From Toy to Instrument: Seven Years of Verifpal

Nadim Kobeissi

Symbolic Software

Abstract. Verifpal is a symbolic protocol analyzer introduced in 2019 to make formal analysis accessible during protocol design. It provides a small modeling language, built-in cryptographic primitives, and attack reports that use the model's names. This paper describes the analysis engine of Verifpal 1.6.7. A backward solver proposes attacker actions, which an executor checks by running all sessions and peer configurations together. An injected value is delivered only when the attacker can derive it from information already available in that execution. A separate query evaluator, or judge, records the result.

We define the model, its five query predicates, and a reference semantics, and prove that every judged execution is a history of that semantics. Executions terminate; search termination is conditional on the solver calls terminating. Evaluation covers 885 secrecy queries across 189 configurations of the fifteen fundamental Noise patterns, varying static-key compromise and protocol checks. At one and two sessions, a separate executor reproduces every reported disclosure, and ProVerif proves all 290 passing counterparts for unbounded sessions. The protocol computations also match 90 external cryptographic test-vector messages. These results support the implementation on the tested models; a Verifpal pass remains the outcome of bounded, incomplete search.

Keywords: symbolic protocol analysis · formal verification · attack search · Dolev–Yao model · Verifpal · ProVerif · Tamarin

1 Introduction

Cryptographic protocols can fail even when their primitives are secure. An attacker may replay a message into another session, cause a key to be accepted by the wrong peer, or make a participant accept a value that no honest party sent. Symbolic analysis studies these failures by treating cryptographic primitives as perfect and giving the attacker control of the network [52]. ProVerif [16] and Tamarin [34] have been used to analyze deployed protocols, including TLS 1.3 [5] and 5G authentication [6], and can prove properties for any number of sessions. Using them requires a model written as processes or rewrite rules; difficult models may also need lemmas or proof guidance. An unsuccessful proof attempt may be inconclusive [19, 36].

Verifpal [1], introduced in 2019, was designed for use during protocol design, including by people with little experience in formal modeling. A model lists each principal's computations, the messages exchanged, and the properties to check. Cryptographic primitives are built in, analysis requires no proof hints, and attack reports use the names from the model. The original paper left the meaning of reported attacks and passing queries informal. Figure 1 summarizes the approaches of the three tools.

E-mail: nadim@symbolic.software (Nadim Kobeissi)

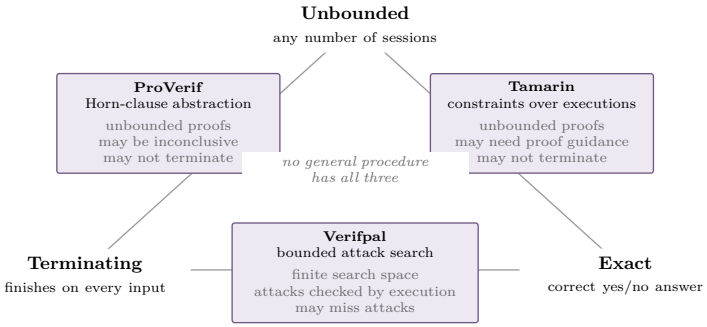


Figure 1: Approaches to symbolic analysis, adapted from Verifpal's comparison guide [3]. Secrecy for unbounded sessions with fresh nonces is undecidable [4], so a general analyzer cannot have all three properties. Positions are schematic and do not imply that a tool satisfies either adjacent property.

The reliability of a verdict also depends on the implementation. In September 2026, pull requests to Tamarin proposed fixes for nine soundness bugs in its prover and twelve bugs in its process translation; some had allowed false lemmas to be proved [31]. This motivates making explicit which parts of an analyzer must be correct for its reports to be sound.

This paper describes the rebuilt analysis engine of Verifpal since version 1.0, today represented by Verifpal 1.6.7. It separates attack search from query evaluation. A backward solver proposes values to inject into protocol messages. An executor then runs all sessions and peer configurations together, delivering each injection only if the attacker can derive it from information already available in that execution. A judge evaluates the queries on the resulting execution (Figure 4). The search cannot record a result or add information to the executor's attacker knowledge.

A failing query comes with an execution that can be replayed from the reported attacker actions. We define a reference semantics for Verifpal models and prove that every judged execution is one of its histories (Theorem 1). This rules out reports that combine incompatible runs or use information before it becomes available. A passing query means that the search found no violating execution within the chosen bounds; the result label makes this scope explicit. Executions always terminate, and search termination is conditional on the solver calls terminating (Theorem 2). In the corpus evaluation, every analysis but one finished within an hour.

Contributions.

- An analysis architecture that checks proposed attacks by executing the whole protocol, with the search outside the code responsible for confirming violations (§2, §5).
- Definitions of execution and the five query predicates, together with a proof that judged executions satisfy the reference semantics (§3, §4).
- A search bounded by the session count and a term-depth limit computed from the model, with result labels that distinguish exhaustion from truncation (§5).
- Explicit sessions and peer configurations, with compromise dated by phase, supporting analysis of Lowe's attack, attacks requiring two responder instances, and forward secrecy (§6).
- A differential evaluation of all fifteen fundamental Noise patterns under systematic static-key compromise and protocol omissions, with external test-vector validation and independent replay of reported attacks (§8.1); corpus, semantic and scaling

```
attacker[active]
principal Alice[
  knows public c0
  generates a
  ga = PUBKEY(a)
]
Alice -> Bob: ga
principal Bob[
  knows public c0
  generates m1, b
  gb = PUBKEY(b)
  gab = DH_KEX(ga, b)
  generates n_e1
  e1 = AEAD_ENC(gab, n_e1, m1, c0)
]
```

```
Bob -> Alice: gb, n_e1, e1
principal Alice[
  gba = DH_KEX(gb, a)
  e1_dec = AEAD_DEC(gba, n_e1, e1, c0)?
]
queries[
  confidentiality? m1
  authentication? Bob -> Alice: e1
]
```

Figure 2: The running example, `simple.vp`. Neither public key is guarded, so the attacker can replace either one.

studies complete the evaluation.

Changes since 2019. The language retains its original structure and adds key encapsulation, threshold secret sharing and FROST-style signatures [7], AEAD with an explicit nonce, weakening annotations that take effect at a chosen phase, and peer scenarios. The engine was rewritten in Rust and is the focus of this paper.

2 Verifpal by Example

We use `simple.vp`, a shipped Verifpal model, as a running example (Figure 2). It describes an unauthenticated Diffie–Hellman exchange followed by one encrypted message.

A `principal` block lists a participant's actions in order. The `knows` declaration introduces public or private long-term material, while `generates` introduces a fresh value for each session. Assignments apply built-in primitives: `PUBKEY(a)` is the public key g^a, `DH_KEX` computes a Diffie–Hellman secret, and `AEAD_ENC` encrypts under a key and an explicit nonce, with associated data. A trailing `?` makes a computation a *check*: if Alice's decryption fails, her run halts.

A message line such as `Alice -> Bob: ga` sends named values, all visible to the attacker. An active attacker can replace or drop each value, except that a *guarded* value, written `[ga]`, cannot be replaced. Guards model out-of-band authentication such as a certificate; they do not hide the value. Models can also disclose values with `leaks`, use `phases` to specify when compromise occurs, and list alternative peer bindings in a `scenarios` block (§6). The manual describes the full language [2].

The `queries` block checks whether the attacker can learn Bob's message (`confidentiality? m1`) and whether Alice can successfully use a ciphertext that Bob did not send her (`authentication? Bob -> Alice: e1`). Verifpal also supports `freshness?`, `unlinkability?` and `equivalence?` queries. Section 4 defines all five predicates.

Both queries fail. In the confidentiality attack (Figure 3), the attacker replaces Alice's public key with `PUBKEY(nil)`, whose private key is the public constant `nil`. Bob therefore computes a shared secret with the attacker and encrypts `m1` under a key the attacker can derive. The report calls this key `gab`, as in the model, and shows its value in the attack.

Finding the attack. Verifpal first expands the model into *runs*: at the default two sessions, two copies each of Alice and Bob. It executes these runs without interference and finds no query violation. The active search then works backward from `m1`: the attacker

```
Fail X confidentiality? m1
Attack trace:
1. Attacker constructs PUBKEY(nil).
2. Attacker replaces ga (sent by Alice to Bob) with PUBKEY(nil). (ga was PUBKEY(a))
3. Attacker observes e1 on the wire, where it is AEAD_ENC(gab, n_e1, m1, c0).
4. Attacker observes gb on the wire.
5. Attacker constructs gab, where it is DH_KEX(gb, nil).
6. Attacker observes n_e1 on the wire.
7. Attacker opens e1 with gab, n_e1, obtaining m1.
-> m1 (m1) is obtained by Attacker.
```

Figure 3: Verifpal's report for `confidentiality? m1`. The wording is verbatim; terminal framing and status glyphs are simplified.

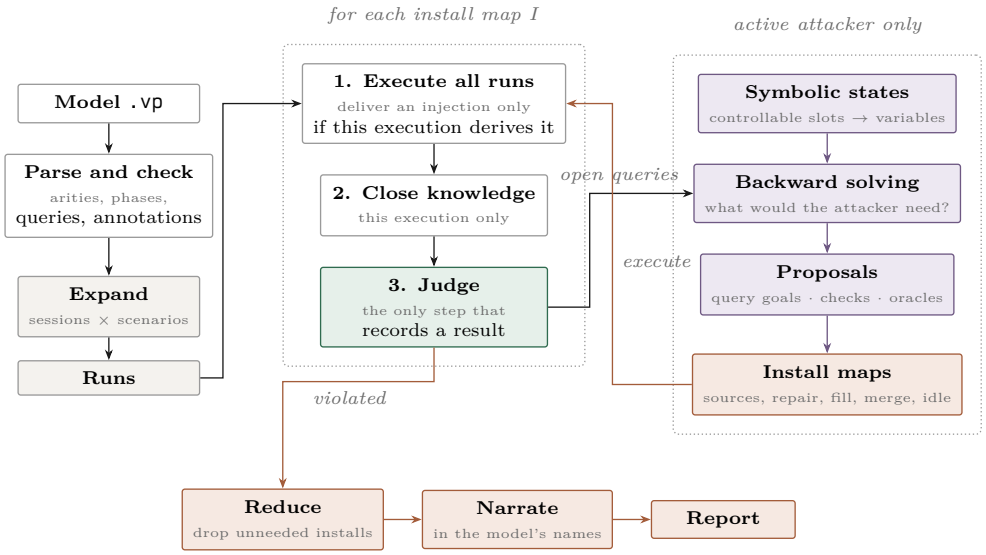


Figure 4: The analysis pipeline. After judging the honest execution, the engine searches for attacks on open queries if the attacker is active. The search proposes install maps. Each map is executed and judged if all its injections are delivered. Only the judge records results.

needs Bob's ciphertext and its key. Since the key depends on the unguarded `ga` received by Bob, the solver proposes replacing `ga` with `PUBKEY(nil)`. This proposal becomes an *install map* specifying the replacement at Bob's receive.

