



Verifpal

Verifpal[®]

Cryptographic Protocol Analysis for the Rest of Us

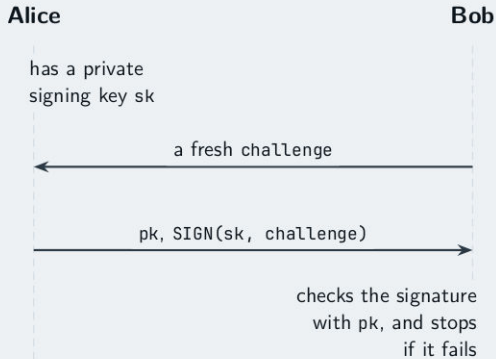
Nadim Kobeissi

Symbolic Software

Swiss Cyber Storm 2026



Can this login be attacked?



Can someone on the network get Bob to accept a signature that Alice didn't make?

Asking Tamarin

Tamarin 1.12.0, same protocol

builtins: signing

rule Alice_Key:

```
[ Fr(~sk) ] --[ Secret(~sk) ]-> [ !Alice_Sk(~sk) ]
```

rule Bob_1:

```
[ Fr(~challenge) ]
```

```
--> [ Out(~challenge), Bob_St(~challenge) ]
```

rule Alice_1:

```
[ !Alice_Sk(~sk), In(challenge) ]
```

```
--[ AliceSends(sign(challenge, ~sk)) ]->
```

```
[ Out(<pk(~sk), sign(challenge, ~sk)>) ]
```

rule Bob_2:

```
[ Bob_St(~challenge), In(<pk, proof>) ]
```

```
--[ BobAccepts(proof),  
    Eq(verify(proof, ~challenge, pk), true) ]->
```

```
[ ]
```

restriction equality:

```
"All x y #i. Eq(x, y) @ #i ==> x = y"
```

lemma authentication:

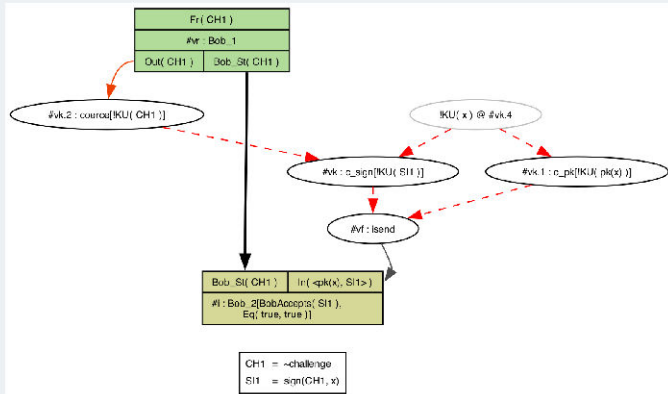
```
"All p #i. BobAccepts(p) @ #i
```

```
==> (Ex #j. AliceSends(p) @ #j & #j < #i)"
```

What you need to learn first

- Each step is a rewrite rule over facts.
- Fresh values, the network and each principal's state are facts you pass around yourself.
- A restriction makes Bob's check stop the run.
- The property is a formula over timepoints.
- In return, Tamarin can prove a property for any number of sessions.

Tamarin's answer



authentication (all-traces):

falsified - found trace (6 steps)

- It's the right attack. SI1 = sign(CH1, x) signs the challenge with a key x that the attacker picked.
- To read it, you need to know what #vk.2, coerce, !KU and lsend mean.

tamarin-prover --prove=authentication --output-dot, Tamarin 1.12.0, drawn with Graphviz.

Tamarin 1.12.0 proves false lemmas

It has no proof kernel, so nothing outside Tamarin checks a Tamarin proof

Issue #936: can B ever decrypt a message?

```
let A = in(x); event Dummy(x)
let B = new sk; out(pk(sk)); in(x);
  let m = adec(x, sk) in event Dec(m)
  else event Fail()
process: A | B
lemma no_dec: "All m #i. Dec(m)@i ==> F"
```

Tamarin's answer

no_dec (all-traces): **verified** (2 steps)

The same model with A's x renamed to y

no_dec (all-traces): **falsified** - found trace (4 steps)

Tamarin 1.12.0, demos/tamarin/. Issue #936 (Hammann); PRs #948–#950, #954, #957, #958 (Milner), status on 19 October 2026. SAPIC+: USENIX Security 2022.

