Ritma / UTL Security Fabric
Pravyom Project – Part I

Pravyom Project

December 2, 2025

What It Is

Ritma's Universal Truth Layer (UTL) is a security fabric that sits between applications and the operating system:

- A live security firewall on every transition (RPC / HTTP / internal flows).
- A policy engine (TruthScript) deciding allow, deny, or allow_with_actions.
- Forensic DigFiles: sealed, Merkle–rooted JSON logs of what actually happened.
- A forensics store and HTTP API for indexing and querying DigFiles.
- A decision event stream feeding a host agent that can enforce at the OS layer.

This slice is the first production piece of the Pravyom project: a practical, policy–driven security OS core that can both enforce and prove what happened.

30 Practical Use Cases

1. Tenant–level firewalling between did:ritma:tenant identities.
2. Service–to–service allowlists for did:ritma:svc endpoints.
3. Zero–trust zones: block public to internal zone flows by default.
4. High–value API monitoring: always seal DigFiles for sensitive endpoints.
5. Abuse / fraud detection with denies plus full forensic trails.
6. Compliance logging for PCI / GDPR with Merkle–rooted dig archives.
7. Incident forensics: reconstruct all transitions around suspicious roots.
8. Policy regression testing with synthetic events.
9. Shadow–mode policy evaluation (decision events without enforcement).
10. Per–tenant policy experiments and A/B testing.
11. API abuse throttling (future) via FlowDecision::Throttle.

12. Network quarantine (future) via IsolationScope and cgroups.
13. Credential misuse detection across actor DIDs.
14. Lateral movement tracking via src_did to dst_did graph analysis.
15. Production change approval anchored in TruthScript rules.
16. Data exfiltration guardrails that seal digs on large exports.
17. Multi–cloud consistency using a single policy and dig format.
18. Security posture scoring per tenant and per zone.
19. Anomaly detection by running ML over DigFile JSON.
20. ZK proof demos using SNARKs on selected flows.
21. Micro–proofs (Distillium) for compact transition proofs.
22. Red–team replay of recorded traffic against new policies.
23. Runbook validation with verifiable evidence of incident steps.
24. Breach impact analysis across all affected roots and entities.
25. Root of trust demos with Merkle roots for all histories.
26. Developer self–service dig inspection for debugging.
27. Security–as–code pipelines for policies and lawbooks.
28. Multi–tenant blast–radius analysis via tenant_id.
29. Regulator–focused evidence bundles from the forensics API.
30. Security OS research platform for future Pravyom work.

Current Condition and Capabilities

Current Condition

- Runs locally as a Rust workspace, no external services required.
- Live policy enforcement in utld: deny or allow transitions based on TruthScript policy, with demo actions for proofs.
- Dig and forensics path: DigFiles written to a local dig directory and to an S3–style forensics tree, plus a dig index in JSONL and SQLite.
- Forensics HTTP API (utl_forensics) for filtered queries and evidence bundles.
- Decision event stream and a logging host agent stub (security_host).

What It Can Do Right Now

- Enforce allow / deny policies on each transition.
- Seal and persist DigFiles with Merkle roots and optional HMAC signatures.
- Index DigFiles with tenant, root, time range, record count, merkle root, policy name, policy version, policy decision, and storage path.
- Serve DigFiles and evidence bundles over a bearer–secured HTTP API.
- Emit structured DecisionEvent records for host–level enforcement.

Truthfulness, Purpose, and Relation to Pravyom

- Truthfulness: digs are sealed, Merkle–rooted, and optionally signed; index entries are derived directly from these DigFiles.
- Purpose: combine live enforcement with verifiable evidence so we can answer what happened and why policy decided it.
- Pravyom: this is Part I of a broader security OS vision that extends into host enforcement, isolation, and stronger proofs.

Founder Story

Ritma / UTL was born out of repeated frustration with security incidents and compliance reviews where nobody could answer a simple question: "what exactly did this system do, and why?" As engineers and operators we had seen too many ad-hoc logs, missing context, and retrospective guesses. The founders built UTL and the Pravyom project to turn that pain into a concrete system: one that enforces policy in real time, records every important decision as a sealed DigFile, and gives both teams and regulators a single, shared source of truth about system behaviour.