The executor checks the map by executing all four runs together. It delivers `PUBKEY(nil)` because the attacker can derive it at that point. Bob then sends `gb`, the nonce and the ciphertext, allowing the attacker to derive `m1`. The judge records the violation, and the engine removes unnecessary installs before producing the report (§7). If an injection could not be derived, its receive would wait; an execution with an undelivered injection would not be judged.

3 The Model

Verifpal represents messages as terms and specifies the attacker's capabilities through rules for the cryptographic primitives [52]. This section defines those terms and rules, followed by the execution semantics.

3.1 Terms and primitives

Definition 1 (Terms)**.** Let $\mathcal{C}$ be the constants of a model, with a distinguished public constant `nil`, and let Σ be the built-in primitives. Terms are

$$t \ ::= \ c \ \mid \ f^{(i)}(t_1, \ldots, t_n) \qquad c \in \mathcal{C},\ f \in \Sigma,$$

where f accepts arity n and i selects one of its outputs; HKDF, for example, derives up to five keys from one call. We write $f(\vec{t})$ when f has one output.

A threshold sharing records both its threshold and the assignment that created it, distinguishing separate sharings of the same secret. All constants, including each session's copy of a generated value, are fixed before analysis. The attacker cannot introduce new ones.

Verifpal has 25 built-in primitives. Three are structural: CONCAT builds a tuple, SPLIT extracts its fields, and ASSERT tests equality. The other 22 cover hashing, key derivation, MACs, Diffie–Hellman, key encapsulation, symmetric, authenticated and public-key encryption, threshold secret sharing, and ordinary, ring, blind and threshold signatures.

Each primitive has a record in a shared table (Appendix A.1). The record specifies arities, outputs, rewrite rules, checks, argument restrictions, weakening annotations, and any rules for nonce reuse or threshold shares. It also specifies what an attacker can learn from a term: for example, holding $\mathsf{ENC}(k, m)$ and k reveals m, corresponding to the rewrite $\mathsf{DEC}(k, \mathsf{ENC}(k, m)) \to m$. The parser, deduction rules, search and executor all use this table. Users need not define primitives, but cannot add their own primitives or equations, and the table's correctness must be trusted.

Equivalence. Terms are compared modulo one equation, for Diffie–Hellman:

$$\mathsf{DH_KEX}(\mathsf{PUBKEY}(a), b) \ \approx \ \mathsf{DH_KEX}(\mathsf{PUBKEY}(b), a),$$

Here $\equiv$ is the least congruence containing this equation. Equivalent terms receive the same hash, allowing knowledge lookups to use hash buckets. Argument restrictions include forbidding PUBKEY around PUBKEY or DH_KEX. These restrictions prevent deriving a shared secret from two public keys, reflecting the computational Diffie–Hellman assumption. They also exclude nested exponents such as g^{abc}.

Checks. An assignment with a trailing `?` halts the run if its rewrite rule does not apply, for example because of a wrong key, a malformed tuple or unequal arguments to ASSERT. Without `?`, an application that does not rewrite remains an opaque value, such as $\mathsf{DEC}(k', c)$.

3.2 What the attacker knows and can build

Each execution maintains its own attacker knowledge, initially `nil` and the public constants. Values disclosed by sends and leaks are added to this knowledge, which is closed after each addition under Table 1. The attacker can *obtain* a value if it holds it, computes it by a successful rewrite from obtainable arguments, or obtains it through a threshold or weakening rule.

Closure operates on held terms; knowing a key does not add every ciphertext under that key to knowledge. Definition 2 specifies which additional terms the attacker can construct.

Table 1: Closure rules for attacker knowledge. The primitive table supplies every rule's details.

Rule	What the attacker gains
DECOMPOSE	the contents of a held term, once it obtains what the primitive's record requires: m from $\mathsf{ENC}(k, m)$ and k; every field of a tuple; the secret and randomness of a KEM ciphertext, given the private key
RECONSTRUCT	a computation that occurs in the execution, once every argument is obtainable
RECOMPOSE	a threshold secret, from τ distinct shares of one τ-of-n sharing
BREAKWEAK	the inputs that a weak annotation in force exposes, such as the preimage of a weak hash
FORWARDREWRITE	the result of a principal's computation whose arguments are all obtainable, such as the signature $\mathsf{SIGN}(sk, m)$ that UNBLIND produces
BUILDREWRITE	the result of applying a rewrite rule itself to a held term, such as unblinding a held blind signature with a known factor
REUSE	what a nonce-reuse pair gives away: two threshold partial signatures under one share and nonce reveal the share; AEAD pairs are described below

Forged terms. Some annotations allow the attacker to construct a term without knowing its secret argument, as with a signature declared forgeable. Decomposing such a term must not disclose that secret. The engine therefore opens a term that could be forged only if a principal built it in this execution and the attacker either received it directly or can derive it without forging that term. If the term is available only through forgery, the secret remains hidden.

AEAD nonce reuse. The AEAD model represents the nonce-reuse behavior of AES-GCM and ChaCha20-Poly1305: a key and nonce determine the keystream masking the plaintext. Two different ciphertexts under the same key and nonce reveal the authentication key, allowing the attacker to re-tag either ciphertext under any associated data. Knowing either plaintext then reveals the other and allows arbitrary ciphertexts to be constructed under that key and nonce. The model does not represent partial knowledge of plaintexts or XOR relations between unknown plaintexts.

Proposition 1 (Closure terminates)**.** *For a fixed execution, closing attacker knowledge reaches the least fixed point of the rules after finitely many steps.*

Proof. Every rule yields a subterm, an output, or a reduct of a term that occurs in the execution or in the initial knowledge, or the combined signature of a partial that occurs there. Reductions and normalization never enlarge a term, and no rule introduces constants. All conclusions therefore lie in a finite set fixed by the execution. Each rule uses knowledge only in positive premises, so the one-step operator is monotone and inflationary on the subsets of that finite set, and iteration reaches its least fixed point above the initial knowledge. □

Definition 2 (Derivability)**.** A term t is *derivable* from knowledge $\mathcal{K}$ if the attacker can obtain it, or if $t = f^{(i)}(\vec{u})$ and every u_k is derivable, with these exceptions:

- a secret argument may be skipped when a forgeable annotation for f under that secret is in force;
- a held $\mathsf{ENC}(k, m)$ under a malleable annotation yields $\mathsf{ENC}(k, m')$ for any derivable m';
- two AEAD ciphertexts under one key and nonce let the attacker re-tag either one and, once one plaintext is known, build any other ciphertext under that key and nonce;

- a threshold share is derivable from τ other shares of its sharing, or from the secret and $\tau - 1$ other shares, but not from the secret alone; and the attacker may combine partial signatures that it makes itself, over shares it obtains or may forge.

Weakening annotations. An annotation can weaken a primitive from a specified phase onward, as in `SIGN[forgeable from phase 2](sk, m)`. The weak annotation exposes declared inputs of HASH, PUBKEY and four encryption primitives. The forgeable annotation permits omitting the secret argument of SIGN, MAC, RINGSIGN, AEAD_ENC and THRESHOLD_SIGN; malleable applies only to ENC. An annotation covers every term computed by that call, including computations on attacker input. Forgery permission applies to the secret itself: marking one signature under *sk* forgeable makes all signatures under *sk* forgeable. Verifpal reports each unannotated occurrence affected by this rule.

3.3 Runs and executions

Before analysis, Verifpal expands the model into a fixed set of *runs*, one for each principal, session and peer configuration (§6). Each run follows its principal's declarations, assignments, leaks, sends and receives in source order, with every action assigned a phase. A message pairs a send in one run with a receive in another; each value is either guarded or unguarded. A run stores values in the *slots* it has reached. A received slot also records its sender and whether its value is *authored*, meaning supplied by the attacker or relayed from a run that held it as authored.

What the attacker does. An active attacker acts on an execution through an install map.

Definition 3 (Install map)**.** An *install map* I is a finite set of attacker instructions, each of one of three kinds: *inject* a term at an unguarded receive slot of a run; *drop* one value that a run receives, guarded or not; or *idle* a run, which then never starts. The empty map gives the *honest* execution.

An injection replaces a value in transit. A drop withholds one value of a message while allowing the others to be delivered; the recipient stops at its first action that needs the missing value. An idle run never starts, so an expansion to k sessions permits *up to* k runs of each principal.

Definition 4 (Joint execution)**.** Fix an install map I. The attacker's knowledge starts with `nil` and the public constants. The execution exec(I) proceeds phase by phase. In phase p, it repeatedly advances every running run whose next action belongs to a phase no later than p:

- an assignment computes and reduces its value; a failed check halts the run, and a checked assignment with several outputs is one check;
- a send records the values the run holds and discloses to the attacker those it does not hold as authored; a leak discloses its value unless it is authored;
- a receive whose sender has sent takes, at each slot, the sent value; at an unguarded slot where I injects a term, it takes that term's normal form instead, provided the term is admissible and derivable (Definition 2) from the attacker's knowledge at that moment, and otherwise waits; a slot that I drops stays empty;
- an action that reads an empty slot stops the run.

When no run can move, the executor may take one receive whose sender has not sent yet, if every slot of that receive is unguarded and carries an injection the attacker can deliver; it picks the earliest-declared such receive and resumes. When nothing more can happen in phase p, every unfinished run is abandoned for good, knowledge is closed (Proposition 1),

and the configuration is archived as Ω_p with knowledge $\mathcal{K}_p$. An injection that the attacker could not deliver by then is *stuck*.

An injection equal to the sender's value is treated as that sent value, with its existing authorship. Early receives occur only when no other action can advance, in declaration order. Since a send depends only on earlier receives, an upstream sender sends before a downstream receive is taken early. An injection that anticipates an honest value is therefore not marked authored.

