

# Nobody Will Read the Lean:

## A Case for High-Level Cryptographic Languages

Nadim Kobeissi  
Symbolic Software  
[nadim@symbolic.software](mailto:nadim@symbolic.software)

October 2026

### Abstract

In September 2026, an AI produced a Lean proof of Fermat’s Last Theorem in eleven days, and AI provers have started on cryptography. Within a few years, an AI may hand you a machine-checked proof that Signal is secure. You won’t need to read it, because Lean will have checked every step. What Lean can’t check is whether the theorem says what you think it says. That depends on its definitions, which the AI will also have written at length for that one proof. This already happens. A Lean library’s security theorem for Schnorr signatures passed the checker, yet as written it held for every adversary and so promised nothing.

When proofs are cheap, the scarce resource is a person who can read what was proved. I argue that we should build high-level cryptographic languages for that person. A protocol’s owners should write its model themselves, short enough for one or two people to read. Every claim the language makes should rest on a checked theorem. And definitions of protocol goals such as forward secrecy should be built in: a small set, versioned and agreed in public the way we agree on AES, so that a property’s name means the same thing in every report. Today it may not: on a one-page Verifpal model, two of five verdicts would flip under other standard definitions, with neither answer wrong. Agreeing on definitions will take years of public argument that better provers can’t shorten, so we should start now.

## 1 Introduction

For most of our field’s history, a machine-checked proof about a real cryptographic system was among the most expensive things a research group could produce. Project Everest’s founding paper lists twenty-two authors working toward a verified replacement for HTTPS [10]. The Tamarin analysis of TLS 1.3 took “approximately 3 months worth of work,” most of it spent writing lemmas [26], and the 5G study needed 124 lemmas [5]. Because proofs were scarce, the field rewarded the people who could produce them and treated tools that reasoned about protocols without proofs as toys. Verifpal, which I built, started out as one of those toys.

AI is now making proofs cheap. In September 2026, Anthropic published a machine-checked proof of Fermat’s Last Theorem in Lean, produced by Claude in eleven days [2]. On 6 October 2026, OpenAI released more than 700 mathematical manuscripts produced by an internal model, and about 42% of their top-line results now have Lean formalizations [67]. AI can produce proofs in bulk. The results we can rely on are still the ones a proof checker has accepted, and only to the extent that someone has read and understood what they state. The Fermat repository says so: its sources “are written to be checked rather than read,” and “what no tool can check is that each intermediate theorem means what its name suggests; that is for the reader to judge” [2].

This paper assumes that within a few years, a machine-checked proof for a protocol the size of Signal or TLS will cost computing time rather than years of expert work. If that holds, the economics of formal methods for cryptography flip. The expensive part of a verification becomes understanding what was proved: reading the definitions and checking that they describe the protocol and the security we care about. So I argue that research effort should move from producing proofs to designing *high-level cryptographic languages*. In such a language, the people responsible for a protocol write its model themselves, clearly enough that one or two readers can follow all of it, and every security claim the language reports is backed by a machine-checked proof against a fixed, public definition. A proof assistant such as Lean holds the generated

proofs underneath. For reasons I’ll explain, the language people read has to be less expressive than the proof assistant.

Section 2 gives the evidence that proofs are getting cheap, Section 3 explains why a checked proof doesn’t settle what was established, Section 4 proposes four requirements, and Section 5 argues for standardized, versioned definitions. Section 6 runs Verifpal on two small models, and Sections 7 to 9 cover existing tools, objections, and what our field should reward.

## 2 Proofs are getting cheap

In 2024, AlphaProof and AlphaGeometry 2 reached silver-medal level at the International Mathematical Olympiad, but humans had to formalize the problems first, and the system could take days per problem [42]. By January 2026, AxiomProver had solved all twelve Putnam 2025 problems in Lean, eight of them within the exam time [4]. Then came large formalizations. Math Inc’s Gauss formalized the strong prime number theorem in about 25,000 lines of Lean in three weeks, starting from scaffolding written by mathematicians [58]. Claude’s proof of Fermat’s Last Theorem rests on 29,511 theorems [2]. OpenAI’s October release pairs its formalizations with prose scope notes that say what each formal statement covers and what it leaves out. Within a day of the release, OpenAI withdrew three manuscripts over a sign error and revised fourteen others. None of the three withdrawn manuscripts had a formalized main result [67]: the error sat in a part that no checker had seen.

Cryptography is following. A study of LLM proof synthesis on F\* projects, including HACL\* and EverCrypt, found that LLMs could automate between a third and a half of the proofs [19]. ShannonProver takes a protocol and security definitions written by a cryptographer and produces EasyCrypt proofs; on case studies such as MEE-CBC and ChaCha20-Poly1305 it “completes within hours proof developments that historically took experts weeks to months” [57]. CatCrypt reports machine-checked security theorems in Lean for 172 cryptographic protocols and constructions, developed in two months by one researcher working with generative AI [73]. And Signal, the Lean FRO and the Beneficial AI Foundation have started verifying Signal’s Rust implementation in Lean with AI assistance [63].

None of the cryptographic work above is yet a complete computational proof of a deployed protocol such as Signal or TLS. Such a proof doesn’t need mathematics as deep as Fermat’s Last Theorem, but it’s a different kind of proof, built from probabilistic games, reductions that must be efficient, and long case analyses over sessions and compromise. I assume AI provers will get there within a few years. If they stall, the assumption fails, and Section 8 explains how much of the argument survives. Kleppmann predicts the same shift for software: as verification becomes automated, “the challenge will move to correctly defining the specification” [47].

## 3 What a proof checker does not tell you

Suppose an AI model hands you a complete Lean verification of the Signal protocol. You don’t need to read its proofs, because Lean’s kernel checks them. But the kernel only confirms that each proof establishes its stated theorem. What a theorem means depends on the definitions it uses: the protocol’s state machine, the attacker, the primitives and the security properties. In an AI-generated development the AI writes those definitions too, and for a real protocol they’re long. If nobody on your team can say what the theorems claim about the version of Signal that ships, then nobody has checked the part that matters.

Large verified systems show the proportions. seL4’s abstract specification is about 4,900 lines of Isabelle, against some 200,000 lines of proof and about 20 person-years of proof effort [46]. Verified crypto libraries keep their specifications small so people can review them; the HACL\* paper calls its specifications “the source of ‘truth’ for our library: the smaller, the better” [79]. An AI writing a protocol’s definitions has no reason to be brief unless the language makes it.

