Verifpal: Cryptographic Protocol Analysis for the Real World

Nadim Kobeissi[1], Georgio Nicolas[1], and Mukesh Tiwari[2]

[1] Symbolic Software
[2] University of Melbourne

Abstract Verifpal is a new automated modeling framework and verifier for cryptographic protocols, optimized with heuristics for common-case protocol specifications, that aims 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.
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. Furthermore, Verifpal's semantics have been formalized within the Coq theorem prover, and Verifpal models can be automatically translated into Coq as well as into ProVerif models for further verification. Verifpal has already been used to verify security properties for Signal, Scuttlebutt, TLS 1.3 as well as the first formal model for the DP-3T pandemic-tracing protocol, which we present in this work. Through Verifpal, we show that advanced verification with formalized semantics and sound logic can exist without any expense towards the convenience of real-world practitioners.

Keywords: formal analysis · protocol analysis · protocol modeling

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 [33]) across complex use-cases (such as support for message synchronization across multiple devices.) 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 [15,19,18].

Automated formal verification tools have seen an encouraging success in helping to model the security of these protocols. Recently, the Signal secure messaging protocol [57], the TLS 1.3 web encryption standard [16], the 5G wireless communication standard [10,35], the Scuttlebutt decentralized messaging protocol [38], the Bluetooth standard [38], the Let's Encrypt certificate issuance system [54,17], the Noise Protocol

Tool	Unbound	Eq-thy	State	Equiv	Link
CPSA [45]	●	○	●	○	○
DEEPSPEC [31]	○	◐	●	●	○
F7 [13]	●	◐	●	○	●
Maude-NPA [49]	●	●	○	●	○
ProVerif [30]	●	◐	○	●	○
Scyther [41]	●	○	○	○	○
Tamarin [72]	●	●	●	●	○
Verifpal	●	◐	●	◐	◐

Figure 1. Comparison between Verifpal and other tools for symbolic security analysis, using established impartial third-party criteria [8]. Verifpal analysis supports unbounded executions (including interleaving protocol sessions), equational theory (although not as refined as Tamarin's), mutable principal states, trace properties and is able to link results to implementations via Coq (and soon Go). Verifpal does not support equivalence properties at the same level as Maude-NPA, ProVerif, Tamarin and DEEPSPEC, but does offer queries for notions of unlinkability between values. Verifpal also focuses on providing a substantially more intuitive overall framework for real-world protocol modeling and analysis through its language and built-in primitive definitions, although such a claim is more tricky to compare.

Framework [58,51] and the WireGuard [47] Virtual Private Network (VPN) protocol [63] 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 [18,15,16], and was, again, only accomplished via collaboration with the designers of the formal verification tools themselves.

1.1 Verifpal's Design Goals

It is important to discern that Verifpal does not aim to produce security proofs in the traditions of tools such as CryptoVerif [21]. In deciding Verifpal's priorities, we slam the brakes at the moment where the learning curve, effort and analysis cost begin to have strongly diminishing returns for the user while still maintaining a responsible level of rigor via a formal treatment of Verifpal's semantics and analysis methodology. Our bet is that this path forward for Verifpal will lead to a hugely more substantial impact for engineers and practitioners than traditional automated proof modeling tools. In this paper, we will for example see how Verifpal makes compromises in analysis completeness that preclude its ability to output full proofs but that greatly increase the likelihood of analysis termination (a significant problem for tools such as ProVerif) without having an apparently significant impact on the analysis of real-world, non-Ivory-Tower protocols.

Verifpal is able to analyze 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.

1.2 Simplifying Protocol Analysis with Verifpal

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, Verifpal's main aim is to appeal more to real-world practitioners, students and engineers without sacrificing comprehensive formal verification features. Verifpal has four main design principles:

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. We argue that this paradigm extends beyond mere convenience, but extends protocol modeling and verification towards a necessary level of intuitiveness for real adoption.

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, **`RINGSIGN`** 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. Furthermore, §3.1 briefly describes Verifpal's use of heuristics in order to avoid non-termination due to state space explosion, a common problem with automated protocol verification tools.

Easy to understand analysis output. Existing tools provide *"attack traces"* that illustrate a deduction using session-tagged values in a chain of symbolic deconstructions. Verifpal follows a different approach: while 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 attack on ephemeral keys.

Compatibility with the Coq theorem prover. The Verifpal language and passive attacker analysis methodology has recently been formalized within the Coq theorem prover [14]. Consequently, Verifpal models can be automatically translated within Coq using the Verifpal software. This allows for further analysis in more established frameworks while also granting a higher level of confidence in Verifpal's analysis methodology. Currently, the Coq work provides a complete formalized illustration of the Verifpal language semantics and of Verifpal analysis under a passive attacker. Our eventual goal is to use Coq as an attestation layer to Verifpal's soundness logic and show that Verifpal analysis results can be attested as sound via the generated Coq implementations. In addition, Verifpal models can also be translated into ProVerif models for an additional parallel verification venue.

1.3 Related Work

Verifpal arrives 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.

Verifpal is heavily inspired by the ProVerif [24,23] protocol verifier, designed by Bruno Blanchet. It does not construct all terms out of Horn clauses [28] in the way that ProVerif does, and it does not use the applied pi-calculus [1] as its modeling language. However, its analysis logic is inspired by ProVerif and is similarly based on the Dolev-Yao model [46]. 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 [16], Let's Encrypt's ACME certificate issuance protocol [17], the Signal secure messaging protocol [57], the Noise Protocol Framework [58], the Plutus network filesystem [25], e-voting protocols [6,44,34,36], FIDO [68] and many more use cases.

The Tamarin [72] 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 [38], TLS [37], WireGuard [48], 5G [10,35], the Noise Protocol Framework [51], multiple e-voting protocols [11,27] and many more use cases.

Scyther[3] [41,12], 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 [42] (used in IPSec), a large amount of Authenticated Key Exchange (AKE) protocols such as HMQV, UM and NAXOS [9], and to check for *"multi-protocol attacks"* [40]. Research focus seems to be moving towards Tamarin, but Scyther is still sometimes used.

AVISPA [4]'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 largely moved to a successor tool, AVANTSSAR [3] which shares many of the same authors. In 2016, a new authentication protocol was designed

[3] Not to be confused with the bug/flying-type Pokémon of the same name, which, despite its *"ninja-like agility and speed"* [66], does not appear to have published work in formal verification.

and prototyped with AVISPA [2]. In 2011, Facebook's *Connect* single sign-on protocol was modeled with AVISPA [64].

FDR [50] is not specifically a protocol verifier, but rather a refinement and equivalence checker for processes written using the Communicating Sequential Processes language [53]. 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 [76].

A performance analysis of symbolic formal verification tools by Lafourcade and Pus [60], conducted in 2015, as well as a preceding study by Cremers and Lafourcade in 2011 [39] found mixed results, with ProVerif coming out on top more often than not.

ProVerif and Tamarin appear to be 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, claims to always yield a proof or an attack, 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 [58].

1.4 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.

Cryptographers, on the other hand, prefer to use *computational* models and do their proofs by hand. A full comparison between these styles [22] 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 [21] 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 bit-strings. Freshly generated nonces and keys are randomly sampled bit-strings that the attacker can guess with some probability (depending on their length). Encryption and decryption are functions on bit-strings to which we may associate standard cryptographic assumptions such as IND-CCA. The attacker is a probabilistic polynomial-time process running in parallel.

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. 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 bit-string.

Recently, the F* programming language [70], which exports type definitions to the Z3 theorem prover [43], has been used to produce implementations of TLS [71] and Signal that are formally verified for functional correctness at the *level of the implementation itself* [69].

1.5 Contributions

We present the following contributions:

- In §1, we introduce Verifpal and provide a comparison against existing automated verification tools in the symbolic model, as well as a recap of the current state of the art.
- In §2, we introduce the Verifpal modeling language complete with syntax and semanticsand provide some justifications for the language's design choices as well as examples.
- In §3, we discuss Verifpal's protocol analysis logic and whether we can be certain that Verifpal will not miss an attack on a protocol model. We also show that Verifpal can find attacks on sophisticated protocols, matching results previously obtained in ProVerif, and demonstrate Verifpal's improved protocol analysis trace output which makes discovered attacks easier to discern for the user.
- In §4, we provide the first formal model of the DP-3T decentralized pandemic-tracing protocol [75], written in Verifpal, with queries and results on unlinkability, freshness, confidentiality and message authentication.
- In §5, we introduce Verifpal's Coq compatibility layer. We show how Verifpal's semantics and verification logic (for passive attacker only) are captured in the Coq theorem prover, as well as how Verifpal can translate arbitrary Verifpal models into Coq and ProVerif for further analysis.
- In §6, we conclude with a discussion of future work.

