Audit Accountable Infrastructure:
Formal Taxonomy, Multi-Plane Cryptographic Logs, and NIST SP 800-53 Attestation

Synrc Research Center
`https://synrc.com`

August 5, 2026

Abstract

This document presents the formal specification, architecture, and theoretical foundation of the `synrc/au` library for high-assurance operating environments based on the ERP.1 three-plane architecture. Built upon standard Erlang/OTP primitives and standard NIST cryptography (ECDSA P-384, SHA-384, HMAC-SHA-512), `synrc/au` enforces strict privilege segregation, fail-secure append-only event logging, deterministic CRDT Merkle-log merging across multi-DC deployments, cold-storage log archiving, and complete compliance with NIST SP 800-53 Rev. 5 Audit and Accountability (AU) controls.

Contents

1 Introduction & Motivation

High-integrity critical systems (defense appliances, state registries, electronic healthcare, infrastructure control systems) demand rigorous mathematical assurance, non-repudiation, and architectural privilege isolation. Traditional POSIX models featuring monolithic `root` processes violate Zero Trust Architecture principles by exposing broad ambient authority.

The ERP.1 architecture reduces the Trusted Computing Base (TCB) to a minimal cryptographic core running on Erlang/OTP on UKI Alpine Linux. The `synrc/au` library isolates audit generation, aggregation, and signature capabilities across three execution planes, guaranteeing that audit logs remain immutable, cryptographically verifiable, and fail-secure under all operating conditions.

2 Audit and Accountability

2.1 ERP.1 Three-Plane Isolation Model

Following the fundamental OS.1 taxonomy defined in `synrc/os/os.txt`, the execution environment is partitioned into three distinct planes:

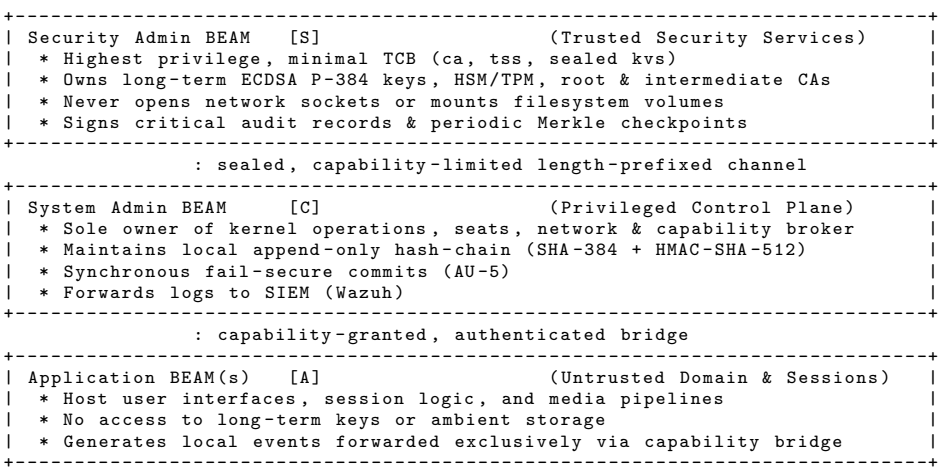

Figure 1: ERP.1 Three-Plane Architecture Isolation Diagram.

2.1.1 Plane Descriptions

1. **Security Admin BEAM [S]**: Houses `synrc/ca`, `synrc/tss`, and sealed `synrc/kvs`. Long-term private keys never leave this plane. It provides RFC 3161 timestamps and ECDSA signatures for critical events.

2. **System Admin BEAM [C]**: Runs `audit_core`. Handles capability broker passing via `SCM_RIGHTS`, maintains the append-only record chain, applies HMAC-SHA-512 integrity tags, and publishes periodic checkpoints.

3. **Application BEAM [A]**: Runs `audit_client`. Untrusted workloads operating under strict Landlock, seccomp-bpf, and cgroups v2 boundaries.

2.2 Formal Distributed Audit Model

2.2.1 Audit Invariants

Audit processing in ERP.1 satisfies four fundamental invariants:

1. **Irreversibility**: Audit entries form an append-only cryptographic hash chain linked via SHA-384 hashes. Past entries cannot be modified or reordered.

2. **Non-Repudiation**: Critical records and batch checkpoints are signed by Security Admin BEAM using ECDSA P-384 keys and sealed with RFC 3161 timestamps.