Knowledge grows monotonically and recorded sends are never retracted. The values reached by the runs are consequently independent of the order in which they advance, giving a unique maximal execution $\mathsf{exec}(I)$. Phases impose barriers: a receive can use only information already disclosed, so a key leaked in phase 1 cannot justify a forgery in phase 0.

4 Judging an Execution

The judge evaluates queries using the configurations archived at phase ends and the execution as a whole. It has no access to the search state.

Runs included in a query. Peer configurations (§6) may include corrupt peers. The runs that *answer* a query at phase p are those whose configurations were honest at phase 0. If a configuration remains honest at p, its run answers for its state at the end of p. If it became corrupt in phase $N \leq p$, the run answers for its state at the end of phase $N - 1$, evaluated against the attacker's knowledge at the end of p. This comparison checks forward secrecy. If no configuration is honest at phase 0, every run answers. A run *sees* its own slots and those created by other answering runs, reading the latter values from their creators. This allows a query to compare two parties' copies of a shared key.

4.1 Query predicates

Authentication and freshness use the following definition of acceptance, based on a value's use within a run.

Definition 5 (Successful use)**.** Fix an execution, a run b and a constant c that b holds. The *uses* of c are b's primitive assignments that depend on c through b's own computations or, if there are none, through other principals' computations. Run b *successfully uses* c by phase p when every use is reached in the execution, except outputs of a checked assignment at which b halts, and some use reached by the end of p is not such an output.

For example, a forged ciphertext may decrypt successfully but fail a later `SPLIT( ... )?`. If that check is the last use, the earlier decryption counts as successful use. If the failure prevents another use from being reached, the value is not accepted.

Definition 6 (Query violation)**.** Fix an execution $\mathsf{exec}(I)$, a phase p and its knowledge $\mathcal{K}_p$. Each answering run is read at the phase it answers for. Query q is *violated at* p when:

- **confidentiality** c: some answering run holds c's slot, and the attacker can obtain its value from $\mathcal{K}_p$;
- **authentication** $A \to B : c$: the queried run b of B answers, holds c's slot as received from a run s, successfully uses c, and one of these holds: the value is not authored, s is not a run of A, and the value has no *emission* (*substitution*); the value is authored and has no emission (*forgery*); or the value is authored and more runs of B's agent successfully use an equivalent value at c's slot, or at a session or scenario copy of it, than there are emissions (*duplicate acceptance*);

- **freshness** c: some answering run r successfully uses c, and either its value contains no generated constant, or the value is *replayed* and another run of r's agent also successfully uses an equivalent value at c's slot or a copy of it;
- **equivalence** $c_1, \ldots, c_n$: in some answering run, every named slot is seen and reached, none holds a failed check, and the values are not all equivalent;
- **unlinkability** $c_1, \ldots, c_n$: some answering run sees two of the named values and a link witness exists for them (Definition 7).

An emission is a send of an equivalent value at c's slot or a copy of it, by a run interchangeable with A, in a message that reaches a run of B's agent directly or through relays, provided some run sends that value unauthored. A value is replayed when it is authored, received from a sender whose copy is replayed, or computed by r from a replayed input. A query with `precondition[P -> T: d]` counts a violation only if the execution sent d from P to T in some phase. The engine judges every query, together with its session and scenario copies, at every phase; a violation of any copy fails the written query.

Interpreting the predicates. Confidentiality concerns the value held in the queried slot during the execution. In the running example, Bob's key is disclosed even though it differs from the key in the honest execution. If the attacker instead replaces a ciphertext with an encryption of `nil`, the recipient's decrypted slot is public but the original message remains secret. The report marks this case with `[attacker-supplied value]`.

Authentication requires a matching send for an accepted value. Copies of A with the same peer binding are interchangeable. A copy with another binding is interchangeable only if neither the slot nor any check before its send depends on a constant rebound by its configuration. A value has no emission if the attacker diverted, halted or silenced the run that would have sent it, even when it equals the honest value. This includes the textbook reflection of a MAC challenge through a second challenger run, and Abadi and Needham's attack on Woo–Lam II [50], in which the attacker supplies the reply that an honest server would have sent. An honest relay does count as an emission, including when it forwards another session's message.

Duplicate acceptances carry the qualifier `[duplicate acceptance]` if the value contains a recipient-generated constant, such as a challenge, and `[duplicate acceptance: no recipient-generated context]` otherwise. In the latter case, repeated acceptance may be intended by the protocol.

Freshness detects a replay only when two runs of the same agent accept the value in one execution. A value from another session passes this test if no second run accepts it.

Definition 7 (Link witness)**.** Take two queried values a, b seen by an answering run, each from that run's copy unless it is authored, otherwise from its creator's. Both must be *observable*: public, a tuple of observable values, or extracted by projection, decomposition, a weakening or a held reuse pair from an executed send or leak of a value that was not authored. Let S be the secret constants shared by the honest, unreduced definitions of a and b; if S is empty and the honest values differ, there is no link. A *witness* is a term w containing a generated or private constant, found by the first of these cases that applies:

- *observed equality*: $a \equiv b$, and w is that value;
- *identifying check*: one runnable check exposes the same identifier w in both values, as signature verification exposes the signer's key;
- *common secret*: the attacker rebuilds both values from a held w distinct from both, which stays derivable when either value is withheld, or one value is a building block of the other;
- *recognized secret*: a runnable check on a held value confirms an argument w tied to the other value.

The identifying and recognized cases are also tried on tuple components and building

blocks of the values. No witness may be buildable, without weakening, from public values and what the attacker injected into a and b, and every witness except an equality of honestly equal values must contain a constant of S.

The condition on S requires a link to depend on a secret shared by the honest values. For example, forcing two tags to $\mathsf{MAC}(k, \mathtt{nil})$ links the runs if both tags use the same secret key k, as in the RFID tracking attack. Different keys yield no link. A passing `unlinkability?` query means that this procedure found no witness; it does not establish indistinguishability.

4.2 Reference semantics and soundness

The executor uses a fixed schedule. We relate its executions to a reference semantics that permits all schedules.

Definition 8 (Histories)**.** Fix an expanded model. A configuration holds the current phase p; for each run, a program counter, the values of the slots it holds with their authorship, and a status (running, halted, stopped, abandoned or complete); the messages emitted so far; the attacker's knowledge; and the knowledge archived at each earlier phase end. A transition advances one running run by one action of a phase no later than p, abandons a running run, or moves to phase $p + 1$:

- an assignment computes and reduces its value; a failed check halts the run, and a checked assignment with several outputs is one action;
- a send emits the run's values and adds those not authored to the attacker's knowledge; a leak adds its value unless it is authored;
- a receive takes, at each slot, either the value emitted by its send, once emitted, or nothing once that value is emitted; and, at an unguarded slot, alternatively any admissible term without a failed embedded check that is derivable from the attacker's knowledge at that point. A taken term other than the emitted value is authored, and so is a relayed authored value;
- an action that reads a slot the run does not hold is never enabled;
- the move to phase $p+1$ is enabled once every running run has passed its last phase-p action. It closes the attacker's knowledge and archives it as $\mathcal{K}_p$.

The attacker never creates a constant. A *history* is a finite sequence of transitions from the initial configuration. We write $\mathsf{IReach}_k(q)$ when some complete history of the k-session expansion violates q under Definition 6, reading each run's values and status at the phase it answers for, knowledge at the end of p, and successful use and preconditions over the whole history.

These transitions are allowed in a bounded-session Dolev–Yao model: runs are sequential processes, guards are authenticated channels, and message values are delivered individually. The attacker may deliver any value it can deduce or construct, or withhold delivery. Our reference semantics additionally fixes the atomic names. Thus $\mathsf{IReach}_k(q)$ implies an attack in the standard model, but the converse can fail for attacks requiring fresh attacker names.

Theorem 1 (Executions are histories)**.** *For every install map I, the steps of $\mathsf{exec}(I)$, in the order the executor takes them and with an abandonment inserted for each unfinished run at each phase end, form a history, and each archived Ω_p is that history's configuration at the end of phase p. Consequently, if q is violated in $\mathsf{exec}(I)$, then $\mathsf{IReach}_k(q)$.*

Proof. By induction over the executor's steps, each step is an enabled transition from the configuration reached so far, after which the two systems agree on program counters, held values and their authorship, statuses, emitted messages and attacker knowledge.

Table 2: What each result establishes. A failure may also carry one of the qualifiers explained after Definition 6.

Output	What it establishes	What it does not
`PASS [search exhausted at` k `sessions]`	No executed install map violates the query; the depth bound rejected no proposal while the query was open.	Security beyond the maps tried, including attacks needing more sessions or proposals the search did not generate.
`PASS [search truncated: term depth]`	No executed install map violates the query, but the depth bound rejected a proposal while it was open.	Coverage of the implemented search space.
`FAIL`	Executing the listed installs violates the query in a history of the reference semantics (Theorem 1).	Agreement between the queried predicate and the intended security property.

Assignments, sends and leaks read only the run's own values, so they compute and halt alike and disclose the same values, since both systems withhold authored ones. The executor never takes an action that reads an empty slot, and the reference never enables one. For a receive, every slot without an injection takes a value the sender's executed send recorded, which is an emitted message of the reference configuration. A drop is taken only after that send, as the reference requires. An injection is delivered only at an unguarded slot, only if it is admissible with no failed embedded check, and only if it is derivable from the executor's knowledge at that moment. That knowledge is the closure of `nil`, the public constants and the values disclosed by sends and leaks that precede the receive in the order, so the term is derivable in the reference configuration too. An injection equal to an emitted value is taken as that value in both systems, and the two assign authorship by the same rule. A receive taken before its sender sends has an injection at every slot, so it needs no emitted value.

The executor leaves phase p only when no run can move: every running run then waits on a receive it cannot take, or is stopped before an action that reads an empty slot, and since knowledge and emitted messages only grow, nothing later in phase p changes that. Abandoning these runs and then moving to phase $p+1$ are enabled, both systems archive the same closed knowledge, and the history reaches Ω_p at the end of each phase p. The judge reads exactly these configurations, with Ω_{N-1} for runs corrupt from phase N, and reads successful use and preconditions over the whole execution, which is the complete history. A violation it finds is therefore a violation under the reference reading, which gives $\mathsf{IReach}_k(q)$. □

Applying the theorem to the implementation requires the executor, closure and judge to implement these definitions correctly. This has not been proved mechanically; §8.1 checks Noise secrecy results against ProVerif and replays reported attacks with a separate executor and deduction checker.