“If the tool’s automated proof search terminates, it returns either a proof of correctness [...] or a counterexample”

The Tamarin manual

- Found in September 2026: 9 soundness bugs in the prover and 12 bugs in SAPIC. Most of the fixes are still unmerged.
- SAPIC+ has a paper proof that its translation is correct. Issue #936 is in the code that implements it.

Asking Verifpal

```
attacker[active]

principal Alice[
  knows private sk
  pk = PUBKEY(sk)
]

principal Bob[ generates challenge ]

Bob -> Alice: challenge

principal Alice[ proof = SIGN(sk, challenge) ]

Alice -> Bob: pk, proof

principal Bob[ _ = SIGNVERIF(pk, challenge, proof)? ]

queries[
  confidentiality? sk
  authentication? Alice -> Bob: proof
]
```

Running it

verifpal.com/workbench, quickstart example. It runs in your browser.

Quickstart: A Substituted Signing Key

VERIFY

PRETTY PRINT

DIAGRAM

COPY LINK

DOWNLOAD

REFERENCES

ANALYSIS COMPLETE
0.3 s elapsed

MODEL.VP

Ctrl+R + enter to verify

VeriPpal ANALYSIS

```
1 attacker[active]
2
3 principal Alice[
4   knows private sk
5   pk = PUBKEY(sk)
6 ]
7
8 principal Bob[
9   generates challenge
10 ]
11
12 Bob → Alice: challenge
13
14 principal Alice[
15   proof = SIGN(sk, challenge)
16 ]
17
18 Alice → Bob: sk, proof
19
20 principal Bob[
21   _ = SIGNVERIFY(pk, challenge, proof)?
22 ]
23
24 queries[
25   confidentiality? sk
26   authentication? Alice → Bob: proof
27 ]
```

RESULTS CONF: COAI

confidentiality? sk
search exhausted at 2 sessions

PASS

authentication? Alice → Bob: proof

FAIL

Attack trace:
1. Attacker constructs PUBKEY(nil).
2. Attacker observes challenge on the wire.
3. Attacker constructs SIGN(nil, challenge).
4. Attacker resolves pk, proof (sent by Alice to Bob) with PUBKEY(nil), SIGN(nil, challenge), fak as PUBKEY(sk), proof as SIGN(sk, challenge).
5. Bob's SIGNVERIFY(PUBKEY(nil), challenge, SIGN(nil, challenge))? passes — the attacker controls one of its inputs.

↳ proof (SIGN(nil, challenge)), sent by Attacker and not by Alice, is successfully used in SIGNVERIFY(pk, challenge, proof)? within Bob's state.

1 of 2 queries failed.

4 analysis messages



Reading the attack

1. Attacker constructs `PUBKEY(nil)`.
 2. Attacker observes challenge on the wire.
 3. Attacker constructs `SIGN(nil, challenge)`.
 4. Attacker **replaces** `pk`, `proof` (sent by Alice to Bob) with `PUBKEY(nil)`, `SIGN(nil, challenge)`. (`pk` was `PUBKEY(sk)`; `proof` was `SIGN(sk, challenge)`)
 5. Bob's `SIGNVERIF(PUBKEY(nil), challenge, SIGN(nil, challenge))`? **passes** --- the attacker controls one of its inputs.
- > `proof (SIGN(nil, challenge))`, sent by Attacker and not by Alice, is successfully used in `SIGNVERIF(pk, challenge, proof)`? within Bob's state.

- The attacker never learns `sk`. It makes its own key from `nil`, a value it always knows, and signs the challenge itself.
- Bob's check passes because he checks against whatever key arrives.

Fixing the model

Before

Alice -> Bob: pk, proof

FAIL x authentication? Alice → Bob: proof

After

Alice -> Bob: [pk], proof

PASS + confidentiality? sk

PASS + authentication? Alice → Bob: proof
[search exhausted at 2 sessions]

- [pk]: the attacker can read pk but can't replace it.
- Something in the real system has to make that true, like a certificate or a key scanned from a QR code. Now the model says so.
- PASS: no attack with up to two sessions of each principal. More on what that means at the end.



