`ssh-tresor-ruby`: Encrypting Files with SSH Agents

An Analysis of SSH-Agent-Gated Encryption at Rest

Ronan Potage
Staff Software Engineer, Cisco Meraki

May 9, 2026

Abstract

`ssh-tresor-ruby` encrypts files by using an active SSH agent as a private-key signing oracle. It does not perform public-key encryption with the SSH public key. Instead, it creates a fresh random master key, encrypts the file with AES-256-GCM, asks the SSH agent to sign random per-key challenges, derives wrapping keys from those signatures using HKDF-SHA256, and stores AES-256-GCM encrypted copies of the master key in key slots. A stored file is therefore encrypted at rest; the distinctive access property is better called *SSH-agent-mediated encryption at rest* or *SSH-agent-gated decryption*. The same construction works with a local SSH agent or a forwarded SSH agent: decryption requires that a matching private-key signing operation be reachable through the active agent socket.

This paper gives a correctness proof and a conditional confidentiality proof for the implemented construction. The proof applies to SSH key algorithms whose agent signatures are stable byte-for-byte for the stored challenge, such as Ed25519 and RSA/SHA2 in the tested OpenSSH environment. The current implementation should not be described as working for all SSH key types: randomized signature schemes, notably ECDSA as exercised by OpenSSH here, do not re-derive the same slot key and therefore do not satisfy the correctness theorem.

1 Executive summary

`ssh-tresor-ruby` implements envelope encryption gated by an SSH agent. The encrypted file contains:

- one AES-256-GCM ciphertext of the plaintext under a fresh 256-bit master key;
- one or more key slots, each containing a random challenge and an AES-256-GCM encrypted copy of the master key;
- the SHA-256 fingerprint of each SSH public key, so the decryptor can select the matching agent key.

The workflow is correctly described as *encryption at rest* for the file stored on disk, with an *SSH-agent-gated* decryption condition:

> The file remains encrypted at rest. Decryption requires an active SSH agent capable of producing a private-key signature for a stored challenge. In local use, that usually means the key has been unlocked and loaded into the agent with `ssh-add`. In remote use, that means the forwarded agent socket is still reachable from the remote host.

That wording is more precise than saying "the SSH key is present." The public key and its fingerprint are not secret and are stored in, or derivable from, the file metadata. What must be present is the private-key operation exposed by the SSH agent.

2 Project behavior and terminology

2.1 The operational flows

The construction supports two common flows.

Local agent flow. A user has a passphrase-protected SSH key on disk. The private key is not directly usable until the user runs `ssh-add` and provides the passphrase. Once the key is loaded into the local SSH agent, `ssh-tresor-ruby` can ask that agent to sign slot challenges and can encrypt or decrypt files. If the key is removed from the agent, the agent is stopped, or `SSH_AUTH_SOCK` no longer points to that agent, the stored `SSHTRESR` blob remains encrypted and cannot be decrypted from the public file metadata alone.

Forwarded-agent remote flow. The motivating remote flow is:

1. A user logs in to a remote host with SSH agent forwarding, for example `ssh -A remote`.
2. The remote shell receives an `SSH_AUTH_SOCK` environment variable pointing to a forwarded agent socket.
3. `ssh-tresor-ruby` running on the remote host asks that forwarded agent to sign random challenges.
4. `ssh-tresor-ruby` derives local wrapping keys from those signatures, encrypts the file, and writes a `SSHTRESR` blob on the remote disk.
5. The user logs out. The forwarded socket is gone. The remote host retains only the encrypted blob.

Both flows are forms of encryption at rest because the stored file is ciphertext. The special property is not merely "at rest" storage encryption, but *agent-gated decryptability*: access to the plaintext is coupled to the active availability of a matching SSH-agent signing capability, whether that agent is local or forwarded.

