Verifpal: Cryptographic Protocol Analysis for Students and Engineers

Nadim Kobeissi
Symbolic Software, NYU Paris

Abstract

Contemporary research in symbolic formal verification has led to confirming security guarantees (as well as finding attacks) in secure channel protocols such as TLS and Signal. However, formal verification in general has not managed to significantly exit the academic bubble.

Verifpal is new software for verifying the security of cryptographic protocols that aims is to work better for real-world practitioners, students and engineers without sacrificing comprehensive formal verification features. In order to achieve this, Verifpal introduces a new, intuitive language for modeling protocols that is easier to write and understand than the languages employed by existing tools. Its formal verification paradigm is also designed explicitly to provide protocol modeling that avoids user error. By modeling principals explicitly and with discrete states, Verifpal models are able to be written in a way that reflects how protocols are described in the real world.

At the same time, Verifpal is able to model protocols under an active attacker with unbounded sessions and fresh values, and supports queries for advanced security properties such as forward secrecy or key compromise impersonation. Verifpal has already been used to verify security properties for Signal, Scuttlebutt, TLS 1.3 and other protocols. It is a community-focused project, and available under a GPLv3 license.

1 Introduction

Internet communications rely on a handful of protocols, such as Transport Layer Security (TLS), SSH and Signal, in order to keep user data confidential. These protocols often aim to achieve ambitious security properties (such as post-compromise security [1]) across complex use-cases (such as support for multiple devices [2].) Given the broad set of operations and states supported by these protocols, verifying that they *do* indeed achieve their desired security goals across all use-case scenarios has proven to be non-trivial.

Automated formal verification tools have seen an encouraging success in helping to model the security of these protocols. Recently, the Signal secure messaging protocol [3], the TLS 1.3 web encryption standard [4], the 5G wireless communication standard [5, 6], the Scuttlebutt decentralized messaging protocol [7], the Bluetooth standard [7], the Let's Encrypt certificate issuance system [8, 9], the Noise protocol framework [10, 11, 12] and the WireGuard [13] Virtual Private Network (VPN) protocol [14] have all been analyzed using automated formal verification.

Despite this increase in the usage of formal verification tools, and despite the success obtained with this approach, automated formal verification technology remains unused outside certain specific realms of academia: an illustrative fact is that *almost all* of the example results cited above have, as a co-author, one of the designers of the automated formal verification tool that was used to obtain the research result. We conjecture that this lack of adoption is leading an increase in the number of weaknesses in cryptographic protocols: in the case of TLS, protocol designers did not use formal verification technology in the protocol's design phase up until TLS 1.3, and that was only due to automated formal verification helping discover a large number of attacks in TLS 1.2 and below [15, 16, 4], and was, again, only accomplished via collaboration with the designers of the formal verification tools themselves.

1.1 A Simpler Approach to Symbolic Verification

Extensive experience with automated formal verification tools has led us to the hypothesis that the prerequisite knowledge, modeling languages and structure in which the tools formalize their results are a significant barrier against wider adoption. Verifpal is an attempt to overcome this barrier. Building upon contemporary research in symbolic formal verification, Verifpals main aim is to appeal more to real-world practitioners, students and engineers without sacrificing comprehensive formal verification features. Verifpal has four main design goals/features:

An intuitive language for modeling protocols. Verifpal's internal logic relies on the deconstruction and reconstruction of abstract terms, similar to existing symbolic verification tools. However, it reasons about the protocol model with *explicit principals*: Alice and Bob exist, they have independent states, they know certain values and perform operations with cryptographic primitives. They send messages to each other over the network, and so on. The Verifpal language is meant to illustrate protocols close to how one may describe them in an informal conversation, while still being precise and expressive enough for formal modeling.

Modeling that avoids user error. Verifpal does not allow users to define their own cryptographic primitives. Instead, it comes with built-in cryptographic functions: **`ENC`** and **`DEC`** representing encryption and decryption, **`AEAD_ENC`** and **`AEAD_DEC`** representing authenticated encryption and decryption, **`DH`** and **`SIGN`** representing asymmetric primitives, etc. — this is meant to remove the potential for users to define fundamental cryptographic operations incorrectly. Verifpal also adopts a global name-space for all constants and does not allow constants to be redefined or assigned to one another. This enforces models that are clean and easy to follow.

Analysis output that's easy to understand. Existing tools provide *"attack traces"* that illustrate a deduction using session-tagged values in a chain of symbolic deconstructions. Verifpal follows a different approach: as it is analyzing a model, it outputs notes on which values it is able to deconstruct, conceive of, or reconstruct. When a contradiction is found for a query, the result is related in a readable format that ties the attack to a real-world scenario. This is done by using terminology to indicate how the attack could have been possible, such as through a mayor-in-the-middle on ephemeral keys.

Integration with the developer's workflow. Verifpal comes with a Visual Studio Code extension that offers syntax highlighting and, soon, live query verification within Visual Studio Code, allowing developers to obtain insights on their model as they are writing it.

Verifpal is founded upon these four design goals, without sacrificing the ability to handle advanced security analysis: the tool is able to verify the security of complex protocols, such as Signal, and query for complex attack scenarios such as post-compromise security and key compromise impersonation, across unbounded session executions of the protocol and with fresh values not being shared across sessions. By giving practitioners this powerful symbolic analysis paradigm in an intuitive package, Verifpal stands a chance at making symbolic formal verification a staple in the diet of any protocol designer.

Verifpal does not aim to replace existing tools: for example, it will likely never support certain features that are easy to accomplish in ProVerif, such as the definition of custom cryptographic primitives (through construction, deconstruction and rewrite rules). Finally, it is important to note that Verifpal currently does not come with a full soundness theorem — this is further discussed in §3.2.

1.2 Related Work

Verifpal is pretty late to the party, arriving roughly two decades since automated formal verification became a research focus. Here, we outline some of the more pertinent formal verification tools, use cases and broader methodologies this research area has seen, and which Verifpal aims to supersede in terms of accessibility and real-world usability.

1.2.1 Other Symbolic Verification Tools and their Use Cases

Verifpal is heavily inspired by the ProVerif [17, 18] protocol verifier, designed by Bruno Blanchet. It does not construct all terms out of Horn clauses [19] in the way that ProVerif does, and it does not use the applied pi-calculus [20] as its modeling language. However, its analysis logic is inspired by ProVerif and is similarly based on the Dolev-Yao model [21]. ProVerif's construction/deconstruction/rewrite logic is also mirrored in Verifpal's own design. ProVerif has been recently used to formally verify TLS 1.2 and TLS 1.3 [4], Let's Encrypt's ACME certificate issuance protocol [9], the Signal secure messaging protocol [3], the Noise protocol framework [10], the Plutus network filesystem [22], e-voting protocols [23, 24, 25, 26], FIDO [27] and many more use cases.

The Tamarin [28] protocol prover also works under the symbolic model, but derives the progeny of its analysis from principals' state transitions rather than from the viewpoint of an attacker observing and manipulating network messages. It is also different from ProVerif in its analysis style, and its modeling language is unique within the domain. Tamarin has been recently used to formally verify Scuttlebutt [7], TLS [29], WireGuard [30], 5G [5, 6], the Noise protocol framework [12, 11], multiple e-voting protocols [31, 32] and many more use cases.

Scyther[1] [34, 35], whose authors also work on Tamarin, offers unbounded verification with guarantees of termination but uses a more accessible and explicit modeling language than Tamarin. Scyther has been used to analyze IKEv1 and IKEv2 [36] (used in IPSec), a large amount of Authenticated Key Exchange (AKE) protocols such as HMQV, UM and NAXOS [37], and to check for *"multi-protocol attacks"* [38]. Research focus seems to be moving towards Tamarin, but Scyther is still sometimes used.