When the model is wrong

```
sanity error: `AEAD_ENC` takes 4 arguments,  
but 3 were given  
--> arity.vp:19:7  
|  
19 |     e1 = AEAD_ENC(gab, m1, c0)  
|         ^^^^^^^^  
|  
= note: its signature is  
  `AEAD_ENC(key, nonce, plaintext, ad)`  
= help: `AEAD_ENC` and `AEAD_DEC` take a  
  nonce as their second argument
```

- The error points at the line and suggests a fix.
- You get the same messages in the workbench, on the command line, and in your editor through `verifpal lsp`.

What if a key is stolen next year?

phase[1] means later, after the messages above it were sent

Static Diffie-Hellman

```
principal Alice[
  knows private a
  ...
  k = DH_KEX(gb, a)
  c1 = AEAD_ENC(k, n1, m1, ad)
]
Alice -> Bob: n1, c1
phase[1]
principal Alice[ leaks a ]
```

FAIL x confidentiality? m1

3. Attacker is handed a by a leaks declaration in Alice.
4. Attacker constructs k.
6. Attacker opens c1 with k, n1, obtaining m1.

Signed ephemeral Diffie-Hellman

```
principal Alice[
  knows private ska
  generates a
  ga = PUBKEY(a)
  siga = SIGN(ska, ga)
]
// ... the same exchange, then:
phase[1]
principal Alice[ leaks ska ]
principal Bob[ leaks skb ]
```

PASS + confidentiality? m1

PASS + authentication? Alice → Bob: c1

- Each run gets a fresh key, signed with a long-term one. Leaking the long-term keys later doesn't open c1. This is forward secrecy.

What if I reuse a nonce?

The attacker only records traffic

```
principal Alice[
  knows private k
  knows public ad, greeting
  generates n, m2
  c1 = AEAD_ENC(k, n, greeting, ad)
  c2 = AEAD_ENC(k, n, m2, ad) // same key, same nonce
]
```

FAIL x confidentiality? m2

1. Attacker observes c2 on the wire.
 2. Attacker observes c1 on the wire.
 3. Attacker recovers m2 from c2: c1 shares its key and nonce, and Attacker knows its plaintext.
- > m2 (m2) is obtained by Attacker.

- Encrypt two messages with the same key and nonce, and the ciphertexts give away the XOR of the plaintexts.
- The greeting is public, so m2 falls.
- Make greeting secret and the query passes.

What if a quantum computer arrives later?

A cipher that breaks later

```
attacker[passive]
```

```
...
```

```
e_now = AEAD_ENC(k, n_now, m_now, nil)
```

```
e_later = AEAD_ENC[weak from phase 2](k,  
    n_later, m_later, nil)
```

PASS + confidentiality? m_now

FAIL x confidentiality? m_later

- weak from phase 2: the attacker records e_later today and breaks it in phase 2.

X25519 plus a post-quantum KEM

```
principal Alice[
```

```
    ss, ct = KEM_ENCAP(ekb, r)
```

```
    dh = DH_KEX(gb, a)
```

```
    k = HKDF(dh, ss, nil)
```

```
    e = AEAD_ENC(k, n, m, nil)
```

```
]
```

```
...
```

```
phase[1]
```

```
principal Alice[leaks a] // discrete log broken
```

```
principal Bob[leaks b]
```

PASS + confidentiality? m

and if Bob also leaks dkb:

FAIL x confidentiality? m

- Breaking discrete log isn't enough to read m. The KEM key has to fall too.

Workbench examples harvest-now-decrypt-later and kem-hybrid.

What if your peer is the attacker?

scenarios: Alice talks to Bob in one run and to Mallory, whose key has leaked, in another

```
principal Mallory[
  knows private mk
  gm = PUBKEY(mk)
  leaks mk
]
// ... Needham-Schroeder as published ...
scenarios[
  Alice[gpeer = gb]
  Alice[gpeer = gm]
]
```

FAIL x confidentiality? nr

4. Attacker opens m1@2 with mk, obtaining d1.
 6. Attacker **replaces** m1 (sent by Alice to Bob) with PKE_ENC(gb, CONCAT(ni@2, ga)). (m1 was PKE_ENC(gb, d1))
 9. Attacker **replays** m2@2 (Bob@2 to Alice@2) from another scenario, where it is PKE_ENC(ga, d2).
 12. Attacker observes m3@2 on the wire, where it is PKE_ENC(gm, nr).
 13. Attacker opens m3@2 with mk, obtaining nr.
- > nr (nr) is obtained by Attacker.**

- This is Lowe's 1995 attack on Needham-Schroeder. Putting Bob's identity in his reply fixes it.