4.3 What a result means

Table 2 states what each output establishes.

Implementation boundary. Implementation errors can invalidate a verifier's conclusions. For example, Tamarin 1.12.0, used in our comparison, proves that no message is decrypted in a two-role process model where the attacker can encrypt a message under a published key. Renaming one variable reverses the verdict [32].

Verifpal restricts the code that can confirm a violation. The result writer requires a verdict token that only the query evaluator can create, after finding a violation in an execution. Only the executor can produce executions. It is called for the honest run, by

the search and witness reducer, and once to date scenario compromise; that last call reads knowledge without judging queries. Injections are delivered only by the executor's receive operation, which first checks derivability. Source-structure tests check these boundaries, and the crate forbids `unsafe` code.

Excluding tests, the executor, judge and result writer comprise 2,678 physical lines of Rust; the knowledge closure, primitive table and term code comprise 4,565; and session and scenario expansion with model construction comprise 1,866. The search and solver add 9,121 lines outside this core. Errors there can miss attacks or waste time, but cannot record violations. Errors in the core can cause false reports or false passes. Reported attacks include executions that can be replayed and checked against the model, as in §8.1.

5 Finding Attacks

The search generates install maps for queries that remain open. Its heuristics affect which attacks are found and how long analysis takes; the executor and judge determine whether a proposal yields a violation. We describe the solver, the construction of install maps, and the conditions for search termination.

5.1 Backward solving

The solver builds a *symbolic state* for each run by replacing attacker-controlled slots with variables and reducing the run's computations over them. A used or forwarded slot is controllable if it arrives unguarded, or if its guarded sender received it through a controllable slot. Public-key slots initially become $\mathsf{PUBKEY}(x)$, preserving the key structure needed for man-in-the-middle attacks. A later pass replaces them with bare variables to search for type confusions in which a key slot accepts another kind of value.

Goals are derived from open queries and checks. A confidentiality goal seeks the queried term; authentication seeks acceptance without an emission; unlinkability seeks a secret argument; and equivalence seeks a value differing from the honest one. Checks supply constraints for successful evaluation. They also supply *oracle* goals: message shapes that another run could emit to perform a cryptographic operation for the attacker. In BAN-Yahalom, for example, a second responder run encrypts a chosen nonce field shaped like the first run's ticket [48].

To solve a goal, the solver checks whether the attacker can already obtain it, leaves a variable free, unifies it with a held or transmitted term, constructs it from solved arguments, or inverts a rewrite rule to find an input that a principal could transform into the goal. It also considers decomposition of transmitted terms, weakening annotations, reuse pairs and threshold combination. Unification uses the Diffie–Hellman equation and expands tuples to the width required by a projection; it does not implement general AC-unification. Bindings that exceed the slot's depth bound (§5.3) are rejected. Remaining variables are filled with `nil`, the attacker's public key $\mathsf{PUBKEY}(\texttt{nil})$, or a held honest value.

In the running example, Bob's slot `ga` becomes $\mathsf{PUBKEY}(x)$ and his key becomes $\mathsf{DH_KEX}(\mathsf{PUBKEY}(x), b) \equiv \mathsf{DH_KEX}(\texttt{gb}, x)$. The confidentiality goal requires this key. Grounding the controllable slots with `nil` gives $x = \texttt{nil}$, so the attacker can derive the key from `nil` and the observed `gb`.

5.2 From proposals to executions

A proposal becomes an install map by assigning terms to the proposing run's receives. For a guarded slot, the injection is placed at the upstream unguarded receives from which

the value is relayed. If a term requires information absent from the honest execution, the search adds *sources*: the installs of the cheapest earlier execution that revealed each required value in time. Each map is executed. Maps with stuck injections may be retried; the others are judged.

Execution outcomes guide several ways of deriving further maps. A *repair* adds bindings to satisfy a check where the proposing run halted. A *fill* supplies a run waiting for an unsent value with the honest value, `nil` or a sibling's value. A stuck map is retried with installs from a retained execution that freshly emits the missing value. A *merge* transfers a map to a session or scenario copy, or combines sources that reveal a value only when executed together. The *idle* pass leaves another sender unstarted or drops the value it needs last. Two later passes use bare public-key variables and sends made fresh by earlier injections as sources for stuck maps. Derived maps undergo the same execution and evaluation as initial proposals.

The search retains executions that add information at any phase: a previously unobtainable value, a principal's emission within the depth bound, a reuse pair, a fresh send, or a new derivation of a known value. It records the *union* of their knowledge by phase and uses this union to generate candidates. The executor still requires each injection to be derivable within the receiving execution. There is no limit on the number of retained executions, since discarding an alternative could exclude attacks that depend on it.

Each search round solves for every run, first using goals and then constructed candidates such as a sibling session's complete message. It then closes the union, merges query sources and retries stuck maps. Rounds continue until the union stops growing. For queries still open, a refined round preserves honest values in selected slots to enable rewrites, followed by one pass each of the idle, unshaped and rerouted searches.

5.3 Termination

Without a term-depth bound, search on the shipped symmetric-key Needham–Schroeder model can generate an unending sequence of injections. A run decrypts a message, hashes it, and encrypts the hash under the same key:

```
d  = AEAD_DEC(k, n, e, nil)
h  = HASH(d)
e2 = AEAD_ENC(k, n2, h, nil)
```

Injecting `e2` and its nonce at `e` and `n` produces $\mathsf{AEAD_ENC}(k, n_2, \mathsf{HASH}(\mathsf{HASH}(m)), \texttt{nil})$. Repeating this process adds another hash layer each time, producing terms absent from the honest protocol. Verifpal limits injections using a depth bound computed from the model before search begins.

Definition 9 (Installable terms)**.** Let $d(M)$ be the greatest nesting depth of any value the model assigns, with named computations expanded but not reduced, and let $\Pi(M)$ contain the subterms of these values and of their reductions. For a slot S of a principal P, let $\mathrm{peel}_P(S)$ be the number of layers P removes from S by a chain of rewrites or splits applied directly to it. A term t may be injected at S if $d(t) \le d(M)$, or if $d(t) \le d(M) + \mathrm{peel}_P(S)$ and t has depth at most $d(M)$ when every subterm equivalent to a member of $\Pi(M)$ counts as depth zero.

The bound allows invented structure up to the depth of the model's deepest computation. Replayed protocol terms may have additional wrapping, limited to the number of layers the recipient removes: decryption followed by splitting permits two extra layers, while hashing permits none. Derived maps reuse admitted terms, honest values and `nil`, keeping every executed install within the bound.

Theorem 2 (Termination)**.** *Every execution terminates. If every call of the solver terminates, the search terminates.*

Proof. An execution advances each run at most once per action and takes each early receive once; each phase ends when nothing can move, and closure terminates by Proposition 1. For the search, the installable terms at each slot form a finite set, since depth and arity are bounded and the constants and sharing tags are fixed before the search. There are finitely many slots and runs, so there are finitely many install maps, counting drops and idled runs, and the search executes each one a bounded number of times. Each execution has a finite closure universe, so the union grows monotonically inside a finite set, and a new round starts only after the union grows. The refined, idle, unshaped and rerouted passes run once each over finite lists. Fills only add installs, repairs only move a halt later, each stuck map is retried once with each source, and each query target is merged once per context, so every chain of derived maps ends. □

Solver termination. Termination of individual solver calls remains unproved. The solver cuts branches with repeated goals, memoizes solved goals, and binds each slot variable at most once per branch within the depth bound. Other recursive calls use strict subterms, reducts, or shapes drawn from transmitted terms, held terms and primitive rules. Although these sets of shapes are finite, they introduce an unbounded number of fresh variables. In the corpus evaluation, every analysis but one finished within its one-hour budget (§8.2).

Limitation 1 (Deep construction). An attack whose injected term nests invented structure deeper than anything the protocol computes, or nests protocol terms more layers than the recipient removes, cannot be found. When the bound cuts a proposal, every query still open at that moment passes with the label `[search truncated: term depth]` instead of `[search exhausted]`.

Complete message-size bounds are known for well-formed and type-compliant protocols [44, 45]; Verifpal assumes neither condition. In Arapinis and Duflot's ill-formed protocol Π_{Toy}, the pair $(\mathsf{PKE_ENC}(pk_B, n), pk_A)$ is encrypted under pk_B. The attacker places this ciphertext in the responder's nonce position, obtains the pair encrypted under its own key, and uses a second responder run to remove the remaining layer. This attack adds wrapping around protocol terms. The model has $d(M) = 10$ before reduction and four after reduction. Verifpal finds the attack at two sessions, including with the bound raised by up to four levels; ProVerif also finds it with two responder copies. One session cannot remove both layers. At that setting the query passes with a truncation label because the depth bound rejected a proposal.

5.4 Search coverage

The fixed constants and depth bound exclude some attacks. The attacker can construct $\mathsf{HASH}(\texttt{nil})$, for example, but cannot introduce a nonce distinct from every model constant. The proposal rules impose further restrictions, adopted after broader versions ran for hours on test models. Proposals use the knowledge union without its closure and, for each run, only information available by its last injectable phase. The idle pass drops one value per silenced sender. Goals are generated only for the queries present in the model, so adding or removing a query can affect whether an attack on another query is found.

6 Sessions and Peers

6.1 Sessions

An expansion to k sessions creates k copies of every principal and message. The copies share long-term values declared with `knows`; generated values and assignment outputs receive distinct names in each session, such as `c#2` in the second. All copies execute

together, allowing the attacker to transfer values between sessions. Any run may be left idle, so k is an upper bound on the number of runs per principal. The default is two sessions. The command line accepts one to sixteen, subject to loader limits of 128 expanded principals and 31 copies of any one principal.

Millen's FFGG protocol illustrates an attack requiring two sessions [47]. Bob reorders the fields of an encrypted tuple and reveals one in the clear. Passing his response through a second Bob run moves the secret into the revealed position. A single session cannot perform this attack, since Bob's only run would have to receive a ciphertext that it has yet to emit. Verifpal, ProVerif, Tamarin and Scyther all find the two-session attack (§8.4).