We already know how correct proofs miss attacks: it has happened many times, with proofs written by experts. The WPA2 four-way handshake had a published proof of key secrecy and session authentication [37]. KRACK broke WPA2 without violating either property; in Vanhoef and Piessens’s words, “the proofs do not model key installation” [77]. TLS 1.3’s pre-shared-key mode had been analyzed with games and with Tamarin. The Selfie attack uses a key shared by a group, a case the Tamarin model excluded by assuming that each pre-shared key belongs to exactly one client and one server [32, 26]. Jackson et al. found new attacks on WS-Security and on DRKey, which had been proved secure in ProVerif and in Coq, because the usual symbolic abstraction of signatures assumes much more than unforgeability [44]. Basin et al. proved certain Mastercard EMV transactions secure, then found that their model assumed the terminal and the issuer always agree on which card brand’s kernel runs the transaction. Their own card brand mixup attack showed otherwise [7, 6]. All of these proofs were correct. Each statement covered less than people thought, and only someone reading the model against the real system noticed.

AI-written statements make this worse. When researchers rechecked miniF2F, a standard benchmark for AI provers, they found discrepancies between the formal and informal statements in more than half of its 488 problems [68]. The Lean reference manual’s guide to validating a proof lists “un-reviewed AI-generated proofs and programs” among the code a checker should treat as possibly malicious, and adds that “it is important to distinguish the question ‘does the theorem have a valid proof’ from ‘what does the theorem statement mean?’” [53]

Cryptographic libraries aren’t exempt. In September 2026, the maintainers of the VCVio Lean library restated several headline security theorems, for Schnorr signatures, Fiat–Shamir, Falcon and part of ML-DSA. The theorems ended with an existential over reductions that carried no efficiency constraint, so, in the pull request’s words, they “hold for every adversary and carry no security content” [30]. Lean’s kernel had accepted all of them except Falcon’s, which still rested on sorry. Kernels have bugs too: in July 2026, an AI-assisted Lean “disproof” of the Collatz conjecture exploited a kernel soundness bug, fixed the same week [62]. Independent checkers reduce that risk. They do nothing about a statement that says the wrong thing.

Verified implementations have the same problem. In February 2026 I audited libcrux, a Rust library whose documentation said that “every implementation in libcrux is formally verified,” along with two related libraries, and reported thirteen vulnerabilities, two of them first found by others [52]. Nine lay outside what the checked proofs covered. One of those, in an ML-KEM interface never translated to  $F^*$ , caused decryption failures for Signal users in the post-quantum ratchet. The other four were inside the verified code. The ML-KEM specification used the wrong rounding constant in decompression and left out an inverse NTT, so proofs against it target the wrong function. A lemma claiming that 12-bit values fit in one bit passed the build because its module was admitted by default. An ML-DSA specification computed  $x^2$  where it meant  $x \cdot y$ , and about forty unproved axioms contradicted it. I found all four by reading, with a text editor and a terminal. I called the gap between what a verification claims and what it establishes *verification theatre*. It doesn’t need AI, just specifications that nobody reads closely, proofs that the build admits without checking, and hand-written reports of what was verified. AI will produce all three faster than anyone can read them.

Tamarin and ProVerif have no small kernel, so a verdict from either one trusts the whole tool. In September 2026, Kevin Milner opened pull requests against Tamarin that fix nine soundness bugs in its main prover, twelve reported bugs in its SAPIC process translation, and more in its equivalence mode; Sven Hammann reported many of them [60]. Several had let Tamarin report a false lemma as verified. In one, a model in which two roles each read an input named  $x$ , “an extremely common idiom,” is translated so that decryption can never succeed. Tamarin 1.12.0, the current release, then proves that no message is ever decrypted, even though the attacker can encrypt one under the published public key [36]. The first batch of fixes, merged on 7 October, changed no verdict in Tamarin’s own regression suite [60].

Protocols also change. Signal adopted PQXDH in 2023. Bhargavan et al. found that nothing

in the first revision of its specification let a party tell a KEM public key from an X25519 public key, and that the protocol relied on a binding property that IND-CCA security doesn't provide. The second revision, a month later, added requirements for both [11]. In 2025 Signal added a post-quantum ratchet and began translating its Rust code to F\* with hax on every change, so that a change that breaks the proofs breaks the build [23]. That keeps the proofs in step with the ratchet's code, but not with the ML-KEM library it calls, where the decryption failures above came from. Nor does it tell anyone whether the new statements still say what the team means; each such change produces statements that someone has to read again. And even correct statements are about a model of the protocol. Checking that the model matches the deployed system has always been the analyst's job. Once proofs are cheap, it becomes most of the job.

## 4 Four requirements

I think the protocol languages we build from now on need four properties. A language either meets each one or it doesn't, and Section 9 suggests how to check.

**Legible at protocol scale.** One or two people should be able to read a complete model, assumptions and claims included, even for a protocol the size of Signal or TLS 1.3. By “read” I mean that a reader can say what each claim promises and point to the lines of the model it depends on. Legibility has to be a design goal from day one, because a language loses it one feature at a time: every construct that lets a modeler say something new also gives every reader something new to check. Code review gives a sense of what one sitting can cover. A widely cited study of review at Cisco advises keeping each review under 200 to 400 lines and under an hour [21], and a model meant to be read in full should fit in a few such sittings. AWS's engineers set a similar target for their TLA+ specifications, which had to “capture the essence of a design in a few hundred lines of precise description” [64].

**Written by its owners.** The people responsible for a protocol should write its model themselves, and that model should be the protocol's specification of record: the text that prose, test vectors and implementations are checked against. Otherwise the reading problem just moves up a level. A team that asks an AI to turn its prose and partial understanding into a formal model gets an artifact that nobody wrote, which then has to be checked against an intent that nobody stated precisely. So the language has to be high-level enough that its users can write it directly. Where AI helps, it should help through refinement that people can follow: the team states its goals in the language's terms, and the AI proposes how to make each one precise, in steps small enough to review before they go into the model. Writing the model is also how gaps come to light. Bhargavan et al. found the gaps in PQXDH's first specification by modeling it [11]. An AI asked to model that revision would have had to fill them in, and nothing would oblige it to show where it guessed. Authorship alone isn't enough, which is why this requirement comes with legibility: two of the libcrux specification errors in Section 3 survived for a year and a half in a specification its own team maintained.