Verifpal is available as free and open source software at `https://verifpal.com`. In addition, Verifpal provides a Visual Studio Code extension that enables it to function as an IDE for the modeling, analysis and verification of cryptographic protocols.

```
Simple Example Protocol

attacker[active]
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)?
]
```

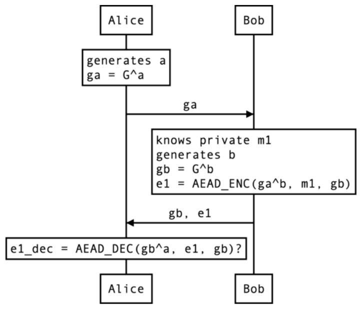

Figure 2. A complete example Verifpal model of a simple protocol is shown on the left.

2 The Verifpal Language

Verifpal's language is meant to be simple while allowing the user to capture comprehensive protocols. We posit that an intuitive language that reads similarly to regular descriptions of secure channel protocols will provide a valuable asset in terms of modeling cryptographic protocols, and design Verifpal's language around that assertion. This is radically different from how the languages of tools such as ProVerif and Tamarin are designed: the former is derived from the applied-pi calculus and the latter from a formalism of state transitions, making it reasonable to say that readability and intuitiveness were not the primary goals of these languages.

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. 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 be impersonated by an active attacker?) that we will ask Verifpal.

2.1 Principals

Figure 2 shows a simple Verifpal model. We first define what kind of attacker Verifpal will use to analyze our model. **`attacker`**`[passive]` indicates a passive attacker, while **`attacker`**`[active]` indicates an active attacker.

We may then declare a principal `Alice` who generates the fresh private constant `a`, then used as her ephemeral private key. Alice then calculates `ga =` **`G`**`^a`. Here, `ga` is

Alice's public Diffie-Hellman key, while `G^a` quite plainly indicates the standard Diffie-Hellman exponentiation g^a. Later, Alice will be able to write `gb^a`, which is how we illustrate the derivation of the shared secret g^{ba} in Verifpal.

2.2 Fundamental Types in Verifpal

Verifpal has three fundamental types: constants, primitives and equations. A constant may have qualifiers such as *freshness* (if declared using **`generates`**). Equations are in the form `G^x^y`. Primitives are one of the various built-in functions in Verifpal, and are defined using Verifpal's internal primitive definition structure. All of these elements are touched upon below.

Constants In Figure 2, `a`, `ga`, `m1`, `b`, `gb`, `e1` and `e1_dec` 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 practitioners 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).[4]
- **`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.
- **`leaks`**: This allows us to specify that the principal will leak an existing constant that they already know to the attacker, rendering the value immediately knowable to the attacker at the point of leakage.
- *Assignment*: A constant may be declared by assigning it to the result of a primitive or equation expression (recall that constants may not be assigned to other constants).

[4] A third qualifier, **`password`**, can be used to declare private constants that are weak or guessable: if they are used directly within, for example, an encryption primitive, and the ciphertext is obtained by the attacker, the attacker will be able to obtain the password value immediately. Therefore, in order to be used safely, values declared using **`knows password`** must first be sent through a password hashing primitive such as **`PW_HASH`**. This allows Verifpal to natively support modeling for cryptographic operations that use weak passwords or other guessable values that do not go through appropriate key derivation mechanisms.

Primitives 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.

Internally in Verifpal's standard implementation, all primitives are defined using a common spec called PrimitiveSpec which restricts how they can be expressed to a set of common rules. Aside from information such as the primitive's names, arity and number of outputs, each PrimitiveSpec defines a primitive solely via a combination of four standard rules:

– Decompose. Given a primitive's output and a defined subset of its inputs, reveal one of its inputs. (Given `ENC(k, m)` and `k`, reveal `m`).
– Recompose. Given a subset of a primitive's outputs, reveal one of its inputs. (Given `a, b`, reveal `x` if `a,b,_ = SHAMIR_SPLIT(x)`).
– Rewrite. Given a matching defined pattern within a primitive's inputs, rewrite the primitive expression itself into a logical subset of its inputs. (Given `DEC(k, ENC(k, m))`, rewrite the entire expression `DEC(k, ENC(k, m))` to `m`).
– Rebuild. Given a primitive whose inputs are all the outputs of some same other primitive, rewrite the primitive expression itself into a logical subset of its inputs. (Given `SHAMIR_JOIN(a, b)` where `a, b, c = SHAMIR_SPLIT(x)`, rewrite the entire expression `SHAMIR_JOIN(a, b)` to `x`).

Core Primitives Verifpal offers the following *"core"* primitives, which perform basic operations that are not necessarily cryptographic in nature, but still often useful in models.

– `ASSERT(MAC(k, m), MAC(k, m))`. Checks the equality of two values, and especially useful for checking MAC equality.
– `CONCAT(a, b...): c`. Concatenates between two to five values into one value.
– `SPLIT(CONCAT(a, b)): a, b`. Splits a concatenation back to its component values. Must contain a `CONCAT` primitive as input; otherwise, Verifpal will output an error.

Coq: Verifpal Symmetric Encryption

```
Definition ENC(key plaintext: constant): constant := ENC_c key plaintext.
Definition DEC(key ciphertext: constant): constant :=
  match ciphertext with
  | ENC_c k m ⇒ match k =? key with
    | true ⇒ m | false ⇒ ENC_c k m end
  | _ ⇒ ciphertext end.
Theorem enc_dec: forall k m: constant, DEC k (ENC k m) = m.
Proof.
  unfold ENC, DEC; intros k m;
  rewrite equal_constant_true; try auto.
Qed.
```

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, BLAKE2s [5]. Takes between 1 and 5 inputs and returns one output.
- **MAC**(key, message): hash. Keyed hash function. Useful for message authentication and for some other protocol constructions.
- **HKDF**(salt, ikm, info): a, b.... Hash-based key derivation function inspired by the Krawczyk HKDF scheme [59]. Essentially, **HKDF** is used to extract more than one key out a single secret value. salt and info help contextualize derived keys. Produces between 1 and 5 outputs.
- **PW_HASH**(a...): x. Password hashing function, similar in practice to, for example, Scrypt [67] or Argon2 [20]. Hashes passwords and produces output that is suitable for use as a private key, secret key or other sensitive key material. Useful in conjunction with values declared using **knows password** a.

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

Coq: Verifpal Authenticated Encryption

```
Theorem aead_enc_dec: forall k m ad: constant,
  AEAD_DEC k (AEAD_ENC k m ad) ad = m.
Proof.
  unfold AEAD_ENC, AEAD_DEC;
  intros k m ad; rewrite equal_constant_true;
  rewrite equal_constant_true; try auto.
Qed.
Theorem aead_enc_dec_2: forall k m ad c: constant,
  c = AEAD_ENC k m ad → m = AEAD_DEC k c ad.
Proof.
  intros k m ad c H.
  rewrite → H. rewrite → aead_enc_dec. reflexivity.
Qed.
```

- **ENC**(key, p): c. Symmetric encryption, similar for example to AES-CBC or to ChaCha20.
- **DEC**(key, **ENC**(key, p)): p. Symmetric decryption.
- **AEAD_ENC**(key, p, ad): c. 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**(key, **AEAD_ENC**(key, p, ad), ad): p. Authenticated decryption with associated data.

– **PKE_ENC**(**G**^key, p): c. Public-key encryption.
– **PKE_DEC**(key, **PKE_ENC**(**G**^key, p)): p. Public-key decryption.

Coq: Verifpal Public Key Semantics

```
Theorem pub_key: forall x: constant, G^( x ) = pub_key_c x.
Proof.
 intros x. destruct x; try reflexivity.
Qed.
(* a private key always has the same public key *)
Theorem pub_key_eq: forall x y: constant,
   x = y → G^( x ) = G^( y ).
Proof.
intros x y H. subst; auto.
Qed.
```

Signature Primitives Verifpal offers a simple signing primitive with a corresponding signature verification function.