Limitation 2 (Sessions). An attack requiring more runs of a principal than the expansion provides cannot be found. Agreement at consecutive session counts does not establish security at larger counts. For example, a secret split into four shares, each recoverable from a different run, remains secret with three sessions but is disclosed with four.

6.2 Peer configurations

A `scenarios` block varies the peer assignment that sessions otherwise repeat. It lists alternative bindings for a principal's `knows` values, such as Alice's intended peer key. Each entry creates a copy of the whole model with that binding, named with suffixes such as `Alice@2` and `c@2`. With n scenarios and k sessions, each principal has nk runs. Analysis covers the listed configurations only.

Verifpal dates a configuration's compromise using the same knowledge rules as attack analysis. It first executes the model honestly with all scenarios at one session. The configuration becomes corrupt at the end of the first phase in which the attacker can obtain a bound private value, or the secret inputs of a bound computed value, under the weakening annotations then in force. Keys disclosed on the wire or recovered from ciphertexts count; public keys and salts do not. Corrupt runs continue to send messages and contribute to attacker knowledge. Section 4 specifies which runs answer queries and which phase of their state is used.

Lowe's attack. In public-key Needham–Schroeder [51, 29], Alice starts a run with Mallory, whose private key is leaked in the model. The attacker decrypts Alice's first message and re-encrypts its contents for Bob. Bob's reply is replayed to Alice's run with Mallory. Alice returns Bob's nonce encrypted for Mallory, disclosing it to the attacker. With two scenarios, Alice bound to Bob and Alice bound to Mallory, Verifpal finds the attack at one session. Lowe's repair adds the responder's identity to the reply [30]. Alice checks this identity against her intended peer and rejects Bob's reply in her run with Mallory; the secrecy query then passes. Increasing the session count alone cannot expose the original attack because session copies preserve peer bindings.

7 Explaining an Attack

The first execution found to violate a query may contain irrelevant installs, including ones proposed for other queries. The engine removes these where possible before generating the report.

Definition 10 (Reduced witness)**.** Let $\mathsf{exec}(I)$ violate q at phase p. The reducer repeatedly removes one entry of I, or two entries at the same receive of one run, and keeps the removal when the smaller map's execution has no stuck injection and still violates q at p. Removing an idle entry lets that run start; removing a drop delivers the value. It stops when no removal applies.

The resulting map is locally minimal under these removals. Trying pairs matters when a recipient checks two fields together, such as a forged message and its signature. Each candidate is executed and judged again, so the reduced execution has itself been checked for the reported violation.

The narrator processes the reduced execution in order through the end of the failing phase. Before each injected receive, it explains how the attacker obtained the term from earlier messages, leaks or recorded derivations. An injection is labeled a *replay* "from another session", "scenario" or "run" if it equals a value sent earlier on a copy of the slot or by another copy of the sender. Other injections are labeled *replacements*, with the honest value noted. The report also lists idle runs, dropped values and checks that succeed on attacker-influenced inputs. Values use the names of the slots holding them (Figure 3); for guarded deliveries, the report locates the injection at the upstream unguarded receive.

8 Evaluation

We compare secrecy judgments with ProVerif, replay reported attacks separately, and measure session and depth costs. Measurements use Verifpal 1.6.7, ProVerif 2.05, Tamarin 1.12.0 and Scyther `v1.3.0-7-gc7af6b6`, on an Apple M5 with ten cores and 16 GiB. Runs are sequential; runtimes are single wall-clock observations. The artifact contains all models, commands and outputs.

8.1 Differential analysis of Noise

Noise [8] defines a family of Diffie–Hellman protocols with different authentication and forward-secrecy guarantees. We generate matched Verifpal and ProVerif models of all three one-way and twelve fundamental interactive patterns, with the engine frozen before generation. Noise Explorer has already analyzed these patterns [9]; we use them to test Verifpal's judgments.

Models and compromise. For every static private key present in a pattern, we consider no disclosure, disclosure before communication, and disclosure in a later phase, after communication has stopped. Their Cartesian product gives 91 configurations. A separate process performs disclosures, so an aborted handshake cannot suppress a scheduled compromise. Ephemeral keys are fresh per session. The peers have fixed intended identities, with expected public keys distributed over authenticated public channels; a received static key must equal the expected key. The queries ask whether the attacker can derive each handshake payload and three subsequent transport payloads, ordered responder–initiator–responder. One-way patterns have one additional initiator payload. Payload constants are private and shared across sessions; some patterns intentionally send early payloads in cleartext. Including both responder transport messages distinguishes exposure before and after initiator key confirmation.

Controlled omissions. We separately remove received-static identity checks, transcript hashes from handshake AEAD associated data, all static-involving DH contributions, or the ephemeral–ephemeral DH contribution. Each applicable omission is evaluated without compromise and with all present static keys disclosed late, giving 98 further configurations. These are deliberately modified protocols. Together, the 189 configurations contain 885 secrecy queries. Pattern selection, compromise schedules, omissions and resource budgets were recorded before the main measurements; every configuration is retained regardless of its outcome.

Encoding checks. A common representation tracks chaining keys, transcript hashes, ordered DH operations, two-output key derivations, nonce resets, checked decryption and

directional transport keys. Honest symbolic execution succeeds in every configuration. Executing the representation with X25519, ChaCha20-Poly1305 and SHA-256 matches all 90 messages in published Cacophony vectors distributed with noise-c [10]: 30 handshake and 60 transport messages across the fifteen patterns. This checks honest computations and message construction, without proving the symbolic translations correct.

Comparison and replay. Each configuration runs at one and two sessions in both tools, and with unbounded replication in ProVerif: 945 processes in total, each limited to 60 seconds and 3 GiB. The ProVerif theory uses DH commutativity, distinct free KDF outputs and perfect AEAD. A derivation that ProVerif cannot reconstruct as an execution remains inconclusive. Only completed processes with all queries parsed enter the verdict comparison; partial results before a resource limit remain in the artifact.

A separate Python executor and deduction checker replays Verifpal's reported replacements. It shares the protocol representation, but no executor, solver or deduction code, with Verifpal. It checks each replacement against the attacker's knowledge at that delivery, executes decryption and identity checks, and requires the queried payload to become derivable. It ignores the report's derivation prose and assertions that checks passed. Its perfect-AEAD theory omits Verifpal's nonce-reuse rules. Tests include attempted use of a future disclosure, unauthorized key substitution and a failed decryption followed by a secret send.

Results. Both session bounds give the same Verifpal verdicts: 290 query positions pass and 595 report disclosure. ProVerif proves all 290 passing counterparts with unbounded sessions. The separate checker validates all 1,190 disclosure reports, including 722 that require network replacements in the replay. No report is rejected or unsupported.

Of 1,770 bounded query pairs, 1,625 receive definite ProVerif answers, all agreeing with Verifpal (Table 3). ProVerif is inconclusive on 135 pairs; two IK runs with two sessions time out, excluding ten queries. Every other process completes. With unbounded sessions, ProVerif reconstructs 253 disclosures and is inconclusive on 342, in addition to the 290 proofs. Table 5 gives the results by pattern.

Table 3: Noise query pairs at one and two sessions. A Verifpal pass records bounded, incomplete search; a ProVerif proof applies to its corresponding model. All queries of a timed-out process are excluded from agreement counts.

ProVerif outcome	Verifpal pass	Verifpal disclosure	Total
Proof	578	0	578
Reconstructed attack	0	1,047	1,047
Inconclusive	0	135	135
Process timed out	2	8	10
Total	580	1,190	1,770

At two sessions, removing static DH contributions exposes 70 previously passing query positions; removing received-static identity checks exposes 36, and removing ephemeral–ephemeral DH exposes 26. Removing transcript binding changes none of its 133 queried positions. The secrecy queries therefore detect several omitted protections, but do not establish channel binding.

For example, an attacker starts a KX responder run with its own ephemeral key. Later disclosure of the initiator's static key exposes the responder's handshake payload and first transport payload. The last responder payload follows decryption of an initiator transport message, which the attacker cannot supply before disclosure. Both tools distinguish the exposures before key confirmation from the subsequent protected payloads; ProVerif proves the latter queries with unbounded replication.

Scope. The experiment covers payload secrecy for fixed peers under the stated compromises. It does not test identity hiding, arbitrary peer populations, PSKs, deferred or fallback patterns, or computational security. NK and XX already occur in the engine's regression models. The two encodings and replay checker share a generator, so a shared modeling error remains possible despite the external vectors. Counts describe this finite grid of related configurations, not independent statistical samples. A Verifpal pass remains bounded, incomplete search.

8.2 Corpus and query interpretation

Of the 1,018 shipped models, 43 describe deployed or published protocols and the running example; the rest are teaching examples, flawed designs and regressions. We analyze 1,015 at two sessions, excluding TLS 1.3 with and without 0-RTT and twelve-message Signal. Under a one-hour, 6 GiB budget, 977 analyses complete, 37 loader-error tests are rejected as intended, and AWS Nitro times out. Completed analyses report 980 violations and 700 passes: 697 exhausted and three truncated. Full counts and the earlier sampled audit remain in the artifact.

The 408 authentication violations include 139 duplicate acceptances, 133 without recipient-generated context, confirmed by a separate source dependency check. Of 571 completed models using guards, 63 compute a guarded value from an unguarded input. Seven semantic probes and the engine's 1,441 tests pass, including checks for result preservation or monotonicity under renaming, guard removal, disclosures and increased session counts.

Classical protocol models. An earlier benchmark covers sixteen classical protocols [48], with the engine frozen and expectations recorded before execution (Table 6). Ten documented attacks are found; three yield weaker use-based violations, BAN-Yahalom needs unsupported associative concatenation, and two require unavailable identity-agreement predicates. Of thirteen repairs, six pass at both session counts, one times out at two, one has an overly strong query, and five violate Verifpal's predicates without contradicting published claims. A recipient may use a value or hold a key before a later check rejects the run. These mismatches remain recorded as such.

8.3 Electronic passports

The fifteen shipped models of ICAO Doc 9303 electronic passport protocols [11] each finish within 16 seconds. They cover BAC and PACE with Secure Messaging, authentication, tracking and remote identity checks. Disclosure of the machine-readable zone (MRZ) decrypts recorded BAC data and links sightings through MACs under a shared static key; the corresponding PACE queries pass [12]. Attacks that distinguish failed checks, such as Chothia and Smirnov's traceability attack, remain outside the value-based unlinkability predicate [13, 14].