A real SDK: can you spot the bug?

WalletConnect One-Click Auth, on the dapp side (shortened)

The wallet answers the dapp's sign-in request with a message signed by the user's account. The dapp checks the signature, then sets up an encrypted session for that account.

```
const isValid = await validateSignedCacao(...);
if (!isValid) {
  reject(getSdkError(...));
}
const address = getDidAddress(payload.iss);
// derive the session key, subscribe, save the session
```

The bug, in the model and in the SDK

A check that doesn't stop the handler is a SIGNVERIF without its question mark

The SDK

```
if (!isValid) {  
    reject(getSdkError(...)); // does not return  
}
```

The model

```
_ = ASSERT(iss_d, gw)?  
_ = SIGNVERIF(iss_d, cp_d, cs_d) // no "?"  
sess_d = HKDF(nil, DH_KEX(gy_pay, x), nil)  
sessreq = AEAD_ENC(sess_d, iv, txdata, c_ad)
```

FAIL x confidentiality? txdata

The attacker names the victim's account and sends a bad signature with its own key. The dapp saves the session anyway, and the attacker reads the next request.

Reproduced on the unmodified SDK, 2.24.0

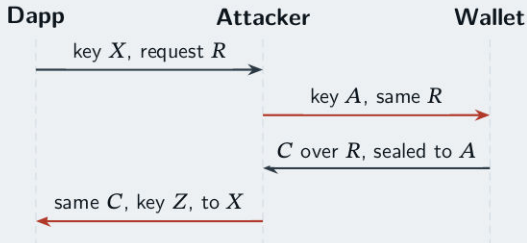
```
authenticate() promise:      REJECTED  
client.session.getAll().length: 1  
account:                    eip155:1:0x1111...1111  
attacker-decrypted method:   personal_sign
```

Present in all 61 stable releases with this method, since 2.12.0 (April 2024). Impact: session impersonation, not the account's signing key.



Fix it, ask again

Checks in the dapp	Result
what shipped	leaks
that the signature is valid W2: a valid signature made for another site	still leaks
that the request fields match W3: the signature doesn't cover the session key	still leaks
that the responder key is signed	no leak found



- W3: the signature covers the request but neither key. The attacker keeps the signature and swaps the keys.

13 models, 33 recorded analyses, Verifpal 1.4.10, 9 September 2026. WalletConnect acknowledged the findings and plans to fix them.

What a result tells you

Verifpal searches for attacks up to a set number of sessions. It doesn't produce proofs.

FAIL with a trace

A run of your model that breaks the query, re-executed before you see it. It's only as accurate as the model.

PASS [search exhausted at 2 sessions]

No attack with up to two sessions of each principal. It says nothing about more.

Use ProVerif or Tamarin for

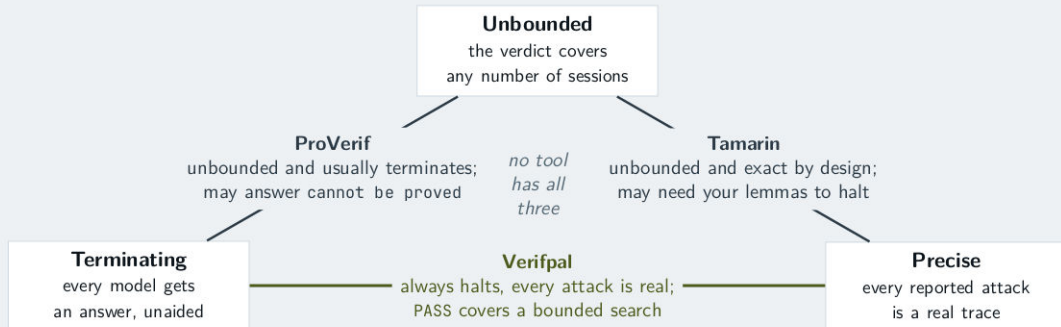
- proofs for any number of sessions
- privacy as indistinguishability
- XOR, pairings, or your own equations

The semantics behind each result are in Verifpal's 2026 paper, ePrint 2026/1654.