3. **Isolation**: Application BEAMs cannot read or mutate audit structures of higher planes.

4. **Completeness**: No privileged action (certificate issuance, key unsealing, ABAC policy change, device grant) can execute without a synchronous audit record commit (fail-secure, AU-5).

2.2.2 Data Models (`#audit_record{}` and `#audit_checkpoint{}`)

Every audit event follows the mandatory schema specified by NIST AU-3:

```
-type audit_event() ::
    auth_login | auth_logout | auth_failure | cert_issue | cert_revoke |
    key_gen | key_rotate | key_unseal | policy_change | privilege_escalation |
    sys_boot | sys_shutdown | process_spawn | process_exit | fs_mount |
    device_grant | net_bind | config_update | user_action | data_access |
    data_mutation | session_start | session_stop | api_call | atom().

-type audit_action() ::
    create | read | update | delete | execute | authenticate | authorize |
    grant | revoke | sign | verify | encrypt | decrypt | unseal | rotate |
    mount | unmount | start | stop | atom().

-record(audit_record, {
    id         :: binary(),              % Monotonic ID (64-bit TS + Rand)
    ts         :: integer(),             % Milliseconds UTC / HLC
    plane      :: security | system | application,
    subject    :: binary(),              % Subject identity
    event      :: audit_event(),         % Formalized event atom
    resource   :: binary(),              % Resource identifier
    action     :: audit_action(),        % Formalized action atom
    outcome    :: success | failure,     % Result
    meta = #{}:: map(),                  % Metadata context
    prev_hash  :: binary(),              % SHA-384 of previous link
    hmac       :: binary(),              % HMAC-SHA-512 (System plane)
    sig        :: binary() | undefined,  % ECDSA P-384 (Security plane)
    tsp        :: binary() | undefined   % RFC 3161 timestamp
}).
```

Periodic Merkle checkpoints and log segment seals utilize the `#audit_checkpoint{}` record schema:

```
-record(audit_checkpoint, {
    node_id      :: binary(),          % Originating node identifier
    from_id      :: binary(),          % First record ID in checkpoint
    to_id        :: binary(),          % Last record ID in checkpoint
    head_hash    :: binary(),          % Top hash of local hash chain
    record_count :: non_neg_integer(), % Total records in segment
    merkle_root  :: binary(),          % SHA-384 Merkle root of records
    sig          :: binary() | undefined, % Security Admin ECDSA P-384 sig
    tsp          :: binary() | undefined  % RFC 3161 timestamp token
}).
```

2.3 Formal Event and Action Taxonomy

To prevent field ambiguity, support deterministic SIEM log processing, and satisfy NIST AU-2 (Event Selection) and AU-3 (Content of Audit Records), `synrc/au` formalizes the allowed atom sets for the `event` and `action` fields of `#audit_record{}`.

2.3.1 Formalized Audit Events (`audit_event()`)

Audit events represent functional domain categories partitioned by execution plane:

2.3.2 Formalized Audit Actions (`audit_action()`)

Audit actions specify the discrete operational action executed on a targeted resource:

2.3.3 Multi-Node CRDT Log Merge

In multi-node / multi-DC environments, each node maintains a local sequential hash chain. Global consensus is achieved using an **Add-only Merkle-CRDT** (G-Set of signed checkpoints):

$$\text{GlobalLog} = \text{G-Set}\left(\{\text{NodeId}, \text{HeadHash}, \text{SignedCheckpoint}, \text{MerkleRoot}\}\right) \tag{1}$$

Merging two node logs L_A and L_B is deterministic and conflict-free:

$$\text{merge}(L_A, L_B) = L_A \cup L_B \tag{2}$$

Linearization sorts records by the tuple $(TS, \text{NodeId}, \text{RecordId})$, producing a deterministic global sequence without altering original node hash chains.