In the remote identity check, a dishonest verifier presents a bank's challenge to the holder's phone as its own and forwards the chip's signature to the bank. Binding the signature to the service's public key makes the query pass at one and two sessions, analogous to WebAuthn's origin binding [15]. The repaired model also exercises the AEAD reuse rules: two bank runs encrypt different challenges under one key and nonce, revealing the tag key but neither challenge, so no forgery follows.

8.4 Comparison with ProVerif and Tamarin

We compare FFGG, unauthenticated Diffie–Hellman, symmetric-key Needham–Schroeder with later compromise, and public-key Needham–Schroeder with and without Lowe's re-

pair [47, 49, 51, 30]. ProVerif's FFGG model comes from its shipped `ffgg.ml` generator; the other models were written for this comparison. The public-key Tamarin models separate honest-peer claims and use a hand-written `sources` lemma based on NSLPK3 [36]. Plain rewrite rules avoid the process translation affected by issue #936 [32]. All runs finish within six seconds.

Of 14 query outcomes in Table 7, eleven agree across the applicable tools under honest-peer claims. Two involve later symmetric-key compromise, which Scyther's released model omits; ProVerif is inconclusive on one. The remaining difference concerns Lowe's repair: Verifpal reports Bob's use of a message constructed from Alice's run with Mallory. Other tools prove agreement at completion, which Bob never reaches because Alice rejects his reply. Appendix A.4 details the properties.

Scyther. Scyther [56, 57] supports unbounded proofs without Diffie–Hellman. We compare server-free public-key Needham–Schroeder and Lowe's repair, symmetric-key Needham–Schroeder, Otway–Rees, Wide-Mouth Frog and Yahalom. The last four use released Scyther models. Its runs have a 20-second watchdog and bound role instances. Both tools distinguish Needham–Schroeder from Lowe's repair on responder-nonce secrecy; authentication differs because Verifpal checks use and replay while Scyther checks non-injective agreement. Scyther proves all eight repaired-protocol claims for unbounded sessions. Detailed timings remain in the artifact.

8.5 Cost of sessions and of the depth bound

The effect of session count varies by model. FFGG and symmetric-key Needham–Schroeder remain under 0.11 seconds through four sessions. The three-message Signal model takes 0.13, 2.5, 28 and 338 seconds at one through four sessions, a twelvefold increase for the fourth session. Peer scenarios further increase the number of runs to analyze.

Sensitivity to the depth bound. A benchmark-only patch raises $d(M)$ and each per-slot allowance by one or two levels. Neither change affects results on the 977 analyzable models, under a 600-second budget and 3 GiB memory limit; AWS Nitro reaches the limit at each setting. These observations establish stability on this corpus, without establishing completeness of the bound.

9 Limitations

Cryptography. Verifpal supports only the primitives and single equation in its registry. It does not represent XOR, pairings, small subgroups, invalid curves, concrete encodings or nested Diffie–Hellman terms such as g^{abc}. The KEM is a deterministic function of a public key and seed, without implicit rejection or the binding properties needed to analyze re-encapsulation [53]. AEAD nonce reuse is modeled coarsely (§3.2). Ideal signatures bind both key and message, an assumption that concrete schemes need not satisfy [33]. The loader cannot detect when a protocol relies on unsupported cryptographic structure. The language is untyped, so it may report type confusions excluded by an implementation's tags or length checks. Domain separation must be represented explicitly.

Search. Passing queries are limited by the session count (Limitation 2), term depth (Limitation 1), fixed constants and proposal rules (§5.4). For each install map, the engine judges the maximal execution at phase barriers. The attacker can idle a run or drop a received value, but cannot stop it before an action that requires no withholdable input.

Limitation 3 (One execution per install map). A violation that exists only in a shorter prefix of a phase, such as an acceptance counted before a matching send of the same phase that the attacker cannot prevent, is not judged.

Properties. The queries have the meanings in Definition 6. Authentication can count a message as accepted when it is used, even if a later check rejects it. This produces violations on some published repairs without contradicting their claimed properties (§8.2). Freshness detects replay through acceptance by two runs of the same agent. The `equivalence?` and `unlinkability?` predicates test values within one execution and cannot establish indistinguishability between executions.

Trust. The proofs are not mechanized. Correctness of the executor, closure, judge and primitive table remains an implementation assumption. The separate Noise checker covers only its theory and confidentiality witnesses.

10 Related Work

Unbounded verifiers. ProVerif translates processes into Horn clauses. This abstraction supports proofs for unbounded sessions but can combine steps that cannot occur together; a derivation that cannot be reconstructed as a trace yields `cannot be proved` [16, 17, 19]. Tamarin solves constraints over executions of multiset rewriting rules [34, 35]. Lemmas, induction and interactive guidance help resolve difficult cases in both tools [36, 37, 18]. Both also check behavioral equivalence [20, 21, 38]. SAPIC+ provides translations from a common process language to ProVerif, Tamarin and DeepSec, with pen-and-paper soundness proofs [24]. Those proofs, like ours, do not establish correctness of the implementation; issue #936 concerns the translation code [32]. Scyther distinguishes bounded results from proofs [56, 57], while CPSA enumerates minimal execution shapes [26, 25].

Bounded and backward search. Complete procedures exist for bounded sessions in several term theories [43, 46]. Constraint solving for bounded processes [40], OFMC's lazy intruder [41], Athena [39] and Maude-NPA [42] represent attacker inputs as variables, often searching backward from a violation. Verifpal uses this approach with a fixed, incomplete set of proposal rules, then checks each candidate by executing the whole protocol. The separation resembles certifying algorithms [27]. The Noise experiment checks confidentiality witnesses separately, but Verifpal has no general certificate format. Results for well-formed and type-compliant protocols bound attack-message size [44, 45]; Verifpal assumes neither condition.

Properties. Lowe classifies authentication properties by their agreement requirements [28]. Verifpal places acceptance at use, which may precede protocol completion. Its phases follow symbolic treatments of compromise [55, 54]. AKISS and DeepSec decide trace equivalence for bounded sessions [22, 23]; Verifpal's `equivalence?` and `unlinkability?` queries test values within an execution.

11 Conclusion

Verifpal 1.6.7 retains the original modeling language while giving execution and query results a formal semantics. Proposed attacks are checked by executing the protocol with attacker knowledge derived within that execution. The soundness theorem relates these executions to the reference semantics, subject to correctness of the implementation. Search remains bounded and incomplete, and passing queries report its outcome within those limits.

Use of generative AI. Claude Opus 5.5 (Anthropic) ran the original benchmarks, wrote the classical protocol models, and performed the initial sampled audit, subsequently repeated by the author. Codex (OpenAI) designed and ran the Noise evaluation, including its generator, replay checker and results discussion. The author takes responsibility for all content.

Acknowledgments

I thank Bruno Blanchet, Vincent Cheval, and the ProVerif team; Mario Raso for his review of authentication and replay semantics; Heba Ajjour (University College of Applied Sciences, Gaza, Palestine) for her analysis of link witnesses and for reporting scenario and unlinkability defects; and my applied cryptography students at the American University of Beirut.

References

[1] N. Kobeissi, G. Nicolas, M. Tiwari. *Verifpal: Cryptographic Protocol Analysis for the Real World.* INDOCRYPT 2020, LNCS 12578, Springer, pp. 151–202. IACR ePrint 2019/971.

[2] N. Kobeissi. *Verifpal Documentation.* Symbolic Software, 2026. https://verifpal.com/docs/

[3] Symbolic Software. *Verifpal: Comparison.* 2026. https://verifpal.com/docs/comparison/

[4] N. A. Durgin, P. D. Lincoln, J. C. Mitchell, A. Scedrov. *Multiset Rewriting and the Complexity of Bounded Security Protocols.* Journal of Computer Security, 12(2):247–311, 2004. https://doi.org/10.3233/JCS-2004-12203

[5] C. Cremers, M. Horvat, J. Hoyland, S. Scott, T. van der Merwe. *A Comprehensive Symbolic Analysis of TLS 1.3.* ACM Conference on Computer and Communications Security (CCS), 2017, pp. 1773–1788.

[6] D. Basin, J. Dreier, L. Hirschi, S. Radomirović, R. Sasse, V. Stettler. *A Formal Analysis of 5G Authentication.* ACM Conference on Computer and Communications Security (CCS), 2018, pp. 1383–1396.

[7] C. Komlo, I. Goldberg. *FROST: Flexible Round-Optimized Schnorr Threshold Signatures.* Selected Areas in Cryptography (SAC 2020), LNCS 12804, Springer, 2021, pp. 34–65.

[8] T. Perrin. *The Noise Protocol Framework.* Revision 34, 11 July 2018. https://noiseprotocol.org/noise.html

[9] N. Kobeissi, G. Nicolas, K. Bhargavan. *Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols.* IEEE European Symposium on Security and Privacy, 2019, pp. 356–370. IACR ePrint 2018/766.

[10] Cacophony contributors. *Noise test vectors*, distributed with noise-c at commit `cfe2541`, file `tests/vector/cacophony.txt`. https://github.com/rweather/noise-c/tree/cfe25410979a87391bb9ac8d4d4bef64e9f268c6/tests/vector

[11] International Civil Aviation Organization. *Doc 9303: Machine Readable Travel Documents, Part 11: Security Mechanisms for MRTDs.* Eighth edition, 2021.

[12] A. Juels, D. Molnar, D. Wagner. *Security and Privacy Issues in E-passports.* SecureComm 2005, IEEE, pp. 74–88.

[13] T. Chothia, V. Smirnov. *A Traceability Attack against e-Passports.* Financial Cryptography and Data Security (FC 2010), LNCS 6052, Springer, pp. 20–34.

[14] M. Arapinis, T. Chothia, E. Ritter, M. Ryan. *Analysing Unlinkability and Anonymity Using the Applied Pi Calculus.* IEEE Computer Security Foundations Symposium (CSF), 2010, pp. 107–121.

[15] W3C. *Web Authentication: An API for Accessing Public Key Credentials, Level 2.* W3C Recommendation, 2021.

[16] B. Blanchet. *Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif.* Foundations of Security Analysis and Design VII (FOSAD 2012/2013 Tutorial Lectures), LNCS 8604, Springer, 2014, pp. 54–87.

[17] B. Blanchet. *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif.* Foundations and Trends in Privacy and Security, 1(1–2):1–135, 2016.