2.2 What the construction is, and is not

`ssh-tresor-ruby` is:

- an envelope encryption scheme;
- a symmetric authenticated encryption scheme for file contents;
- an SSH-agent-mediated key wrapping scheme for the data master key;
- a way to make a stored file unreadable whenever the matching SSH-agent signing capability is unavailable, under the threat model in Section 8.

`ssh-tresor-ruby` is not:

- public-key encryption using the SSH public key;
- a replacement for full-disk encryption;
- protection from a malicious machine while encryption or decryption is actively running;
- protection if an attacker records the slot signature, master key, plaintext, or process memory during the active session.

3 Implementation overview

The repository contains the core construction in four Ruby modules:

File	Cryptographic role
`lib/ssh_tresor/crypto.rb`	Random generation, HKDF-SHA256, AES-256-GCM encryption and decryption.
`lib/ssh_tresor/agent.rb`	SSH agent connection, identity listing, public-key fingerprinting, signing request construction.
`lib/ssh_tresor/tresor.rb`	Envelope encryption orchestration: create slots, recover the master key, encrypt and decrypt payloads.
`lib/ssh_tresor/format.rb`	Binary and armored `SSHTRESR` v3 file format.

The project uses Ruby's standard and bundled libraries:

- `SecureRandom.random_bytes` for 32-byte challenges, 32-byte master keys, and 12-byte GCM nonces;
- `OpenSSL::KDF.hkdf` for HKDF-SHA256, corresponding to RFC 5869 [2];
- `OpenSSL::Cipher.new("aes-256-gcm")` for AES-256-GCM authenticated encryption, whose mode is specified by NIST SP 800-38D [1];
- `UNIXSocket` and `SSH_AUTH_SOCK` for the SSH agent protocol, which is a de-facto protocol currently documented as an IETF Internet-Draft [3];
- `Digest::SHA256` to compute OpenSSH-style SHA-256 public-key fingerprints.

4 The implemented file format

`lib/ssh_tresor/format.rb` defines `SSHTRESR` v3:

```
Header:  SSHTRESR (8) + version (1) + slot_count (1)
Slot:    fingerprint (32) + challenge (32) + nonce (12) + encrypted_key (48)
Data:    nonce (12) + ciphertext including 16-byte AES-GCM auth tag
```

The code constants are:

Listing 1: File-format constants from lib/ssh_tresor/format.rb.

```
MAGIC = "SSHTRESR".b
VERSION = 0x03
FINGERPRINT_SIZE = 32
CHALLENGE_SIZE = 32
NONCE_SIZE = 12
AUTH_TAG_SIZE = 16
MASTER_KEY_SIZE = 32
ENCRYPTED_KEY_SIZE = MASTER_KEY_SIZE + AUTH_TAG_SIZE
SLOT_SIZE = FINGERPRINT_SIZE + CHALLENGE_SIZE + NONCE_SIZE + ENCRYPTED_KEY_SIZE
HEADER_SIZE = 10
```

For each slot, the stored fields have the following security meaning:

Field	Secret?	Meaning
`fingerprint`	No	`SHA256` digest of the SSH public key blob. It selects the matching agent key.
`challenge`	No	32-byte random value that the agent must sign to recreate the slot key.
`nonce`	No	12-byte AES-GCM nonce for encrypting the master key in that slot.
`encrypted_key`	Yes, as ciphertext	32-byte master key encrypted with AES-GCM plus a 16-byte authentication tag.

The data section stores a separate 12-byte GCM nonce and the AES-GCM ciphertext of the plaintext under the master key.