[1]Not to be confused with the bug/flying-type Pokémon of the same name, which, despite its *"ninja-like agility and speed"* [33], does not appear to have published work in formal verification.

AVISPA [39]'s modeling language is somewhat similar to Verifpal's: both have a focus on describing *"actors"* with *"roles"*, and explicitly attempt to allow the user to illustrate the protocol intuitively, as if describing actors in a theatrical play. Despite this, work on AVISPA seems to have stalled in 2005. The tool does not seem to be actively developed and appears to have been superseded by the AVANTSSAR [40] project. In 2016, a new authentication protocol was designed and prototyped with AVISPA [41]. In 2011, Facebook's *Connect* single sign-on protocol was modeled with AVISPA [42].

FDR [43] is not specifically a protocol verifier, but rather a refinement and equivalence checker for processes written using the Communicating Sequential Processes language [44]. CSP can be used to illustrate processes that capture secure channel protocols, and security queries can be illustrated as refinements or properties resulting from these processes. In that sense, FDR can act as a protocol verifier. In 2014, an RFID authentication protocol was formally verified using FDR [45].

A performance analysis of symbolic formal verification tools by Lafourcade and Pus [46], conducted in 2015, as well as a preceding study by Cremers and Lafourcade in 2011 [47] found mixed results, with ProVerif coming out on top more often than not.

ProVerif and Tamarin appear to the the current titans of the symbolic verification space, and they tend to compliment each other due to diverging design decisions: for example, ProVerif does not require human assistance for verification, but sometimes may not terminate and may also sometimes find false attacks (although it is proven not to miss attacks.) Tamarin, on the other hand, always terminates, but may require human assistance, therefore making it less suited for fully automated analysis — in some cases, fully automated analysis can be necessary to achieve certain research goals [10].

1.2.2 Formal Verification Paradigms

Verifpal, as well as all of the tools cited above, analyze protocols in the *symbolic* model. There are other methodologies in which to formally verify protocols, including the computational model or, for example, by using SMT solvers. We choose the symbolic model as the focus of our research due to its academic success record in verifying contemporary protocols and due to its propensity for fully automated analysis. It should be noted, however, that more precise analysis can often be achieved using the aforementioned formal verification methodologies.

Traditionally, *symbolic* models are favored the security protocol verification community for ease of automated analysis. Cryptographers, on the other hand, prefer to use *computational* models and do their proofs by hand. A full comparison between these styles [48] is beyond the scope of this work; here we briefly outline their differences in terms of the tools currently used in the field.

ProVerif, Tamarin, AVISPA and other tools analyze symbolic protocol models, whereas tools such as CryptoVerif [49] verify computational models. The input languages for both types of tools can be similar. However, in the symbolic model, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). In ProVerif, for example, the attacker is an arbitrary process running in parallel with the protocol, which can read and write messages on public channels and can manipulate them symbolically.

In the computational model, messages are concrete bitstrings. Freshly generated nonces and keys are randomly sampled bitstrings that the attacker can guess with some probability (depending on their length). Encryption and decryption are functions on bitstrings to which we may associate standard cryptographic assumptions such as IND-CCA. The attacker is a probabilistic polynomial-time process running in parallel.

Queries can also be modeled similarly in symbolic and computational models as between events, but analysis differs: in symbolic analysis, we typically ask whether the attacker can derive a secret, whereas in the computational model, we ask whether it can distinguish a secret from a random bitstring.

The analysis techniques employed by the two tools are quite different. Symbolic verifiers search for a protocol trace that violates the security goal, whereas computational model verification tries to construct a cryptographic proof that the protocol is equivalent (with high probability) to a trivially secure protocol. Symbolic verifiers are easy to automate, while computational model tools, such as CryptoVerif, are semi-automated: it can search for proofs but requires human guidance for non-trivial protocols.

Recently, the F* programming language [50], which exports type definitions to the Z3 theorem prover [51], has been used to produce an implementation of the Signal secure messaging protocol that is formally verified for functional correctness at the *level of the implementation itself* [52]. Microsoft Research's Project Everest [53] is attempting to accomplish the same thing for HTTPS, also using F* [54].

1.3 Contributions

W present the following contributions:

- Introduction of Verifpal and comparison against existing automated verification tools in the symbolic model (§1).
- The Verifpal modeling language with justifications for the language's design choices as well as examples (§2).
- Verifpal's protocol analysis logic and whether we can be certain that Verifpal will not miss an attack on a protocol model (§3).
- Modeling of the Signal secure messaging protocol using Verifpal, with queries for confidentiality, authentication, forward secrecy and key compromise impersonation (§4).

Verifpal is free and open source software, available for download at `https://verifpal.com`.

2 The Verifpal Language

Verifpal's language is meant to be simple while allowing the user to capture comprehensive protocols. When describing a protocol in Verifpal, we begin by defining whether the model will be analyzed under a *passive* or *active* attacker. Then, we define the *principals* engaging in activity other than the attacker. These could be Alice and Bob, a Server and one or more Clients, etc.

⟨*verifpal*⟩ ::= ⟨*attacker*⟩ ⟨*principal*⟩ (⟨*principal*⟩ | ⟨*message*⟩)+ ⟨*queries*⟩
⟨*attacker*⟩ ::= ‘`attacker [`’ (‘`active`’ | ‘`passive`’) ‘`]`’
⟨*principal*⟩ ::= ‘`principal`’ ⟨*string*⟩ ‘`[`’ (⟨*knows*⟩ | ⟨*generates*⟩ | ⟨*assignment*⟩)+ ‘`]`’
⟨*knows*⟩ ::= ‘`knows `’ (‘`private`’ | ‘`public`’) ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)*
⟨*generates*⟩ ::= ‘`generates `’ ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)*
⟨*assignment*⟩ ::= ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)* ‘` = `’ (⟨*primitive*⟩ | ⟨*equation*⟩)
⟨*message*⟩ ::= ⟨*string*⟩ ‘ ’ ⟨*string*⟩ ‘`: `’ (⟨*constant*⟩ | ⟨*guardedConstant*⟩) (‘`,`’ (⟨*constant*⟩ | ⟨*guardedConstant*⟩))*
⟨*queries*⟩ ::= ‘`queries[`’ (⟨*confidentialityQuery*⟩ | ⟨*authenticationQuery*⟩)* ‘`]`’
⟨*confidentialityQuery*⟩ ::= ‘`confidentiality? `’ ⟨*constant*⟩
⟨*authenticationQuery*⟩ ::= ‘`authentication? `’ ⟨*string*⟩ ‘ ’ ⟨*string*⟩ ‘`: `’ ⟨*constant*⟩
⟨*constant*⟩ ::= ⟨*string*⟩
⟨*guardedConstant*⟩ ::= ‘`[`’ ⟨*constant*⟩ ‘`]`’
⟨*primitive*⟩ ::= ⟨*primitiveName*⟩ ‘`(`’ (⟨*constant*⟩ | ⟨*primitive*⟩ | ⟨*equation*⟩) (‘`,`’ (⟨*constant*⟩ | ⟨*primitive*⟩ | ⟨*equation*⟩))* ‘`)`’ [‘`?`’]
⟨*equation*⟩ ::= ⟨*constant*⟩ ‘`^`’ ⟨*constant*⟩
⟨*primitiveName*⟩ ::= ‘`HASH`’ | ‘`HKDF`’ | ‘`AEAD_ENC`’ | ‘`AEAD_DEC`’ | ‘`ENC`’ | ‘`DEC`’ | ‘`HMAC`’ | ‘`HMACVERIF`’ | ‘`SIGN`’ | ‘`SIGNVERIF`’
⟨*string*⟩ ::= ⟨*stringElement*⟩+
⟨*stringElement*⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | _ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 1: Verifpal language syntax.