**Backed by proofs.** The language's semantics and its security definitions are formalized once in a proof assistant. Each model is translated into theorems about those semantics, one per claim, and the proofs are generated, by AI or any other method, and checked by the kernel. If the language says a model has forward secrecy, the kernel has accepted a theorem stating that the translated model satisfies the language's definition of forward secrecy. If no proof turns up, the language makes no claim, and where it can, it shows a counterexample instead. The record of what was established comes from the checker, not from documentation kept by hand, and a claim whose proof was admitted or assumed is not a claim the language makes. All four libcrux errors in verified code survived because the proofs that would have exposed them were admitted or assumed [52]. The translation is part of what a reader trusts, so it has to be proved correct

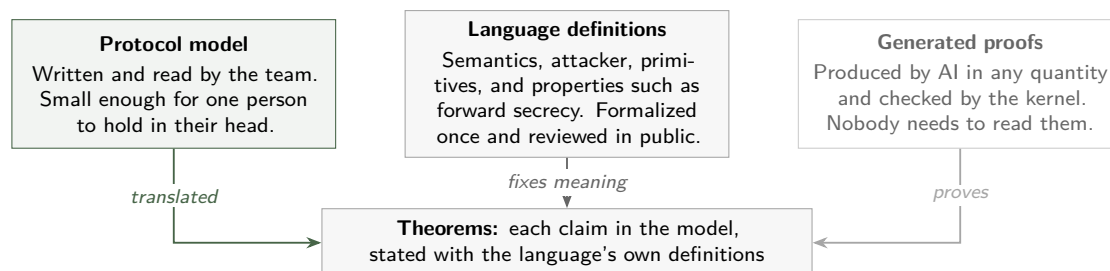


**Figure 1:** Who reads what. The team reads the model, the community reviews the language’s definitions and its translation once, and nobody needs to read the generated proofs.

once or checked separately for each model. SAPIC+ proves its translations to ProVerif, Tamarin and DeepSec correct on paper [20], but a proof on paper doesn’t cover the code that implements it, and the SAPIC bugs in Section 3 were in that code. Noise\* verifies its protocol compiler in F\*, so the proof covers the compiler that runs [40].

I use Lean as the example host because most current work on AI theorem proving targets it, but nothing here depends on Lean: computational definitions might live in EasyCrypt, SSProve, or a Lean library such as VCVio [76].

**Opinionated.** The security definitions are built into the language, and there are few of them. By security definitions I mean protocol goals such as forward secrecy, post-compromise security, authentication and unlinkability, not primitive notions such as IND-CCA2. A modeler chooses which properties to claim about which values, but can’t write a new definition of forward secrecy. Primitives get the same treatment one level down: the language has a fixed registry, and each primitive has a fixed symbolic abstraction tied to named computational assumptions. Section 5 argues for this requirement, which I expect our field to find the hardest to swallow.

Figure 1 shows how the requirements divide the reading. To trust a result, you trust the model, which your team reads; the language’s semantics and definitions, which the community reads once; the translation, which is proved once; and the proof checker’s kernel. That’s the whole verification boundary [52], and nobody has to read the proofs. In a Tamarin or Lean development today, the model, the property statements and any restrictions or axioms are written afresh for each protocol, and anyone who wants to know what was established has to read all of them.

## 5 Fixed definitions

The fourth requirement asks a language to decide in advance what forward secrecy, post-compromise security, authentication, unlinkability and zero knowledge mean, and to offer those definitions instead of letting each analyst write their own. Our field has treated the freedom to write your own definitions as a strength, and for research it is one. For protocols that people deploy, it has become a source of confusion.

The disagreement is old and well documented. Lowe opened his 1997 hierarchy by noting that “there is no clear consensus in the academic literature about precisely what ‘authentication’ means” [55]. For key exchange, Cremers showed that the CK, CK-HMQV and eCK models are formally incomparable, with attacks that each model misses and another catches [28]. Boyd and Gellert found, in the forward secrecy literature, “different terminology used to describe the same thing, and ... the same terminology used to describe different things” [16]. For group key exchange, Poettering et al. report that “effectively no two security analyses of such protocols were conducted in the same formal model” [70]. Unlinkability is the starkest case: the e-passport BAC protocol was shown linkable under one formalization and proved unlinkable under another, and the two camps argued about it in print for years [3, 39, 41].

Single protocols are no different. Signal has been analyzed as a multi-stage key exchange with game-based definitions [22], in ProVerif and CryptoVerif [50], as a double ratchet with its own notions of forward secrecy and healing [1], and in the UC framework [14]. Each analysis states its own version of what was proved, and the last one describes six variants of the double ratchet that are insecure under its definition but secure under earlier ones. Apple’s PQ3 was analyzed with games and with Tamarin; the Tamarin authors found that injective agreement holds only under assumptions about the session-handling layer that the game-based analysis didn’t consider [74, 54]. Even RFC 9846, which replaced RFC 8446 in 2026, describes TLS 1.3’s security properties informally and points to several papers for their formal definitions [71].

So two reports that both claim “forward secrecy” may not be comparable, and you can’t tell without reading both sets of definitions. Cheap proofs don’t remove that reading.

Assumptions about primitives also turn out to be too weak, leaving protocols to rely on properties that nobody had named. Weak Fiat–Shamir is sound only for statements fixed in advance. Bernhard et al. showed that Helios used it, so the earlier proofs of Helios did not apply [9], and Dao et al. later found 36 weak Fiat–Shamir implementations across 12 proof systems [29]. The fix was to name the missing property and say which protocols need it. A fixed library of protocol goals can hold several notions of one property, too. Lowe was right that several notions of authentication are valid goals, and a library can offer aliveness, non-injective agreement and injective agreement as separate, named definitions. What it shouldn’t allow is a new meaning under an old name.

We already accept this trade-off for primitives. Projects don’t design their own block ciphers: we standardize a few, study them for years, and give them names and versions. I propose that we treat protocol security definitions the same way, with a small set of named, versioned definitions that are formalized once and reviewed in public. A report would then say which definition it established, by name and version, the way a protocol specification names AES-256-GCM rather than “a block cipher in a suitable mode.” Symbolic and computational proofs establish different guarantees, and computational soundness results connect the two only under restrictions [24]. So each built-in property needs both a symbolic and a computational definition, and a report should state the two results separately: proved symbolically, proved computationally under named assumptions, or not attempted.