Plane	Event Atom	Semantics & Triggering Context
Security	`auth_login`	User or service principal authentication attempt
	`auth_logout`	Session termination or revocation
	`auth_failure`	Authentication rejection or credential failure
	`cert_issue`	X.509 / TLS Certificate issuance by Security CA
	`cert_revoke`	Certificate revocation or CRL update
	`key_gen`	Cryptographic keypair generation (ECDSA / AES)
	`key_rotate`	Key lifecycle rotation or re-keying operation
	`key_unseal`	Master key unsealing or vault hardware access
	`policy_change`	ABAC / RBAC security policy modification
	`privilege_escalation`	Elevating capability or ambient authority
System	`sys_boot`	Operating system or kernel initialization
	`sys_shutdown`	System shutdown, reboot, or emergency halt
	`process_spawn`	BEAM process or container execution spawn
	`process_exit`	Process termination or abnormal crash
	`fs_mount`	Storage volume / Landlock directory mount
	`device_grant`	Hardware seat or device capability assignment
	`net_bind`	Network socket bind or outbound connection
	`config_update`	Kernel parameter or system config update
Application	`user_action`	User interface or workflow activity
	`data_access`	Database read or query operation
	`data_mutation`	Record insertion, update, or deletion
	`session_start`	Application domain session initialization
	`session_stop`	Application domain session destruction
	`api_call`	Service API invocation or RPC message

Table 1: Formal Taxonomy of Allowed Audit Event Atoms.

2.4 NIST Cryptographic Profile

All cryptographic primitives are standard FIPS/NIST compliant:

2.5 NIST SP 800-53 Rev. 5 AU Control Attestation

2.6 Comprehensive Audit Usage & Code Examples

2.6.1 1. Starting the Audit Service

To initialize the System Admin BEAM aggregator with ECDSA P-384 Security keys:

```
{PubKey, PrivKey} = audit_crypto:generate_keypair(),
MacKey = audit_crypto:generate_mac_key(),

{ok, Pid} = audit_core:start_link(#{
     node_id     => <<"DC1-Node01">>,
     boot_time   => erlang:system_time(millisecond),
     mac_key     => MacKey,
     sec_keypair => {PubKey, PrivKey}}).
```

Action Atom	Operational Semantics
`create`	Instantiation of a new record, entity, or resource
`read`	Accessing, inspecting, or querying resource content
`update`	Modifying or mutating existing resource state
`delete`	Removing, purging, or soft-deleting a resource
`execute`	Launching or invoking an executable, task, or process
`authenticate`	Verifying identity credentials
`authorize`	Evaluating access control capabilities or permissions
`grant`	Bestowing privileges, roles, or capability tokens
`revoke`	Rescinding privileges, roles, or capability tokens
`sign`	Calculating cryptographic digital signatures
`verify`	Validating signatures, Merkle proofs, or hashes
`encrypt`	Encrypting data at rest or in transit
`decrypt`	Decrypting encrypted payloads
`unseal`	Accessing sealed cryptographic keys or vaults
`rotate`	Replacing cryptographic keys or certificates
`mount`	Mounting filesystems or capability trees
`unmount`	Unmounting filesystems or capability trees
`start`	Launching a service, node, or application process
`stop`	Terminating a service, node, or application process

Table 2: Formal Taxonomy of Allowed Audit Action Atoms.

Purpose	Algorithm	Specification	NIST
Chain	Merkle Hashing	SHA-384 (or SHA-512)	NIST FIPS 180-4
Message Integrity	HMAC-SHA-512	NIST FIPS 198-1	
Digital Signature	ECDSA P-384	NIST FIPS 186-4 (secp384r1)	
Timestamp	RFC 3161	SHA-384 / SHA-512 TimeStampToken	
Key Generation	ECDH P-384	NIST SP 800-56A	

Table 3: NIST Cryptographic Profile for `synrc/audit`.

2.6.2 2. Logging Standard Application Events (AU-12)

From any Application plane process:

```
ok = audit_client:log(
     application,
     <<"user_session_42">>,
     user_login,
     <<"/api/v1/auth">>,
     authenticate,
     success,
     #{<<"ip">> => <<"192.168.1.100">>, <<"seat">> => <<"seat0">>}).
```