Once we have described the actions of more than one principal, it's time to illustrate the *messages* being sent across the network. Then, after having illustrated the principals' actions and their messages, we may finally describe the questions, or *queries* (can a passive attacker read the first message that Alice sent to Bob? Can Alice can be impersonated by an active attacker?) that we will ask Verifpal.

2.1 Principals

First, we must define what kind of attacker Verifpal will use to analyze our model. The syntax for this is pretty simple: **`attacker`**`[passive]` indicates a passive attacker, while **`attacker`**`[active]` indicates an active attacker.

We may then declare a principal `Alice` which knows the public constants `c0, c1` and the private constant `m1`, which will act as the secret message Alice will want to send to Bob later. Since `c0` and `c1` are declared as known publicly, they are immediately also known to the attacker. The same, of course, is not true of `m1`. Alice also *generates* a random value `a`. She will use this value as her private key.

New Principal: Alice

```
principal Alice[
  knows public c0, c1
  knows private m1
  generates a
]
```

```
New Principal:  Bob

principal Bob[
  knows public c0, c1
  knows private m2
  generates b
  gb = G^b
]
```

Notice how Bob also calculates `gb = G^b`. Here, `gb` is Bob's public Diffie-Hellman key, while `G^b` quite plainly indicates the standard Diffie-Hellman exponentiation g^b. Later, Alice will be able to write `gb^a`, which is how we illustrate g^{ba} in Verifpal.

2.2 Constants

In the above examples, `c0, c1, m1, m2, b, gb` are all *constants*. Certain rules apply on constants in Verifpal:

- **Immutability.** Once assigned, constants cannot be reassigned.
- **Global name-space.** If Bob declares or assigns some constant `c`, Alice cannot declare a constant `c` even if Bob declares or assigns his constant privately.
- **No referencing.** Constants cannot be assigned to other constants, but only to primitives or equations.

These rules exist in order to encourage the user to write Verifpal models that will hopefully be cleaner and easier to read. Let's summarize the different ways that exist to declare constants, and how they differ from one another:

- **`knows`**: A principal may be described as having prior knowledge of a constant. The qualifiers **`private`** and **`public`** describe whether this constant that they have knowledge of is supposed to be considered known by everyone else (including the attacker) or just by them. Constants declared this way are considered to be, well, constant, across every execution of the protocol (i.e. they are not unique for every different time the protocol is executed).
- **`generates`**: This allows a principal to describe a *"fresh"* value, i.e. a value that is re-generated every time the protocol is executed. A good example of this could be an ephemeral private key. Such values (and all values derived using these values) are not kept between different protocol session executions.
- **Assignment**: A constant may be declared by assigning it to the result of a primitive or equation expression. Remember, however, that constants may not be assigned to other constants.

2.3 Primitives

In order to describe cryptographic protocols, we will of course need cryptographic primitives.

Simple Example Protocol

```
attacker[active]
principal Bob[]
principal Alice[
  generates a
  ga = G^a
]
Alice -> Bob: ga
principal Bob[
  knows private m1
  generates b
  gb = G^b
  e1 = AEAD_ENC(ga^b, m1, gb)
]
Bob -> Alice: gb, e1
principal Alice[
  e1_dec = AEAD_DEC(gb^a, e1, gb)?
]
```

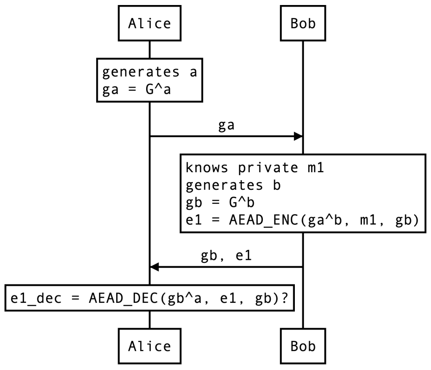

Figure 2: A complete example model of a simple protocol is shown on the left. On the right, a helpful diagram is provided to illustrate how modeling in Verifpal works. The diagram on the right is not part of Verifpal's modeling language and is simply provided here as a visual aid.

In Verifpal, cryptographic primitives are essentially *"perfect"*. That is to say, hash functions are perfect one way functions, and not susceptible to something like length extension attacks. It is also not possible to model for, say, encryption primitives that use 40-bit keys, which could be guessed easily, since encryption functions are perfect pseudo-random permutations, and so on. This is a common element of formal verification tools that function under the symbolic model.

2.3.1 Hashing Primitives

Verifpal offers the following hashing primitives, which aim to capture classical cryptographic hashing, keyed hashing and hash-based key derivation:

- **HASH**(a, b...): x. Secure hash function, similar in practice to, for example,

Simple Example Protocol: Queries

```
queries[
  confidentiality? e1
  confidentiality? m1
  authentication? Bob -> Alice: e1
]
```

Figure 3: Queries for confidentiality and authentication checks on the model described in Figure 2.

BLAKE2s [55]. Takes an arbitrary number of input arguments ≥ 1, and returns one output.

- `HMAC(key, message): hash`. Keyed hash function. Useful for message authentication and for some other protocol constructions.
- `HMACVERIF(HMAC(k, m), HMAC(k, m)): x`. Checks the equality of two HMAC primitive outputs.
- `HKDF(salt, ikm, info): a, b...`. Hash-based key derivation function inspired by the Krawczyk HKDF scheme [56]. Produces an arbitrary number of outputs ≥ 1.

2.3.2 Encryption Primitives

Verifpal offers the following encryption primitives, which aim to capture unauthenticated encryption, and authenticated encryption with associated data:

- `ENC(key, plaintext): ciphertext`. Symmetric encryption, similar for example to AES-CBC or to ChaCha20.
- `DEC(key, ENC(key, plaintext)): plaintext`. Symmetric decryption.
- `AEAD_ENC(key, plaintext, ad): ciphertext`. Authenticated encryption with associated data. `ad` represents an additional payload that is not encrypted, but that must be provided exactly in the decryption function for authenticated decryption to succeed. Similar for example to AES-GCM or to ChaCha20-Poly1305.
- `AEAD_DEC(k, AEAD_ENC(k, p, ad), ad): p`. Authenticated decryption with associated data.

2.3.3 Signature Primitives

Verifpal offers a simple signing primitive with a corresponding signature verification function:

- `SIGN(key, message): signature`. Classic signature primitive. Here, `key` is a private key, for example `a`.
- `SIGNVERIF(G^k, m, SIGN(k, m)): m`. Verifies if signature can be authenticated. If key `k` was used for `SIGN`, then `SIGNVERIF` will expect `G^k` as the key value.

2.3.4 Checked Primitives

In Verifpal, `AEAD_DEC`, `HMACVERIF`, and `SIGNVERIF` are *"checkable"* primitives: if we add a question mark (`?`) after one of these primitives, then model execution will abort should `AEAD_DEC` fail authenticated decryption, or should `HMACVERIF` fail to find its two provided inputs equal, or should `SIGNVERIF` fail to verify the signature against the provided message and public key. For example: `SIGNVERIF(k, m, s)?` makes this instantiation of `SIGNVERIF` a *"checked"* primitive.

If we are analyzing under a passive attacker, then Verifpal will only execute the model once. Therefore, if a checked primitive fails, the entire verification procedure will abort.

Under an active attacker, however, Verifpal is forced to execute the model once over for every possible permutation of the inputs that can be affected by the attacker. Therefore, a failed checked primitive may not abort all executions. Also, messages obtained before the failure of the checked primitive are still valid for analysis in future sessions.