Standards bodies already do this for primitives. NIST’s post-quantum call told submitters which definitions counted, IND-CCA2 for KEMs and EUF-CMA for signatures, and added that each “should be taken as a statement of what NIST will consider to be a relevant attack” [66]. Its 2022 call for additional signatures listed the BUFF properties as desirable [65, 27]. A current IRTF draft asks specification authors to “express security definitions in a formal language” and to list the properties a construction does not provide [75]. Notions such as IND-CCA2 sit behind the primitive registry as named assumptions; they aren’t the protocol goals this section is about.

The Noise Protocol Framework is the closest thing we have to this design. Its specification gives every handshake pattern a fixed table of authentication and confidentiality levels for its payloads [69]. Girol et al. showed what happens when fixed definitions are informal: the levels are “informal and non-trivial to interpret,” they don’t grow stronger monotonically as they appear to, and a systematic analysis finds 74 distinct combinations of guarantees where the specification lists 9 [35]. I read this as a reason to give fixed definitions formal semantics and version numbers. Their own tool derived its 74 results from a small, fixed vocabulary of adversary capabilities and goals.

Fixed definitions don’t have to be lists of properties. In the real-versus-ideal paradigm, a protocol is secure if no environment can tell its executions apart from those of an ideal functionality run with a simulated attacker [18]. The functionality is a short program in which a trusted party does what the protocol is meant to do. An ideal secure channel, for instance, shows the adversary only each message’s length and lets it decide only whether and when to deliver. Symbolic tools state unlinkability the same way, against a system in which every session uses a

| Query                                   | 1 session | 2 sessions | What decides a failure            |
|-----------------------------------------|-----------|------------|-----------------------------------|
| <b>confidentiality?</b> m               | holds     | holds      |                                   |
| <b>authentication?</b> Bob -> Alice: gb | holds     | holds      |                                   |
| <b>authentication?</b> Alice -> Bob: ga | holds     | fails      | injective counting of acceptances |
| <b>authentication?</b> Alice -> Bob: e  | holds     | holds      |                                   |
| <b>equivalence?</b> k_a, k_b            | holds     | fails      | runs compared by session index    |

**Table 1:** Verifpal 1.6.5 on Figure 2. “Holds” means that the bounded search found no attack at that number of sessions per principal; three sessions give the same results.

fresh identity [3, 39]. This form fits Section 4’s requirements: it states a goal as a program that the protocol’s owners can read, and refinement can start from it. But it moves the hard reading into what the functionality leaks and what it lets the adversary decide. A functionality that hands the adversary every input and lets it choose every output is realized by every protocol; it’s the real-versus-ideal version of a theorem that holds for every adversary. Functionalities are also usually written for the analysis at hand, as in the UC analysis of Signal [14]. The remedy is the same: a small library of ideal functionalities with named, versioned leakage.

A language built on such a set won’t be able to express some protocols. I think that’s an acceptable price for a language people can read. The W3C’s Rule of Least Power gives the same advice for the web, to “use the least powerful language suitable,” and notes that “less powerful languages are usually easier to secure” [8]. Some of the definitions will turn out to be wrong. A small shared set is still much easier to fix than many private ones, because a fix to a shared definition reaches every model that uses it.

## 6 What Verifpal shows

I’ve spent seven years building Verifpal [49, 51] around legible models that engineers write themselves and fixed definitions, and its current engine goes part of the way toward proofs.

A Verifpal model is a sequence of principal blocks and messages between named principals, close to how engineers sketch a protocol on a whiteboard. Primitives come from a fixed registry of 25, each with fixed rewrite rules, so a model never contains an equational theory. A value in square brackets in a message travels over an authenticated channel. A **phase** line separates periods of time, and a **leaks** line in a later phase hands values to the attacker, which is how a model asks about forward secrecy. An annotation such as `PUBKEY[weak from phase 1]` says that a primitive’s protection fails from some point on. There are five query types: confidentiality, authentication, freshness, unlinkability and equivalence. Their meaning is fixed by the language [51, §5], and a model can’t change it. People have used Verifpal on Signal, TLS 1.3, MTPROTO and ProtonMail, and during the design of Zoom’s end-to-end encryption.

Figure 2 is a signed ephemeral Diffie–Hellman handshake. Alice and Bob exchange signed ephemeral keys, derive a session key, and Alice sends one encrypted message. Both identity private keys leak in phase 1. Table 1 gives Verifpal’s results.

The confidentiality query, read together with the phase-1 leaks, is Verifpal’s built-in form of forward secrecy: a value that an honest run held at the end of phase 0 must stay undervivable by the attacker in every later phase. A reader doesn’t need to know how the engine checks this. They need to know what the query means, and that fits in a sentence.

Table 1 also shows two failures, and neither is an attack on the session key. With two sessions, the attacker replays Alice’s first flight to a second run of Bob. Both of Bob’s runs accept the same signed ephemeral key, which Alice sent once. Verifpal’s authentication query counts acceptances against sends, so it reports a duplicate acceptance, and the tool prints a note telling the reader to “read it as a replay-protection question about this flight rather than as a forgery.” Under Lowe’s non-injective agreement the same query holds; under injective agreement

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> attacker[active]  principal Alice[   knows private a_id   ga_id = PUBKEY(a_id) ] principal Bob[   knows private b_id   gb_id = PUBKEY(b_id) ] Alice -&gt; Bob: [ga_id] Bob -&gt; Alice: [gb_id]  principal Alice[   generates a   ga = PUBKEY(a)   sig_a = SIGN(a_id, ga) ] Alice -&gt; Bob: ga, sig_a  principal Bob[   _ = SIGNVERIF(ga_id, ga, sig_a)?   generates b   gb = PUBKEY(b)   sig_b = SIGN(b_id, CONCAT(ga, gb))   k_b = HKDF(nil, DH_KEX(ga, b),     CONCAT(ga, gb)) ] </pre> | <pre> Bob -&gt; Alice: gb, sig_b  principal Alice[   _ = SIGNVERIF(gb_id,     CONCAT(ga, gb), sig_b)?   k_a = HKDF(nil, DH_KEX(gb, a),     CONCAT(ga, gb))   generates m, n   e = AEAD_ENC(k_a, n, m, nil) ] Alice -&gt; Bob: n, e  principal Bob[   m_b = AEAD_DEC(k_b, n, e, nil)? ]  phase[1] principal Alice[leaks a_id] principal Bob[leaks b_id]  queries[   confidentiality? m   authentication? Bob -&gt; Alice: gb   authentication? Alice -&gt; Bob: ga   authentication? Alice -&gt; Bob: e   equivalence? k_a, k_b ] </pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Figure 2:** A signed ephemeral Diffie–Hellman handshake in Verifpal. Identity public keys are distributed in advance over an authenticated channel, and both identity private keys leak in phase 1. Wrapped lines are single lines in the model file.