– **SIGN**(key, m): sig. Classic signature primitive. Here, key is a private key.
– **SIGNVERIF**(**G**^k, message, **SIGN**(k, m)): m. Verifies if the signature can be authenticated. If key a was used for **SIGN**, then **SIGNVERIF** will expect **G**^a as the key value.
– **RINGSIGN**(k_a, **G**^k_b, **G**^k_c, m): sig. Ring signature. In ring signatures, one of three parties (Alice, Bob or Charlie) signs a message. The resulting signature can be verified using the public key of any of the three parties, and the signature does not reveal the signatory, only that they are a member of the signing ring (Alice, Bob or Charlie). The first value provided as an argument must be the private key of the actual signer, while the subsequent two arguments must be the public keys of the other potential signers. Paired with **RINGSIGNVERIF**.
– **BLIND**(k, m): m. Message blinding primitive, useful for implementing blind signatures [29]. Here, the sender uses the secret "blinding factor" k in order to blind message m, which can then be sent to the signer, who will be able to produce a signature on m without knowing m. Used in conjunction with **UNBLIND**.
– **UNBLIND**(k, m, **SIGN**(a, **BLIND**(k, m))): **SIGN**(a, m). Once **BLIND**(k, m) is signed by the signer, the sender can convert **SIGN**(a, **BLIND**(k, m)) to **SIGN**(a, m) by unblinding the message using their secret blinding factor k. The resulting unblinded signature can then be used as if it were a regular signature by a over m.

Secret Sharing Primitives Verifpal offers a simple interface for modeling Shamir Secret Sharing [73], which allows a secret (such as a key) to be split into multiple shares such that only some (and not all) of these shares are required to reconstitute it.

– **SHAMIR_SPLIT**(k): s1, s2, s3. In Verifpal, we allow splitting the key into three shares such that only two shares are required to reconstitute it.

```
Coq: Verifpal Ring Signatures (Partial)
```

```
Theorem ringsignverif_verif1: forall a b c m: constant,
  m = RINGSIGNVERIF (G^( a )) (G^( b )) (G^( c )) m (
    RINGSIGN a (G^( b )) (G^( c )) m).
Proof.
  unfold RINGSIGN, RINGSIGNVERIF. intros a b c m.
  simpl. rewrite equal_constant_true. simpl. reflexivity.
Qed.
Theorem ringsignverif_order_sign1: forall a b c m: constant,
  m = RINGSIGNVERIF (G^( a )) (G^( b )) (G^( c )) m (
    RINGSIGN a (G^( c )) (G^( b )) m).
Proof.
  unfold RINGSIGN, RINGSIGNVERIF. intros a b c m.
  simpl. rewrite equal_constant_true. simpl. reflexivity.
Qed.
```

– **`SHAMIR_JOIN`**`(sa, sb): k`. Here, `sa` and `sb` must be two distinct elements out of the set `(s1, s2, s3)` in order to obtain `k`.

If 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 — and messages obtained before the failure of the checked primitive are still valid for analysis, perhaps even in future sessions.

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

```
Coq: Verifpal Diffie-Hellman Semantics

Theorem dh_commutativity: forall x y,
  (DH (G^( x )) y) = (DH (G^( y )) x).
Proof.
  intros x y. rewrite dh_eq. rewrite dh_eq.
  rewrite ← mult_commute. reflexivity.
Qed.
```

Messages, Guarded Constants, Checked Primitives and Phases 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 line of the 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.

In Verifpal, **`ASSERT`**, **`SPLIT`**, **`AEAD_DEC`**, **`SIGNVERIF`** and **`RINGSIGNVERIF`** are *"checkable"* primitives: if we add a question mark (`?`) after one of these primitives, then the model execution will abort should **`AEAD_DEC`** fail authenticated decryption, or should **`ASSERT`** 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.

Phases allow Verifpal to express notions of temporal logic, which allow for reliable modeling of post-compromise security properties such as forward secrecy or future secrecy. When modeling with an active attacker, a new phase can be declared:

```
Example: Phases

Bob→ Alice: b1
phase[1]
principal Alice[leaks a2]
```

In the above example, the attacker won't be able to learn `a2` until the execution of everything that occurred in phase 0 (the initial phase of any model) is concluded. Furthermore, the attacker can only manipulate `a2` within the confines of the phases in which it is communicated. That is to say, the attacker will have knowledge of `b1` when doing analysis in phase 1, but won't be able to manipulate `b1` in phase 1. The attacker won't have knowledge of `a2` during phase 0, but will be able to manipulate `b1` in phase 0. Phases are useful to model scenarios where, for example, the attacker manages to steal Alice's keys strictly *after* a protocol has been executed, allowing the attacker to use their knowledge of that key material, but only outside of actually injecting it into a running protocol session.

Values are learned at the earliest phase in which they are communicated, and can only be manipulated within phases in which they are communicated, which can be more than one phase since Alice can for example send a2 later to Carol, to Damian, etc. Importantly, values derived from mutations of `b1` in phase 0 cannot be used to construct new values in phase 1.

2.3 Queries

Here are examples of three different types of queries:

```
Simple Example Protocol: Queries

queries[
  confidentiality? m1
  authentication? Bob→ Alice: e1
  unlinkability? ga, m1
]
```

The above example is drawn from Verifpal's current four query types:

- **Confidentiality Queries:** Confidentiality queries are the most basic of all Verifpal queries. We ask: *"can the attacker obtain* `m1`*?"* — where `m1` is a sensitive message. If the answer is yes, then the attacker was able to obtain the message, despite it being presumably encrypted. When used in conjunction with phases, confidentiality queries can however be used to model for advanced security properties such as forward secrecy.
- **Authentication Queries:** Authentication queries rely heavily on Verifpal's notion of "*checked"* or *"checkable"* primitives. 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.
- **Freshness Queries:** Freshness queries are useful for detecting replay attacks, where an attacker could manipulate one message to make it seem valid in two different contexts. In passive attacker mode, a freshness query will check whether a value is "fresh" between sessions (i.e. if it has at least one composing element that is generated, non-static). In active attacker mode, it will check whether a value can be rendered

"non-fresh" (i.e. static between sessions) and subsequently successfully used between sessions.
- **Unlinkability Queries:** Protocols such as DP-3T (see §4), voting protocols and RFID-based protocols posit an "unlinkability" security property on some of their components or processes. Definitions for unlinkability vary wildly despite the best efforts of researchers [74,52,7], but in Verifpal, we adopt the following definition: *"for two observed values, the adversary cannot distinguish between a protocol execution in which they belong to the same user and a protocol execution in which they belong to two different users."*

Based on the above, Verifpal introduced in version 0.12.0 experimental support for a notion of unlinkability based on the following checks. For an unlinkability query evaluating two values `a` and `b`:

- First, Verifpal checks to see if `a` and `b` satisfy freshness. If they do not, the query fails. Similarly to regular freshness queries, if an attacker can coerce a value to be non-fresh across sessions, then it is non-fresh and the query fails.
- If `a` and `b` both satisfy freshness, Verifpal then checks to see if the attacker can determine them as being the output of the same primitive or as having a *common source*. For example, the first and second output of the same **`HKDF`** construction with the same inputs. Of course, `a` and `b` can indeed be the outputs of that **`HKDF`** and be unlinkable; unless the attacker is able to reconstruct that same **`HKDF`** primitive and thereby use it to determine that both values are the outputs of it.

We note that unlinkability queries are especially experimental, since it is likely that these two notions are not sufficient to fully capture unlinkability between values, and future versions of Verifpal may expand this definition with additional notions.

2.4 Query Options

Imagine that we want to check if Alice will only send some message to Carol if it has first authenticated it from Bob. This can be accomplished by adding the **`precondition`** option to the authentication query for `e`:

```
Query Options Example

queries[authentication? Bob→ Alice: e[
    precondition[Alice→ Carol: m2]]]
```

The above query essentially expresses: *"The event of Carol receiving* `m2` *from Alice shall only occur if Alice has previously received and authenticated an encryption of* `m2` *as coming from Bob."*

3 Analysis in Verifpal

Verifpal's active attacker analysis methodology follows a simple set of procedures and algorithms. The overall process is comprised of five steps (see Figure 3 for an illustration):