Checked primitives are supposed to offer an elegant way to express session abortion in the event of an unexpected failure. This could be, for example, a client finding an invalid signature for a server certificate in a Verifpal model of TLS 1.3.

2.4 Equations

Equations are special expressions intended to capture public key generation (useful for both Diffie-Hellman and signatures), as well as shared secret agreement (useful for Diffie-Hellman).

As we saw earlier, `G^a` indicates the public key obtained from value a. This public key can be used both for signing primitives as well as for Diffie-Hellman shared secret agreement. Let's look at some other example equations in Verifpal:

Example Equations

```
principal Server[
  generates x
  generates y
  gx = G^x
  gy = G^y
  gxy = gx^y
  gyx = gy^x
]
```

In the above, `gxy` and `gyx` are considered equivalent by Verifpal. In Verifpal, all equations must have the constant `G` as their root generator. This mirrors Diffie-Hellman behavior. Furthermore, all equations can only have two constants (`a^b`), but as we can see above, equations can be built on top of other equations (as in the case of `gxy` and `gyx`).

2.5 Messages and Guarded Constants

Sending messages over the network is simple. Only constants may be sent within messages:

Example Messages

```
Alice -> Bob: ga, e1
Bob -> Alice: [gb], e2
```

In the first message above, Alice is the sender and Bob is the recipient. Notice how Alice is sending Bob her long-term public key `ga = G^a`. An active attacker could intercept `ga` and replace it with a value that they control. But what if we want to model our protocol such that Alice has pre-authenticated Bob's public key `gb = G^b`? This is where *guarded constants* become useful.

In the second message from the above example, we see that, `gb` is surrounded by brackets (`[]`). This makes it a *"guarded"* constant, meaning that while an active attacker can still read it, they cannot tamper with it. In that sense it is guarded against the active attacker.

Guarded constants are intended to provide an elegant way to express certain properties, such as mutual authentication of public keys, and the out-of-band authentication of data streams or pre-shared key material.

2.6 A Simple Complete Example

Figure 2 provides a full model of a naïve protocol where Alice and Bob only ever exchange unauthenticated public keys (`G^a` and `G^b`). Bob then proceeds to send an encrypted message to Alice using the derived Diffie-Hellman shared secret to encrypt the message. We then want to ask Verifpal three questions:

1. Can the attacker obtain the ciphertext?
2. Can the attacker obtain the plaintext?
3. Can the attacker impersonate Bob and deliver a tampered ciphertext to Alice that nevertheless still authenticates?

Figure 3 shows what these queries would look like in Verifpal. While simple, the Verifpal language is sufficiently expressive in order to elegantly capture complex protocols, such as Signal, as we will illustrate in §4.

3 Analysis in Verifpal

Verifpal is a protocol verifier; unlike some other automated formal verification tools [49], it does not produce game-based proofs of the protocols that it analyzes. Instead, it digests models representing the execution of a protocol under a very specific scenario enacted by principals that act in a specific way. Verifpal's goal is to then attempt to find contradictions to the queries presented by the user. For example, the second query in Figure 3 is understood by Verifpal as, essentially, the user telling them: *"I bet you can't obtain `m1`!"*

Authentication queries are trickier than confidentiality queries. In the example authentication query shown in Figure 3, we ask: *"if Bob successfully decrypts and authenticates `e1`, does that necessarily mean that Alice sent `e1` to Bob?"* The implication is that if the attacker was able to successfully convince Bob to validate the decryption of `e1`, then an impersonation attack could have occurred where the attacker was able to impersonate Alice. Authentication queries rely heavily on Verifpal's notion of *"checked"* or *"checkable"* primitives, as seen in §2.3.4.

Intuitively, the goal of authentication queries is to ask whether Bob will rely on some value `e1` in an important protocol operation (such as signature verification or authenticated decryption) if and only if he received that value from Alice. If Bob is successful in using `e1` for signature verification or a similar operation without it having been necessarily sent by Alice, then authentication is violated for `e1`, and the attacker was able to impersonate Alice in communicating that value. Note that we don't check for the authentication of plaintext `m1` — that is because `m1` is only obtainable by Bob once decryption succeeds, which only happens if **`AEAD_DEC`** is successfully re-writable back into the input values to **`AEAD_ENC`**, i.e. if the primitive passes the check.

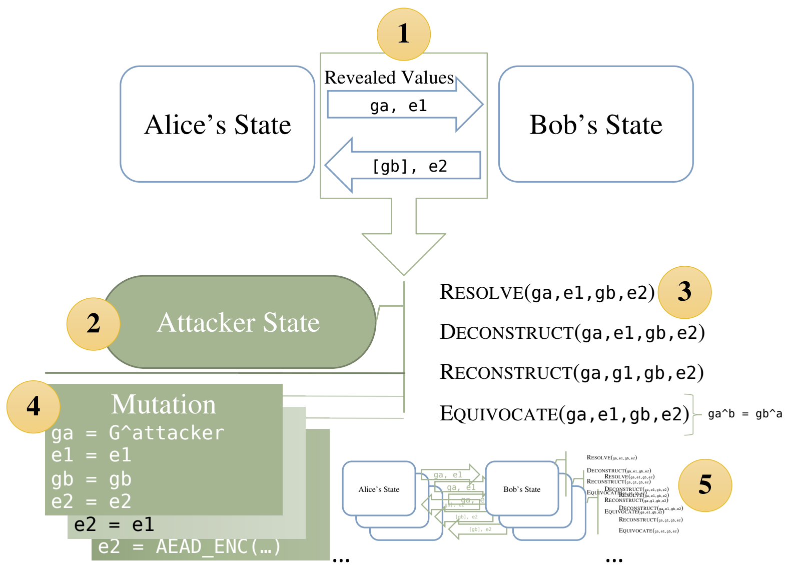

Figure 4: Verifpal analysis methodology.

3.1 Analysis Methodology

Verifpal's active attacker analysis methodology (Figure 4) follows a simple set of procedures and algorithms. The overall process is comprised of five phases:

1. **Gather values.** Attacker passively observes a protocol execution and gathers all values shared publicly between principals.

2. **Insert learned values into attacker state.** Attacker's state ($\mathcal{V}_A$) obtains newly learned values.

3. **Apply transformations.** Attacker applies the four main *"transformations"* on all obtained values (these transformations are detailed below.)

4. **Prepare mutations for next session.** If the attacker has learned new values due to the transformations executed in the previous step, they create a combinatorial table of all possible value substitutions, and from that, derive a set of all possible value substitutions across future executions of the protocol on the network.

5. **Iterate across protocol mutations.** Attacker proceeds to execute the protocol across sessions, each time *"mutating"* the execution by mayor-in-the-middling a value. Attacker then returns to step 1 of this list. The process continues so long as the attacker keeps learning new values.

After each phase, Verifpal checks to see if it has found a contradiction to any of the queries specified in the model and informs the user if such a contradiction is found. The four main transformations mentioned above are the following:

- RESOLVE. Resolves a certain constant to its assigned value (for example, a primitive or an equation). Executed on $\mathcal{V}_A$, the set of all values known by the attacker.

- Deconstruct. Attempts to deconstruct a primitive or an equation. In order to deconstruct a primitive, the attacker must possess sufficient values to satisfy the primitive's rewrite rule. For example, the attacker must possess `k` and `e` in order to obtain `m` by deconstructing `e = ENC(k, m)` with `k`. In order to deconstruct an equation, the attacker must similarly possess all but one private exponent. Executed on $\mathcal{V}_A$, the set of all values known by the attacker.