Demo Run With Real Output

Environment Setup

```
cd ~/Documents/connector/ritma

export UTLD_POLICY=creat/policies/security_policy.json
export UTLD_DIG_INDEX=./dig_index.jsonl
export UTLD_DIG_DIR=./dig
export UTLD_FORENSICS_DIR=./forensics
export UTLD_DIG_INDEX_DB=./dig_index.sqlite
export UTLD_DECISION_EVENTS=./decision_events.jsonl

export UTL_FORENSICS_ADDR=127.0.0.1:9101
export UTL_FORENSICS_TOKEN=secret
```

Start the UTL Daemon

```
cargo run -p utld
```

This starts the UTL daemon, listening on the Unix socket configured by `UTLD_SOCKET` (default `/tmp/utld.sock`).

Start the Forensics HTTP API

```
cargo run -p utl_forensics
```

The forensics service listens on `UTL_FORENSICS_ADDR`, for example 127.0.0.1:9101.

List Digs From the CLI (Real Output)

```
file_id=343694725911866759420742646674130501 03 root_id=300 tenant_id=acme
    time_range=1764640664-1764640664 records=4 merkle_root=
    c4957a59949451674121d9475b00d64f7f5d3e166ff47235e54445fe1cdf42ec policy
    =<none> decision=<none> path=./dig/root-300_file
    -343694725911866759420742646674130501 03_1764640664.dig.json
file_id=123323533102211976749380422689014311888 root_id=300 tenant_id=acme
     time_range=1764640664-1764640664 records=1 merkle_root=98
    db1a1e45c0856d298ed8aedd3d05e96c18f97a73932297ab7b361ce0ad78a4 policy=
    security_policy decision=allow_with_actions path=./dig/root-300_file
    -123323533102211976749380422689014311888_1764640664.dig.json
file_id=65298172895416657786460545580706427572 root_id=300 tenant_id=acme
    time_range=1764640664-1764640664 records=1 merkle_root=2
    ce7baecec2e262ec6e0948db1beda4a17c7ac99fd360b2ed6eeaa5538308ee2 policy=
    security_policy decision=allow_with_actions path=./dig/root-300_file
    -65298172895416657786460545580706427572_1764640664.dig.json
file_id=82267068329764365758771882348027784020 root_id=300 tenant_id=acme
    time_range=1764640664-1764640664 records=1 merkle_root=2
    b8547baa14609d6b509fadaf5a82f1bfe6510249b3f880a03d4a28a50cb467c policy=
    security_policy decision=allow_with_actions path=./dig/root-300_file
    -82267068329764365758771882348027784020_1764640664.dig.json
file_id=27107409509634130926566997685651 1155421 root_id=300 tenant_id=acme
     time_range=1764640664-1764640664 records=1 merkle_root=805
    e12c4874210c5c93fd39f390f3be0900ee4d509817cae9bea67e758bbdc83 policy=
    security_policy decision=allow_with_actions path=./dig/root-300_file
    -27107409509634130926566997685651 1155421_1764640664.dig.json
```

```
file_id=10379481144482977843600473118107171642 2 root_id=300 tenant_id=acme
    time_range=1764640664-1764640664 records=1 merkle_root=3
   aef17166eefe64632b078aa05fa6955a7debe2df7cccc837a6f6c0c62061040 policy=
   security_policy decision=allow_with_actions path=./dig/root-300_file
   -10379481144482977843600473118107171642 2_1764640664.dig.json
```

Inspect One Dig by ID

```
# Replace the file_id value with one from the listing above
cargo run -p utl_cli -- \
  dig-inspect-id \
  --file-id 82267068329764365758771882348027784020 \
  --limit 5
```

This resolves the DigFile path from the index and prints a filtered summary of records (tenant, event_kind, severity, policy_decision, and so on).

Query Via the Forensics HTTP API

```
curl -H "Authorization: Bearer secret" \
  "http://127.0.0.1:9101/digs?tenant=acme&root_id=300&policy_decision=
     allow_with_actions&show_path=true" \
  | jq
```

The response is a JSON array of indexed digs matching the filters, including resolved paths when `show_path=true`.

Run the Host Agent Stub

```
export SECURITY_EVENTS_PATH=./decision_events.jsonl
cargo run -p security_host
```

The host agent reads DecisionEvent records and logs firewall and cgroup intents using the security_os traits.