| Variant                     | DH broken | KEM broken | Bob signs ek | <code>confidentiality? m</code> |
|-----------------------------|-----------|------------|--------------|---------------------------------|
| hybrid_dh_broken (Figure 3) | yes       | no         | yes          | holds                           |
| hybrid_kem_broken           | no        | yes        | yes          | holds                           |
| hybrid_both_broken          | yes       | yes        | yes          | fails                           |
| hybrid_ek_unsigned          | yes       | no         | no           | fails                           |

**Table 2:** Four variants of Figure 3 under Verifpal 1.6.5. Results are the same at one, two and three sessions.

it fails [55]. The equivalence query fails for a different reason. It compares the key of Alice’s first run with the key of Bob’s first run, and the attacker can cross the sessions so that Alice’s first run completes with Bob’s second. Every key is still shared only by honest runs, but the two runs the query compares no longer agree. A definition that partners runs by their transcripts, as Bellare–Rogaway-style models do, would find nothing wrong.

So in a one-page model, two of five answers depend on a definition the reader may not have in mind. Another tool could give the opposite answers for the same protocol, and neither tool would be wrong. Verifpal keeps its definitions fixed, explains each verdict in the model’s own names, and pins each definition down with regression models: the repository holds 869, many of which isolate a single distinction, such as replay versus forgery or known versus forgeable. No single tool can settle whether its definitions are the right ones. That takes public review, and reports that name the version of each definition they applied, as Section 5 proposes.

Hybrid post-quantum key exchange is now deployed in Signal and iMessage, among many others [11, 54]. The idea is that a session key stays secret as long as either component, the classical Diffie–Hellman exchange or the post-quantum KEM, holds up. Verifpal has no query

|                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> attacker[active]  principal Bob[   knows private b_id   gb_id = PUBKEY(b_id) ] Bob -&gt; Alice: [gb_id]  principal Bob[   generates b, dk   gb = PUBKEY[weak from phase 1](b)   ek = PUBKEY(dk)   sig = SIGN(b_id, CONCAT(gb, ek)) ] Bob -&gt; Alice: gb, ek, sig  principal Alice[   knows private m   generates a, r, n   ga = PUBKEY[weak from phase 1](a)   _ = SIGNVERIF(gb_id, CONCAT(gb, ek), sig)? </pre> | <pre> ss, ct = KEM_ENCAP(ek, r) k = HKDF(CONCAT(ga, gb, ct), CONCAT(DH_KEX(gb, a), ss), nil) e = AEAD_ENC(k, n, m, nil) ] Alice -&gt; Bob: ga, ct, n, e  principal Bob[   ss_b = KEM_DECAP(dk, ct)   k_b = HKDF(CONCAT(ga, gb, ct), CONCAT(DH_KEX(ga, b), ss_b), nil)   m_b = AEAD_DEC(k_b, n, e, nil)? ]  phase[1]  queries[   confidentiality? m ] </pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Figure 3:** A hybrid handshake in which the Diffie–Hellman half breaks in phase 1. Bob signs both of his ephemeral public keys, and the session key depends on both shared secrets.

for this, so the analyst has to spell it out. Figure 3 shows such a handshake, and Table 2 shows four variants I wrote. The first two each break one component in phase 1 with a *weak* annotation. The third breaks both, to check that the model isn’t secure for a trivial reason. The fourth breaks Diffie–Hellman and has Bob sign only his Diffie–Hellman share, so the attacker can swap in its own encapsulation key and read the message once the classical half falls.

The hybrid claim is the first two rows together, checked against the third, and nothing in the language connects them. The analyst has to know that the property needs one model per component, that a model where everything breaks should fail, and that each component must be authenticated. Analyses of hybrid protocols rebuild this structure by hand, each in its own terms. A language for post-quantum migration should define hybrid secrecy once: a value stays secret in every execution in which at least one listed component holds. The language would then generate the component cases and the sanity case itself.

The KEM also shows what a reader has to trust beyond the definitions. Verifpal’s KEM leaves out ciphertext validity, implicit rejection and the binding properties that re-encapsulation attacks exploit [51, §9], so the attack that Bhargavan et al. found in PQXDH lies outside the model [11]. A legible language has to show omissions like these next to the result, as Verifpal already does for the weakening annotations a user writes.

The 2026 Verifpal paper proves that every execution Verifpal’s judge evaluates is a history of a reference semantics, so every reported attack is an explicit execution of the model, and that the search terminates [51]. Those proofs are on paper, though, and a passing query, meaning no violation was found at the chosen number of sessions, isn’t yet a theorem. The usual way to close that gap has been to ask engineers to learn a proof tool and write lemmas. AI-generated proofs offer another: keep a language that engineers can read, and turn every passing query into a checked theorem. Figure 4 sketches what the theorems behind Figure 2 would state.

## 7 Existing tools

Protocol languages with built-in properties are an old idea. Casper let the user choose properties from a fixed menu taken from Lowe’s hierarchy [56], CAPSL and OFMC’s Alice-and-Bob

```

-- Produced from signed_dh.vp by a translation proved correct once.
def signedDH : Protocol := ...
-- Statements: one per query. Proofs: generated, checked by the kernel, not read.
theorem signedDH_m_sym : Sym.Confidentiality.v1 signedDH "m" := by ...
theorem signedDH_m_comp :
  Comp.Confidentiality.v1 signedDH "m"
  [GapDH.v1 X25519, EUF_CMA.v1 Ed25519, AEAD.v1 ChaCha20Poly1305, PRF.v1 HKDF]
  (bound := ...) := by ...