- Reconstruct. Attempts to reconstruct primitives and equations given that the attacker possesses all of the component values. Executed on $\mathcal{V}_A$, the set of all values known by the attacker, as well as on $\mathcal{V}_P$, the values known by the principal whose state is currently being evaluated by the attacker.

- Equivocate. Determines if the attacker can reconstruct or equivocate any values within $\mathcal{V}_P$ from $\mathcal{V}_A$. If so, then these equivalent values are added to $\mathcal{V}_A$.

Verifpal's goal is to obtain as many values as is logically possible from their viewpoint as an attacker on the network. As a passive attacker, Verifpal can only do this by deconstructing the values made available as they are shared between principals, and potentially reconstructing them into different values. As an active attacker, Verifpal can modify unguarded constants as they cross the network. Each modification could result in learning new values, so an unbounded number of modifications can occur over an unbounded number of protocol executions. *"Fresh"* (i.e. generated) values are not kept across different protocol executions, as they are assumed to be different for every session of the protocol.

An active attacker can also generate their own values, such as a key pair that they control, and fabricate new values that they use as substitutes for any unguarded constants sent between principals. If, during a protocol execution, a checked primitive fails, that session execution is aborted and the attacker moves on to the next one. However, values obtained thus far in that particular session execution are kept.

Verifpal also keeps track of which values are used where, the path a value takes until it arrives into the state of a principal, and who first declared or generated a value. This information is used in order to analyze for contradictions to authentication queries.

3.2 Soundness of Results

Verifpal has so far been used in order to model TLS, Signal, Scuttlebutt, Telegram, ProtonMail and some other protocols. So far, all of its results have been in line with previous analyses of these protocols. But anecdotal evidence is not sufficient in order to declare with full confidence that Verifpal qualifies as a proven formal verification framework.

In order for Verifpal to qualify as a mature formal verification framework, it must provide a soundness theorem in which it can demonstrate that its methodology cannot miss an attack. A formal soundness theorem is currently a work in progress, and is expected to be completed as the Verifpal tool evolves and matures during real-world user testing. In this section, we nevertheless present an outline of Verifpal's formal analysis methodology, such that we can say with a high degree of confidence that:

- If an attacker is unable to obtain a value `m`, then `m` is necessarily confidential for the protocol described in the Verifpal model.

- If an attacker cannot find more than one way in which value `e` can be communicated between principals `A` and `B` such that `B` later employs `e` as an argument to a rewrite-capable primitive or equation, then `e` is necessarily authenticated under `A` $\rightarrow$ `B` for the protocol described in the Verifpal model.

It is important to note that we do not currently explicitly seek to rule out false attacks (i.e. false positives.) Our central argument is that the analysis logic described in §3.1 is sufficient in order to capture all possible confidentiality and authentication attacks within the language defined in Figure 1:

3.2.1 Value Construction

Protocol analysis always begins from the point of view of the attacker. The initial set of values that the attacker can know are necessarily constants, since only constants can be exchanged within network messages (Figure 1). *"Pure"* constants (constants that are declared via a **`knows`** or **`generates`** expression and not via assignment) resolve to themselves (`x` $\rightarrow$ `x`). Assigned constants resolve to either a primitive or an equation. Primitives can take constants, primitives or equations as arguments but always return constants. Equations can only take constants as arguments (effectively exponents).

3.2.2 Deconstructions, Rewrites, and Checks

Verifpal primitives have two kinds of potential rules:

- **Decomposition rules** allow principals and the attacker to obtain the value of a primitive's argument by knowing the primitive's output and only some of the primitive's other arguments. For example, knowing `e = ENC(k, m)` and `k` allows us to obtain `m`. **`AEAD_ENC`**, **`AEAD_DEC`**, **`ENC`** and **`DEC`** have decomposition rules.
- **Rewrite rules** allow principals and the attacker to rewrite a primitive's assigned value if certain conditions are satisfied. For example, `d = AEAD_DEC(k, e, a)` would be rewritten to `d = p` if `e = AEAD_ENC(k, p, a)`. When we *"check"* a primitive (see §2.3.4), a failed rewrite is essentially what we are terming as a *"failed check"* — checks simply make it such that failed rewrites abort session execution at that point. **`HMACVERIF`**, **`SIGNVERIF`**, **`AEAD_DEC`** and **`DEC`** have rewrite rules.

3.2.3 Genealogy of Values

In Verifpal, once a constant is known, generated or assigned, an immutable *creator* value is assigned to it defining the principal responsible for creating it. As the value travels across the network, a *sender* chain is built tracking its genealogy. For example, if Alice creates a value `m` and sends it to Bob, and if Bob then sends it to Carol, then `m` would have Alice as its creator and a sender chain of `Alice` $\rightarrow$ `Bob` $\rightarrow$ `Carol`.

When an attacker is tasked with contradicting an authentication query, it attempts to find out if a scenario exists in which a value is used in a primitive (or worse, triggers a valid rewrite rule) that does not follow the sender chain decreed by the authentication query.

3.2.4 Mutations and Guarded Constants

Except for guarded constants (see §2.5), the attacker can, at will, substitute any constant with any other, including constants crafted by the attacker. The goal of these substitutions is to execute the protocol in every possible permutation of constant-to-value assignments based on the values known by the attacker. Each unguarded constant risks being permuted with:

- **Other constants and values from the protocol** that have been revealed to the attacker.
- **New primitive and equation declarations** constructed from values that have been revealed to the attacker.
- **Malicious values** crafted by the attacker, including for example malicious public keys or malicious signatures under key pairs generated and owned by the attacker.

As noted in §3.1, once the attacker gains new values through this process, the permutation table is recalculated and the the set of executions begins anew. Protocol analysis ends when no new values are known to the attacker after a complete run of all possible permutations. The goal of this step is to obtain a full search of all runs of the protocol under all possible discoverable values, given the assumption that §3.1 allows the attacker to obtain all obtainable values.

Mutations and transformations are executed recursively. That is, if executing any one of Resolve, Deconstruct, Reconstruct and Equivocate leads to new values being discovered, then that transformation is executed recursively until no new values are found. If any new values are found, the series of four transformations is also re-executed recursively in its totality until no new values are obtainable by the attacker. Once that is the case, we move on to the next mutation.

Our core assumption regarding the completeness and reliability of Verifpal's analysis methodology is that the above is sufficient to, within Verifpal's language, capture all values knowable to the attacker, as well as all sender chains possible within a protocol given an attacker.

4 Case Study: Signal in Verifpal

Introduced in 2014, the Signal protocol started off as the core of the eponymous Signal messaging app for Android and iOS devices. In the following years it was also adopted by WhatsApp [57], Facebook Messenger, Skype and other applications. Today, it is responsible for encrypted communications on at least a billion devices worldwide, competing with Apple's iMessage protocol and Telegram's MTProto protocol.

4.1 Security Goals

Aside from targeting obvious security goals such as message confidentiality and mutual authentication for principals, Signal differentiated itself from predecessors as well as from

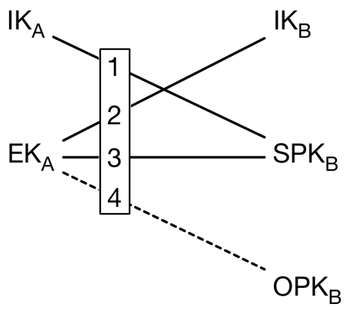

Figure 5: Signal's *"X3DH"* authenticated key exchange. IK_A and IK_B represent Alice and Bob's long-term key pairs. EK_A represents Alice's ephemeral session key pair. SPK_B and OPK_B represent Bob's signed ephemeral pre-key and one-time ephemeral pre-key. Three Diffie-Hellman shared secret calculations, and one optional Diffie-Hellman shared secret calculation, are conducted.

its competitor protocols by offering some ambitious security properties. The core design element behind these features is the fact that in Signal, each principal has essentially two types of key pairs: *long-term key pairs*, which serve to authenticate the identity of Alice and Bob to one another, are used exclusively for signing and for session establishment and that never change, and *ephemeral key pairs*, which last at most for a handful of messages and are used solely for encryption. The point of this approach is target the following security goals:

- **Forward-secure authenticated key exchange.** After a Signal session is established between Alice and Bob, revealing any or both parties' long-term keys does not reveal the contents of any of their messages. Since long-term keys are the only key material that remains on-device for extended periods of time, it can be assumed that this security goal is supposed to guard against device theft.
- **Per-message forward secrecy and post-compromise security.** If Alice or Bob's state were to be compromised at any point in time, the number of past and future messages, relevant to the last message sent at time of compromise, is limited.

