# Capsule: A Protocol for Secure Collaborative Document Editing

Nadim Kobeissi

Symbolic Software
nadim@symbolic.software

**Abstract.** Today's global society strongly relies on collaborative document editing, which plays an increasingly large role in sensitive workflows. While other collaborative venues, such as secure messaging, have seen secure protocols being standardized and widely implemented, the same cannot be said for collaborative document editing. Popular tools such as Google Docs, Microsoft Office365 and Etherpad are used to collaboratively write reports and other documents which are frequently sensitive and confidential, in spite of the server having the ability to read and modify text undetected.

Capsule is the first formalized and formally verified protocol that addresses secure collaborative document editing. Capsule provides confidentiality and integrity on encrypted document data, while also guaranteeing the ephemeral identity of collaborators and preventing the server from adding new collaborators to the document. Capsule also, to an extent, prevents the server from serving different versions of the document being collaborated on.

In this paper, we provide a full protocol description of Capsule. We also provide formal verification results on the Capsule protocol in the symbolic model. Finally, we present a full software implementation of Capsule, which includes a novel formally verified signing primitive implementation.

**Keywords:** collaborative document editing, secure document editing protocol, formal verification, proverif, privacy enhancing technology

## 1 Introduction

Collaborative document editing software such as Google Docs and Microsoft Office365 has become indispensable in today's work environment. In spite of no confidentiality guarantees, large working groups still find themselves depending on these services even for drafting sensitive documents. Peer pressure, a rush to agree on a working solution and the sheer lack of alternatives have created an ecosystem where a majority of sensitive drafting is currently conducted with no end-to-end encryption guarantees whatsoever. Google and Microsoft servers have full access to all documents being collaborated upon and so do any parties with read access to the internal databases used by these services.

Capsule aims to solve this problem by being the first formally specified and formally verified protocol for secure collaborative document editing. Capsule comes with security goals that are validated in the symbolic model using the ProVerif [1] automated protocol verifier. It is also presented with a full client and server implementation, published as open source software.

We note that Capsule does not target the much simpler problem of encrypting cloud documents that are at rest, such as files stored on Google Drive on Microsoft OneDrive. Rather, the Capsule Protocol targets files being actively, collaboratively edited in real time.

Capsule's more involved use case scenario is responsible for providing a class of security guarantees vaguely resembling that of group secure messaging but without a need for forward secrecy. Another element that Capsule has to deal with is allowing the management of a document's incremental history with a server that cannot read the document's contents. This is accomplished by storing the document as *an authenticated hash chain of encrypted, signed diffs:* when a new participant joins a Capsule document, they pull the entire document hash chain, decrypting and parsing every diff until the entire document is reconstructed. Once within the document, each participant pushes and pulls encrypted diff blocks into this hash chain. Utilizing a hash chain is not only more efficient than more naïve approaches, but also makes it more difficult for the server to provide differing document histories to participants, without forking the hash chain entirely and with little payoff.

Capsule uses symmetric encryption to guarantee document confidentiality and integrity. Cryptographic signatures are used for authentication, although we employ only *ephemeral authentication* since identities are valid for the lifetime of the document. Symmetric primitives are also used for generating a proof value that disallows the server from adding non-existent participants to the document, in spite of these fake participants already being unable to access any document plaintext.

### 1.1 Security Goals

Capsule aims to guarantee the following security goals. In the following, a *valid participant* is any entity with access to the Capsule collaborative document's shared master secret. This master secret is generated upon document creation and is shared manually by the document creator in order to allow access to the document.

- **Participant List Integrity.** Only participants with access to the Capsule collaborative document's shared master secret may appear as valid entries on the participant list. Illegitimate entries injected by the server or any other parties must be detectable by valid participants.
- **Confidentiality.** Any changes made to a collaborative document may only be viewed by valid participants.
- **Integrity.** Encrypted diffs that are appended to the Capsule document's hash chain by a particular author cannot be tampered with by any other party.

- **Ephemeral Authentication.** The identity keys which are generated for the lifetime of the Capsule document session must successfully identify the participant that owns them and disallow impersonation. Public identity keys must benefit from Trust on First Use logic and be verifiable out-of-band.
- **Transcript Consistency.** A malicious server cannot selectively omit changes to the collaborative document short of entirely forking the hash chain. Furthermore, in no scenario may a malicious server maliciously *append* changes to the collaborative document.