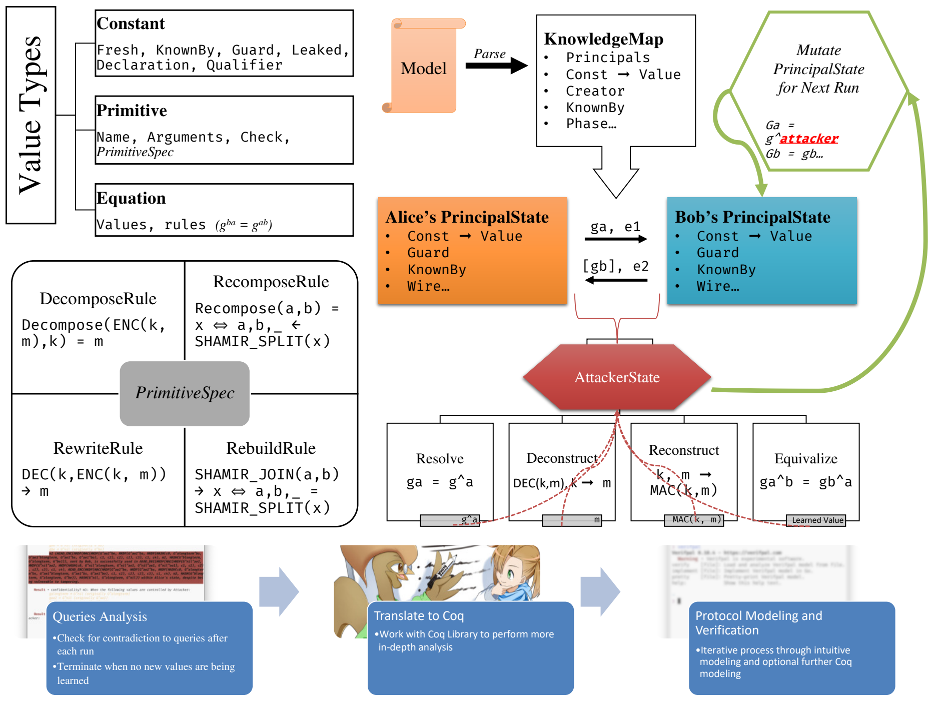
Value Types
Constant
Fresh, KnownBy, Guard, Leaked, Declaration, Qualifier
Primitive
Name, Arguments, Check, PrimitiveSpec
Equation
Values, rules ($g^{ba} = g^{ab}$)
DecomposeRule
Decompose(ENC(k, m),k) = m
RecomposeRule
Recompose(a,b) = x ⇔ a,b,_ ← SHAMIR_SPLIT(x)
PrimitiveSpec
RewriteRule
DEC(k,ENC(k, m)) → m
RebuildRule
SHAMIR_JOIN(a,b) → x ⇔ a,b,_ = SHAMIR_SPLIT(x)
Model
Parse
KnowledgeMap
Principals
Const → Value
Creator
KnownBy
Phase…
Mutate PrincipalState for Next Run
Ga = g^attacker
Gb = gb…
Alice's PrincipalState
Const → Value
Guard
KnownBy
Wire…
ga, e1
[gb], e2
Bob's PrincipalState
Const → Value
Guard
KnownBy
Wire…
AttackerState
Resolve
ga = g^a
g^a
Deconstruct
DEC(k,m), k → m
m
Reconstruct
k, m → MAC(k,m)
MAC(k, m)
Equivalize
ga^b = gb^a
Learned Value
Queries Analysis
•Check for contradiction to queries after each run
•Terminate when no new values are being learned
Translate to Coq
•Work with Coq Library to perform more in-depth analysis
Protocol Modeling and Verification
•Iterative process through intuitive modeling and optional further Coq modeling

Figure 3. Verifpal analysis methodology. On the left, the three fundamental types usable in Verifpal models are illustrated. As noted in §2.2, all primitives are defined via a standard PrimitiveSpec structure with four logical rules. On the right, a model analysis is illustrated: first, the Verifpal model is parsed and translated into a global immutable *"knowledge map"* structure from which a *"principal state"* is derived for each declared principal. Based on the messages exchanged between these principal states, the attacker obtains values to which it can recursively apply the four transformations discussed in §3 before executing mutated sessions while still following the heuristics touched upon in §3.1, until it is unable to learn new values.

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 transformations (detailed below) on all obtained values.
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 step, 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 reconstruct 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.
- Equivalize. Determines if the attacker can reconstruct or equivalize 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 it 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.

3.1 Preventing State Space Explosion

A common problem among symbolic model protocol verifiers is that for complex protocols, the space of the user states and value combinations that the verifier must assess becomes too large for the verifier to terminate in a reasonable time. Verifpal optimizes for this problem via certain heuristic techniques: first, Verifpal separates its analysis into a number of *stages* in which it gradually allows itself to modify more and more elements of principals' states. Only in later stages are the internal values of certain primitives (which are labeled *"explosive"* in their PrimitiveSpec) mutated. Verifpal also imposes other restrictions, such as limiting the maximum number of inputs and outputs of any primitive to five. Thus, Verifpal achieves unbounded state analysis, similarly to ProVerif, but also applies a set of heuristics that are hopefully more likely to achieve termination in a more reasonable time for large models (such as those seen for TLS 1.3 or Signal with more than three messages). Verifpal also leverages multi-threading and other such techniques to achieve faster analysis. Verifpal's stages segment its search strategy in essentially the following way, with the aim to hold back infinite mutation recursion depth as far as possible, unless queries cannot be contradicted without it:

- **Stage 1:** All of the elements of passive attacker analysis, plus constants and equation exponents may be mutated to `nil` only and not to each other (for equations, this means that `g^a` mutates to `g^nil` but not to `g^b`).
- **Stage 2:** All of the elements of Stage 1, plus non-explosive primitives are mutated but without exceeding a call depth that is pre-determined in relation to the way in which they were employed by principals in the Verifpal model. For example, `HASH(HASH(x))` will not mutate to `HASH(HASH(HASH(y)))` (since the call depth is deeper in the mutation), and `ENC(HASH(k), G^y)` will not mutate to `ENC(PW_HASH(k), k)` (since the *"skeleton"* of the original primitive does not employ `PW_HASH`, but `HASH`, and employs an equation (`G^y`) as the second argument and not a constant (`k`)).
- **Stage 3:** All of the elements of Stage 2, with the inclusion of explosive primitives.
- **Stage 4:** All of the elements of Stage 3, with the addition of constants and equation exponents being replaced with one another and not just `nil`.
- **Stage 4 and beyond:** All of the elements of Stage 3, with the addition of primitives being allowed a mutation depth of $n - 3$ where n represents the current Stage, so long as the resulting mutations have the same *"skeleton"* as defined in Stage 2.

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. We present in this section an outline of Verifpal's formal analysis methodology, in addition to the formalized semantics and analysis logic of the Verifpal Coq Library discussed in §5. From all of this, we draw an incomplete, semi-formal, in-progress set of results that aim to eventually show that:

- If an attacker is unable to obtain a value `m`, then Verifpal will answer that the query passes 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` will be deemed as authenticated `A` $\rightarrow$ `B` for the protocol described in the Verifpal model.

Formally, Verifpal is unable to claim that it never misses an attack in any model that can be expressed within its language. However, our hope is that Verifpal would not miss attacks affecting models of, or resembling, "real-world protocols". Our rationale is that given Verifpalś goals, it is preferable to avoid risking non-terminating analysis in order

⟨model⟩ ::= ⟨attacker⟩ ⟨principal⟩ (⟨principal⟩ | ⟨message⟩ | ⟨phase⟩)+ ⟨queries⟩

⟨attacker⟩ ::= ‘`attacker[`’ (‘`active`’ | ‘`passive`’) ‘`]`’

⟨principal⟩ ::= ‘`principal`’ ⟨string⟩ ‘`[`’ (⟨knows⟩ | ⟨generates⟩ | ⟨leaks⟩ | ⟨assignment⟩)+ ‘`]`’

⟨knows⟩ ::= ‘`knows `’ (‘`private`’ | ‘`public`’ | ‘`password`’) ⟨constant⟩ (‘`,`’ ⟨constant⟩)*

⟨generates⟩ ::= ‘`generates `’ ⟨constant⟩ (‘`,`’ ⟨constant⟩)*

⟨leaks⟩ ::= ‘`leaks `’ ⟨constant⟩ (‘`,`’ ⟨constant⟩)*

⟨assignment⟩ ::= ⟨constant⟩ (‘`,`’ ⟨constant⟩)* ‘` = `’ (⟨primitive⟩ | ⟨equation⟩)

⟨message⟩ ::= ⟨string⟩ ‘` → `’ ⟨string⟩ ‘`: `’ (⟨constant⟩ | ⟨guardedConstant⟩) (‘`,`’ (⟨constant⟩ | ⟨guardedConstant⟩))*

⟨phase⟩ ::= ‘`phase[`’ ⟨number⟩ ‘`]`’

⟨queries⟩ ::= ‘`queries[`’ (⟨confidentialityQuery⟩ | ⟨authenticationQuery⟩ | ⟨freshnessQuery⟩ | ⟨unlinkabilityQuery⟩)* ‘`]`’

⟨confidentialityQuery⟩ ::= ‘`confidentiality? `’ ⟨constant⟩ ⟨queryOptions⟩?