Control	Description	Plane	Implementation in `synrc/audit`
AU-1	Audit Policy and Procedures	[C]	Defined in CMDB metadata.
AU-2	Event Selection	[C],[A]	Configurable event masks per plane.
AU-3	Content of Audit Records	[C]	Mandatory `#audit_record{}` schema.
AU-4	Storage Capacity	[C]	ETS / KVS sizing and segmentation.
AU-5	Response to Processing Failures	[C]	Synchronous fail-secure `gen_server` call.
AU-6	Review, Analysis, Reporting	[C],[S]	Offline verifier (`audit_verify`) & SIEM export.
AU-7	Audit Reduction	[C]	Filtered queries and Merkle proofs.
AU-8	Time Stamps	[C],[S]	UTC ms / HLC + RFC 3161 timestamps.
AU-9	Protection of Audit Info	[C],[S]	Append-only SHA-384 chain + HMAC-SHA-512.
AU-10	Non-Repudiation	[S]	ECDSA P-384 signatures by Security plane.
AU-11	Audit Record Retention	[C]	Archiving tool (`audit_archive`) for cold storage.
AU-12	Audit Generation	[A],[C]	Real-time generation via capability bridge.

Table 4: NIST SP 800-53 Rev. 5 AU Controls Attestation Matrix.

2.6.3 3. Logging Critical Security Events with ECDSA Signatures

For high-assurance operations requiring non-repudiation (AU-10):

```
ok = audit_client:log_critical(
     security,
     <<"security_admin">>,
     cert_issue,
     <<"CA-Root-2026">>,
     sign,
     success,
     #{<<"subject_dn">> => <<"CN=SystemAdmin">>}).
```

2.6.4 4. Generating and Verifying Merkle Checkpoints

To generate a signed checkpoint for active records:

```
Checkpoint = audit_core:get_checkpoint(),
Valid      = audit_checkpoint:verify(Checkpoint, PubKey),
io:format("Checkpoint valid: ~p~n", [Valid]).
```

2.6.5 5. Multi-Node CRDT Log Merging, Delta & Sync

Merging, diffing, and linearizing audit streams from independent nodes:

```
SetA          = audit_merge:add_node_log(audit_merge:new_log_set(), CheckpointA, RecordsA),
SetB          = audit_merge:add_node_log(audit_merge:new_log_set(), CheckpointB, RecordsB),
MergedSet     = audit_merge:merge(SetA, SetB),
true          = audit_merge:verify_merged(MergedSet, PubKey),
LinearRecords = audit_merge:linearize(MergedSet, #{plane => security, limit => 50}),

%% Calculate synchronization delta and missing entries diff
DiffSet       = audit_merge:diff(SetA, SetB),
DeltaSet      = audit_merge:delta(MergedSet, #{<<"DC1-Node01">> => 10}),
Stats         = audit_merge:stats(MergedSet).
```

2.6.6 6. Pure Offline Verification (audit_verify)

Perform independent offline verification of a record chain and Merkle inclusion:

```
GenesisHash = audit_chain:genesis_hash(<<"DC1-Node01">>, BootTime),
case audit_verify:verify_chain(Records, GenesisHash, MacKey, PubKey) of
    ok -> io:format("Chain integrity verified.~n");
    {error, {tampered_record, Index, BadRecord}} ->
        io:format("Tampering detected at record ~p!~n", [Index])
end,
IsIncluded = audit_verify:verify_inclusion(Record, Checkpoint).
```

2.6.7 7. Cold Storage Log Archiving & Sealing (audit_archive)

Splitting, sealing, and serializing log segments for long-term cold storage (AU-11):

```
{Segment, Remaining} = audit_archive:cut_segment(Records, CutoffTimestamp),
SealedCP             = audit_archive:seal_segment(Segment, <<"DC1-Node01">>, PrivKey),
ArchiveBin           = audit_archive:serialize_archive(Segment, SealedCP),

{ok, RestoredCP, RestoredRecords} = audit_archive:deserialize_archive(ArchiveBin).
```

2.6.8 8. SIEM & OSCAL Export (audit_export)

Exporting log entries for Wazuh / Syslog or compliance evidence:

```
CEFString = audit_export:to_siem(Record),
JSONData  = audit_export:to_json(Records),
OscalMap  = audit_export:to_oscal(Checkpoint, Records).
```

2.7 Erlang Module API Reference

This section provides the complete reference specification for all exported functions across the 12 modules in `synrc/au`.

2.7.1 audit_client

High-level Application Plane client interface for emitting audit events to `audit_core`.