### 1.2 Threat Model

Capsule assumes the following threat model, which is fairly standard for protocols aiming to provide end-to-end security:

- **Untrusted Network.** We assume that an attacker controls the network and so can intercept, tamper with and inject network messages.
- **Malicious Server.** We assume that a Capsule server may be potentially interested in misleading users with regards to the document's history and in the apparent list of identities participating in a collaborative document editing session.

### 1.3 Related Work

To the best of our knowledge, the only prior existing work regarding collaborative document encryption is CryptPad [2], an open source web client. CryptPad shares similarities with Capsule especially in that both use a hash chain of encrypted diffs in order to manage document collaboration and to reconstruct the document. However, CryptPad adopts a more relaxed threat model of an "honest but curious cloud server" and does not appear to guard against a server interfering with the document's list of participants or its history. Meanwhile, Capsule explicitly guards against a server injecting false participants by requiring a certain proof from all participants. CryptPad's software implementation is also limited within a web browser and unlike Capsule's, does not employ formally verified cryptographic primitives.

## 2 Primitives

In designing the Capsule protocol, we wanted to focus on obtaining the smallest attack surface possible on an architectural cryptographic level. This is why our low-level cryptographic operations comprise of a restricted subset that is simple to illustrate. The practical details of which cipher suites are chosen to satisfy the security goals of these primitives is discussed in §5.2.

- **Hashing.** $\mathsf{HASH}(x) \longrightarrow y$. A standard one-way cryptographic hash function.
- **Hash-Based Key Derivation.** $\mathsf{HKDF}(k, \texttt{salt}) \longrightarrow (k_0, k_1, k_2, k_3)$. HKDF functions [3] are proven to be pseudorandom functions under a random oracle model, which is useful for aspects of Capsule's protocol design.
- **Encryption and Decryption.**
  - **Encryption.** $\mathsf{BENC}(ek, mk, p) \longrightarrow (\mathsf{ENC}_p, n)$. Where $ek$ is an encryption key and $mk$ is an authentication key. An authentication tag is generated, as is typical with authenticated symmetric ciphers.
  - **Decryption.** $\mathsf{BDEC}(ek, mk, \mathsf{ENC}_p, n) \longrightarrow \{p, \perp\}$ depending on whether decryption can be authenticated.
- **Signatures.**
  - **Key Generation.** $\mathsf{EDGEN}(sk) \longrightarrow pk$.
  - **Signing.** $\mathsf{EDSIGN}(sk, x) \longrightarrow \mathsf{SIG}_x$.
  - **Signature Verification.** $\mathsf{EDVERIF}(pk, x, sig_x) \longrightarrow \{\top, \perp\}$ depending on whether signature verification succeeds.

## 3 Protocol Description

During the lifetime of a Capsule collaborative document, there are only two subprotocols that a participant has to follow. The first is the key generation subprotocol which produces the necessary key material to conduct the session. The second is the hash chain [4] protocol, which we call *DiffChain*. By pushing and pulling encrypted diffs to the diff chain, participants can reconstruct the document upon joining the session and then begin exchanging modifications.

### 3.1 Key Material

The following key material is necessary for all participants.

**Client Key Materials** A client $A$ owns the following key material in relationship to collaborative document $V$:

- $V_k \xleftarrow{R} \{0,1\}^{128}$, a randomly generated master secret, selected by the document creator or otherwise obtained from the document creator. Acts as the lifetime token for legitimately joining a collaborative document.
- $(V_{ek}, V_{mk}, V_{sp}, V_{id}) \longleftarrow \mathsf{HKDF}(V_k, \texttt{CAPSULECORP})$.[1] A set of symmetric subkeys used for encryption, message authentication, proof of being a legitimate participants and to derive the document ID used by the server for bookkeeping.
- $AV_{sk} \xleftarrow{R} \{0,1\}^{256}$, a signing private key.
- $AV_{pk} \longleftarrow \mathsf{EDGEN}(AV_{sk})$.
- $AV_{pv} \longleftarrow \mathsf{PVCALC}(V_{sp}, V_{id}, A, AV_{pk}) = \mathsf{HMAC}(V_{sp},\ A \| V_{id} \| AV_{pk})$

[1] `CAPSULECORP` represents a salt encoded as a string.

In the above, PVCALC is simply a shorthand function for the HMAC construction following it.