Aside from these security-centric features, Signal also offers *asynchronous (*"offline"*) session establishment:* Alice is able to establish a Signal session with Bob and send a message even if Bob's phone is turned off. When Bob turns his phone back on, he will immediately receive Alice's message (even if, at the time, Alice's phone is off.)

Signal has already been formally verified using ProVerif [3], with results for confidentiality, authentication, post-compromise security and even leading to the discovery of a new key compromise impersonation attack. Here, we show that Verifpal can meet these exact same results, and in a smaller, simpler model (Figure 6).

4.2 Principals

Our first step in Verifpal will be to model Signal's essential protocol components and then to illustrate how these components can be used by Alice and Bob in order to conduct a Signal session.

4.2.1 Modeling the Key Exchange

Figure 5 illustrates how Signal's authenticated key exchange works. When initiating a session with Bob, Alice will perform four Diffie-Hellman operations:

1. Between Alice's long-term private key and Bob's *"signed pre-key"*, an ephemeral public key that Bob has pre-emptively generated, signed using his long-term private key, and stored on the Signal server.

2. Between Alice's ephemeral private key, generated for this session, and Bob's long-term public key.

3. Between Alice's ephemeral private key and Bob's signed pre-key.

4. Between Alice's ephemeral private key and Bob's *"one-time pre-key"*, an ephemeral public key that Bob has pre-emptively generated and stored on the Signal server. Unlike the signed pre-key, it is not signed. Signed pre-keys are rotated roughly once a week, while one-time pre-keys are only used once. This is simply because signing is a slow and computationally expensive process, and having Bob's phone sign every one-time pre-key (of which a server could store hundreds at a time) would be somewhat inefficient.

The four values obtained above are then hashed into a single value known as the master secret. Alice can also include an encrypted message along with her key exchange message. So, let's declare Alice and Bob in Verifpal:

Signal: Initializing Alice and Bob

```
attacker[active]
principal Alice[
  knows public c0, c1, c2, c3, c4
  knows private alongterm
  galongterm = G^alongterm
]
principal Bob[
  knows public c0, c1, c2, c3, c4
  knows private blongterm, bs
  generates bo
  gblongterm = G^blongterm
  gbs = G^bs
  gbo = G^bo
  gbssig = SIGN(blongterm, gbs)
]
```

Now, let's have Alice initiate a session with Bob and derive a master secret, which she stores as `amaster`:

```
Signal:  Alice Initiates Session with Bob

Bob -> Alice: [gblongterm], gbssig, gbs, gbo
principal Alice[
  generates ae1
  gae1 = G^ae1
  amaster = HASH(c0, gbs^alongterm, gblongterm^ae1, gbs^ae1, gbo^ae1)
  arkba1, ackba1 = HKDF(amaster, c1, c2)
]
```

4.2.2 Modeling Messages and the Double Ratchet

Since long-term keys are only employed in master secret derivation, and since we want to achieve per-message forward secrecy and post-compromise security, we want to both *authenticate* future messages based on Alice and Bob's identities while keeping them *confidential* using perpetually fresh ephemeral shared secrets.

```
Signal:  Alice Encrypts Message 1 to Bob

principal Alice[
  generates m1, ae2
  gae2 = G^ae2
  valid = SIGNVERIF(gblongterm, gbs, gbssig)?
  akshared1 = gbs^ae2
  arkab1, ackab1 = HKDF(akshared1, arkba1, c2)
  akenc1, akenc2 = HKDF(HMAC(ackab1, c3), c1, c4)
  e1 = AEAD_ENC(akenc1, m1, HASH(galongterm, gblongterm, gae2))
]
Alice -> Bob: [galongterm], gae1, gae2, e1
```

Notice how Alice generates a second fresh ephemeral key pair, `(ae2, gae2 = G^ae2)`, and mixes it with the master secret in order to derive two symmetric keys, `ackab1` will be used for encryption, while `arkab1` will only be used to derive future pairs of symmetric keys in the same fashion, thereby keeping a relationship back to the master secret, which ensures that all future derived keys are mixed with the key material that provided authentication in the master secret.

Notice also how the `SIGNVERIF` primitive is checked — if Alice can't verify the signature of Bob's signed pre-key `gbs` using Bob's long-term signing public key `gblongterm`, then the entire session is aborted.

Finally, notice how we are guarding `gblongterm` and `galongterm` from being modified by an active attacker while in transit – this achieves a model where Alice and Bob have mutually pre-authenticated one another's long-term public keys.

Alice then encrypts her chosen plaintext message `m1` to produce ciphertext `e1`. Notice how the Signal protocol specifies that a hash of the public keys used in this session must go as associated data to the message encryption primitive. This helps achieve a property known as *session* or *channel binding*.

Bob will first need to generate the shared master secret. Bob will be able to decrypt Alice's first message. Then, Bob will send his reply, encrypting his message `m2` to produce ciphertext `e2`:

```
Signal:  Bob Derives Shared Master Secret

principal Bob[
  bmaster = HASH(c0, galongterm^bs, gae1^blongterm, gae1^bs, gae1^bo)
  brkba1, bckba1 = HKDF(bmaster, c1, c2)
]
```

```
Signal:  Bob Decrypts Alice's Message 1

principal Bob[
  bkshared1 = gae2^bs
  brkab1, bckab1 = HKDF(bkshared1, brkba1, c2)
  bkenc1, bkenc2 = HKDF(HMAC(bckab1, c3), c1, c4)
  m1_d = AEAD_DEC(bkenc1, e1, HASH(galongterm, gblongterm, gae2))
]
```

```
Signal:  Bob Encrypts Message 2 to Alice

principal Bob[
  generates m2, be
  gbe = G^be
  bkshared2 = gae2^be
  brkba2, bckba2 = HKDF(bkshared2, brkab1, c2)
  bkenc3, bkenc4 = HKDF(HMAC(bckba2, c3), c1, c4)
  e2 = AEAD_ENC(bkenc3, m2, HASH(gblongterm, galongterm, gbe))
]
Bob -> Alice: gbe, e2
```

For good measure, we model a final message `m3` sent from Alice to Bob, after Alice decrypts Bob's message:

```
Signal:  Alice Decrypts Message 2

principal Alice[
  akshared2 = gbe^ae2
  arkba2, ackba2 = HKDF(akshared2, arkab1, c2)
  akenc3, akenc4 = HKDF(HMAC(ackba2, c3), c1, c4)
  m2_d = AEAD_DEC(akenc3, e2, HASH(gblongterm, galongterm, gbe))
]
```

```
Signal:  Alice Encrypts Message 3 to Bob

principal Alice[
  generates m3, ae3
  gae3 = G^ae3
  akshared3 = gbe^ae3
  arkab3, ackab3 = HKDF(akshared3, arkba2, c2)
  akenc5, akenc6 = HKDF(HMAC(ackab3, c3), c1, c4)
  e3 = AEAD_ENC(akenc5, m3, HASH(gblongterm, galongterm, gae3))
]
Alice -> Bob: gae3, e3
```

```
Signal:  Bob Decrypts Message 3

principal Bob[
  bkshared3 = gae3^be
  brkab3, bckab3 = HKDF(bkshared3, brkba2, c2)
  bkenc5, bkenc6 = HKDF(HMAC(bckab3, c3), c1, c4)
  m3_d = AEAD_DEC(bkenc5, e3, HASH(gblongterm, galongterm, gae3))
]
```

Now that we've modeled a fairly illustrative and representative execution of the Signal protocol between Alice and Bob, covering an authenticated key exchange as well as three messages, we're finally ready to ask Verifpal some tough questions and to analyze if, and how, our model of Signal achieves its desired security goals.