5 Cryptographic construction

5.1 Notation

Let:

- $P \in \{0,1\}^*$ be the plaintext file.
- $K_M \leftarrow \{0,1\}^{256}$ be a randomly generated master key.
- $n_D \leftarrow \{0,1\}^{96}$ be the data AES-GCM nonce.
- $C_D = \mathrm{Enc}_{\mathrm{GCM}}(K_M, n_D, P)$ be the data ciphertext plus authentication tag.
- (pk_i, sk_i) be the SSH public/private key pair represented by an agent key.
- $F_i = \mathrm{SHA256}(\mathrm{ssh\_public\_key\_blob}_i)$ be the stored 32-byte fingerprint.
- $r_i \leftarrow \{0,1\}^{256}$ be a random challenge for slot i.
- $\sigma_i = \mathrm{Sign}_{sk_i}(r_i)$ be the raw signature bytes returned by the SSH agent.
- K_i is the 32-byte slot wrapping key:

$$K_i = \mathrm{HKDF}_{\mathrm{SHA256}}(\sigma_i, \mathrm{salt}, \mathrm{info}, 32),$$

where `salt` is `"ssh-tresor-v3"` and `info` is `"slot-key-derivation"`.

- $n_i \leftarrow \{0,1\}^{96}$ be the slot AES-GCM nonce.
- $E_i = \mathrm{Enc}_{\mathrm{GCM}}(K_i, n_i, K_M)$ be the encrypted master key plus authentication tag.

The file stores:

$$B = \left(\texttt{SSHTRESR}, 3, \ell, \{(F_i, r_i, n_i, E_i)\}_{i=1}^{\ell}, n_D, C_D\right).$$

5.2 Encryption algorithm

For selected agent keys $pk_1, \ldots, pk_\ell$:

1. Generate K_M with `SecureRandom.random_bytes(32)`.
2. For each selected key, generate a random challenge r_i.
3. Ask the SSH agent to compute $\sigma_i = \mathrm{Sign}_{sk_i}(r_i)$.
4. Derive $K_i = \mathrm{HKDF}_{\mathrm{SHA256}}(\sigma_i, \texttt{"ssh-tresor-v3"}, \texttt{"slot-key-derivation"}, 32)$.
5. Encrypt K_M as $E_i = \mathrm{Enc}_{\mathrm{GCM}}(K_i, n_i, K_M)$.
6. Encrypt the file as $C_D = \mathrm{Enc}_{\mathrm{GCM}}(K_M, n_D, P)$.
7. Serialize the slots and data ciphertext as `SSHTRESR` v3.

This is implemented by `encrypt_with_keys` and `create_slot`:

Listing 2: Envelope encryption and slot creation from lib/ssh_tresor/tresor.rb.

```ruby
def encrypt_with_keys(agent, keys, plaintext)
  master_key = Crypto.random_master_key
  slots = keys.map { |key| create_slot(agent, key, master_key) }
  data_nonce = Crypto.random_nonce
  ciphertext = Crypto.encrypt(master_key, data_nonce, plaintext)

  TresorBlob.new(slots: slots, data_nonce: data_nonce, ciphertext: ciphertext)
end

def create_slot(agent, key, master_key)
  challenge = Crypto.random_challenge
  signature = agent.sign(key, challenge)
  slot_key = Crypto.derive_key(signature)
  nonce = Crypto.random_nonce
  encrypted_key = Crypto.encrypt(slot_key, nonce, master_key)

  Slot.new(
    fingerprint: key.fingerprint_bytes,
    challenge: challenge,
    nonce: nonce,
    encrypted_key: encrypted_key
  )
end
```

5.3 Decryption algorithm

Given a `SSHTRESR` blob and an active SSH agent:

1. List the public keys currently available through the agent.
2. For each agent key, find a slot whose stored fingerprint equals `SHA256(public_key_blob)`.
3. Ask the agent to sign the stored challenge r_i, producing σ'_i.
4. Derive $K'_i = \mathrm{HKDF}_{\mathrm{SHA256}}(\sigma'_i, \texttt{"ssh-tresor-v3"}, \texttt{"slot-key-derivation"}, 32)$.
5. Decrypt E_i under K'_i to recover K_M.
6. Decrypt C_D under K_M to recover P.