Among the above values, $AV_{pv}$ is called a *proof of participation value*: Since the server does not know $V_{sp}$, it cannot generate a $xV_{pv}$ for any possible participant $x$. Validating this value, therefore, protects the collaborative document editing session from containing illegitimate participants who were not given access to $V_k$.

**Server Key Materials** A server $S$ obtains access to the following key materials in relationship to collaborative document $V$:

- $V_{id}$, used as the server-side identifier for the document.
- $AV_{pk}$ as Alice's ephemeral identity. Optionally, the server can also use this to validate DiffChain blocks sent in by Alice for commitment.
- $AV_{pv}$, used by Alice to prove that her identity is being served as a legitimate participant to the collaborative document.

### 3.2 Session Setup

Suppose Alice ($A$) wants to create a new collaborative document. She generates the keys mentioned in §3.1 and then communicates the following key material to server[2] $S$:

$$A \xrightarrow{(V_{id}, AV_{pk}, AV_{pv})} S$$

The server creates the new document identified server-side as $V_{id}$ and stores the triple $(A, AV_{pk}, AV_{pv})$ as the first participant to the document.

Session setup is now complete. In order to invite a fellow collaborator into the document, Alice simply shares $V_k$. Alice can choose to encode $V_k$ as a string or QR code to make sharing easier. Once Bob obtains $V_k$, he too can immediately generate all necessary key material to join the document, encrypt and authenticate diff information and prove to the rest of the participants that his participation is legitimate.

When Bob ($B$) joins the document, the following occurs (the last message is repeated for every existing participant:)

$$B \xrightarrow{(V_{id}, BV_{pk}, BV_{pv})} S$$

$$B \xleftarrow{(V_{id}, AV_{pk}, AV_{pv})} S$$

At this stage, Bob may carry out several validation operations:

[2] Transport layer protections for this and other communications listed below are beyond the scope of this protocol. For simplicity, we assume they are protected with a transport layer security model equivalent to a regular TLS 1.3 deployment.

- Verifying that $V_k$ maps to $V_{id}$.
- Depending on mutual authentication requirements, verifying that $AV_{pk}$ is expected for some known identity for Alice.
- Verifying that $\mathsf{HMAC}(V_{sp}, A \| V_{id} \| AV_{pk}) = AV_{pv}$.

### 3.3 Managing Collaborative Document History with DiffChain

As a protocol, Capsule needs to provide a fast, efficient method to allow for the continuous update of a document by an unbounded number of participants and for the constant synchronization of this document between the participants. Aside from homomorphic encryption, which currently does not appear to be ready for use in this kind of real-world system, the other clear potential solution seemed to be an encrypted, append-only authenticated log of diff information. We construct such a data structure and call it DiffChain.

Pulling a DiffChain to reconstruct a collaborative document upon joining it is very similar to pulling the Bitcoin blockchain and parsing it in order to reconstruct the currency's current value and activity by going through all transactions since the original block. A DiffChain block contains the following data:

- **Encrypted Diff.** Alice generates this encrypted diff using $\mathsf{BENC}(V_{ek}, V_{mk}, \texttt{diff})$.
- **Hash of Previous Block.** This is a standard element in hash chain constructions.
- **ID of Current Block.** This is conventionally a UNIX timestamp. It is appended by the server.
- **ID of Previous Block.** This allows for faster lookups.
- **Signature.** All of the above fields are included in a single signature, with the exception of the ID of the current block since it is appended by the server.

**Responsible pushing and pulling.** As a rule, clients should not push any new blocks before obtaining confirmation from the server that their local diff state is current. This will help avoid merge conflicts.

**Diffing algorithm.** The Capsule library comes equipped with an efficient diff generation algorithm that features a JSON-compatible syntax for compatibility. Independent implementations are free to adapt their own diff payload representations, as this does not strictly affect the security or operation of the cryptographic protocol itself.

### 3.4 Protocol Variant: Unauthenticated Encryption

We observe that if all participants accept ciphertexts signed only by identities with a valid $XV_{pv}$ value for any participant $X$, then the authentication component of the $\mathsf{ENC}$ becomes redundant. This observation allows potentially replacing $\mathsf{ENC}$ with a unauthenticated encryption cipher, which could lead to improved performance.