4.3 Queries and Analysis

Given that Signal is a secure messaging protocol, we certainly want to check whether `m1`, `m2` and `m3` are confidential against an active attacker. We also want to check if an attacker can impersonate any of the principals in sending one of the above messages. Formulating these queries in Verifpal is straightforward:

```
Signal:  Message Queries

queries[
  confidentiality? m1
  authentication? Alice -> Bob: e1
  confidentiality? m2
  authentication? Bob -> Alice: e2
  confidentiality? m3
  authentication? Alice -> Bob: e3
]
```

```
Signal:  Initial Results

Verifpal! verification completed at 12:36:53
```

This indicates that Verifpal was unable to find a contradiction to any of the queries. This goes hand in hand with previous academic formal verification work on Signal [3]: if Alice and Bob initiate a session with mutual pre-authentication, and if Alice is aborting the session should Bob's signed pre-key not pass signature verification, then the Signal protocol achieves confidentiality and authentication for messages sent between the two parties.

Recall that Signal also aims to achieve forward secrecy and post-compromise security. Let's see what happens if we leak Alice's long-term private key, by adding the following line right before she encrypts and sends `m3`:

```
Signal:  Alice Leaks Long-Term Private Key

Alice -> Bob: alongterm
```

By re-running the analysis, we see that Alice's messages m1 and m3 are still confidential against an active attacker. However, an interesting result appears:

```
Signal:  Key Compromise Impersonation for Alice

Deduction! m2 is obtained by the attacker as m2
Deduction! e2, sent by Attacker and not by Bob and resolving to AEAD_ENC(bkenc3,
    m2, HASH(gblongterm, galongterm, gbe)), is used in primitive AEAD_DEC(akenc3,
    e2, HASH(gblongterm, galongterm, gbe)) in Alice's state
```

It appears that leaking Alice's long-term private key allowed the attacker to impersonate Bob *to* Alice. The explanation is that Signal is vulnerable to a *key compromise impersonation attack* — compromising Alice's long-term private key does not only allow the attacker to impersonate Alice to others, but it also allows them to impersonate others to Alice. This result again matches previous analyses of Signal [3].

In order to understand how this attack works, let's look at Figure 5. Armed with Alice's long-term private key, the attacker can now perform the Diffie-Hellman operation marked *"1"* in her name as well as the others, thereby faking a session initiation. Let's remove the line we added to test forward secrecy and try something else. If we uncheck Alice's usage of **SIGNVERIF**, we see that results don't change. But what happens if we then also unguard Bob's long-term public key as it is being sent to Alice?

```
Signal:  Mayor-in-the-Middle Attack on Bob

Result! confidentiality? m1: m1 is obtained by the attacker as m1
Result! authentication? Alice -> Bob: e1: e1, sent by Attacker and not by Alice
    and resolving to AEAD_ENC(akenc1, m1, HASH(galongterm, gblongterm, gae2)), is
    used in primitive AEAD_DEC(bkenc1, e1, HASH(galongterm, gblongterm, gae2)) in
    Bob's state
Result! confidentiality? m3: m3 is obtained by the attacker as m3
Result! authentication? Alice -> Bob: e3: e3, sent by Attacker and not by Alice
    and resolving to AEAD_ENC(akenc5, m3, HASH(gblongterm, galongterm, gae3)), is
    used in primitive AEAD_DEC(bkenc5, e3, HASH(gblongterm, galongterm, gae3)) in
    Bob's state
```

The attacker was able to compromise all of the messages that Alice sent to Bob, since they were able to fully impersonate Bob as he interacted with Alice in the session.

Tweaking our model and re-running analysis is central to getting the most insight out of Verifpal. By making some very simple changes to our model, we were quickly able to go from a fully secure model to one that showed us whether forward secrecy would be achieved in the event of a long-term private key compromise, and then to another that provided a warning on the importance of mutual pre-authentication.

5 Limitations and Future Work

Verifpal currently does not possess a full soundness theorem, and this is slated as the most urgent future work as the tool matures and develops due to its exposure to the world. Verifpal does possess a formalized analysis methodology, as described in §3. This analysis comes with two major known limitations:

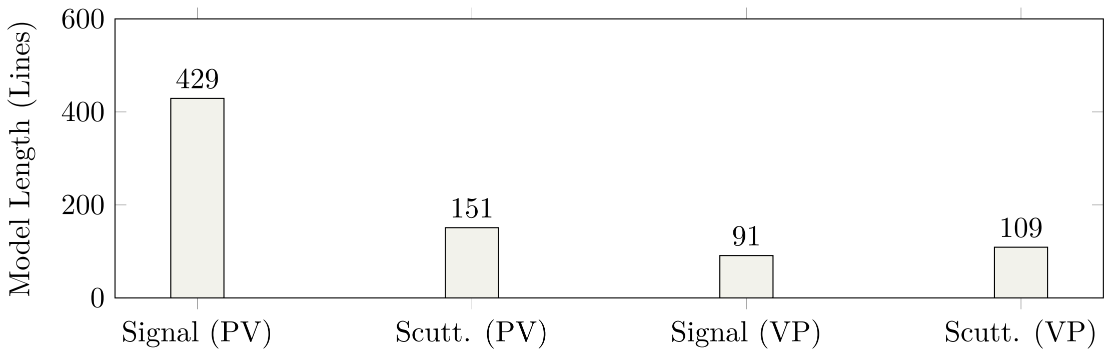

Figure 6: Comparison of the length and complexity of modeling Scuttlebutt (handshake only) and Signal (full protocol with three messages) in ProVerif (PV) and Verifpal (VP), with similar scenarios for Alice and Bob, and with similar queries.

1. Reconstruct is largely limited to reconstructing values known by principals, and will not attempt to construct arbitrary values outside of those used and expressed within **`principal`** declarations (with $\mathcal{V}_P$.)

2. Fresh values are not kept between sessions. This is expected behavior for many symbolic analysis tools, but, in Verifpal, it may lead to some less complete analysis for attacks based on intra-session fresh values, especially, for example, parallel or *"multi-protocol"* session executions.

Current effort is focused largely on further studying the limitations of Verifpal's analysis methodology and on deriving countermeasures that may lead to a more comprehensive analysis. For example, allowing the attacker to keep certain fresh values between protocol executions could lead to an easier modeling for parallel sessions.

Verifpal does not plan to be a competitor to aforementioned tools such as ProVerif or Tamarin. This is due to our plan to prioritize usability over features, leading Verifpal to have no road map to support, for example, declaring custom primitives or rewrite rules. Verifpal also lacks the fine control that tools such as ProVerif can offer over how protocol processes are executed: Verifpal has no notion of protocol *phases*, cannot differentiate between parallel and non-parallel processes, cannot query for indifferentiability or observational equivalence [58, 59], and also does not support many other advanced features. Verifpal cannot also model state transitions with the same precision enjoyed by Tamarin. At least for the moment, this is explicit and by design: Verifpal aims to differentiate itself from other verifiers by focusing on usability first and features last. So far, the results have been encouraging: Verifpal was able to test for advanced security goals in protocols such as Signal from its very first release.