- `log(Subject, Event, Resource, Action, Outcome, Meta) -> ok | {error, term()}`
  Submits a standard audit event defaulting to the `application` plane.
- `log(Plane, Subject, Event, Resource, Action, Outcome, Meta) -> ok | {error, term()}`
  Submits a standard audit event targeting the specified execution plane (`security`, `system`, or `application`).
- `log_critical(Plane, Subject, Event, Resource, Action, Outcome, Meta) -> ok | {error, term()}`
  Submits a high-assurance audit event requiring Security Admin ECDSA P-384 signature and RFC 3161 timestamping.

2.7.2 audit_core

OTP gen_server managing local ETS storage, append-only hash chain linking, and checkpoint generation.

- start_link() -> {ok, pid()} | {error, term()}
  Starts the audit core server with default node options.
- start_link(Opts::map()) -> {ok, pid()} | {error, term()}
  Starts the audit core server with custom options (node_id, boot_time, mac_key, sec_keypair).
- stop() -> ok
  Gracefully stops the audit_core server.
- log_event(Record::#audit_record{}) -> ok | {error, term()}
  Synchronously appends a standard audit record to the local hash chain and ETS table.
- log_critical_event(Record::#audit_record{}) -> ok | {error, term()}
  Synchronously appends a signed critical audit record to the local hash chain and ETS table.
- get_head_hash() -> binary()
  Returns the current 384-bit SHA-384 head hash of the node's local audit chain.
- get_records() -> [#audit_record{}]
  Retrieves all audit records stored in the local ETS table in chronological sequence.
- get_checkpoint() -> #audit_checkpoint{}
  Generates a signed #audit_checkpoint{} spanning all currently accumulated records.
- set_security_keys(PubKey::binary(), PrivKey::binary()) -> ok
  Dynamically configures or updates the Security Admin ECDSA P-384 keypair.

2.7.3 audit_record

Constructs, validates, encodes, and formats individual NIST AU-3 audit record instances.

- new(Plane, Subject, Event, Resource, Action, Outcome, Meta) -> #audit_record{}
  Instantiates and validates a new #audit_record{} with automatic monotonic ID generation.
- allowed_events() -> [audit_event()]
  Returns the list of standard formalized audit event atoms.

- `allowed_actions() -> [audit_action()]`
  Returns the list of standard formalized audit action atoms.
- `is_allowed_event(Event::atom()) -> boolean()`
  Checks whether a given atom belongs to the formalized `audit_event()` set.
- `is_allowed_action(Action::atom()) -> boolean()`
  Checks whether a given atom belongs to the formalized `audit_action()` set.
- `validate(Record::term()) -> ok | {error, term()}`
  Enforces NIST AU-3 schema validation on record fields.
- `canonical_payload(#audit_record{}) -> binary()`
  Computes a deterministic binary payload encoding (excluding transient hashes/signatures) used for hashing and MAC generation.
- `encode(#audit_record{}) -> binary()`
  Encodes an audit record into a deterministic binary term format.
- `decode(Bin::binary()) -> {ok, #audit_record{}} | {error, term()}`
  Decodes a binary term into a validated `#audit_record{}`.
- `new_id() -> binary()`
  Generates a monotonic 16-byte record identifier (64-bit UTC ms timestamp + 8-byte random entropy).
- `now_ms() -> integer()`
  Returns current UTC timestamp in milliseconds.

2.7.4 `audit_chain`

Implements append-only hash chain linking, HMAC tag calculation, and link verification.

- `genesis_hash(NodeId::binary(), BootTime::integer()) -> binary()`
  Computes the root genesis hash for a node audit chain (`ERP.1-AUDIT-ROOT:NodeId:BootTime`).
- `append_record(Record, PrevHash, MacKey) -> #audit_record{}`
  Calculates `prev_hash` via SHA-384 and `hmac` via HMAC-SHA-512 over `(PrevHash || Payload)`.
- `append_critical_record(Record, PrevHash, MacKey, SecPrivKey) -> #audit_record{}`
  Appends standard chain hashes and attaches an ECDSA P-384 signature and RFC 3161 timestamp token.
- `verify_record(Record, PrevPrevHash, MacKey, SecPubKey) -> boolean()`
  Validates single record hash continuity, HMAC tag, ECDSA signature, and timestamp token.

2.7.5 audit_checkpoint

Constructs and verifies Merkle checkpoints over log segments.