⟨authenticationQuery⟩ ::= ‘`authentication? `’ ⟨string⟩ ‘` → `’ ⟨string⟩ ‘`: `’ ⟨constant⟩ ⟨queryOptions⟩?

⟨freshnessQuery⟩ ::= ‘`freshness? `’ ⟨constant⟩ ⟨queryOptions⟩?

⟨unlinkabilityQuery⟩ ::= ‘`unlinkability? `’ ⟨constant⟩ ‘`,`’ ⟨constant⟩ (‘`,`’ ⟨constant⟩)* ⟨queryOptions⟩?

⟨queryOptions⟩ ::= ‘`[`’ ⟨queryOption⟩* ‘`]`’

⟨queryOption⟩ ::= ‘`precondition`’ ‘`[`’ ⟨message⟩ ‘`]`’

⟨constant⟩ ::= ⟨string⟩

⟨guardedConstant⟩ ::= ‘`[`’ ⟨constant⟩ ‘`]`’

⟨primitive⟩ ::= ⟨primitiveName⟩ ‘`(`’ (⟨constant⟩ | ⟨primitive⟩ | ⟨equation⟩) (‘`,`’ (⟨constant⟩ | ⟨primitive⟩ | ⟨equation⟩))* ‘`)`’ [‘`?`’]

⟨equation⟩ ::= ⟨constant⟩ ‘`^`’ ⟨constant⟩

⟨primitiveName⟩ ::= ‘`BLIND`’ | ‘`UNBLIND`’ | ‘`RINGSIGN`’ | ‘`RINGSIGNVERIF`’ | ‘`PW_HASH`’ | ‘`HASH`’ | ‘`HKDF`’ | ‘`AEAD_ENC`’ | ‘`AEAD_DEC`’ | ‘`ENC`’ | ‘`DEC`’ | ‘`MAC`’ | ‘`ASSERT`’ | ‘`CONCAT`’ | ‘`SPLIT`’ | ‘`SIGN`’ | ‘`SIGNVERIF`’ | ‘`PKE_ENC`’ | ‘`PKE_DEC`’ | ‘`SHAMIR_SPLIT`’ | ‘`SHAMIR_JOIN`’

Figure 4. Verifpal regular language syntax.

to account for attacks that are unlikely to occur in real-world protocol constructions. This leaves us with the problematically subjective definition of what constitutes a "real-world protocol", and implies that Verifpal will for the first few years of its existence require work on grounding and expressing more clearly the constraints of the protocols which can be expressed and for which missed attacks can truly be ruled out.

Our central argument is that the analysis logic described in this section is sufficient in order to capture a majority of confidentiality and authentication attacks within the language. We further buttress this claim with the formalization of Verifpal's semantics and analysis logic in Coq, as shown in §5.

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. *"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).

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.

Mutations and Guarded Constants Except for guarded constants, 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 earlier, once the attacker gains new values through this process, the permutation table is recalculated and 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 the methodology 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 Equivalize 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.

3.3 Analysis Results of Real-World Protocols

It is important to understand that the measures Verifpal takes to encourage analysis termination, as touched upon earlier in §3.1, do not affect the comprehensiveness of results that Verifpal can obtain from the analysis of real-world protocols. Verifpal ships with an integration testing suite comprised of 54 testing protocols. Of these, we highlight the following non-trivial protocols which have also been modeled in other symbolic analysis tools:

- **Signal** is modeled in Verifpal as well as in ProVerif [57] and in Tamarin [32]. All three analyses obtain matching results when checking for message confidentiality, authentication and post-compromise security. Post-compromise security is modeled using temporal logic (see §2.2) in all three analysis frameworks.
- **Scuttlebutt** is modeled in Verifpal as well as in ProVerif [61], CryptoVerif and in Tamarin [38]. All three analyses obtain matching results when checking for message confidentiality and authentication queries.
- Verifpal also obtained results matching state-of-the art analysis on the Telegram MT-Proto "secure chat" protocol [55,62], Firefox Sync and ProtonMail's email encryption feature towards recipients that do not use ProtonMail [56].

Queries were contradicted (or not contradicted) in the same scenarios across Verifpal, ProVerif and Tamarin, depending on which key materials were leaked, and when. Various forms of partial state leakage were tested. Aside from the above relatively sophisticated protocols, Verifpal obtained matching results on many variants of Needham-Schroeder, the "FFGG" "parallel attack" protocol discussed in 3.4, and over 50 other test protocols, some of which are mirrored in ProVerif's own test suite.

Finally, Verifpal was used by the popular Zoom telecommunications software in May 2020 during the entire conception and design process of their revised [26] end-to-end encryption protocol. During this collaboration, Verifpal not only helped the Zoom team design their protocol from scratch but also spotted non-obvious attacks which the Zoom team were able to fix prior to publication.

Listing 1.1. ProVerif Attack Trace

```
new skB: skey creating skB_2 at {1}
out(c, ~M) with ~M = pk(skB_2) at {3}
new n1_1: nonce creating n1_2 at {9} in copy a
new n2_1: nonce creating n2_2 at {10} in copy a
out(c, (~M_1,~M_2)) with ~M_1 = n1_2, ~M_2 = n2_2 at
      {11} in copy a
new n1_1: nonce creating n1_3 at {9} in copy a_1
new n2_1: nonce creating n2_3 at {10} in copy a_1
out(c, (~M_3,~M_4)) with ~M_3 = n1_3, ~M_4 = n2_3 at
      {11} in copy a_1
in(c, (~M_3,~M_1)) with ~M_3 = n1_3, ~M_1 = n1_2 at {5}
       in copy a_2
out(c, ~M_5) with ~M_5 = encrypt((n1_3,n1_2,M),pk(skB_2
      )) at {6} in copy a_2
in(c, ~M_5) with ~M_5 = encrypt((n1_3,n1_2,M),pk(skB_2)
      ) at {12} in copy a_1
out(c, (~M_6,~M_7)) with ~M_6 = n1_2, ~M_7 = encrypt((
      n1_2,M,n1_3),pk(skB_2)) at {14} in copy a_1
in(c, ~M_7) with ~M_7 = encrypt((n1_2,M,n1_3),pk(skB_2)
      ) at {12} in copy a
out(c, (~M_8,~M_9)) with ~M_8 = M, ~M_9 = encrypt((M,
      n1_3,n1_2),pk(skB_2)) at {14} in copy a
The attacker has the message ~M_8 = M.
```

Listing 1.2. Verifpal Attack Trace

```
Result • confidentiality? m — When:
 n1 → nil ← mutated by Attacker (was n1)
 n2 → nil ← mutated by Attacker (was n2)
 msg → PKE_ENC(G^skb, CONCAT(nil, n1, m))
 clear → CONCAT(nil, n1, m)
 x → nil
 y1 → n1
 y2 → m
 unnamed_0 → ASSERT(nil, n1)?
 msg2 → PKE_ENC(G^skb, CONCAT(n1, m, n1)) ←
       obtained by Attacker

m is obtained:
 msg → PKE_ENC(G^skb, CONCAT(n1, m, n1)) ←
       mutated by Attacker
   (was PKE_ENC(pkb, CONCAT(n1, n2, m)))
 clear → CONCAT(n1, m, n1)
 x → n1
 y1 → m ← obtained by Attacker
 y2 → n1
 unnamed_0 → ASSERT(n1, n1)?
 msg2 → PKE_ENC(G^skb, CONCAT(m, n1, n1))
m (m) is obtained by Attacker.
```

Figure 5. ProVerif and Verifpal attack traces for the protocol discussed in §3.4.

3.4 Improving Readability of Protocol Analysis Traces

ProVerif and Verifpal both ship with a set of example protocol models. Of those protocol models, the “FFGG” protocol [65] is included due to it requiring a parallel attack in order for confidentiality queries to be contradicted. We take Verifpal and ProVerif’s models of FFGG and modify them to be as functionally and structurally similar as possible.[5]

Figure 5 shows a ProVerif trace compared to a Verifpal trace for the confidentiality query contradiction on message `m` in FFGG.[6] The Verifpal trace shows the two parallel sessions required for the attack to be pulled off, clearly noting which values had to be mutated by the attacker alongside their original resolved values. Each session ends with the message that had to be obtained by the attacker and re-used in the following session for the attack to work. In this case, `msg2`, which resolved to `PKE_ENC(G^skb, CONCAT(n1, m, n1))`, was injected by the active attacker to replace `msg` in the second session as it traveled across the network.