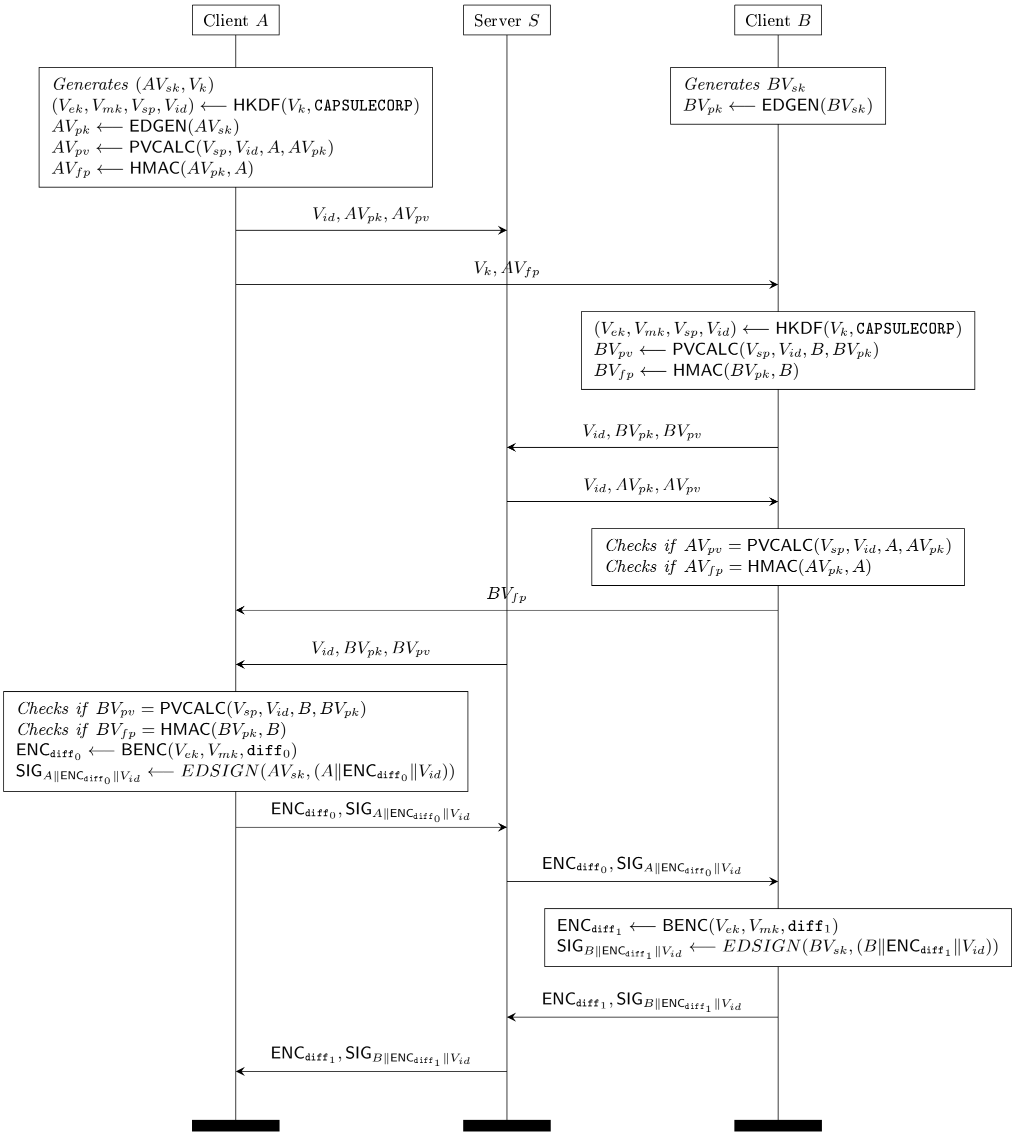
Client $A$
Server $S$
Client $B$
*Generates* $(AV_{sk}, V_k)$
$(V_{ek}, V_{mk}, V_{sp}, V_{id}) \longleftarrow \mathsf{HKDF}(V_k, \texttt{CAPSULECORP})$
$AV_{pk} \longleftarrow \mathsf{EDGEN}(AV_{sk})$
$AV_{pv} \longleftarrow \mathsf{PVCALC}(V_{sp}, V_{id}, A, AV_{pk})$
$AV_{fp} \longleftarrow \mathsf{HMAC}(AV_{pk}, A)$
*Generates* $BV_{sk}$
$BV_{pk} \longleftarrow \mathsf{EDGEN}(BV_{sk})$
$V_{id}, AV_{pk}, AV_{pv}$
$V_k, AV_{fp}$
$(V_{ek}, V_{mk}, V_{sp}, V_{id}) \longleftarrow \mathsf{HKDF}(V_k, \texttt{CAPSULECORP})$
$BV_{pv} \longleftarrow \mathsf{PVCALC}(V_{sp}, V_{id}, B, BV_{pk})$
$BV_{fp} \longleftarrow \mathsf{HMAC}(BV_{pk}, B)$
$V_{id}, BV_{pk}, BV_{pv}$
$V_{id}, AV_{pk}, AV_{pv}$
*Checks if* $AV_{pv} = \mathsf{PVCALC}(V_{sp}, V_{id}, A, AV_{pk})$
*Checks if* $AV_{fp} = \mathsf{HMAC}(AV_{pk}, A)$
$BV_{fp}$
$V_{id}, BV_{pk}, BV_{pv}$
*Checks if* $BV_{pv} = \mathsf{PVCALC}(V_{sp}, V_{id}, B, BV_{pk})$
*Checks if* $BV_{fp} = \mathsf{HMAC}(BV_{pk}, B)$
$\mathsf{ENC}_{\texttt{diff}_0} \longleftarrow \mathsf{BENC}(V_{ek}, V_{mk}, \texttt{diff}_0)$
$\mathsf{SIG}_{A\|\mathsf{ENC}_{\texttt{diff}_0}\|V_{id}} \longleftarrow EDSIGN(AV_{sk}, (A\|\mathsf{ENC}_{\texttt{diff}_0}\|V_{id}))$
$\mathsf{ENC}_{\texttt{diff}_0}, \mathsf{SIG}_{A\|\mathsf{ENC}_{\texttt{diff}_0}\|V_{id}}$
$\mathsf{ENC}_{\texttt{diff}_0}, \mathsf{SIG}_{A\|\mathsf{ENC}_{\texttt{diff}_0}\|V_{id}}$
$\mathsf{ENC}_{\texttt{diff}_1} \longleftarrow \mathsf{BENC}(V_{ek}, V_{mk}, \texttt{diff}_1)$
$\mathsf{SIG}_{B\|\mathsf{ENC}_{\texttt{diff}_1}\|V_{id}} \longleftarrow EDSIGN(BV_{sk}, (B\|\mathsf{ENC}_{\texttt{diff}_1}\|V_{id}))$
$\mathsf{ENC}_{\texttt{diff}_1}, \mathsf{SIG}_{B\|\mathsf{ENC}_{\texttt{diff}_1}\|V_{id}}$
$\mathsf{ENC}_{\texttt{diff}_1}, \mathsf{SIG}_{B\|\mathsf{ENC}_{\texttt{diff}_1}\|V_{id}}$