- `create(NodeId, Records, HeadHash, SecPrivKey) -> #audit_checkpoint{}`
  Builds a `#audit_checkpoint{}`, computes the Merkle tree root over record canonical payloads, and applies an optional ECDSA signature.
- `verify(Checkpoint::#audit_checkpoint{}, SecPubKey) -> boolean()`
  Validates ECDSA P-384 signature and RFC 3161 timestamp token on a checkpoint.
- `compute_merkle_root([#audit_record{}]) -> binary()`
  Computes the binary Merkle root hash (SHA-384) over a list of audit records.

2.7.6 audit_merge

Add-only Merkle-CRDT G-Set engine for multi-node log synchronization, diffing, delta tracking, and linearization.

- `new_log_set() -> log_set()`
  Returns an empty G-Set map.
- `add_node_log(LogSet, Checkpoint, Records) -> log_set()`
  Adds or updates a node log entry in the G-Set, merging records if the node exists.
- `merge(LogSetA, LogSetB) -> log_set()`
  Computes the deterministic CRDT G-Set union of two log sets.
- `linearize(LogSet) -> [#audit_record{}]`
  Produces a total order sequence of all records across nodes sorted by `(TS, NodeId, RecordId)`.
- `linearize(LogSet, Opts::map()) -> [#audit_record{}]`
  Linearizes records with filtering options (`from_ts`, `to_ts`, `plane`, `limit`).
- `global_merkle_root(LogSet) -> binary()`
  Computes a cluster-wide global Merkle root over sorted node head hashes.
- `verify_merged(LogSet, SecPubKey) -> boolean()`
  Verifies all checkpoints and record hash chains contained within a merged log set.
- `diff(LogSetA, LogSetB) -> log_set()`
  Calculates missing records present in `LogSetB` but absent in `LogSetA`.
- `delta(LogSet, KnownCounts::map()) -> log_set()`
  Generates a delta log set containing only records newer than specified per-node counts.

- `apply_delta(LocalSet, DeltaSet) -> log_set()`
  Applies a delta log set into a local log set via CRDT merge.
- `stats(LogSet) -> map()`
  Computes aggregate cluster metrics (`node_count`, `total_records`, `min_timestamp`, `max_timestamp`, `global_merkle_root`).

2.7.7 `audit_verify`

Pure offline cryptographic chain and checkpoint verifier.

- `verify_chain(Records, GenesisHash, MacKey, SecPubKey) -> ok | {error, {tampered_record, integer(), term()}}`
  Iteratively verifies chain continuity, HMAC tags, and ECDSA signatures from genesis.
- `verify_checkpoint(Checkpoint, SecPubKey) -> boolean()`
  Verifies a checkpoint's signature and timestamp offline.
- `verify_inclusion(Record, Checkpoint) -> boolean()`
  Verifies whether a record payload hash is included in a checkpoint's Merkle root.

2.7.8 `audit_archive`

Cold storage segmentation, sealing, and serialization primitives.

- `cut_segment(Records, CutoffTS) -> {Segment, Remaining}`
  Partitions audit records at a timestamp boundary.
- `seal_segment(Segment, NodeId, SecPrivKey) -> #audit_checkpoint{}`
  Seals a record segment into a signed `#audit_checkpoint{}`.
- `serialize_archive(Segment, Checkpoint) -> binary()`
  Compresses and serializes a segment and its checkpoint into binary format.
- `deserialize_archive(Bin::binary()) -> {ok, #audit_checkpoint{}, [#audit_record{}]} | {error, term()}`
  Deserializes and validates an archive binary blob.

2.7.9 `audit_export`

Formats audit evidence into standard SIEM, JSON, and OSCAL representations.

- `to_json(RecordOrRecords) -> binary()`
  Formats record(s) into structured JSON.
- `to_siem(Record) -> binary()`
  Formats an audit record into Wazuh / Syslog Common Event Format (CEF).

- `to_oscal(Checkpoint, Records) -> map()`
  Formats checkpoint and records into a NIST OSCAL Assessment Results structure.

2.7.10 `audit_crypto`

FIPS/NIST cryptographic primitives (ECDSA P-384, SHA-384, HMAC-SHA-512, RFC 3161 timestamps).