Unbounded, terminating, precise

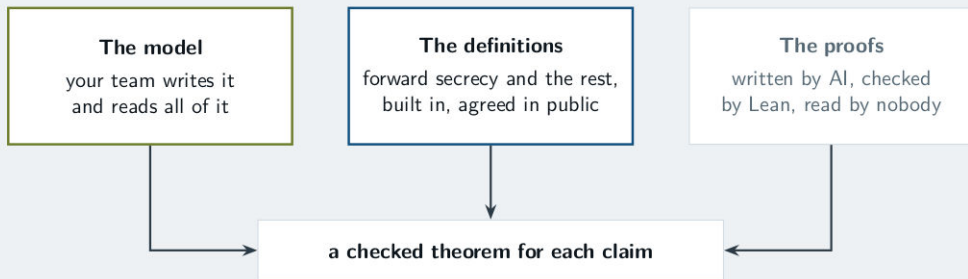
A tool with all three would decide an undecidable problem



Unbounded secrecy with fresh nonces is undecidable (Durgin, Lincoln, Mitchell, Scedrov, J. Computer Security 2004); with bounded sessions it is NP-complete (Rusinowitch, Turuani, TCS 2003). ProVerif's termination is proven for tagged protocols (Blanchet, Podelski, TCS 2005). Verifpal's search is heuristic inside its bound (ePrint 2026/1654, §3).

When an AI writes the proof, what do you read?

From my 2026 position paper, *Nobody Will Read the Lean*



- A Lean security theorem for Schnorr signatures passed the checker but held for every adversary, so it promised nothing.
- Verifpal already has the short models and the built-in definitions. The paper argues for the third box: a checked theorem behind every PASS.

Try it yourself

At verifpal.com/workbench

1. Open the quickstart example and press Verify.
2. Change pk to [pk] and verify again.
3. Open kem-hybrid, add dkb to Bob's leaks, and verify again.

verifpal.com workbench, manual, examples
github.com/symbolicsoft/verifpal GPL-3.0, Rust

From Toy to Instrument: Seven Years of Verifpal. ePrint 2026/1654.
Proving the Account, Losing the Session: Authentication Failures in WalletConnect and TON Connect. 2026.

Used at

adyen

 **BOSCH**

Google

 **PARFIN**

 **Rayls**

TRAILBLAZERS

zoom



ETH zürich



Mit



Opens the quickstart example

Thank you. Questions?

How ProVerif reaches unbounded results

Automatic and unbounded, by analyzing an abstraction of the protocol

- The protocol becomes Horn clauses, which forget how often a message was sent and in what state. Resolution saturates them.
- Sound but incomplete: a derivation may match no execution. When ProVerif can't rebuild a trace, it answers cannot be proved.
- Termination is proven for tagged protocols. For state, such as counters or revocation, the user can add lemmas, axioms and restrictions (since 2022).

ProVerif 2.05, symmetric-key Needham-Schroeder

Try adding a [precise] option on it.

Trying with the initial derivation tree instead.

Could not find a trace corresponding to this derivation.

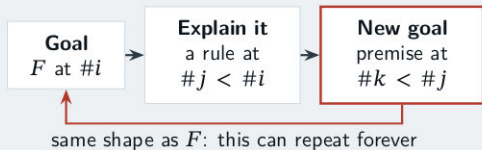
RESULT inj-event(AliceAcceptsFromBob(x)) $\Rightarrow$
inj-event(BobSendsToAlice(x)) cannot be proved.

- Tamarin and Verifpal both find an attack on this protocol. ProVerif can't decide, and asks you to annotate the model.

Blanchet, FnT Privacy and Security 2016; Allamigeon, Blanchet, CSFW 2005; Blanchet, Podelski, TCS 2005; Blanchet, Cheval, Cortier, S&P 2022.
Output: ePrint 2026/1654 artifact, bench/outputs/needham-schroeder.proverif.txt, Table 7.

How Tamarin reaches unbounded results

Unbounded and exact, with the user's help when the search loops



- The method is sound and complete, but it may not halt: a loop in the protocol becomes an infinite proof tree.
- The user breaks the loop with induction, a sources lemma, a reuse lemma, or an oracle that picks the next goal.

“In fully automatic mode, Tamarin either returns a proof that the property holds, or a counterexample/attack if the property is violated, or it may not terminate as the underlying problem is undecidable.”

Basin et al., Tamarin's developers, on 5G AKA. CCS 2018.