As discussed in §1.2, one of Verifpal’s design principles is to improve the readability of protocol analysis traces. In line with this goal, Verifpal’s trace also makes it easier to see how the mutation of preceding values affects the resolution of values that are composed of those mutated values. In longer, more complex protocol models, Verifpal is able to still output traces of relatively similar size and simplicity, whereas the growth of complexity and length in ProVerif traces is more substantial.

[5] The full Verifpal and ProVerif FFGG models are available at https://source.symbolic.software/verifpal/verifpal/-/tree/master/examples.

[6] ProVerif is also capable of outputting graphical representations of attack traces, which could make them easier to read and understand.

4 Case Study: Contact Tracing

During the COVID-19 pandemic, a rise was observed in the number of proposals for privacy-preserving pandemic and contact tracing protocols. Arguably the most popular and well-analyzed of these proposals is the Decentralized Privacy-Preserving Proximity Tracing (DP-3T) protocol [75], which aims to *"simplify and accelerate the process of identifying people who have been in contact with an infected person, thus providing a technological foundation to help slow the spread of the SARS-CoV-2 virus"*, and to *"minimize privacy and security risks for individuals and communities and guarantee the highest level of data protection."*

4.1 Modeling DP-3T in Verifpal

To demonstrate DP-3T, we will assume that the principals participating in this simulation are the following:

- A population of 3 individuals: Alice, Bob, and Charlie, each of them possessing a smartphone: `SmartphoneA`, `SmartphoneB`, and `SmartphoneC` respectively;
- A Healthcare Authority serving this population;
- A Backend Server, that individuals can communicate with to obtain daily information.

We begin by defining an attacker which matches with our security model, which, in this case, is an active attacker. We then proceed to illustrate our model as a sequence of days in which DP-3T is in operation within the life cycle of a pandemic.

Day 0: Setup Phase We assume that no new individuals were diagnosed with the disease on Day 0 of using DP-3T. This means that the Healthcare Authority and the Backend Server will not act at this stage and we can simply ignore them for now.

The DP-3T specification states that every principal, when first joining the system, should generate a random secret key (`SK`) to be used for one day only. For every `SK` value, and the knowledge of a public "broadcast key" value, principals should compute multiple Unique Ephemeral ID values (`EphID`) using a combination of a PRG and a PRF. The method of generating `EphID` is analogous with the HKDF function from Verifpal. We could add the following lines of code to our file in order to model Alice's `SmartphoneA`:

DP-3T: SmartphoneA, B and C Setup

```
principal SmartphoneA[
  knows public BroadcastKey
  generates SK0A
  EphID00A, EphID01A, EphID02A = HKDF(nil, SK0A, BroadcastKey)
]
```

Whenever two principals would come to be in physical proximity of each other, they would automatically exchange `EphID`s. Once a principal uses an `EphID` value, they discard it and use another one when performing an exchange with another principal.

Let's imagine that Alice and Bob came into contact. It would mean that Alice sent `EphID00A` in a message to Bob and that Bob sent `EphID00B` to Alice. Further, let's say that in the conclusion of Day 0, Bob sits behind Charlie in the Bus.

```
DP-3T: EphID Communication

SmartphoneA→ SmartphoneB: EphID00A
SmartphoneB→ SmartphoneA: EphID00B
SmartphoneC→ SmartphoneB: EphID01C
SmartphoneB→ SmartphoneC: EphID01B
```

Day 1 The Backend Server will automatically publish the `SK` values of people who were infected to the members of the general population. These values were previously unpublished and thus were private and only known by their generators and the server.

```
DP-3T: BackendServer Communication

principal BackendServer[
  knows private infectedPatients0
]
BackendServer→ SmartphoneA: infectedPatients0 // Also to SmartphoneB/C
```

Every day starting from Day 1, DP-3T mandates that principals will generate new `SK` values. The new value will be equal to the hash of the `SK` value from the day before. Principals will also generate `EphIDs` just like before.

```
DP-3T: EphID Generation

principal SmartphoneA[
  SK1A = HASH(SK0A)
  EphID10A, EphID11A, EphID12A = HKDF(nil, SK1A, BroadcastKey)
]
// Similar principal blocks for SmartphoneB/C here
```

Thankfully, Alice, Bob and Charlie were committed to self-confinement and have stayed at home, so they did not exchange `EphIDs` with anyone.

Day 2 A similar sequence of events takes place. Since it is sufficient to define the values that we will need later on in our model, we will just define a block for Alice.

```
DP-3T: EphID Generation

principal SmartphoneA[
  SK2A = HASH(SK1A)
  EphID20A, EphID21A, EphID22A = HKDF(nil, SK2A, BroadcastKey)
]
```

Fast-Forward to Day 15 Unfortunately, Alice tests positive for COVID-19. Since this breaks the routine that happened between Day 1 and Day 15, we will announce a new phase (see §2.2) in our protocol model:

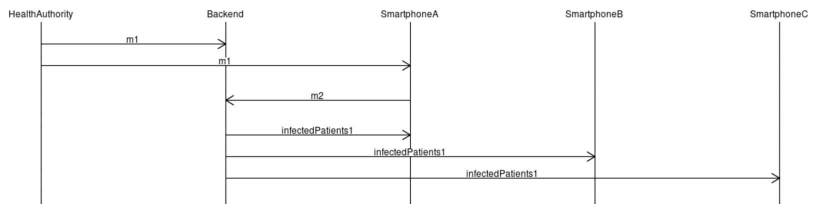

Figure 6. A summary of the parties and network exchanges involved in Day 15 of our Verifpal model of the DP-3T protocol.

DP-3T: Declaring a New Phase

```
phase[1]
```

Alice decides to announce her infection anonymously using DP-3T. This means that she will have to securely communicate `SK1A` (her `SK` value from 14 days ago) to the Backend Server, using a unique trigger token provided by the healthcare authority. Assuming that the Backend Server and the Healthcare Authority share a secure connection, and that a key `ephemeral_sk` has been exchanged off the wire by the Healthcare Authority, Alice, and the Backend Server, the Healthcare Authority will encrypt a freshly generated triggerToken using `ephemeral_sk` and send it to both Alice and the Backend Server.

DP-3T: Sending Tokens to HealthCareAuthority

```
principal HealthCareAuthority[
  generates triggerToken
  knows private ephemeral_sk
  m1 = ENC(ephemeral_sk, triggerToken)
]
HealthCareAuthority→ BackendServer : [m1]
HealthCareAuthority→ SmartphoneA : m1
```

Then, Alice would have to use an AEAD cipher to encrypt `SK1A` using `ephemeral_sk` as the key and `triggerToken` as additional data and send the output to the BackendServer. Note that Alice can only obtain `triggerToken` after decrypting `m1` using `ephemeral_sk`.

DP-3T: Communicating with BackendServer

```
principal SmartphoneA[
  knows private ephemeral_sk
  m1_dec = DEC(ephemeral_sk, m1)
  m2 = AEAD_ENC(ephemeral_sk, SK1A, m1_dec)
]
SmartphoneA→ BackendServer: m2
```

The Backend Server will now have to decrypt `m1` to receive the `triggerToken` in the same way that Alice did, then attempt to decrypt `m2`. If that decryption was successful, the server would obtain `SK1A` and would be sure that the value came from Alice because it is only Alice who knows both `triggerToken` and `SK1A` at the same time as defined in the protocol.

Finally, the Backend Server will add `SK1A` to the list of infected patients previously defined, and then send this list to all of the individuals in this community.

DP-3T: Updating List of Infected Patents

```
  principal BackendServer [
    knows private ephemeral_sk
    m2_dec = AEAD_DEC(ephemeral_sk, m2, DEC(ephemeral_sk, m1))?
    infectedPatients1 = CONCAT(infectedPatients0, m2_dec)
  ]
  BackendServer→ SmartphoneA: infectedPatients1 // Also to SmartphoneB/C
```

Everything that happened in Day 15 can be summarized in Figure 6.

4.2 DP-3T Analysis Results

Since `SK1A` is now shared publicly, the DP-3T software running on anyone's phone should be able to re-generate all `EphID` values generated by the owner of `SK1A` starting from 14 days prior to the day of diagnosis. These values would then be compared with the list of `EphIDs` they have received. Everyone who came in contact with Alice will therefore be notified that they have exchanged `EphIDs` with someone who has been diagnosed with the illness without revealing the identity of that person.