The implementation is direct:

Listing 3: Slot-based decryption from lib/ssh_tresor/tresor.rb.

```ruby
def decrypt_with_slot(agent, key, slot, blob)
  signature = agent.sign(key, slot.challenge)
  slot_key = Crypto.derive_key(signature)
  master_key = Crypto.decrypt(slot_key, slot.nonce, slot.encrypted_key)
  Crypto.decrypt(master_key, blob.data_nonce, blob.ciphertext)
end
```

6 How the SSH agent is used

`ssh-tresor-ruby` connects to the active agent through `SSH_AUTH_SOCK`:

Listing 4: Agent socket connection from lib/ssh_tresor/agent.rb.

```ruby
def self.connect
  socket_path = ENV["SSH_AUTH_SOCK"]
  raise AgentError, "SSH agent not available\nHint: Is SSH_AUTH_SOCK set? Try running:
     eval $(ssh-agent) && ssh-add" if socket_path.nil? || socket_path.empty?

  new(UNIXSocket.new(socket_path))
rescue SystemCallError => e
  raise AgentError, "Failed to connect to SSH agent: #{e.message}"
end
```

The socket named by `SSH_AUTH_SOCK` may be a local agent socket or an agent-forwarding socket created by `ssh -A`. `ssh-tresor-ruby` does not need to distinguish those cases. It only requires that the socket expose an agent identity capable of signing the stored slot challenge. For a passphrase-protected private key, this usually means the user has already unlocked and loaded the key into the agent with `ssh-add`; the encrypted private-key file on disk is not itself used by `ssh-tresor-ruby`.

It requests signatures with the SSH agent signing operation:

Listing 5: Signing request from lib/ssh_tresor/agent.rb.

```ruby
def sign(key, data)
  flags = key.ssh_type == "ssh-rsa" ? SSH_AGENT_SIGN_REQUEST_RSA_SHA2_256 : 0
  payload = SSHEncoding.byte(SSH_AGENTC_SIGN_REQUEST) +
```

```
            SSHEncoding.string(key.blob) +
            SSHEncoding.string(data) +
            SSHEncoding.uint32(flags)

  response = request(payload)
  reader = SSHEncoding::Reader.new(response)
  type = reader.byte
  raise AgentError, "SSH agent refused signing request" if type == SSH_AGENT_FAILURE
  raise AgentError, "Unexpected SSH agent response type #{type}" unless type ==
    SSH_AGENT_SIGN_RESPONSE

  signature_blob = reader.string
  signature_reader = SSHEncoding::Reader.new(signature_blob)
  signature_reader.string
  signature_reader.string
end
```

The private key is never read by this code. The code sends a public key blob and data to be signed, then receives signature bytes. For RSA keys, it sets the OpenSSH flag requesting RSA/SHA-256 signatures, aligning the implementation with the modern RSA/SHA2 SSH algorithms standardized in RFC 8332 [4].

7 AES-GCM and HKDF implementation

The cryptographic primitives are wrapped in `lib/ssh_tresor/crypto.rb`:

Listing 6: Key derivation and AES-GCM from lib/ssh_tresor/crypto.rb.

```
def derive_key(signature)
  OpenSSL::KDF.hkdf(
    signature,
    salt: "ssh-tresor-v3",
    info: "slot-key-derivation",
    length: 32,
    hash: "SHA256"
  )
end

def encrypt(key, nonce, plaintext)
  cipher = OpenSSL::Cipher.new("aes-256-gcm")
  cipher.encrypt
  cipher.key = key
  cipher.iv = nonce
  cipher.auth_data = "".b

  ciphertext = cipher.update(plaintext.b) + cipher.final
  ciphertext + cipher.auth_tag
rescue OpenSSL::Cipher::CipherError => e
  raise Error, "Encryption failed: AES-GCM encryption failed: #{e.message}"
end
```

The corresponding decrypt function splits the final 16 bytes as the GCM authentication tag, sets `cipher.auth_tag`, and raises `DecryptionError` if OpenSSL rejects the tag. Therefore an incorrect

slot key, an incorrect master key, or a corrupted ciphertext does not produce unauthenticated plaintext.