Even Needham-Schroeder-Lowe, Tamarin 1.12.0

lemma types [sources, reuse]:

```
"(All na m1 #i. IN_B_1_na(na, m1) @ #i ==>
  ((Ex #j. KU(na) @ #j & #j < #i) |
   (Ex #j. OUT_A_1(m1) @ #j & #j < #i)))"
```

- --auto-sources leaves cases open on the textbook protocol. This lemma, from Tamarin's own NSLPK3 example, closes them.

Meier, Schmidt, Cremers, Basin, CAV 2013; Cortier, Delaune, Dreier, Klein, JCS 2022; Basin, Dreier, Hirschi, Radomirović, Sasse, Stettler, CCS 2018. Model: ePrint 2026/1654 artifact, models/tamarin/ns1-pk.spthy.

Are they proofs?

Yes, with conditions

- They're about the model. A proof of WalletConnect's specification would still miss the SDK's missing return.
- They hold under the algebra the modeler chose, and a wrong equational theory fails silently.
- They exist only when the search halts. On real protocols, that has taken months.
- For privacy, both tools prove diff-equivalence, which many true unlinkability properties fail.
- They're only as sound as the tool's code. Nothing outside ProVerif or Tamarin rechecks a proof.

At scale, in the authors' own words

[TLS 1.3, Tamarin \(CCS 2017\)](#)

Cremers et al.: “approximately 3 months worth of work ... the vast majority of that is the process of writing lemmas.” Up to 100 GB of RAM in a proof.

[5G AKA, Tamarin \(CCS 2018\)](#)

Basin et al.: 124 lemmas and oracles in Python. “Tamarin's fully automatic mode fails to prove relevant security properties and even extremely weak properties such as the full executability of the protocols.”

[TLS 1.3 + 1.2, ProVerif \(S&P 2017\)](#)

Bhargavan, Blanchet and me: “hours on a powerful workstation” for all modes.

These proofs were research projects by the tools' authors, done once the design had stopped changing. Cremers et al., CCS 2017; Basin et al., CCS 2018; Bhargavan, Blanchet, Kobeissi, S&P 2017.



What the bound gives Verifpal

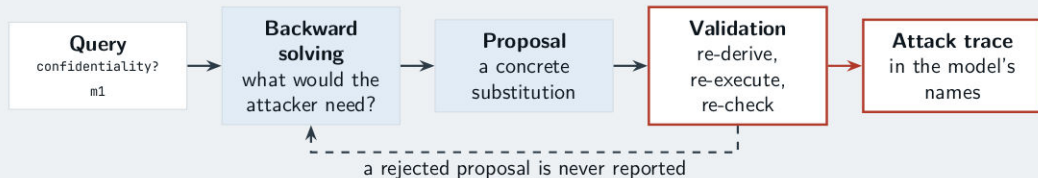
What follows from searching a bounded space

	ProVerif	Tamarin	Verifpal
Sessions	unbounded	unbounded	k , default 2
Halts unaided	tagged only	no	yes
Third verdict	cannot be proved	none; may loop	none
You supply	lemmas	lemmas, oracles	nothing
Queries	event queries	temporal logic	five kinds
Algebra	no XOR	DH, XOR	fixed 25 primitives
Attack output	renamed trace	graph	your names
Proof	unbounded	unbounded	none

Proving that no attack exists and finding one are different questions. On real protocols, the first has taken experts months; the second usually takes seconds to minutes.

Backup: how the search works

Backward from the query, with every attack checked before it's reported



- For m1 to leak, the attacker needs the key and the ciphertext; for the key, an exponent; and so on, down to things it saw or can build.
- A small trusted part of the engine re-derives every attacker term and re-runs the protocol. A solver bug can make Verifpal miss an attack, but not report one that isn't in the model.

Backup: the 25 primitives

Built in, each with one fixed meaning

Hashing and keys

`HASH` `HKDF` `MAC`
`PUBKEY` `DH_KEX`

Encryption

`ENC` `DEC`
`AEAD_ENC` `AEAD_DEC`
`PKE_ENC` `PKE_DEC`

Structure

`CONCAT` `SPLIT` `ASSERT`

Signatures

`SIGN` `SIGNVERIF`
`RINGSIGN` `RINGSIGNVERIF`
`BLIND` `UNBLIND`

Post-quantum

`KEM_ENCAP` `KEM_DECAP`

Threshold (FROST, RFC 9591)

`THRESHOLD_SPLIT`
`THRESHOLD_SIGN`
`THRESHOLD_JOIN`

- You can't declare a signature that leaks its key or a cipher the attacker can open.
- Six of them take ?.
- Security that rests on other algebra, such as XOR, pairings or small subgroups, can't be modeled faithfully. Use another tool for that.