[18] B. Blanchet, V. Cheval, V. Cortier. *ProVerif with Lemmas, Induction, Fast Subsumption, and Much More.* IEEE Symposium on Security and Privacy, 2022, pp. 69–86.

[19] ProVerif authors. *Use ProVerif.* https://bblanche.gitlabpages.inria.fr/proverif/usage.html

[20] B. Blanchet, M. Abadi, C. Fournet. *Automated Verification of Selected Equivalences for Security Protocols.* Journal of Logic and Algebraic Programming, 75(1):3–51, 2008.

[21] V. Cheval, B. Blanchet. *Proving More Observational Equivalences with ProVerif.* POST 2013, LNCS 7796, Springer, pp. 226–246.

[22] R. Chadha, V. Cheval, Ş. Ciobâcă, S. Kremer. *Automated Verification of Equivalence Properties of Cryptographic Protocols.* ACM Transactions on Computational Logic, 17(4):1–32, 2016. https://doi.org/10.1145/2926715

[23] V. Cheval, S. Kremer, I. Rakotonirina. *DEEPSEC: Deciding Equivalence Properties in Security Protocols—Theory and Practice.* IEEE Symposium on Security and Privacy, 2018, pp. 529–546. https://doi.org/10.1109/SP.2018.00033

[24] V. Cheval, C. Jacomme, S. Kremer, R. Künnemann. *SAPIC+: Protocol Verifiers of the World, Unite!* USENIX Security Symposium, 2022, pp. 3935–3952.

[25] S. F. Doghmi, J. D. Guttman, F. J. Thayer. *Searching for Shapes in Cryptographic Protocols.* Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 4424, Springer, 2007, pp. 523–537.

[26] M. D. Liskov, J. D. Guttman, J. D. Ramsdell, P. D. Rowe, F. J. Thayer. *Enrich-by-Need Protocol Analysis for Diffie–Hellman.* Foundations of Security, Protocols, and Equational Reasoning, LNCS 11565, Springer, 2019, pp. 135–155.

[27] R. M. McConnell, K. Mehlhorn, S. Näher, P. Schweitzer. *Certifying Algorithms.* Computer Science Review, 5(2):119–161, 2011.

[28] G. Lowe. *A Hierarchy of Authentication Specifications.* IEEE Computer Security Foundations Workshop (CSFW), 1997, pp. 31–43.

[29] G. Lowe. *An Attack on the Needham–Schroeder Public-Key Authentication Protocol.* Information Processing Letters, 56(3):131–133, 1995.

[30] G. Lowe. *Breaking and Fixing the Needham–Schroeder Public-Key Protocol Using FDR.* Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 1055, Springer, 1996, pp. 147–166.

[31] K. Milner. *Soundness/input validation/general correctness fixes for regular proving, for diff mode, and for SAPIC.* Tamarin prover pull requests #948–#950, 17 September 2026, being resubmitted in parts from #954 (merged 7 October 2026). https://github.com/tamarin-prover/tamarin-prover/pull/948

[32] S. Hammann. *SAPIC: builtin equation variables collide with renamed process variables in destructor let; decryption rule can never fire (false "verified").* Tamarin prover issue #936, 15 September 2026. https://github.com/tamarin-prover/tamarin-prover/issues/936

[33] D. Jackson, C. Cremers, K. Cohn-Gordon, R. Sasse. *Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures.* ACM Conference on Computer and Communications Security (CCS), 2019, pp. 2165–2180.

[34] B. Schmidt, S. Meier, C. Cremers, D. Basin. *Automated Analysis of Diffie–Hellman Protocols and Advanced Security Properties.* IEEE Computer Security Foundations Symposium (CSF), 2012, pp. 78–94.

[35] S. Meier, B. Schmidt, C. Cremers, D. Basin. *The TAMARIN Prover for the Symbolic Analysis of Security Protocols.* Computer Aided Verification (CAV), LNCS 8044, Springer, 2013, pp. 696–701.

[36] The Tamarin Team. *Tamarin Prover Manual.* https://tamarin-prover.com/manual/

[37] V. Cortier, S. Delaune, J. Dreier, E. Klein. *Automatic Generation of Sources Lemmas in TAMARIN: Towards Automatic Proofs of Security Protocols.* Journal of Computer Security, 30(4):573–598, 2022. https://doi.org/10.3233/JCS-210053

[38] D. Basin, J. Dreier, R. Sasse. *Automated Symbolic Proofs of Observational Equivalence.* ACM CCS 2015, pp. 1144–1155.

[39] D. X. Song. *Athena: A New Efficient Automatic Checker for Security Protocol Analysis.* IEEE Computer Security Foundations Workshop (CSFW), 1999, pp. 192–202.

[40] J. Millen, V. Shmatikov. *Constraint Solving for Bounded-Process Cryptographic Protocol Analysis.* ACM CCS 2001, pp. 166–175.

[41] D. Basin, S. Mödersheim, L. Viganò. *OFMC: A Symbolic Model Checker for Security Protocols.* International Journal of Information Security, 4(3):181–208, 2005.

[42] S. Escobar, C. Meadows, J. Meseguer. *Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties.* Foundations of Security Analysis and Design V (FOSAD 2007/2008/2009 Tutorial Lectures), LNCS 5705, Springer, 2009, pp. 1–50.

[43] M. Rusinowitch, M. Turuani. *Protocol Insecurity with a Finite Number of Sessions and Composed Keys Is NP-Complete.* Theoretical Computer Science, 299(1–3):451–475, 2003.

[44] M. Arapinis, M. Duflot. *Bounding Messages for Free in Security Protocols.* Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 4855, Springer, 2007, pp. 376–387. https://doi.org/10.1007/978-3-540-77050-3_31

[45] R. Chrétien, V. Cortier, A. Dallon, S. Delaune. *Typing Messages for Free in Security Protocols.* ACM Transactions on Computational Logic, 21(1):1–52, 2020. https://doi.org/10.1145/3343507

[46] Y. Chevalier, R. Küsters, M. Rusinowitch, M. Turuani. *Deciding the Security of Protocols with Diffie–Hellman Exponentiation and Products in Exponents.* Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 2914, Springer, 2003, pp. 124–135.

[47] J. K. Millen. *A Necessarily Parallel Attack.* Workshop on Formal Methods and Security Protocols (FMSP'99), Trento, Italy, July 1999.

[48] J. Clark, J. Jacob. *A Survey of Authentication Protocol Literature: Version 1.0.* Technical report, University of York, 1997.

[49] D. E. Denning, G. M. Sacco. *Timestamps in Key Distribution Protocols.* Communications of the ACM, 24(8):533–536, 1981.

[50] M. Abadi, R. Needham. *Prudent Engineering Practice for Cryptographic Protocols.* IEEE Transactions on Software Engineering, 22(1):6–15, 1996.

[51] R. M. Needham, M. D. Schroeder. *Using Encryption for Authentication in Large Networks of Computers.* Communications of the ACM, 21(12):993–999, 1978.

[52] D. Dolev, A. Yao. *On the Security of Public Key Protocols.* IEEE Transactions on Information Theory, 29(2):198–208, 1983.

[53] C. Cremers, A. Dax, N. Medinger. *Keeping Up with the KEMs: Stronger Security Notions for KEMs and Automated Analysis of KEM-Based Protocols.* ACM CCS 2024, pp. 1046–1060. IACR ePrint 2023/1933.

[54] K. Cohn-Gordon, C. Cremers, L. Garratt. *On Post-Compromise Security.* IEEE Computer Security Foundations Symposium (CSF), 2016, pp. 164–178.

[55] D. Basin, C. Cremers. *Modeling and Analyzing Security in the Presence of Compromising Adversaries.* ESORICS 2010, LNCS 6345, Springer, pp. 340–356.

[56] C. Cremers. *The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols.* Computer Aided Verification (CAV), LNCS 5123, Springer, 2008, pp. 414–418.

[57] C. Cremers. *Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement.* ACM CCS 2008, pp. 119–128.

A Supplementary material

This appendix provides the primitive registry, supplementary evaluation tables, query correspondences and an AI-use record.

A.1 Primitive registry

Table 4 lists Verifpal 1.6.7's 25 built-in primitives, with arguments numbered from zero. "Holding" specifies what the attacker learns from a held term; Definition 2 specifies which terms it can construct. For a rewrite marked *check*, a pattern mismatch halts a checked assignment.

Table 4: The primitive registry of Verifpal 1.6.7.

Primitive(s)	**Arity → outputs**	**Rewriting and knowledge**	**Restrictions and annotations**
ASSERT	$2 \to 1$	Check succeeds exactly when its arguments are equivalent. Its value is the application itself, which the attacker can build from obtainable arguments.	Core check.
CONCAT, SPLIT	2–5 → 1; 1 → 1–5	Holding a concatenation reveals every field; $\mathsf{SPLIT}^{(i)}(\mathsf{CONCAT}(t_1, \ldots, t_n)) \to t_i$ when output i exists (check). An unchecked SPLIT that fails is an ordinary value.	Core structural primitives.
HASH	1–5 → 1	Free constructor.	weak reveals every argument.
HKDF	$3 \to 1$–5	Free, distinct outputs; no output reveals another or an input.	—
AEAD_ENC, AEAD_DEC	$4 \to 1$; $4 \to 1$	$\mathsf{AEAD_DEC}(k, n, \mathsf{AEAD_ENC}(k, n, m, a), a) \to m$ (check). Holding ENC with k and n reveals m. Two held, inequivalent ENC terms under one k and n form a reuse pair: either can be re-tagged under any obtainable a'; a member's m is revealed once another member's plaintext is obtainable; once some member's plaintext is held, any $\mathsf{AEAD_ENC}(k, n, m', a')$ is constructible, except a member of the pair.	ENC: weak reveals m; forgeable exempts key position 0. DEC: checking key at position 0.
ENC, DEC	2 → 1; 2 → 1	$\mathsf{DEC}(k, \mathsf{ENC}(k, m)) \to m$. Holding either form with k reveals its second argument.	ENC: weak reveals m; malleable varies position 1. DEC: not a check.
MAC	$2 \to 1$	Free constructor.	forgeable exempts key position 0.
PUBKEY, DH_KEX	1 → 1; 2 → 1	$\mathsf{DH_KEX}(\mathsf{PUBKEY}(a), b) \equiv \mathsf{DH_KEX}(\mathsf{PUBKEY}(b), a)$.	PUBKEY's argument forbids PUBKEY and DH_KEX; weak reveals it. DH positions 0/1 forbid DH / {PUBKEY, DH}.
SIGN, SIGNVERIF	2 → 1; 3 → 1	$\mathsf{SIGNVERIF}(\mathsf{PUBKEY}(sk), m, \mathsf{SIGN}(sk, m)) \to \mathtt{nil}$ (check).	SIGN: forgeable exempts position 0. VERIF identifies the signer key.
PKE_ENC, PKE_DEC	2 → 1; 2 → 1	$\mathsf{PKE_DEC}(sk, \mathsf{PKE_ENC}(\mathsf{PUBKEY}(sk), m)) \to m$. Holding PKE_ENC and sk reveals m.	ENC: weak reveals m. DEC: not a check.
THRESHOLD_SPLIT, THRESHOLD_JOIN	1 → 2–16; 2–16 → 1	$\mathsf{THRESHOLD_SPLIT}_\tau$ shares a secret; τ distinct held shares of one sharing reveal it. A share is obtainable only from τ other shares, or from the secret and $\tau - 1$ other shares. JOIN over τ distinct shares rewrites to the secret; over τ partials $\mathsf{THRESHOLD_SIGN}(s_i, n_i, c, m)$ agreeing on c and m, with each $\mathsf{PUBKEY}(n_i)$ in c, to $\mathsf{SIGN}(k, m)$; over τ values $\mathsf{PUBKEY}(s_i)$, to $\mathsf{PUBKEY}(k)$. The attacker may also join partials it makes itself.	The threshold $2 \leq \tau \leq$ outputs and the sharing's identity are part of the term. JOIN is not a check.
THRESHOLD_SIGN	$4 \to 1$	Free constructor over a share, a nonce, commitments and a message. Holding a partial with its nonce, commitments and message reveals the share, as do two partials under one share and nonce; a term the attacker could mint counts only as described in §3.2.	forgeable exempts the share.