DP-3T: Queries

```
queries[
  // Check if values shared 15 days before testing get flagged
  confidentiality? EphID02A
  // Check if Alice's previous EphIDs can be computed by passerbys
  confidentiality? EphID10A, EphID11A, EphID12A, EphID20A, EphID21A, EphID22A
  // Is the server able to Authenticate Alice as the sender of m2?
  authentication? SmartphoneA→ BackendServer: m2
  // Unlinkability of HKDF values
  unlinkability? EphID02A, EphID00A, EphID01A
]
```

The results of our initial modeling in Verifpal suggest to us the following:

- No `EphIDs` generated by Alice are known by any parties before Alice announces her illness.
- `EphID02A` remains confidential even after Alice declaring her illness. Note that it was generated 15 days before Alice got tested.
- All of the following values `EphID10A`, `EphID11A`, `EphID12A`, `EphID20A`, `EphID21A`, `EphID22A` have been recoverable by an attacker in `phase[1]` after Alice announces her illness.

These results come in line with what is expected from the protocol. We note that the security of communication channels between Healthcare Authorities, Backend Servers, and Individuals have not been defined, and we have placed our hypothetical security conditions in order to focus on quickly sketching the DP-3T protocol.

While further analysis will be required in order to better elucidate the extent of the obtained security guarantees, Verifpal radically speeds up this process by allowing for the automated translation of easy-to-write Verifpal models to full-fat Coq and ProVerif models, as discussed in §5.

5 Verifpal in Coq and ProVerif

```
Protocol: test.vp
attacker[passive]
principal Bob [ knows private a ]
principal Alice [
  knows private a
  generates ma
  ka = HASH(a)
  c = ENC(ka, ma)
]
Alice→ Bob: c
principal Bob [
  kb = HASH(a)
  mb = DEC(kb, c)
]
phase[1]
Alice [ leaks a ]
queries[ confidentiality? ma ]
```

Figure 7. A simple Verifpal model used in order to illustrate the Coq Library.

Verifpal's core verification logic and semantics can be captured in Coq using our Verifpal Coq Library. This library includes high level functions that can be used to perform analysis on any valid protocol modeled using the Verifpal language. Additionally, a Verifpal functionality has been developed that automatically generates Coq code which uses the high level functions from our library, when input with a protocol file. This automates the process of translating Verifpal models into representations that could be further analysed using Coq's powerful paradigm of constructive logic. Once executed, this Coq code would yield results for the queries defined in the protocol model.

Parallel analysis confirmation in Coq and ProVerif. In addition to being able to output Coq implementations of Verifpal models, Verifpal is also able to translate Verifpal protocol models into ProVerif models. A similar approach is used: the generated models include a pre-defined library implementing all Verifpal primitives in the applied-pi calculus. ProVerif tables are used to keep track of principal states, with principal blocks being converted to `let` declarations. A public channel is used to exchange values and to potentially leak them to the attacker. Finally, the top-level process is declared as a parallel execution of all principal `let` declarations. This latter formulation of the Verifpal model in ProVerif allows us to make use of ProVerif's ability to model the parallel execution of processes.

By providing robust support for automatic translation of arbitrary models into Coq and ProVerif, Verifpal simultaneously allows for its own semantics to be defined more concretely and in relationship to established verification paradigms, while also increasing confidence in its own verification methodology by mirroring its results on security queries within the analysis framework of tools that have existed for decades.

Verifpal semantics in Coq. We define several types to capture all of the primitives of the Verifpal language in Coq. For example, we have defined `constant`, `Principal`, and `knowledgemap` as inductive types to capture the notions of *constant*, *principal* and *knowledgemap* from Verifpal respectively. Whenever a principal declares, generates, assigns, leaks, or receives a message, an item of knowledge would be added to their state.

Suppose that Alice wants to send `c` to Bob, and that the latest `knowledgemap` contains Alice's internal state `a`, `ma`, `ka` as well as Bob's state, most relevantly `a`. We use `send_message` to send `c` from Alice to Bob and thereby update the `knowledgemap` of both principals. Bob's state gets updated with the value `c`, to contain both `a` and `c`, after the function is executed. All of the primitives supported by Verifpal are formally specified in our Coq library. Outputs of certain primitives are defined as sub-types of the type `constant`.

Coq: Constant Definition

```
Inductive constant : Type :=
  | value_c (name: string)
  | ENC_c (key message: constant)
  | HASH1_c (value: constant)
  | ...
```

As an illustrative example, we demonstrate a lemma that proves decidable equality between elements of type `constant`. This lemma essentially captures the functionality of the **`ASSERT`** core primitive.

Coq: Constant Equality Lemma

```
Lemma equal_constant_true : forall (c : constant), c =? c = true.
Proof.
  induction c; simpl; try firstorder.
  apply string_equality. reflexivity.
  rewrite IHc1, IHc2, IHc3, IHc4; auto.
  rewrite IHc1, IHc2, IHc3, IHc4, IHc5; auto.
  rewrite IHc1, IHc2, IHc3, IHc4; auto.
  rewrite IHc1, IHc2, IHc3, IHc4, IHc5; auto.
  rewrite IHc1, IHc2, IHc3, IHc4; auto.
  apply string_equality. reflexivity.
Qed.
```

When Alice performs `c = ENC(ka, ma)`, and then sends `c` over the wire, we would expect that the decryption of `c` would only yield the plaintext `ma` if and only if the key used to decrypt `c` is the same one that was used for encrypting `ma`, as defined in our formalization of the `DEC` primitive (see §2). We provide additional lemmas to prove that our model satisfies the behavior expected from our primitives. For example, we can prove that `DEC(kb, ENC(ka, ma))` would yield `ma` using the `enc_dec` Theorem (see §2).

Verifpal analysis in Coq. Using the functionality provided by the Verifpal Coq library, and the Coq code generation feature of Verifpal, it is possible to perform a symbolic execution of any protocol that can be modeled using Verifpal. In addition, it is possible to independently run the axioms on which our primitives and analysis methodology are defined by simply running the included proofs that are written using the Ltac tactics language supported by Coq. The passive attacker methodology in the Verifpal Coq Library is analogous to that defined in §3:

1. The attacker can gather values: any value leaked, or declared as public is automatically added to the attacker's list of knowledge. In addition, any value sent over the wire is known by the attacker.
2. The attacker attempts to apply transformations on the values learned. The definiton of these transformations accompany our primitive definitions and can be independently verifiable.
3. This process is repeated so long as the attacker was able to learn new values.

We formalize this methodology using an `Attacker` inductive type. An instance of type `Attacker` contains the attacker type, a list of `constant` values that are known by the attacker, as well as the mutability status for every item of knowledge. `constant_meta` acts as a wrapper type for `constant` with the purpose of adding metadata relevant to the declearation of a `constant`. `constant_meta`, along with some helper types, is defined as follows:

Coq: constant_meta Helper Types

```
Inductive qualifier : Type := | public | private | password.
Inductive declaration : Type := | assignment | knows | generates.
Inductive guard_state : Type := | guarded | unguarded.
Inductive leak_state : Type := | leaked | not_leaked.
Inductive constant_meta: Type :=
  | constant_meta_c (c: constant) (d: declaration) (q: qualifier)
    (created_by name: string) (l: leak_state)...
```

`constant_meta` elements are stored inside the `Principal` data structure and constitute the principal's knowledge. Any value that is transmitted over the wire, is also sent as a `constant_meta` along with its corresponding metadata.

Step 1 of the analysis methodology is modeled with the help of two functions:

– `absorb_message_attacker` enables an `Attacker` to learn any value when it is being sent over the wire.
– `absorb_knowledgemap_attacker` enables an `Attacker` to iterate over `Principal` elements found in the `knowledgemap` and their lists of `constant_meta` items. The attacker

can learn a constant_meta that they come across strictly if its (l:leak_state) value equals leaked or if its (q:qualifier) equals **public**, otherwise the value is ignored.

At the end of phase[0] of the example protocol, the attacker would have learned the constant c because it was sent over the wire. At the end of phase[1], the attacker would have learned a in addition to c because it was leaked by Alice.

In phase[1], the attacker was able to reconstruct HASH1_c a after learning a then consequently attempted **DEC**(HASH1_c a)c. As discussed earlier, the **DEC** operation would reveal the plaintext if the key provided is equivalent to the encryption key. Developing further we obtain **DEC**(HASH1_c a)(ENC_c ka ma) then **DEC**(HASH1_c a)(**ENC**(HASH1_c a)ma), the attacker would then automatically apply the enc_dec lemma (shown in §2) to deduce ma and add it to its knowledge. It is worth noting that all transformations that can be applied by the attacker, just like primitives, are accompanied with independently provable lemmas and theorems.