Verifpal is also fully capable of supporting a more nuanced definition of primitives recently seen in other symbolic verifiers — for example, recent, more precise models for signature schemes [8] in Tamarin can be fully integrated into Verifpal's design.

We also plan to add support for more primitives as these are suggested by the Verifpal user community. We also plan to integrate Verifpal better into existing IDEs, such as Visual Studio Code integration, in order to provide engineers live feedback on their protocol as they model it.

6 Conclusion

Verifpal's language is easier and more intuitive than that of existing tools, while still allowing to express complex protocols, such as the Signal secure messenger, and to query for advanced security properties such as forward secrecy. These design elements should allow Verifpal to offer students, engineers, developers and protocol designers a way to prototype protocols that works for them. We believe that Verifpal's verification framework gives it full jurisdiction over maturing its language and feature set, such that it can grow to satisfy the fundamental verification needs of protocol developers without having the barrier-to-entry present in tools such as ProVerif and Tamarin.

Verifpal is currently available as free and open source software, allowing it to grow both in terms of both confidence in the soundness of its own analysis, and in terms of features offered for newcomers to the world of formal protocol verification.

Acknowledgments Verifpal is fundamentally inspired by Bruno Blanchet's decades of research into automated formal verification, and would not exist without his work. We also thank Georgio Nicolas and Sasha Lapiha for their feedback.

Funding was provided through the NGI0 PET Fund, a fund established by NLnet with financial support from the European Commission's Next Generation Internet program, under the aegis of DG Communications Networks, Content and Technology under grant agreement №825310.

References

[1] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, 2016.

[2] Andreas Straub. OMEMO encryption. 2018.

[3] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[4] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P)*, pages 483–502. IEEE, 2017.

[5] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1383–1396. ACM, 2018.

[6] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. *2019 Network and Distributed System Security Symposium (NDSS)*, 2019.

[7] Cas Cremers and Dennis Jackson. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman. *IEEE Computer Security Foundations Symposium (CSF)*, 19, 2019.

[8] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In *ACM CCS 2019*, 2019.

[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Nadim Kobeissi. Formal modeling and verification for domain validation and ACME. In *International Conference on Financial Cryptography and Data Security*, pages 561–578. Springer, 2017.

[10] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[11] Guillaume Girol. Formalizing and verifying the security protocols from the Noise framework. Master's thesis, ETH Zurich, 2019.

[12] Andris Suter-Dörig. Formalizing and verifying the security protocols from the Noise framework, 2018.

[13] Jason A Donenfeld. WireGuard: Next generation kernel network tunnel. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[14] Lipp Benjamin, Bruno Blanchet, and Karthikeyan Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[15] Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy (S&P)*, pages 98–113. IEEE, 2014.

[16] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy (S&P)*, pages 535–552. IEEE, 2015.

[17] Bruno 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] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design VII*, pages 54–87. Springer, 2013.

[19] Ashok K Chandra and David Harel. Horn clause queries and generalizations. *The Journal of Logic Programming*, 2(1):1–15, 1985.

[20] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[21] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[22] Bruno Blanchet and Avik Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Symposium on Security and Privacy (S&P)*, pages 417–431. IEEE, 2008.

[23] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *IEEE Computer Security Foundations Symposium*, pages 195–209. IEEE, 2008.

[24] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[25] Véronique Cortier and Cyrille Wiedling. A formal analysis of the norwegian e-voting protocol. In *International Conference on Principles of Security and Trust*, pages 109–128. Springer, 2012.

[26] Cas Cremers and Lucca Hirschi. Improving automated symbolic analysis of ballot secrecy for e-voting protocols: A method based on sufficient conditions. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[27] Olivier Pereira, Florentin Rochet, and Cyrille Wiedling. Formal analysis of the FIDO 1. x protocol. In *International Symposium on Foundations and Practice of Security*, pages 68–82. Springer, 2017.

[28] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In Stephen Chong, editor, *IEEE Computer Security Foundations Symposium (CSF), Cambridge, MA, USA, June 25-27, 2012*, pages 78–94. IEEE, 2012.

[29] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788. ACM, 2017.

[30] Jason A Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol. Technical report, Technical Report, 2017.

[31] David Basin, Saša Radomirovic, and Lara Schmid. Alethea: A provably secure random sample voting protocol. In *IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 283–297. IEEE, 2018.

[32] Alessandro Bruni, Eva Drewsen, and Carsten Schürmann. Towards a mechanized proof of selene receipt-freeness and vote-privacy. In *International Joint Conference on Electronic Voting*, pages 110–126. Springer, 2017.

[33] Professor Oak. Kanto Regional Pokédex. *Kanto Region Journal on Pokémon Research*, 19, 1996.

[34] C.J.F. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[35] David A. Basin and Cas J.F. Cremers. Degrees of security: Protocol guarantees in the face of compromising adversaries. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.

[36] C.J.F. Cremers. Key exchange in IPsec revisited: formal analysis of IKEv1 and IKEv2. In *Proceedings of the 16th European conference on Research in computer security*, ESORICS, pages 315–334, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] David Basin and Cas Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *Computer Security - ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 340–356. Springer, 2010.

[38] C.J.F. Cremers. Feasibility of multi-protocol attacks. In *Proc. of The First International Conference on Availability, Reliability and Security (ARES)*, pages 287–294, Vienna, Austria, April 2006. IEEE Computer Society.

[39] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *International conference on computer aided verification*, pages 281–285. Springer, 2005.

[40] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–282. Springer, 2012.

[41] Ruhul Amin, SK Hafizul Islam, Arijit Karati, and GP Biswas. Design of an enhanced authentication protocol and its verification using AVISPA. In *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*, pages 404–409. IEEE, 2016.

[42] Marino Miculan and Caterina Urban. Formal analysis of Facebook Connect single sign-on authentication protocol. In *SOFSEM*, volume 11, pages 22–28. Citeseer, 2011.

[43] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[44] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[45] Bae Woo-Sik. Formal verification of an RFID authentication protocol based on hash function and secret code. *Wireless personal communications*, 79(4):2595–2609, 2014.

[46] Pascal Lafourcade and Maxime Puys. Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In *International Symposium on Foundations and Practice of Security*, pages 137–155. Springer, 2015.

[47] Cas J.F. Cremers, Pascal Lafourcade, and Philippe Nadeau. Comparing state spaces in automatic protocol analysis. In *Formal to Practical Security*, volume 5458/2009 of *Lecture Notes in Computer Science*, pages 70–94. Springer Berlin / Heidelberg, 2009.

[48] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Principles of Security and Trust (POST)*, pages 3–29, 2012.

[49] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar on Applied Formal Protocol Verification*, page 117, 2007.

[50] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in F. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17, 2017.

[51] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[52] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *IEEE Symposium on Security and Privacy (S&P)*, page 0. IEEE, 2019.

[53] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[54] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in F. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17, 2017.

[55] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-OHearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[56] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology (CRYPTO)*, pages 631–648. IACR, 2010.

[57] Open Whisper Systems. WhatsApp encryption overview, 2016.

[58] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *International Conference on Principles of Security and Trust*, pages 226–246. Springer, 2013.

[59] Hiroyuki Okazaki, Yuichi Futa, and Kenichi Arai. Suitable symbolic models for cryptographic verification of secure protocols in ProVerif. In *2018 International Symposium on Information Theory and Its Applications (ISITA)*, pages 326–330. IEEE, 2018.