Table 4 (continued)

Primitive(s)	Arity → outputs	Rewriting and knowledge	Restrictions and annotations
RINGSIGN, RINGSIGNVERIF	$4 \to 1$; $5 \to 1$	Verification yields nil (check) when the message matches and the three listed public keys are a permutation of the signer's and the two other ring keys.	SIGN: forgeable exempts the signer key. VERIF identifies no signer.
BLIND, UNBLIND	$2 \to 1$; $3 \to 1$	Holding $\mathsf{BLIND}(r, m)$ and r reveals m; $\mathsf{UNBLIND}(r, m, \mathsf{SIGN}(sk, \mathsf{BLIND}(r, m))) \to \mathsf{SIGN}(sk, m)$.	—
KEM_ENCAP, KEM_DECAP	$2 \to 2$; $2 \to 1$	$\mathsf{KEM_ENCAP}^{(1)}(pk, r)$ is the shared secret and $\mathsf{KEM_ENCAP}^{(2)}(pk, r)$ the ciphertext. Holding output 2 and the private key reveals output 1 and r; KEM_DECAP rewrites to output 1 (check).	ENCAP: key and seed restrictions; weak reveals output 1. DECAP: checking key at position 0.

A.2 Noise results by pattern

Table 5 breaks down the population and outcomes of §8.1. Each row includes the stated compromise schedules and applicable omissions. The artifact's `noise-queries.csv` records every matched query outcome, and `noise-summary.json` contains these aggregates.

Table 5: Noise outcomes by pattern. Verifpal's verdicts are identical at one and two sessions; each configuration–payload position is counted once here. Every passing counterpart has an unbounded ProVerif proof, and every Verifpal disclosure is replayed separately at both bounds. ProVerif's attack column requires a reconstructed execution.

			Verifpal	ProVerif, unbounded		
Pattern	**Configs.**	**Queries**	**Disclosures**	**Proofs**	**Attacks**	**Inconcl.**
N	7	14	10	4	10	0
K	13	26	18	8	0	18
X	15	30	20	10	0	20
NN	3	15	15	0	15	0
NK	9	45	37	8	16	21
NX	11	55	50	5	30	20
KN	9	45	34	11	26	8
KK	15	75	36	39	10	26
KX	17	85	53	32	16	37
XN	11	66	56	10	42	14
XK	17	102	56	46	5	51
XX	17	102	66	36	15	51
IN	11	55	44	11	44	0
IK	17	85	43	42	10	33
IX	17	85	57	28	14	43
Total	189	885	595	290	253	342

A.3 Held-out recall benchmark

Table 6 lists the models evaluated in §8.2; the artifact includes their preregistered expectations. *Found* denotes the documented attack at the session count shown. *Other* denotes a query violation through a different, weaker execution. The flawed Diffie–Hellman model uses the textbook protocol from the running example, which its writer consulted for syntax.

Table 6: Held-out recall models. Reasons for failures of repaired models: *use*, a recipient uses a replayed message before a later check rejects it; *held key*, a run holds a key before a later check halts it; *short split*, a short `SPLIT` of a longer tuple returns its first fields; *query too strong*, the query asks for agreement the protocol does not promise; *replay*, a replay the abstracted timestamp would prevent.

Protocol	Documented attack	Flawed model	Repaired model
Needham–Schroeder PK	man in the middle with a dishonest peer (Lowe)	found, 1 session	holds (Lowe's fix)
Denning–Sacco PK	signed key re-encrypted for a third party (Abadi–Needham)	found, 1	fails: held key; replay
SPLICE/AS	re-signed request (Hwang–Chen)	found, 1	holds
Diffie–Hellman	man in the middle	found, 1	holds (signed keys)
Needham–Schroeder SK	replay of a compromised key (Denning–Sacco)	found, 1	fails: use; held key
Otway–Rees	type flaw: name tuple taken as key	found, 1	fails: query too strong
Neuman–Stubblebine	type flaw: nonce taken as key (Hwang et al.)	found, 1	fails: held key
Wide-Mouth Frog	replayed server message (Lowe)	found, 2	—
ISO/IEC 9798-2 3-pass	reflection	found, 2	holds (identities)
MAC challenge–response	reflection	found, 2	holds (identities)
Andrew Secure RPC	message 2 replayed as message 4	other	holds (BAN's fix)
Kao–Chow v1	replay of a compromised key	other	holds, 1 session; no verdict at 2
Woo–Lam Π	interleaved runs (Abadi–Needham)	other; attack not found	fails: use
BAN-Yahalom	associativity of concatenation (Syverson)	missed	fails: short split
CCITT X.509	re-signed encrypted data	not expressible	—
Station-to-Station	unknown key share	not expressible	—

A.4 Property correspondence across tools

Table 8 gives the query correspondence for §8.4. All encodings define secrecy through attacker derivability and restrict claims to honest-peer runs. ProVerif uses an event monitor for the nonce, Tamarin uses separate honest-initiator facts, and Verifpal and Scyther restrict the queried run directly. In every tool, the attacker can still interact with corrupt-peer runs.

Authentication results depend on the chosen acceptance point. Verifpal checks a value's origin at successful use and compares the number of accepting runs with the number of sending runs. Correspondence queries relate modeler-defined events, usually at protocol completion. Neither predicate generally implies the other.

Needham–Schroeder–Lowe. At one session with two peer scenarios, Alice runs with both Bob and Mallory. Using Mallory's leaked key, the attacker decrypts Alice's message, recovers her nonce and identity, and encrypts the pair for Bob. Bob decrypts the

Table 7: Query outcomes for Verifpal (Vp), ProVerif (PV), Tamarin (Tm) and Scyther (Sc). ✗: attack found; ✓: no attack, which is bounded exhaustion for Verifpal and a proof for the other tools; *nb*: no Scyther attack within its bounds; *cbp*: ProVerif cannot prove the query; —: not modeled. *Model*: the tools model different assumptions; *property*: the properties differ. [a] Scyther has no Diffie–Hellman. [b] Scyther's released model omits the later key disclosure and queries a different secret. [c] non-injective claim. [d] proved with the hand-written sources lemma. [e] injective in ProVerif, non-injective in Tamarin.

Protocol	Property	Vp	PV	Tm	Sc	Note
FFGG	secrecy of the payload	✗	✗	✗	✗	
DH	secrecy of the payload	✗	✗	✗	—[a]	
	authentication	✗	✗[e]	✗	—[a]	
NS-sym	secrecy of the responder's nonce	✗	✗	✗	—[b]	
	authentication, $A \to B$	✗	✗[e]	✗	*nb*[b]	model
	authentication, $B \to A$	✗	*cbp*[e]	✗	*nb*[b]	model
NS-PK	secrecy of the initiator's nonce	✓	✓	✓	✓	
	secrecy of the responder's nonce	✗	✗	✗	✗	
	agreement, responder on initiator	✗	✗	✗	✗	
	agreement, initiator on responder	✓	✓	✓	✓	
NSL	secrecy of the initiator's nonce	✓	✓	✓	✓	
	secrecy of the responder's nonce	✓	✓	✓	✓	
	agreement, responder on initiator	✗	✓	✓[d]	✓[c]	property
	agreement, initiator on responder	✓	✓	✓	✓	

Table 8: Query correspondence. Verifpal's freshness, unlinkability and equivalence queries are left out because at least one other tool has no counterpart. Public-key correspondence events range over the peer key and both nonces. The public-key claims in ProVerif, Tamarin and Scyther are non-injective, and Scyther restricts them to honest-actor runs. Some ProVerif queries in the symmetric and Diffie–Hellman models are injective; Table 7 marks them.

Tool	`confidentiality? x`	`authentication? A -> B: x`
ProVerif	`query attacker(x)` for an unscoped name; an event reachability monitor for the honest-peer nonce	event correspondence at the model's acceptance point; `inj-event` where stated
Tamarin	`lemma secrecy: All x #i. Secret(x)@i ⟹ not(Ex #j. K(x)@j)`	non-injective correspondence at the chosen acceptance event; completion in the public-key models
Scyther	`claim(R, Secret, x)`	`claim(R, Niagree)` and `claim(R, Nisynch)`

ciphertext, splits the pair and checks Alice's identity. These uses violate Verifpal's origin query because no Alice run sent that ciphertext. Because the reply names Bob, Alice rejects it in her run with Mallory. She does not return Bob's nonce, so he cannot reach the completion event checked by ProVerif, Tamarin and Scyther. Their correspondences therefore hold. The reverse query passes in Verifpal because any reply Alice accepts was sent to her by a Bob run.