```

**Figure 4:** A sketch of the statements behind the confidentiality query of Figure 2. This is not working Lean, and the names are invented. A reader checks three things: that `signedDH` comes from the model they read, what `Confidentiality.v1` means in each setting, and which named assumptions the computational statement rests on.

| Tool                             | Input                        | Properties      | Model | Machine-checked       |
|----------------------------------|------------------------------|-----------------|-------|-----------------------|
| Casper, CAPSL, OFMC              | Alice and Bob                | built in        | sym.  | none                  |
| scyther-proof, PSPSP             | Alice and Bob; transactions  | partly built in | sym.  | Isabelle proofs       |
| Noise Explorer                   | Noise patterns               | built in        | sym.  | none (ProVerif runs)  |
| Noise*                           | Noise patterns               | built in        | sym.  | F*, verified compiler |
| Verifpal                         | principals and messages      | built in        | sym.  | none (bounded search) |
| Tamarin, ProVerif                | rewriting rules; pi calculus | user-written    | sym.  | none                  |
| SAPIC+                           | stateful pi calculus         | user-written    | sym.  | none                  |
| DY*, DyLean                      | F* or Lean code              | user-written    | sym.  | F* or Lean            |
| hax                              | Rust                         | user-written    | both  | F*, Rocq, Lean        |
| Owl                              | its own language             | user-written    | comp. | none (Verus for code) |
| CryptoVerif, EasyCrypt, Squirrel | calculi, programs            | user-written    | comp. | tool-checked scripts  |
| SSProve, CryptHOL, VCVio         | proof assistant              | user-written    | comp. | Rocq, Isabelle, Lean  |

**Table 3:** Where existing tools stand on the requirements. “Built in” means that the user chooses from properties the language defines.

notation had built-in goals [31, 61], and scyther-proof and PSPSP generate Isabelle proofs from high-level descriptions [59, 38]. Noise Explorer, which I co-authored, generates ProVerif models and queries for any Noise pattern from the specification’s security levels [48].

Table 3 places these tools next to current ones. Two current tools come closest to what I’m proposing. hax translates Rust to F\*, Rocq, Lean, ProVerif and SSProve, so it already reaches symbolic and computational back ends [13]. But it starts from implementations. Its output grows with the code, its properties are written by the user, and its authors note that they “do not yet provide formal guarantees” for the correctness of its translations [13]. For ProVerif, the top-level processes and the security queries are still written by hand [12]. SAPIC+ works at the protocol level and compiles one model to ProVerif, Tamarin and DeepSec through translations proved correct on paper for restricted fragments [20]. All of its targets are symbolic, and its properties are lemmas the analyst writes.

On the computational side, Owl fixes its cryptographic assumptions in its typing rules but leaves the security goals to the user [34], and CryptoVerif ships a library of primitive assumptions [15]. Lean now has protocol frameworks of its own: DyLean and LeanDY for symbolic trace properties, VCVio for game-based proofs, and a machine-checked link between a symbolic and a computational model by Dziembowski et al. [78, 45, 76, 33]. These are the right foundations for backing claims with proofs. None of them fixes the definitions of security, though, and none aims at a model a team can read in one sitting. I found no tool that combines a protocol-level language a team can read, a small set of built-in definitions, and generated, kernel-checked proofs against both symbolic and computational versions of those definitions.

## 8 Objections

**Real protocols need expressiveness.** Some do. New security notions, unusual primitives and research protocols need a language in which anything can be stated, and Lean, EasyCrypt, Tamarin and ProVerif should stay the place for that work. My claim is narrower. Deployed protocols mostly claim a small set of properties, and those are the properties whose results most need to be comparable. Verifpal’s five query types, with phases and weakening annotations, have been enough to model the protocols listed in Section 6, though that section also shows a property, hybrid secrecy, that they can only express by hand. When a protocol needs a property outside the built-in set, the language should say so. The team can then propose a new definition through the public process, or move to an expressive tool and accept the reading cost there.

**An AI can check the statements for us.** Kevin Buzzard writes that he “would definitely trust the opinion of a frontier LLM if it said that a formal (Lean) and informal (English)” statement agreed [17]. For mathematics that’s plausible, because the informal statement is fixed and public. For protocols, the agreed informal statement is exactly what we’re missing. There is no shared English statement of Signal’s forward secrecy to compare a Lean theorem against (Section 5). An AI that compares a theorem with a fixed, public definition performs a useful check. An AI that explains a theorem in its own words gives the team an unchecked account of the one part the kernel didn’t check. Fixed definitions turn the second situation into the first.

**Proofs about protocols won’t get that cheap.** Maybe not, or not soon. Most of the argument doesn’t depend on the timeline, though. Fixed definitions make results comparable whether proofs are cheap or not, and a legible model is easier to review whoever writes the proof. If proofs stay expensive, the proof requirement falls back to what Verifpal offers today: a bounded search whose reported attacks are executions of the model, and whose results each state what they establish. Cheap proofs just make the case more urgent, because they multiply the number of results someone has to read.

**Fixed definitions will be wrong, or will freeze.** Some of them will be wrong. When IND-CCA turned out to be too weak for some protocols, researchers named the missing binding properties, and later analyses could state which ones they assumed [25, 72]. A versioned library of protocol goals makes the same kind of fix visible. A report says it established `FS.v1`; when `FS.v2` appears, every model in a corpus can be proved again, which is cheap by assumption, and the results that change are the ones to look at. As for freezing, we haven’t frozen our primitives. We retire them when the evidence demands it, as we did with MD5, SHA-1 and RC4.

**We should verify implementations instead.** We should verify both. Implementation proofs such as HACLS\* and the hax toolchain show that code does what its specification says [79, 13]. They don’t say what the protocol is meant to achieve, and their specifications grow with the code. The libcrux errors in Section 3 sat in such specifications, and they were a risk even though the code was correct: a developer who trusted a specification and changed the code to match would have imported its error [52]. A protocol-level model is where the security goals get written down. The two should meet, with implementation proofs targeting the same definitions that protocol-level claims use.

**Nobody can agree on the definitions.** We manage it for primitives, through open calls, public comment and years of analysis, run by bodies such as NIST and the IRTF’s CFRG. The IETF TLS working group’s formal analysis triage team may already ask the group “for clarification on what security properties are intended for a particular change” [43]. The same process can work for definitions, starting with the few properties most deployed protocols claim.

## 9 What we should reward

If this argument is right, our research priorities should change in four ways.

First, we should build shared, versioned libraries of protocol security definitions in a proof assistant, with a symbolic and a computational version of each property, and argue about them in public the way we argue about primitives. Each definition should come with test cases: known attacks it must flag, such as Lowe’s attack on Needham–Schroeder, KRACK, Selfie and the PQXDH key confusion, and protocols it must accept. A definition that can’t tell these apart is wrong in a way anyone can check.

Second, we should build proved translations from protocol languages to those definitions, symbolic and computational, so a language’s claims can’t drift from its checked theorems.