8 Threat model

The security claim is intentionally narrow and operational:

> After the matching SSH-agent signing capability is no longer reachable, an attacker who obtains only the `SSHTRESR` file cannot recover the plaintext, except with negligible probability, unless they also obtain a matching private-key signing capability or recorded key material from the active session.

The claim assumes:

Assumption 1 (Cryptographic random generation)**.** `SecureRandom.random_bytes` returns unpredictable bytes for challenges, master keys, and nonces.

Assumption 2 (AES-GCM security)**.** AES-256-GCM is confidential and ciphertext-authentic for messages encrypted under a key with non-repeated nonces, as specified in NIST SP 800-38D [1].

Assumption 3 (HKDF security)**.** HKDF-SHA256 maps the SSH signature bytes to a pseudorandom 32-byte key for attackers who do not know, cannot compute, and did not record those signature bytes. This is the normal KDF role described by RFC 5869 [2].

Assumption 4 (Agent signature capability)**.** For supported key types, the SSH agent can reproduce the same signature bytes for the same stored challenge, and an attacker without the private key or agent oracle cannot compute those bytes.

Assumption 5 (Session isolation)**.** The attacker did not compromise the process performing encryption or decryption, did not alter the `ssh-tresor-ruby` binary being run, did not read process memory, and did not record the agent signature, slot key, master key, or plaintext while the agent was reachable.

Assumption 6 (Agent unreachability)**.** After the access window ends, the attacker cannot access an SSH agent containing the matching key and cannot cause such an agent to sign the stored challenge. In local use, this can mean the key has not been loaded with `ssh-add`, has been removed from the agent, or the agent has stopped. In remote use, this can mean the SSH session has closed and the forwarded socket is gone.

These assumptions are exactly the boundary of the user story. If the machine running `ssh-tresor-ruby` is malicious during encryption or decryption, it can log the plaintext or the signature returned by the agent. No file encryption tool can protect data from a machine that is currently handling the plaintext and is allowed to run modified code.

9 Correctness proof

Definition 1 (Supported key for this construction)**.** An SSH agent key is supported by this construction if, for a fixed challenge r, the agent returns the same signature bytes σ during encryption and later decryption, and those bytes verify as a signature under the corresponding public key.

Lemma 1 (Slot recovery)**.** *For a supported key i, decryption of slot i recovers the exact master key K_M generated during encryption.*

Proof. During encryption, the slot challenge is r_i and the agent returns $\sigma_i = \mathrm{Sign}_{sk_i}(r_i)$. The implementation derives:

$$K_i = \mathrm{HKDF}_{\mathrm{SHA256}}(\sigma_i, \texttt{"ssh-tresor-v3"}, \texttt{"slot-key-derivation"}, 32).$$

It then stores:

$$E_i = \mathrm{Enc}_{\mathrm{GCM}}(K_i, n_i, K_M).$$

During decryption, the stored challenge r_i is sent to the same agent key. Because the key is supported, the agent returns the same bytes $\sigma'_i = \sigma_i$. HKDF is deterministic, so $K'_i = K_i$. AES-GCM decryption under the same key and nonce accepts the authentication tag and returns the original 32-byte plaintext of the slot, namely K_M. □

Theorem 1 (End-to-end correctness)**.** *For any plaintext P encrypted for at least one supported SSH agent key that is later available in the agent,* `ssh-tresor-ruby` *decrypts the* `SSHTRESR` *blob to exactly P.*

Proof. By Lemma 1, the matching slot recovers the same master key K_M generated during encryption. The data ciphertext was computed as $C_D = \mathrm{Enc}_{\mathrm{GCM}}(K_M, n_D, P)$. AES-GCM correctness says that decrypting C_D with the same K_M and n_D returns P and accepts the tag. This is exactly the code path in `decrypt_with_slot`. □

10 Confidentiality proof

Theorem 2 (Post-agent unreadability)**.** *Under the assumptions in Section 8, after the matching SSH-agent signing capability is unavailable, the* `SSHTRESR` *file is computationally unreadable to an attacker who possesses only the stored blob.*

Proof. The stored blob exposes F_i, r_i, n_i, E_i, n_D, and C_D. It does not store σ_i, K_i, K_M, or P.