Mathematical and Technical Commentary

Core Model

- IDs: each root and entity is identified by a 128–bit value; these are carried through transitions and digs.
- DIDs: security identities use structured strings such as `did:ritma:tenant:acme` and `did:ritma:svc:acme:pu`
- Events: each transition is turned into an EngineEvent with a kind and a map of typed fields for the policy engine.

Policy Engine

A TruthScript policy maps EngineEvent values to EngineAction results. The main actions used today are:

- Deny with a human–readable reason.
- SealCurrentDig to force sealing and indexing of the current DigFile.
- RequireSnarkProof and RequireDistilliumProof as cryptographic demos.
- FlagForInvestigation and RecordField for richer forensic context.

For each RecordTransition, utld evaluates the policy, injects metadata back into the event parameters, emits a DecisionEvent, and then applies actions.

DigFiles and Merkle Roots

DigFiles are sequences of DigRecords containing timestamps, parameters, and other structured data. A Merkle tree is built over these records; the Merkle root is stored in the DigFile and the dig index. This gives a compact, robust commitment to the full record set.

Indexing and Forensics Store

- JSONL index: dig_index.jsonl accumulates DigIndexEntry lines.
- SQLite mirror: an optional dig_index.sqlite mirrors the index for fast queries.
- Forensics store: DigFiles are mirrored into a forensics directory using an S3–style path layout by tenant and date.

Index entries include time ranges, tenant, root, record count, Merkle root, policy metadata, and the forensic storage path.

Decision Events and Host Agent

DecisionEvent records capture:

- Core identifiers: tenant_id, root_id, entity_id, event_kind.
- Policy context: policy_name, policy_version, policy_decision, policy_rules, policy_actions.

- Optional identity context: src_did, dst_did, actor_did, src_zone, dst_zone.

The security_host binary reads these events, prints human–readable logs, and invokes the security_os FirewallController and CgroupController traits to show how a real enforcement agent would behave.

Command Cheat–Sheet

```
# UTL daemon
cargo run -p utld

# Forensics HTTP API
cargo run -p utl_forensics

# Host agent stub
cargo run -p security_host

# List registered roots
cargo run -p utl_cli -- roots-list

# Register a root
cargo run -p utl_cli -- root-register --root-id 300 --root-hash <hex> \
  --param tenant_id=acme

# Record a transition
cargo run -p utl_cli -- tx-record --entity-id <id> --root-id 300 \
  --signature <hex> --data "payload" --addr-heap-hash <hex> \
  --hook-hash <hex> --logic-ref "ref" --wall "boundary" \
  --param tenant_id=acme

# List digs from the index
cargo run -p utl_cli -- digs-list --show-path

# Inspect a dig by file_id
cargo run -p utl_cli -- dig-inspect-id --file-id <file_id> --limit 5
```

Glossary and Terminology

This section explains key terms used throughout Ritma / UTL, both new concepts and those that are native to the infrastructure.

Core Platform

UTL (Universal Truth Layer): the security fabric that evaluates policies for each transition and records what really happened.

utld: the main UTL daemon. It receives NodeRequest messages (RegisterRoot, RecordTransition, BuildDigFile, and so on), evaluates policies, enforces allow / deny decisions, and seals DigFiles.

utl_forensics: the forensics HTTP service. It reads the dig index and serves filtered lists of digs and evidence bundles over a bearer–secured REST API.

utl_cli: the command-line client used to register roots, record transitions, list digs, and inspect DigFiles.

security_host: a host agent stub that reads DecisionEvent records and shows how a real firewall / isolation controller would be driven.

security_os: a crate that defines abstract traits such as FirewallController, CgroupController, IsolationScope, IsolationProfile, and FlowDecision, which describe how a host OS would enforce policy.

Pravyom: the broader research and product programme. This document covers Part I: the policy / truth / forensics core.

Identity and Scope

tenant_id: identifies a tenant or customer (for example "acme"). Many policies and forensic queries are scoped per tenant.

root_id: a 128–bit identifier for a root of truth. A root groups related transitions and DigFiles (for example all events for a particular deployment or ledger).

entity_id: a 128–bit identifier for an entity participating in transitions under a given root.

zone: a logical or network zone (for example public, internal, admin). Policies often reason about src_zone and dst_zone.