**Fig. 1.** Capsule protocol session network messages. Here, $A$ initiates a session with $S$ and then communicates session key $V_k$ and her own fingerprint $AV_{fp}$ to $B$ out of band. After $B$ joins and all checks pass, $A$ commits an example encrypted diff, $\texttt{diff}_0$ and $B$ follows up with an example diff $\texttt{diff}_1$. More involved sessions can include clients $C$, $D$, etc. with a non-deterministic diff commit schedule.

*Proof.* We assume that all clients are honest. We assume that HMAC is a one-way keyed hash function, that $V_k$ is secret and that $AV_{pk}$ is authenticated as a public signing identity belonging to $A$:

- If value $AV_{pv}$ is validated, this proves that generator $A$ must possess $V_{sp}$.
- Given that $V_k$ is secret, ownership of $V_{sp}$ in turn proves that $A$ must possess $V_k$.
- At no point in a correct execution of the protocol are the values $(V_{ek}, V_{mk}, V_{sp})$ leaked by any participant.
- $V_k$ maps into $(V_{ek}, V_{mk}, V_{sp}, V_{id})$. Therefore, proving ownership of $V_{sp}$ under a secret $V_k$ is equivalent to proving ownership of $(V_{ek}, V_{mk}, V_{sp}, V_{id})$. This satisfies the original purpose of validating the proof of participation value.
- $A$ cannot produce $AV_{pv}$ without also being able to produce $(V_{ek}, V_{mk})$. Since signature verification disallows tampering, any unauthenticated ciphertext encrypted under $V_{ek}$ and signed by $A$ is equivalent to a ciphertext encrypted under $V_{ek}$, authenticated using $V_{mk}$ and signed by $A$. □

## 4 Symbolic Verification with ProVerif

We describe Capsule using the applied-pi calculus and, using the ProVerif symbolic protocol verifier, verify that the Capsule protocol meets its security goals under its proposed threat model.