To decrypt the file, an attacker must recover P from C_D. By AES-GCM confidentiality, this requires the master key K_M except with negligible probability. The only stored copies of K_M are the slot ciphertexts E_i. By AES-GCM confidentiality for each slot, recovering K_M from E_i requires the corresponding slot key K_i except with negligible probability.

For each slot, K_i is the HKDF-SHA256 output derived from $\sigma_i = \mathrm{Sign}_{sk_i}(r_i)$. The challenge r_i is public, but by Assumption 4 an attacker without the private key or an agent oracle cannot compute the signature bytes σ_i. By Assumption 3, without σ_i the HKDF output is computationally indistinguishable from an unknown 32-byte key. By Assumption 6, after the access window ends the attacker has no matching signing oracle. Therefore the attacker cannot recover any slot key K_i except with negligible probability.

Since the attacker cannot recover any K_i, they cannot recover K_M from any slot. Since they cannot recover K_M, they cannot recover P from the data ciphertext. Thus the file is unreadable after the active SSH agent capability disappears, except with negligible probability under the stated assumptions. □

10.1 Hybrid view

The same argument can be expressed as a standard sequence of games:

1. Game 0 is the real `ssh-tresor-ruby` encryption output.
2. Game 1 replaces each HKDF-derived slot key with an independent random 256-bit key. Assumptions 3 and 4 make this change indistinguishable to an attacker without the signatures.
3. Game 2 replaces each encrypted master key with an encryption of zeros under the random slot key. AES-GCM confidentiality makes this indistinguishable.
4. Game 3 replaces the data ciphertext with an encryption of zeros under the random master key. AES-GCM confidentiality makes this indistinguishable.

In Game 3 the blob contains no information about the plaintext. Therefore the real blob in Game 0 leaks no computationally useful information about the plaintext beyond length and public metadata.

11 Why agent availability matters

The file is readable while a matching agent is reachable because the agent can be asked to sign the stored challenge. The file becomes unreadable to anyone who has only the stored blob after that signing capability is unavailable. In local use, this can happen when the key is not loaded into the agent or the agent is stopped. In remote use, this can happen when logout removes the forwarded socket from the remote host. At that point, the stored blob contains only:

- public fingerprints;
- public challenges;
- public nonces;
- AES-GCM ciphertexts and tags.

None of those values is enough to compute the slot key. The missing value is the agent signature over the stored challenge. Removing access to the matching SSH agent removes the ability to obtain that value.

12 Compatibility boundary: deterministic signatures

The current design derives key material from the exact signature bytes returned by the agent. This has an important consequence:

> The same key, signing the same stored challenge, must return the same byte string during decryption that it returned during encryption.

This property holds for Ed25519 as specified by EdDSA, which does not require a unique random number for each signature [5]. It also holds for the OpenSSH RSA/SHA2 path exercised by this project, where the code requests the `rsa-sha2-256` signature flag. It does not hold for conventional randomized ECDSA signatures.

The local verification performed while preparing this paper found:

Key type	CLI round trip	Interpretation
Ed25519	Succeeded	Fits the correctness theorem.
RSA 3072	Succeeded	Fits the correctness theorem in the tested OpenSSH/Ruby environment.
ECDSA P-256	Failed	The decryption path could not recover a matching slot key; this is consistent with randomized ECDSA signatures.

Therefore, the project can accurately claim the proved property for deterministic-signature SSH keys. To claim support for every SSH key listed by `ssh-agent`, the implementation would need to change. The most direct hardening is to reject key types that are not known to produce stable signature bytes, or to use a different construction that does not depend on signature byte reproducibility.