DID (Decentralised Identifier): a structured string such as `did:ritma:tenant:acme` or `did:ritma:svc:acme:public_api` used to name tenants, services, zones, and actors.

Data Structures

NodeRequest: the JSON message type sent to utld. Examples include RegisterRoot, RecordTransition, and BuildDigFile.

RecordTransition: a NodeRequest variant representing one transition (for example an HTTP request) with signatures, data, and a parameter container (p_container).

DigFile: a sealed, append–only JSON log of DigRecords for a given root over a time range. Each DigFile has a Merkle root and may be signed.

DigIndexEntry: a small index record for each DigFile containing file id, root id, tenant id, time range, record count, Merkle root, policy metadata, and storage path.

DecisionEvent: a structured record emitted whenever a policy evaluation produces actions. It contains event kind, policy decision, rules, actions, and optional DIDs and zones.

Storage and Formats

JSONL: JSON Lines format; each line of a file is a separate JSON object. Used for the dig index and decision events log.

S3–style path layout: an object storage style path such as `forensics/<tenant>/<YYYY>/<MM>/<DD>/root-<r` The forensics_store crate writes DigFiles in this layout on the local filesystem.

Forensics store: the on-disk directory tree that holds DigFiles in the S3–style layout.

Dig index: the combination of dig_index.jsonl and optional dig_index.sqlite that lets services quickly find DigFiles by tenant, root, time, and policy decision.

Cryptography and Integrity

Merkle tree / Merkle root: a hash tree built over DigRecords; the root hash commits to the entire set of records in a DigFile.

HMAC: a keyed hash (Hash-based Message Authentication Code) used to sign DigFiles so that tampering can be detected.

SNARK (zkSNARK): a succinct zero-knowledge proof system. In this codebase it is used in demo form to show how a policy could require a cryptographic proof before allowing a transition.

Distillium micro-proofs: compact proofs for specific properties of a root or transition, used here as a placeholder for more advanced proofs.

Policy and Enforcement

TruthScript: the policy language used by the PolicyEngine to decide allow, deny, or allow_with_actions.

Lawbook: a curated set of policies and constraints for a tenant or environment, encoded in TruthScript (and, in future, CUE).

PolicyEngine: the component inside utld that takes an EngineEvent and a policy and returns EngineActions (Deny, SealCurrentDig, proof requirements, and so on).

EngineAction: an action produced by the PolicyEngine, such as Deny or SealCurrentDig.

FlowDecision: an enum in security_os that describes whether a flow should be allowed, denied, or throttled. The host agent maps DecisionEvents into FlowDecisions.

IsolationScope / IsolationProfile: types in security_os that describe how a process or container should be isolated (for example which cgroup or namespace settings to apply).

How to Evaluate Your System with Ritma / UTL

This section outlines a practical way for a team to evaluate their own system using the current Ritma / UTL capabilities.

1. Choose a Pilot Scope

- Pick one or two tenants (for example a non-production tenant such as "acme-sandbox").
- Choose a handful of services and flows that matter (for example public API calls into an internal service, or admin actions on critical data).
- Register one or more roots in utld that correspond to these critical areas.

2. Wire UTL into the Path

- Point the selected flows through utld (for example via an HTTP shim, sidecar, or client library).
- Ensure each RecordTransition includes tenant, DIDs, zones, and relevant business fields in the parameter container.

3. Define an Initial Policy

- Start with a small TruthScript policy that encodes obvious rules: which tenants and zones may talk, which actors may call which services, and when digs should be sealed.
- Run in allow-with-actions or shadow mode first, so that you collect DecisionEvents and DigFiles without breaking traffic.

4. Generate Realistic Traffic

- Replay real traces (if available) or run normal staging / pre-prod workloads through the instrumented path.
- Intentionally trigger a few known-bad scenarios (for example a forbidden zone crossing) to exercise deny paths and deny-forensics.

5. Inspect Digs and Decisions

- Use utl_cli and the forensics HTTP API to list and inspect DigFiles by tenant, root, time, and policy decision.
- Check that the Merkle-rooted digs contain enough context to answer "what happened" for your key scenarios.
- Review DecisionEvent logs (or run security_host) to see whether the live decisions match your expectations.