- `generate_keypair() -> {PubKey, PrivKey}`
  Generates a fresh ECDSA P-384 (`secp384r1`) keypair.
- `generate_mac_key() -> binary()`
  Generates a 256-bit symmetric key for HMAC calculation.
- `hash(Data::binary()) -> binary()`
  Computes SHA-384 hash of binary data.
- `hash(Algo, Data::binary()) -> binary()`
  Computes specified digest (`sha384` or `sha512`).
- `hmac(Key, Data) -> binary()`
  Computes HMAC-SHA-512 over data using symmetric key.
- `sign(PrivKey, Data) -> binary()`
  Signs data digest using ECDSA P-384 with SHA-384 digest.
- `verify(PubKey, Data, Signature) -> boolean()`
  Verifies ECDSA P-384 signature over data.
- `timestamp_token(Digest, TS) -> binary()`
  Generates a simulated RFC 3161 timestamp token for a given digest and timestamp.
- `verify_timestamp(Token, Digest, TS) -> boolean()`
  Verifies an RFC 3161 timestamp token.

2.7.11 `audit_app` & `audit_sup`

OTP application lifecycle and supervisor specification.

- `audit_app:start(StartType, StartArgs) -> {ok, pid()} | {error, term()}`
  Starts the `au` OTP application and supervisor tree.
- `audit_app:stop(State) -> ok`
  Stops the application.
- `audit_sup:start_link() -> {ok, pid()} | {error, term()}`
  Starts the `audit_sup` supervisor (`one_for_one` strategy supervising `audit_core`).
- `audit_sup:init(Args) -> {ok, {SupFlags, [ChildSpec]}}`
  OTP supervisor callback specifying child worker parameters.

3 Related Work, Historical Lineage, and Ecosystem Comparison

3.1 Historical Lineage

Cryptographic audit logging and tamper-evident event recording rest upon foundational work across distributed security literature:

1. **Schneier & Kelsey Forward-Secure Logging (1998, 1999)** [1, 2]: Schneier and Kelsey established the core model for secure audit logs on untrusted machines using evolving hash chains ($H_i = \text{SHA}(H_{i-1} \parallel W_i)$) and forward-secure symmetric MAC keys deleted after writing. `synrc/au` builds upon this linear chain model, extending it to a multi-plane Erlang execution architecture where key generation and signature authorization reside strictly inside the isolated Security Admin BEAM [S].

2. **Merkle Trees & Audit Proofs (Merkle 1987, Crosby & Wallach 2009)** [3, 4]: Merkle's 1987 binary tree structures and Crosby & Wallach's efficient data structures for tamper-evident logging established logarithmic $\mathcal{O}(\log N)$ inclusion verification and consistency proofs. `synrc/au` utilizes SHA-384 Merkle trees inside `audit_checkpoint` and for multi-DC global cluster root generation (`global_merkle_root/1`).

3. **Certificate Transparency & RFC 6962 (Laurie et al., 2013)** [5]: Established publicly auditable append-only Merkle tree logs for public TLS certificate issuance. `synrc/au` incorporates append-only Merkle tree semantics while providing native AEAD AES-256-GCM encryption at rest (SC-28) for enterprise confidentiality.

4. **Google Transparency Dev Initiative** [6]: Modern zero-dependency Go ecosystem (`https://transparency.dev`) provides modular, tile-based Merkle tree storage (`tessera`) and witness verification networks for application transparency. `synrc/au` shares this zero-dependency philosophy, implementing native Erlang/OTP cryptographic log storage and verification.

5. **NIST SP 800-53 Rev. 5 Standards (2020)** [8]: Outlines mandatory security controls for US Federal and FedRAMP High baseline environments.

3.2 Modern Zero-Dependency Competitor Analysis Across Languages

We analyze `synrc/au` against modern, standalone, minimal-dependency audit and transparency logging implementations in Go, C99, and Rust:

System	Language / TCB	Dependency Model	Isolation Architecture	Cryptographic Security	Replication / Merging
`synrc/au`	Erlang/OTP	OTP `crypto`	Process boundary ([S],[C],[A])	SHA-384 Chain + ECDSA P-384 + AES-256-GCM AEAD	**Add-only Merkle-CRDT G-Set**
`kauditd`	Linux Kernel	POSIX Kernel	Ring 0 Kernel vs Ring 3 Userspace	Plaintext sequential logs (Requires external syslog)	Manual syslog forwarder
`transparency.dev`	Go	Tessera	Process boundary	Tile Merkle Tree + Witness Signatures	Static HTTP tiles / Checkpoint witnesses
`sigstore.dev`	Go	Trillian	Container / Microservice	Merkle Transparency Tree + Rekor Sigstore	DB-backed consensus
`CT-Go`	Go	Go stdlib	Process boundary	RFC 6962 Merkle Tree + Ed25519	Centralized log server
`tpm2-tss-audit`	C99, `libcrypto`, TPM2	Hardware TPM2 Enclave	TPM2 HMAC / PCR event log signatures	Single node hardware log	
`sunlight`	Rust `std`, `ring`	Memory-safe process	RFC 6962 Merkle Tree + Ed25519	Static tile log storage	

Table 5: Cross-Language Zero-Dependency Audit System Comparison.

3.3 Key Architectural Differentiators of `synrc/au`

1. **Three-Plane Process Isolation**: C99 system loggers (`auditd`) rely on monolithic POSIX processes, exposing signing keys to local root compromise. `synrc/au` isolates key generation and signing inside the Security Admin BEAM [S], preventing Application [A] or Control [C] planes from accessing long-term private keys.

2. **Confidentiality at Rest (SC-28 / AU-9(3))**: Standard transparency loggers (Rekor, CT-Go, `transparency.dev`) are public plaintext ledgers. `synrc/au` natively encrypts payload metadata using AES-256-GCM AEAD authenticated encryption.

3. **Asynchronous Conflict-Free Merging**: Uses Add-only Merkle-CRDT G-Set semantics to merge logs across multi-DC deployments deterministically without centralized database locks.

4 Conclusion

I dedicate this work to my daugther Michelle.

The `synrc/audit` library provides a minimal, self-contained, and cryptographically verifiable audit infrastructure for ERP.1 systems. By combining standard NIST algorithms, Erlang/OTP supervision, three-plane capability separation, CRDT Merkle log merging, cold storage log archiving, and comprehensive export capabilities, it fulfills the full spectrum of NIST SP 800-53 Rev. 5 AU requirements without external dependencies.

References

[1] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 2, pp. 159–176, 1998.

[2] B. Schneier and J. Kelsey, "Secure audit logs supporting tamper detection and forward-secrecy," in *Proceedings of the 10th USENIX Security Symposium*, 1999, pp. 139–165.

[3] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the Theory and Application of Cryptographic Techniques (CRYPTO '87)*, Springer, 1987, pp. 369–378.

[4] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *Proceedings of the 18th USENIX Security Symposium*, 2009, pp. 317–334.

[5] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," *IETF RFC 6962*, Jun. 2013.

[6] Google Transparency Engineering, "Verifiable Data Structures and Transparency Logs," *Google Transparency Dev Initiative*, 2024. [Online]. Available: `https://transparency.dev`

[7] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato, "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)," *IETF RFC 3161*, Aug. 2001.

[8] National Institute of Standards and Technology (NIST), "Security and Privacy Controls for Information Systems and Organizations," *NIST Special Publication 800-53*, Revision 5, Sep. 2020.

[9] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)," *FIPS PUB 186-4*, Jul. 2013.

[10] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)," *FIPS PUB 180-4*, Aug. 2015.

[11] National Institute of Standards and Technology (NIST), "The Keyed-Hash Message Authentication Code (HMAC)," *FIPS PUB 198-1*, Jul. 2008.

[12] D. E. Bell and L. J. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," *MITRE Corp Technical Report MTR-2997*, Jul. 1975.

[13] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free Replicated Data Types," in *Symposium on Self-Stabilizing Systems (SSS 2011)*, Springer, 2011, pp. 386–400.

[14] Linux Audit Subsystem Project, "Linux Audit Framework (`kauditd` & `auditd`)," *Linux Kernel Documentation*, 2024. [Online]. Available: `https://docs.kernel.org/trace/events.html`

[15] Sigstore Community, "Rekor: Signature Transparency Log for Software Supply Chain Security," *Linux Foundation Open Source Project*, 2022. [Online]. Available: `https://www.sigstore.dev`

[16] Google Transparency Team, "Trillian: Verifiable Data Structures for Verifiable Logs," *GitHub Repository*, 2023. [Online]. Available: `https://github.com/google/trillian`