13 Integrity and corruption behavior

AES-GCM provides both confidentiality and authentication. `ssh-tresor-ruby` relies on this twice:

- slot decryption authenticates the encrypted master key;
- data decryption authenticates the encrypted file content.

The implementation's `Crypto.decrypt` method rejects too-short ciphertexts and converts OpenSSL GCM tag failures into `DecryptionError`:

Listing 7: Authentication failure handling from lib/ssh_tresor/crypto.rb.

```ruby
def decrypt(key, nonce, ciphertext_with_tag)
  raise DecryptionError, "ciphertext too short" if ciphertext_with_tag.bytesize <
    AUTH_TAG_SIZE

  ciphertext = ciphertext_with_tag.byteslice(0, ciphertext_with_tag.bytesize -
    AUTH_TAG_SIZE)
  tag = ciphertext_with_tag.byteslice(-AUTH_TAG_SIZE, AUTH_TAG_SIZE)

  cipher = OpenSSL::Cipher.new("aes-256-gcm")
  cipher.decrypt
  cipher.key = key
  cipher.iv = nonce
  cipher.auth_tag = tag
  cipher.auth_data = "".b

  cipher.update(ciphertext) + cipher.final
rescue OpenSSL::Cipher::CipherError
  raise DecryptionError, "authentication failed - wrong key or corrupted data"
end
```

This means that tampering with an encrypted key slot, data ciphertext, nonce, or tag causes decryption failure rather than silent plaintext corruption. The current code uses empty associated data, so public header fields are not separately authenticated as AAD; tampering with them can still cause parsing failure, slot selection failure, or authentication failure.

14 Nonce and randomness analysis

GCM requires nonce uniqueness under a fixed key. `ssh-tresor-ruby` uses 96-bit random nonces:

Listing 8: Random sizes from lib/ssh_tresor/crypto.rb.

```ruby
CHALLENGE_SIZE = 32
MASTER_KEY_SIZE = 32
NONCE_SIZE = 12
AUTH_TAG_SIZE = 16

def random_challenge
  SecureRandom.random_bytes(CHALLENGE_SIZE)
end

def random_master_key
  SecureRandom.random_bytes(MASTER_KEY_SIZE)
end

def random_nonce
  SecureRandom.random_bytes(NONCE_SIZE)
end
```

The construction uses a fresh master key for each newly encrypted file and a fresh slot challenge for each key slot. Therefore nonce collision risk under the same AES-GCM key is already structurally low:

- the data key K_M is fresh per new encryption;
- each slot key K_i is derived from a fresh challenge signature;
- adding a key creates a new slot key and does not re-encrypt file data.

For a uniformly random 96-bit nonce space, the birthday collision probability after q encryptions under one key is approximately $q(q-1)/2^{97}$. In normal `ssh-tresor-ruby` use, q is effectively one per key for the data ciphertext and one per slot key for the wrapped master key.

15 Relation to encryption at rest

"Encryption at rest" usually means that data stored on persistent media is encrypted when not actively being processed. `ssh-tresor-ruby` satisfies that storage property for the output file: the storage medium receives only the `SSHTRESR` blob, not the plaintext. However, the phrase by itself does not capture the central security mechanism. A clearer description is:

> `ssh-tresor-ruby` provides SSH-agent-mediated encryption at rest: files are stored encrypted, and decryption requires a live SSH agent with a matching private key operation.

For the user story, an even more operational phrase is:

> `ssh-tresor-ruby` provides SSH-agent-gated decryption: the stored file can be decrypted only while a matching private-key operation is reachable through `SSH_AUTH_SOCK`, whether the agent is local or forwarded.

Both phrases are technically accurate. The first connects to the common storage-security category; the second explains the operational gate that makes the file unreadable when the matching agent capability is no longer reachable.