6. Define Success Criteria

- Mean-time-to-understand (MTTU): how long it takes to reconstruct an incident from DigFiles versus previous logging approaches.
- Coverage: how many critical flows are now both enforced by policy and recorded in digs.
- Auditability: whether an external reviewer could understand the policy decisions and evidence using only DigFiles, the dig index, and DecisionEvents.

7. Iterate on Policy and Scope

- Add new rules, tenants, and flows based on what you learn.
- Gradually move from pure observation to enforcement (from allow-with-actions to real denies) once confidence is high.

This evaluation loop can be run in a free pilot, a scoped paid pilot, or as part of a broader enterprise deployment plan.

Business Model and Go-To-Market

This section sketches how Ritma / UTL moves from free pilots to paid pilots and then to an enterprise programme, and where revenue is generated at each stage.

Phase 1: Free Pilots

In the first phase we run a small number of free pilots with design partners. The goals are:

- Prove technical fit: can UTL enforce the required policies and produce usable digs and evidence bundles in their environment?

- Prove operational fit: can their teams adopt the CLI, API, and dashboards without excessive friction?
- Co-design repeatable reference architectures for specific verticals (e.g., fintech, SaaS, infra-as-a-service).

During this phase we do not charge licence or usage fees. We may, however, ask for:

- Access to realistic traffic and policies for tuning the engine.
- Rights to anonymised metrics and case studies that feed into the product story.

Phase 2: Proofed / First Paid Pilots

Once we have proof-of-value from free pilots, we convert a subset into paid pilots. These are typically structured as small, time-bound engagements (for example 3–6 months) with clear success criteria.

Possible charging models in this phase:

- A fixed pilot fee that covers onboarding, policy authoring support, and managed operation of the UTL stack.
- Optional professional services for custom integrations (billing based on days or sprints).
- Usage-based components for heavy forensic storage or very high decision volumes.

Customers in this phase are paying primarily for:
(a) reduced incident and compliance risk and
(b) faster investigations enabled by structured digs and decision events.

Phase 3: Enterprise Programme

For enterprises that validate the pilot, we move to a programme–level engagement. At this stage we expect:

- Multiple clusters and environments (prod, staging, regions).
- Tight integration with existing SIEM, ticketing, and compliance workflows.
- Shared roadmaps for deeper host enforcement and governance features.

Typical enterprise revenue components:

- A base subscription per tenant or per cluster for the policy and enforcement engine.
- Volume-based pricing for decision throughput (for example number of policy evaluations per month) and forensic storage consumed.
- Optional premium features (advanced ZK proofs, long-term evidence retention, advanced analytics) as add-on modules.
- Ongoing professional services for new lawbooks, audits, or migrations.

Why Customers Will Pay (Grounded in Current Maturity)

Even at the current stage of the codebase, customers gain concrete value:

- A running utld daemon that enforces allow / deny decisions from a real policy file (TruthScript) on every RecordTransition.
- Automatic DigFile sealing on SealCurrentDig and on Deny actions, with DigFiles written to a local dig directory and mirrored into an S3–style forensics tree.
- A dig index (JSONL plus optional SQLite) that records tenant, root, time ranges, record counts, Merkle roots, policy decisions, and storage paths.
- A forensics HTTP service (utl_forensics) that lets them filter digs by tenant, root, time window, and policy decision, and fetch evidence bundles over a bearer–secured API.
- A DecisionEvent stream that explains why each policy fired, with a host agent stub (security_host) proving how this can drive firewall / isolation controllers.
- A working CLI (utl_cli) that can list digs, inspect by file id, and introspect roots and transitions using the same code paths.

Because this is backed by concrete code rather than slideware, there are customers who will pay earlier in the lifecycle:

- Teams with painful incident forensics or audit workloads can buy a paid pilot that focuses purely on DigFiles, the dig index, and the forensics API, even before full host enforcement arrives.
- Regulated SaaS and fintechs can justify a pilot fee because UTL already produces structured, sealed evidence that shortens audits and improves regulator conversations.
- Security teams that currently spend days grepping logs can see immediate time savings from the CLI and API, which is a direct and measurable ROI.

In practice the path is:
design partner (free) $\rightarrow$ proofed pilot (paid, scoped) $\rightarrow$ enterprise programme (recurring subscription plus usage and services). This keeps the early stage focused on proving value while establishing clear surfaces where value is monetised once the product is trusted.