Backup: queries

confidentiality? m

Can the attacker obtain m ?

authentication? $A \rightarrow B: x$

When B uses x , did A send it, exactly once? A failure is either *origin* or *duplicate acceptance*.

freshness? x

Can a stale x be accepted?

unlinkability? a, b

Can the attacker tell that a and b belong to the same party?

equivalence? a, b

Do two parties end up holding the same value?

- precondition limits a query to runs that reached a given message. `--auto-queries` lets Verifpal propose the questions.

Backup: sessions

Two per principal by default; `--sessions k` for more

Info . Analyzing 2 parallel sessions per principal;
per-session values and principals are
suffixed #2.

- Copies share long-term keys but get fresh nonces. The attacker can carry messages between runs, so replays and reflections are in reach.
- authentication? is injective: a value replayed from another run shows up as a *duplicate acceptance*.

Wall-clock seconds by session count

Model	$k=1$	$k=2$	$k=4$
Needham–Schroeder	0.02	0.09	0.79
Signal, 3 messages	0.17	1.4	11.2
TLS 1.3	0.36	2.5	15.5
Signal, 12 messages	3.3	92	502

Apple M5, single observations, Verifpal 1.4.2.



Backup: readability, measured

Share of identifier tokens in a counterexample that the modeler wrote

	Verifpal	ProVerif	Tamarin
Unauthenticated DH	84 %	82 %	40 %
FFGG	92 %	37 %	57 %
Needham–Schroeder	91 %	51 %	65 %
Signal (three messages)	100 %	—	—

Text counterexamples, strict match against identifiers in each tool's own model; Tamarin figures are for its proof-script output. This measures presentation, not analysis power. Data: ePrint 2026/1654, bench/metrics.json.

Backup: result categories

Output	Established	Not established
PASS, exhausted	No contradiction at the reported session count; no term rejected for depth.	Security; any larger session count; completeness of the search.
PASS, truncated	No contradiction among explored proposals; at least one rejected for depth.	Even exhaustion of the implemented search.
FAIL, reproduced and grounded	The reduced state reproduces the outcome; every installed term is re-derived in order.	A certified single-history trace.
[duplicate acceptance]	Two recipient runs used one send.	Completion agreement.
[attacker-supplied value]	The disclosed runtime value was the attacker's own input.	Disclosure of the honest value.

Condensed from ePrint 2026/1654, Table 5. The command line prints FAIL for every contradicted category and appends the bracketed qualifier to the label.

Backup: the command line

```
verifpal verify protocol.vp
verifpal verify protocol.vp --sessions 3
verifpal verify protocol.vp --auto-queries
verifpal verify protocol.vp --fail-on-attack
verifpal verify protocol.vp --format json
verifpal verify protocol.vp --format html
verifpal verify protocol.vp --format tex
verifpal verify protocol.vp --result-code -q
verifpal pretty protocol.vp --write
verifpal diagram protocol.vp
verifpal lsp
```

- `--format tex` writes a LaTeX report with the sequence chart, the verdicts and a diagram of each attack.
- `--format json` gives every verdict and trace step as data, for editor integrations.
- `--fail-on-attack` fails a CI build on a new attack.
`--result-code` pins expected verdicts.
- `lsp` serves the same diagnostics to any editor.
- 43 example protocols ship with the source, including TLS 1.3, Signal, PQXDH and HPKE.



Backup: track record

Published and industrial use

- Signal, Scuttlebutt, TLS 1.3 (2019 paper); first formal model of DP-3T
- Informed the design of Zoom's end-to-end encryption
- Used at Google, Bosch, Trail of Bits, Adyen; in Delta Chat, PrivateBin, Salt Channel
- Taught at Lund, AUB, Sydney, MIT
- WalletConnect and SimpleX (2026)

Cross-tool comparison, 2026

- 11 of 14 selected verdicts agree with ProVerif, Tamarin and Scyther after aligning property scope
- Finds FFGG rotation, Lowe's relay, Denning–Sacco replay and the Π_{Toy} nesting attack
- 588 contradictions across 437 models; a fixed-seed audit of 30 found no false report

verifpal.com/community; ePrint 2026/1654, sections 5–8. The audit interval (0–11.4%) does not establish attack soundness; it is a sample.