In the symbolic model, cryptographic primitives are logically perfect black boxes: hash functions are one-way mappings, encryption functions are permutations, random values are unique and perfectly indistinguishable. Processes can then be executed in parallel. In this setting, a Dolev-Yao [5] active attacker exists and attempts to reconstruct inputs and outputs in such a way as to reach goals defined by queries and events.

### 4.1 Capsule Processes in ProVerif

In ProVerif, we describe a single public channel, `pub`, intending to represent regular Internet network exchanges. Then, two types of client processes are described. We illustrate them here in an abbreviated format.[3]

**Writer Client Processes** The writer process running under identity $W$ joins a document $V$ using a pre-shared $V_k$. It calculates $WV_{pk}$, emits a `ProofSent` event and sends $(V_{id}, W, WV_{pk}, WV_{pv})$ over the network. What follows next is an unbounded execution of the `writeBlock` process, with unbounded newly instantiated plaintext diff. Note that for modeling purposes, we "tag" the plaintext diff using the construction `secretdiff` which takes in a value of type `plaintext` and produces a value of type `plaintext`. This constructor is paired with a reductor, `issecretdiff`, that simply reveals whether its input value was constructed using `secretdiff`.

[3] Capsule models and reference implementations are available at https://symbolic.software/capsule/.

```
let writerClient(vk:key, w:principal, sk:key) =
let pk = edgen(sk) in
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
let pv = pvcalc(vsp, vid, w, pk) in
event ProofSent(w, pv);
out(pub, (vid, w, pk, pv));
!(
  new diffx:plaintext;
  writeBlock(vk, w, sk, secretdiff(diffx))
).
```

The `writeBlock` process is self-explanatory:

```
let writeBlock(vk:key, w:principal, sk:key, diffx:plaintext) =
let pk = edgen(sk) in
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
let ediff = benc(vek, vmk, diffx) in
let ediffsig = edsign(
  sk, concat3(prin2bit(w), ediff, key2bit(vid))
) in
event Pushed(vid, w, pk, ediff, ediffsig);
out(pub, (vid, w, pk, ediff, ediffsig)).
```

**Reader Client Processes** The reader process running under participant identity $R$ comes equipped with a value $UV_{fp} \longleftarrow \mathsf{HMAC}(UV_{pk}, U)$ that serves as a fingerprint for authenticating public identities and their public signing keys out of band.

Mirroring the writer processes above, the reader process begins by similarly generating the requisite values and communicating them over the network. Then, an unbounded execution of the `readBlock` process occurs.

```
let readerClient(vk:key, r:principal, sk:key, ufp:bitstring) =
let pk = edgen(sk) in
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
let pv = pvcalc(vsp, vid, r, pk) in
event ProofSent(r, pv);
out(pub, (vid, r, pk, pv));
!(readBlock(vk, r, ufp)).
```

The `readBlock` process receives over the network a triple of candidate values for $(U, UV_{pk}UV_{pv})$ with a header matching the $V_{id}$ value that the reader was able to obtain using the $V_k$ the process was initialized with. Then, the internal $UV_{fp}$ and $UV_{pv}$ are checked against the candidate values received over the network.

If the check succeeds, a ProofVerified event is emitted. Then, an encrypted diff along with its signature is received over the network with a header matching $(V_{id}, U, UV_{pk})$. Once signature and authenticated encryption checks pass, the *Pulled* event is emitted.

```
let readBlock(vk:key, r:principal, ufp:bitstring) =
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
in(pub, (=vid, user0:principal, upk0:key, upv0:bitstring));
let ufp0 = hmac(upk0, prin2bit(user0)) in
let upv = pvcalc(vsp, vid, user0, upk0) in
if ((ufp0 = ufp) && (upv0 = upv)) then (
  event ProofVerified(user0, r, upv);
  in(pub, (
    =vid, =user0, =upk0,
    ediff0:bitstring, ediff0sig:bitstring
  ));
  let sigver0 = edverif(
    upk0, ediff0sig,
    concat3(prin2bit(user0), ediff0, key2bit(vid))
  ) in
  if (sigver0 = true) then (
    let vs0:valid = bdec(vek, vmk, ediff0) in
    let (v0:bool, s0:plaintext) = validunpack(vs0) in
    if (v0 = true) then (
      event Pulled(vid, user0, r, upk0, ediff0, ediff0sig)
    ))).
```

