) to sign each D_i with the private key, yielding the set $\{S_i\}$.
3. Sends $\{S_i\}$ to the server.

To verify the proof, the server,

1. Randomly selects m different parts of file.
2. Requests $\{D_i\}$ from Bob (or Alice, or anyone).
3. Verifies signatures S_i on D_i.

Problem: This requires a modified signature algorithm. This could be done either by modifying a signature algorithm so that it produces tiny signatures (each of them can be verified with normal public key, but are weak, because the signatures are short). We have not yet found a way to create appropriately short signatures that can still be verified with an EC public key.

3. MAC scheme

To provide a proof, Bob:

1. Splits data into small parts $\{D_i\} = \{D_1, D_2 \dots D_n\}$.
2. Calculates shared key f for Bob and the auditing server, $f = sQ_S$ (where Q_S is public key of the server, s is private key of Bob).
3. Obtains a short MAC (message authentication code) for each piece D_i, using f as a symmetric key, yielding the set $\{MAC_i\}$ and sends it to the server.

To verify the proof server:

1. Randomly selects m different parts of file.
2. Requests $\{D_i\}$ from Bob (or Alice, or anyone).
3. Recovers shared key f like $f=rQ_B$ (where Q_B is public key of Bob, r is private key of the server).
4. Verifies signatures MAC_m on D_m

Problem: Bob can give f to Alice, who will calculate correct proofs. It does not harm Bob's security because f does not reveal any information about s.

There is one more practical way of constructing the proof-of-custody using bilinear pairings, but they require usage of novel cryptography (supersingular elliptic curves) that is not very common.