Third, we should measure legibility directly. Two measures are easy to start with. One is how much text a person has to read to trust a result, counting the model and each definition it invokes. The other is whether engineers who didn’t write a model can find a mistake planted in it, such as a dropped signature check, an extra leak or a constant copied from the wrong function. That’s how mutation testing measures test suites.

Fourth, our conferences should reward work that makes the trusted statement smaller, alongside work that proves more. A paper that cuts what a reader must check from forty pages of definitions to four can add as much assurance as a new proof. Artifact evaluation could ask a reviewer to read the statement of each main theorem and write down what it promises, and standards bodies could treat a protocol’s formal model as its specification of record.

## 10 Conclusion

Cheap proofs move the cost of assurance from proving to reading. A team that receives an AI-generated verification of its protocol still has to understand what the theorems say about the protocol it ships. Today that means reading definitions written for that one analysis, in a language few engineers know, at a length few people will read in full. I’ve argued for protocol languages that a team writes itself as its specification of record and can read in full, that report a claim only when a kernel has checked a theorem behind it, and that build in a small set of named, versioned definitions of protocol goals reviewed by the whole community. Verifpal shows that a language people write and read, with fixed definitions, is practical, and Lean and the protocol libraries now being built on it put the proofs within reach. The language doesn’t have to be Verifpal. Agreeing on its definitions will take years of public argument, and better provers won’t shorten that, so I think we should start now.

**Acknowledgments.** I thank Lucas Meier for thoughtful discussions and feedback.

## References

- [1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. EUROCRYPT 2019, pp. 129–158.
- [2] Anthropic. Formalizing Fermat’s Last Theorem. Blog post, 4 September 2026. <https://www.anthropic.com/news/formalizing-fermats-last-theorem>. Proof repository: <https://github.com/anthropics/fermats-last-theorem>.
- [3] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. IEEE CSF 2010, pp. 107–121.
- [4] Axiom Math. AxiomProver at Putnam 2025. GitHub repository, January 2026. <https://github.com/AxiomMath/Putnam2025>.
- [5] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirović, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. ACM CCS 2018, pp. 1383–1396.

- [6] David Basin, Ralf Sasse, and Jorge Toro-Pozo. Card brand mixup attack: Bypassing the PIN in non-Visa cards by using them for Visa transactions. *USENIX Security 2021*, pp. 179–194.
- [7] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV standard: Break, fix, verify. *IEEE S&P 2021*, pp. 1766–1781.
- [8] Tim Berners-Lee and Noah Mendelsohn, editors. The rule of least power. *W3C TAG Finding*, 23 February 2006.
- [9] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat–Shamir heuristic and applications to Helios. *ASIACRYPT 2012*, pp. 626–643.
- [10] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, et al. Everest: Towards a verified, drop-in replacement of HTTPS. *SNAPL 2017, LIPIcs 71*, pp. 1:1–1:12.
- [11] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. Formal verification of the PQXDH post-quantum key agreement protocol for end-to-end secure messaging. *USENIX Security 2024*, pp. 469–486.
- [12] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. Formal security and functional verification of cryptographic protocol implementations in Rust. *ACM CCS 2025*, pp. 2729–2743.
- [13] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, et al. hax: Verifying security-critical Rust software using multiple provers. *VSTTE 2024, LNCS 15525*, pp. 96–119, 2025. Tool: <https://github.com/cryspen/hax>.
- [14] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. *CRYPTO 2022*, pp. 784–813.
- [15] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing* 5(4):193–207, 2008.
- [16] Colin Boyd and Kai Gellert. A modern view on forward security. *The Computer Journal* 64(4):639–652, 2021.
- [17] Kevin Buzzard. Comment on Matt Baker, “Some early morning thoughts on the OpenAI dump.” 8 October 2026. <https://mattbaker.blog/2026/10/08/some-early-morning-thoughts-on-the-openai-dump/>.
- [18] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *IEEE FOCS 2001*, pp. 136–145.
- [19] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for SMT-assisted proof-oriented programming. *ICSE 2025*, pp. 1755–1767.
- [20] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. SAPIC+: Protocol verifiers of the world, unite! *USENIX Security 2022*, pp. 3935–3952.
- [21] Jason Cohen. Code review at Cisco Systems. In *Best Kept Secrets of Peer Code Review*, Smart Bear, 2006, pp. 63–87.
- [22] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *Journal of Cryptology* 33(4):1914–1983, 2020.
- [23] Graeme Connell and Rolfe Schmidt. Signal Protocol and post-quantum ratchets. *Signal blog*, 2 October 2025. <https://signal.org/blog/spqr/>.
- [24] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning* 46(3–4):225–259, 2011.
- [25] Cas Cremers, Alexander Dax, and Niklas Medinger. Keeping up with the KEMs: Stronger security notions for KEMs and automated analysis of KEM-based protocols. *ACM CCS 2024*, pp. 1046–1060.
- [26] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. *ACM CCS 2017*, pp. 1773–1788.
- [27] Cas Cremers, Samed Düzl , Rune Fiedler, Marc Fischlin, and Christian Janson. BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures. *IEEE S&P 2021*, pp. 1696–1714.
- [28] Cas Cremers. Examining indistinguishability-based security models for key exchange protocols: The case of CK, CK-HMQV, and eCK. *ASIACCS 2011*, pp. 80–91.
- [29] Quang Dao, Jim Miller, Opal Wright, and Paul Grubbs. Weak Fiat–Shamir attacks on modern proof systems. *IEEE S&P 2023*, pp. 199–216.
- [30] Quang Dao. Name the reductions in security bounds instead of quantifying over them. *VCVio pull request #707*, merged 13 September 2026. <https://github.com/Verified-zkEVM/VCVio/pull/707>.