**Attacker and Top-level Processes** Before declaring the top-level process, we must declare a process where the attacker attempts to obtain the plaintext for any diff:

```
let attackerTryingToObtainPlaintext() =
in(pub, x:plaintext);
if (issecretdiff(x) = true) then (
event PlaintextObtainedByAttacker(x)
).
```

Now, we can finally declare the top-level process. We model an unbounded number of sessions, each with a new $V_k$ and its own unbounded instantiations of reader and writer principals with their own unique signing key pairs.

```
process
out(pub, msk) |
!(
  new vkx:key;
  !(
    new alice:principal;
    new bob:principal;
new ask:key;
new bsk:key;
out(pub, (alice, bob, sigpk(ask), sigpk(bsk)));
!(
  writerClient(vkx, alice, ask) |
  writerClient(vkx, bob, bsk) |
  readerClient(vkx, alice, ask, hmac(sigpk(bsk), prin2bit(bob))) |
  readerClient(vkx, bob, bsk, hmac(sigpk(ask), prin2bit(alice))) |
  attackerTryingToObtainPlaintext()
)))
```

## 4.2 Security Goals in the Symbolic Model

We apply ourselves on the security goals described in §1.1 in order to model and verify participant list integrity, confidentiality, integrity and ephemeral authentication in ProVerif.[4]

**Participant List Integrity** We assert that if a proof of participation value $XV_{pv}$ is verified by participant $Y$, then there must exist an identity $X$ which has issued $XV_{pv}$. As shown in §3.1, this means that whoever asserts identity $X$ must also possess $V_{sp}$:

$$\mathsf{event\ ProofVerified}(X, Y, XV_{pv}) \implies \mathsf{event\ ProofIssued}(X, XV_{pv})$$

**Confidentiality** We simply query for whether the attacker can trigger the plaintext obtention event described above for any `plaintext`-type bitstring $m$:

$$\mathsf{query\ event\ PlaintextObtainedByAttacker(m)}$$

**Integrity and Ephemeral Authentication** We assert that if a DiffChain block $(\mathsf{ENC}_{\mathtt{diff}}, \mathsf{SIG}_{X\|\mathsf{ENC}_{\mathtt{diff}}\|V_{id}})$ was received by $Y$ as part of document $V$, then this block will successfully authenticate and decrypt as originating from identity $X$ if and only if it was pushed by $X$ for document $V$.

$$\mathsf{event\ Pulled}(V_{id}, X, Y, XV_{pk}, \mathsf{ENC}_{\mathtt{diff}}, \mathsf{SIG}_{X\|\mathsf{ENC}_{\mathtt{diff}}\|V_{id}}) \implies \\ \mathsf{event\ Pushed}(V_{id}, X, XV_{pk}, \mathsf{ENC}_{\mathtt{diff}}, \mathsf{SIG}_{X\|\mathsf{ENC}_{\mathtt{diff}}\|V_{id}})$$

## 4.3 Results Under a Dolev-Yao Attacker

All queries mentioned above complete successfully. By tweaking the model, we are also able to quickly test for the event where an ephemeral identity is not verified out-of-band by the other participants. Since all encryption is symmetric and no asymmetric key agreement ever occurs in the Capsule protocol, the outcome of this is minor and does not result in the compromise of confidential information.

[4] Transcript consistency, while not explicitly modeled in ProVerif, is obtained through the hash chain structure of DiffChains, as described in §3.3.

## 5 Software Implementation

Capsule is provided with a client/server reference implementation called Capsulib. It is meant to allow for quick deployment and testing of the Capsule protocol in production.

### 5.1 Capsulib: a Capsule Client/Server Implementation

Capsulib Server is a Node.js application that uses Redis as a database backend but is otherwise self-contained. It achieves low latency by communicating over WebSockets. Its main purpose is to hold a DiffChain record for every document and to facilitate the exchange of encrypted blocks.

Capsulib Client is meant to be executed within an Electron runtime. It provides a user interface, the Capsulib cryptographic library and the full Capsule client protocol implementation.