16 Evidence from the repository

The test suite exercises the core properties:

- AES-256-GCM encrypt/decrypt succeeds and wrong-key decryption fails;
- binary and armored `SSHTRESR` formats round trip;
- the public `Vault` API encrypts, decrypts, and lists slots;
- CLI integration starts an SSH agent, creates Ed25519 keys, encrypts, decrypts, adds a key slot, removes a key slot, and confirms the remaining slot decrypts.

The verification run performed for this paper completed successfully:

```
bundle exec rspec
10 examples, 0 failures
```

Additional key-type probing performed for this paper found successful round trips for Ed25519 and RSA, and a failed round trip for ECDSA. That result is not a failure of AES-GCM or HKDF; it is a consequence of deriving persistent key material from signature bytes when the signature scheme is randomized.

17 Recommendations and Future Improvements

The construction is sound for deterministic agent signatures under the stated threat model. The remaining recommendations focus on compatibility-conscious hardening and future format evolution:

1. Reject unsupported randomized key types during encryption, preferably before writing a file. A conservative initial allowlist would include `ssh-ed25519` and `ssh-rsa` with RSA/SHA2 signatures.
2. Deferred for compatibility: domain-separating the signed challenge would make the signature purpose explicit, but it would change the bytes sent to the SSH agent and therefore break compatibility with the existing Rust `ssh-tresor` format. This should only be reconsidered as part of a coordinated format-version change across implementations.
3. Consider authenticating selected public header fields as AES-GCM associated data in a future format version. The current format remains safe for plaintext confidentiality, but AAD would tighten whole-blob integrity semantics.

18 Conclusion

`ssh-tresor-ruby` implements a compact and defensible construction for SSH-agent-gated file encryption. It encrypts file contents with a fresh random master key, protects that master key with AES-GCM key slots derived from live SSH-agent signatures, and stores only public challenges, public

fingerprints, nonces, and ciphertexts. Under standard assumptions for SecureRandom, HKDF-SHA256, AES-256-GCM, and deterministic SSH agent signatures, the mathematical argument is straightforward: after the matching agent signing capability disappears, an attacker with only the stored blob cannot recreate the slot key, cannot unwrap the master key, and cannot decrypt the file.

The accurate claim is therefore:

> `ssh-tresor-ruby` provides SSH-agent-mediated encryption at rest for deterministic-signature SSH keys. Files remain unreadable without a matching SSH-agent signing capability, whether the intended workflow uses a local agent loaded by `ssh-add` or a forwarded agent exposed through an SSH session. Decryption becomes possible again only when that matching agent capability is made available. This claim excludes compromise of an active encryption or decryption session, where plaintext, signature bytes, or derived keys could be recorded.

Acknowledgments

The author thanks Harald Hoyer for publishing the original Rust implementation of `ssh-tresor`, available at https://github.com/haraldh/ssh-tresor. That project established the practical SSH-agent-based encryption pattern that inspired this Ruby implementation. The author also acknowledges OpenAI Codex for assistance in drafting, editing, and typesetting this paper.

References

[1] Morris J. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D, 2007. https://www.nist.gov/publications/recommendation-block-cipher-modes-operation-galoiscounter-mode-gcm-and-gmac

[2] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869, May 2010. https://www.rfc-editor.org/rfc/rfc5869

[3] Damien Miller. *SSH Agent Protocol*. IETF Internet-Draft `draft-ietf-sshm-ssh-agent-16`, February 2026. https://datatracker.ietf.org/doc/draft-ietf-sshm-ssh-agent/16/

[4] Denis Bider. *Use of RSA Keys with SHA-256 and SHA-512 in the Secure Shell (SSH) Protocol*. RFC 8332, March 2018. https://www.rfc-editor.org/rfc/rfc8332

[5] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032, January 2017. https://www.rfc-editor.org/rfc/rfc8032

[6] Ruby Documentation. *OpenSSL::KDF*. https://docs.ruby-lang.org/en/4.0/OpenSSL/KDF.html

[7] Ruby Documentation. *OpenSSL::Cipher*. https://docs.ruby-lang.org/en/4.0/OpenSSL/Cipher.html