- [31] Grit Denker and Jonathan Millen. CAPSL integrated protocol environment. DISCEX 2000, pp. 207–221.
- [32] Nir Drucker and Shay Gueron. Selfie: Reflections on TLS 1.3 with PSK. *Journal of Cryptology* 34(3):27, 2021.
- [33] Stefan Dziembowski, Grzegorz Fabiański, Daniele Micciancio, and Rafał Stefański. Computationally-sound symbolic cryptography in Lean. *Cryptology ePrint Archive*, Paper 2025/1700.
- [34] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. Owl: Compositional verification of security protocols via an information-flow type system. *IEEE S&P* 2023, pp. 1130–1147.
- [35] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David Basin. A spectral analysis of Noise: A comprehensive, automated, formal analysis of Diffie–Hellman protocols. *USENIX Security* 2020, pp. 1857–1874.
- [36] Sven Hammann. SAPIC: builtin equation variables `x.1/x.2` 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>.
- [37] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. *ACM CCS* 2005, pp. 2–15.
- [38] Andreas V. Hess, Sebastian Mödersheim, Achim D. Brucker, and Anders Schlichtkrull. PSPSP: A tool for automated verification of stateful protocols in Isabelle/HOL. *Journal of Computer Security* 33(6):425–469, 2025.
- [39] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for unbounded verification of privacy-type properties. *Journal of Computer Security* 27(3):277–342, 2019.
- [40] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise\*: A library of verified high-performance secure channel protocol implementations. *IEEE S&P* 2022, pp. 107–124.
- [41] Ross Horne and Sjouke Mauw. Discovering ePassport vulnerabilities using bisimilarity. *Logical Methods in Computer Science* 17(2):24, 2021.
- [42] Thomas Hubert, Rishi Mehta, Laurent Sartran, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature* 651:607–613, 2026.
- [43] IETF TLS Working Group. Formal Analysis Triage Team (FATT) process overview. <https://github.com/tlswg/tls-fatt>.
- [44] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. *ACM CCS* 2019, pp. 2165–2180.
- [45] Simon Jeanteur, Lorenzo Veronese, Magdalena Solitro, and Matteo Maffei. LeanDY: Type-based and trace-based symbolic protocol verification in Lean. *arXiv:2607.03406*, 2026.
- [46] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. seL4: Formal verification of an OS kernel. *SOSP* 2009, pp. 207–220.
- [47] Martin Kleppmann. Prediction: AI will make formal verification go mainstream. Blog post, 8 December 2025. <https://martin.kleppmann.com/2025/12/08/ai-formal-verification.html>.
- [48] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. *IEEE EuroS&P* 2019, pp. 356–370.
- [49] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world. *INDOCRYPT* 2020, pp. 151–202. First released as *Cryptology ePrint Archive*, Paper 2019/971.
- [50] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. *IEEE EuroS&P* 2017, pp. 435–450.
- [51] Nadim Kobeissi. From toy to instrument: Seven years of Verifpal. *Cryptology ePrint Archive*, Paper 2026/1654.
- [52] Nadim Kobeissi. Verification theatre: False assurance in formally verified cryptographic libraries. *Cryptology ePrint Archive*, Paper 2026/192.
- [53] Lean FRO. Validating a Lean proof. *The Lean Language Reference*, version 4.34.0, 2026. <https://lean-lang.org/doc/reference/4.34.0/ValidatingProofs/>.
- [54] Felix Linker, Ralf Sasse, and David Basin. A formal analysis of Apple’s iMessage PQ3 protocol. *USENIX Security* 2025, pp. 5015–5034.
- [55] Gavin Lowe. A hierarchy of authentication specifications. *IEEE CSFW* 1997, pp. 31–43.
- [56] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security* 6(1–2):53–84, 1998.

- [57] Yiping Ma, Yu-Lin Tsai, Mayank Rathee, Deevashwer Rathee, François Dupressoir, Pierre-Yves Strub, and Raluca Ada Popa. ShannonProver: Towards automating formal cryptographic proofs. arXiv:2607.02847, 2026.
- [58] Math Inc. Introducing Gauss, an agent for autoformalization. September 2025. <https://www.math.inc/gauss>.
- [59] Simon Meier, Cas Cremers, and David Basin. Strong invariants for the efficient construction of machine-checked protocol security proofs. IEEE CSF 2010, pp. 231–245.
- [60] Kevin Milner. Soundness/input validation/general correctness fixes for regular proving, for diff mode, and for SAPIC. Tamarin prover pull requests #948, #949 and #950, 17 September 2026; first part merged as #954, 7 October 2026. <https://github.com/tamarin-prover/tamarin-prover/pull/954>.
- [61] Sebastian Mödersheim. Algebraic properties in Alice and Bob notation. ARES 2009, pp. 433–440.
- [62] Leonardo de Moura. Postmortem for kernel soundness bug #14576. Blog post, 1 August 2026. <https://leodemoura.github.io/blog/2026-8-1-postmortem-for-kernel-soundness-bug-14576/>.
- [63] Leonardo de Moura. Signal Shot: The platform is ready. Blog post, 20 April 2026. <https://leodemoura.github.io/blog/2026-4-20-signal-shot-the-platform-is-ready/>.
- [64] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. Communications of the ACM 58(4):66–73, 2015.
- [65] NIST. Call for additional digital signature schemes for the post-quantum cryptography standardization process. September 2022.
- [66] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. December 2016. <https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf>.
- [67] OpenAI. Mathematical manuscripts and proof artifacts: repository README, scope notes and history.md, 6–8 October 2026. Accessed 10 October 2026. <https://github.com/openai/math>.
- [68] Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. miniF2F-Lean revisited: Reviewing limitations and charting a path forward. NeurIPS 2025. arXiv:2511.03108.
- [69] Trevor Perrin. The Noise protocol framework, revision 34. 2018. <https://noiseprotocol.org/noise.html>.
- [70] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange. CT-RSA 2021, pp. 148–176.
- [71] Eric Rescorla. The Transport Layer Security (TLS) protocol version 1.3. RFC 9846, July 2026.
- [72] Sophie Schmieg. Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK. Cryptology ePrint Archive, Paper 2024/523.
- [73] Bas Spitters. CatCrypt: From Rust to cryptographic security in Lean. Cryptology ePrint Archive, Paper 2026/604.
- [74] Douglas Stebila. Security analysis of the iMessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/357.
- [75] Nick Sullivan and Christopher A. Wood. Guidelines for writing cryptography specifications. Internet-Draft draft-irtf-cfrg-cryptography-specification-03, July 2026.
- [76] Devon Tuma, Quang Dao, James Waters, Alexander Hicks, and Nicholas Hopper. VCVio: Verified cryptography in Lean via oracle effects and handlers. Cryptology ePrint Archive, Paper 2026/899.
- [77] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. ACM CCS 2017, pp. 1313–1328.
- [78] Théophile Wallez and Cas Cremers. Bob DyLean: A framework for the symbolic analysis of cryptographic protocols in Lean. Cryptology ePrint Archive, Paper 2026/1493.
- [79] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACl\*: A verified modern cryptographic library. ACM CCS 2017, pp. 1789–1806.