### 5.2 Cryptographic Cipher Suite

We expose the following cryptographic functions:

- **Hashing.** $\mathsf{HASH}(x) \longrightarrow y$. BLAKE2s is used as the hash function.
- **Hash-Based Key Derivation.** $\mathsf{BKDF}(k, \texttt{salt}) \longrightarrow (k_0, k_1, k_2, k_3)$. BLAKE2s is also used for key derivation, producing four 16-byte outputs.
- **Encryption and Decryption.** BLAKE2 is used for encryption and decryption. We hash the encryption key over a counter and a nonce to generate a keystream. BLAKE2 is then used as a keyed hash with a MAC key to generate an HMAC value over the ciphertext.
  - **Encryption.** $\mathsf{BENC}(ek, mk, p) \longrightarrow (\mathsf{ENC}_p, n)$.
  - **Decryption.** $\mathsf{BDEC}(ek, mk, \mathsf{ENC}_p, n) \longrightarrow p$.
- **Signatures.** Ed25519 is chosen due to its speed and minimal input validation requirements.
  - **Key Generation.** $\mathsf{EDGEN}(sk) \longrightarrow pk$.
  - **Signing.** $\mathsf{EDSIGN}(sk, x) \longrightarrow \mathsf{SIG}_x$.
  - **Signature Verification.** $\mathsf{EDVERIF}(pk, x, sig_x) \longrightarrow \{\top, \bot\}$.

### 5.3 Formally Verified Cryptographic Primitives in WebAssembly

While Capsulib is written in JavaScript, much of the Capsulib cryptographic primitives library is automatically generated as WebAssembly (WASM) [6] code. We use a toolchain that provides strong verification guarantees, hence ruling out potentially catastrophic bugs in the cryptographic layer.

Ed25519 is specified in F* [7], an ML-like programming language with support for program verification via the use of a dependent type system, effect annotations and SMT-based automation. Our specifications are executable, which allows us to run them against the RFC test vectors to ensure their correctness.

The cryptographic algorithms themselves are written in Low*, a subset of F* that enjoys an optimized compilation scheme to low-level targets. The Low* implementation of our algorithms is shown to match the original F* specification, which ensures functional correctness and memory safety.

Based on this, we rule out both incorrect math and memory errors such as buffer overflows. In addition to function correctness and memory safety, we enforce basic side-channel resistance, by restricting secrets to a very limited set of operations. In effect, this ensures that secret-manipulating code is branch-free and cannot access memory using a secret index. This helps rule out classic timing and cache attacks.

Once verified, Low* programs can be compiled to low-level targets using the Kremlin [8] compiler, which currently supports C and WASM as backends. Compared to Emscripten, the Kremlin compiler offers a much smaller trusted computing base, smaller resulting WASM files and almost no dependencies on untrusted code, such as a `libc` implementation.

The cryptographic primitives that we compile into WebAssembly using this process are originally part of the HACL* [9] cryptographic library, specified in F*.

## 6 Conclusion

In this paper, we have presented Capsule, the first formally verified protocol for secure collaborative document editing. We believe Capsule will address an important need in the space of privacy enhancing technologies. We have provided a full description of the Capsule protocol and symbolic verification results for its security goals under the specified threat model.

Finally, we also provide Capsulib, a reference implementation of Capsule based on modern web technologies. While Capsulib is built for web use, it is also the first software project to employ the HACL* formally verified cryptographic library by translating it into WASM. By adopting such technologies, Capsule protocol is able to be presented complete with a practical implementation that follows best practices and achieves sound practical security in deployment.

## Acknowledgments

We would like to thank Karthikeyan Bhargavan, Denis Merigoux and Jonathan Protzenko for various feedback and assistance. We also thank Vitalik Buterin for encouragement to pursue this work.

## References

1. Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *International Conference on Principles of Security and Trust*, pages 226–246. Springer, 2013. 2
2. CryptPad: Zero Knowledge, Collaborative Real Time Editing, 2016. `https://cryptpad.fr/what-is-cryptpad.html`. 3
3. Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. Cryptology ePrint Archive, Report 2010/264, 2010. `https://eprint.iacr.org/2010/264`. 4
4. Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981. 4
5. Paul Syverson, Catherine Meadows, and Iliano Cervesato. Dolev-Yao is no better than machiavelli. Technical report, Naval Research Lab, Washington DC Center for High Assurance Computing Systems, 2000. 8
6. WebAssembly Core Specification, 2018. `https://webassembly.github.io/spec/core/bikeshed/index.html`. 12
7. Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in F. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17, 2017. 12
8. Peng Wang, Karthikeyan Bhargavan, Jean-Karim Zinzindohoué, Abhishek Anand, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Extracting from F* to C: a progress report. 13
9. Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017. 13