Verifpal queries are analogous to decidable processes and help us reason about protocols. The confidentiality query defined would translate to *"is the attacker able to obtain the value* ma *after the protocol is executed?"* To answer this, we search in the attacker's knowledge for a value that is equal to ma using the search_by_name_attacker function; if such a value is found, the query "fails", otherwise it "passes". In this case the query would fail, as the attacker was able to obtain ma by applying the methodology from the previous section. Generating a Coq implementation of the protocol discussed will yield an identical result, and could allow the user to verify the soundness of this result by executing the proofs included in the code.

6 Discussion and Conclusion

Verifpal's focus on prioritizing usability leads it to have no road map to support, for example, declaring custom primitives or rewrite rules as supported in ProVerif and Tamarin. However, future work focuses on giving Verifpal the fine control that tools such as ProVerif can offer over how protocol processes are executed. However, Verifpal has recently managed to gain support for protocol *phases* and parametrized queries (useful for modeling post-compromise security) as well as querying for unlinkability and other advanced features. Verifpal also ships with a Visual Studio Code extension that turns Verifpal into essentially an IDE for the modeling, development, testing and analysis of protocol models. The extension offers live analysis feedback and diagram visualizations of models being described and supports translating models automatically into Coq. We plan to also launch within the coming weeks support for translating Verifpal models into prototype Go implementations immediately, allowing for live real-world testing of described protocols.

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 [54] 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 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 for Windows, Linux, macOS and FreeBSD, along with a user manual that goes more in-depth into the Verifpal language and analysis methodology, at `https://verifpal.com`.

Acknowledgements

We thank Bruno Blanchet, Benjamin Lipp, Denis Merigoux and the Verifpal users active on the mailing list 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. Abadi, M., Blanchet, B., Fournet, C.: The applied pi calculus: Mobile values, new names, and secure communication. J. ACM **65**(1), 1:1–1:41 (2018). https://doi.org/10.1145/3127586, `http://doi.acm.org/10.1145/3127586`
2. Amin, R., Islam, S.H., Karati, A., Biswas, G.: Design of an enhanced authentication protocol and its verification using AVISPA. In: 2016 3rd International Conference on Recent Advances in Information Technology (RAIT). pp. 404–409. IEEE (2016)
3. Armando, A., Arsac, W., Avanesov, T., Barletta, M., Calvi, A., Cappai, A., Carbone, R., Chevalier, Y., Compagna, L., Cuéllar, J., 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. pp. 267–282. Springer (2012)
4. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P.H., Héam, P.C., Kouchnarenko, O., Mantovani, J., et al.: The AVISPA tool for the automated validation of internet security protocols and applications. In: International conference on computer aided verification. pp. 281–285. Springer (2005)
5. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. In: International Conference on Applied Cryptography and Network Security. pp. 119–135. Springer (2013)
6. Backes, M., Hritcu, C., Maffei, M.: Automated verification of remote electronic voting protocols in the applied pi-calculus. In: IEEE Computer Security Foundations Symposium. pp. 195–209. IEEE (2008)
7. Baelde, D., Delaune, S., Moreau, S.: A Method for Proving Unlinkability of Stateful Protocols. Ph.D. thesis, Irisa (2020)
8. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: SoK: Computer-aided cryptography. In: IEEE Symposium on Security and Privacy (S&P). IEEE (2021)
9. Basin, D., Cremers, C.: Modeling and analyzing security in the presence of compromising adversaries. In: Computer Security - ESORICS 2010. Lecture Notes in Computer Science, vol. 6345, pp. 340–356. Springer (2010)
10. Basin, D., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., Stettler, V.: A formal analysis of 5G authentication. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1383–1396. ACM (2018)

11. Basin, D., Radomirovic, S., Schmid, L.: Alethea: A provably secure random sample voting protocol. In: IEEE 31st Computer Security Foundations Symposium (CSF). pp. 283–297. IEEE (2018)
12. Basin, D.A., Cremers, C.J.: 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. Lecture Notes in Computer Science, vol. 6247, pp. 1–18. Springer (2010)
13. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(2), 1–45 (2011)
14. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
15. Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zinzindohoue, J.K.: A messy state of the union: Taming the composite state machines of TLS. In: IEEE Symposium on Security and Privacy (S&P). pp. 535–552. IEEE (2015)
16. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified models and reference implementations for the TLS 1.3 standard candidate. In: IEEE Symposium on Security and Privacy (S&P). pp. 483–502. IEEE (2017)
17. Bhargavan, K., Delignat-Lavaud, A., Kobeissi, N.: Formal modeling and verification for domain validation and ACME. In: International Conference on Financial Cryptography and Data Security. pp. 561–578. Springer (2017)
18. Bhargavan, K., Lavaud, A.D., Fournet, C., Pironti, A., Strub, P.Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: IEEE Symposium on Security and Privacy (S&P). pp. 98–113. IEEE (2014)
19. Bhargavan, K., Leurent, G.: On the practical (in-) security of 64-bit block ciphers: Collision attacks on http over tls and openvpn. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 456–467 (2016)
20. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: new generation of memory-hard functions for password hashing and other applications. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 292–302. IEEE (2016)
21. Blanchet, B.: CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In: Dagstuhl seminar on Applied Formal Protocol Verification. p. 117 (2007)
22. Blanchet, B.: Security protocol verification: Symbolic and computational models. In: Principles of Security and Trust (POST). pp. 3–29 (2012)
23. Blanchet, B.: Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In: Foundations of Security Analysis and Design VII, pp. 54–87. Springer (2013)
24. Blanchet, B.: 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)
25. Blanchet, B., Chaudhuri, A.: Automated formal analysis of a protocol for secure file sharing on untrusted storage. In: IEEE Symposium on Security and Privacy (S&P). pp. 417–431. IEEE (2008)
26. Blum, J., Booth, S., Gal, O., Krohn, M., Len, J., Lyons, K., Marcedone, A., Maxim, M., Mou, M.E., O'Connor, J., et al.: E2e encryption for Zoom meetings (2020), `https://github.com/zoom/zoom-e2e-whitepaper`
27. Bruni, A., Drewsen, E., Schürmann, C.: Towards a mechanized proof of selene receipt-freeness and vote-privacy. In: International Joint Conference on Electronic Voting. pp. 110–126. Springer (2017)
28. Chandra, A.K., Harel, D.: Horn clause queries and generalizations. The Journal of Logic Programming **2**(1), 1–15 (1985)

29. Chaum, D.: Blind signatures for untraceable payments. In: Advances in cryptology. pp. 199–203. Springer (1983)
30. Cheval, V., Blanchet, B.: Proving more observational equivalences with ProVerif. In: International Conference on Principles of Security and Trust. pp. 226–246. Springer (2013)
31. Cheval, V., Kremer, S., Rakotonirina, I.: DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice. Research report, INRIA Nancy (May 2018), https://hal.inria.fr/hal-01698177
32. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 451–466. IEEE (2017)
33. Cohn-Gordon, K., Cremers, C., Garratt, L.: On post-compromise security. In: IEEE Computer Security Foundations Symposium (CSF). pp. 164–178. IEEE (2016)
34. Cortier, V., Wiedling, C.: A formal analysis of the norwegian e-voting protocol. In: International Conference on Principles of Security and Trust. pp. 109–128. Springer (2012)
35. Cremers, C., Dehnel-Wild, M.: Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. 2019 Network and Distributed System Security Symposium (NDSS) (2019)
36. Cremers, C., Hirschi, L.: 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)
37. Cremers, C., Horvat, M., Hoyland, J., Scott, S., van der Merwe, T.: A comprehensive symbolic analysis of TLS 1.3. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1773–1788. ACM (2017)
38. Cremers, C., Jackson, D.: Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman. IEEE Computer Security Foundations Symposium (CSF) **19** (2019)
39. Cremers, C.J., Lafourcade, P., Nadeau, P.: Comparing state spaces in automatic protocol analysis. In: Formal to Practical Security. Lecture Notes in Computer Science, vol. 5458/2009, pp. 70–94. Springer Berlin / Heidelberg (2009)
40. Cremers, C.: Feasibility of multi-protocol attacks. In: Proc. of The First International Conference on Availability, Reliability and Security (ARES). pp. 287–294. IEEE Computer Society, Vienna, Austria (April 2006), http://www.win.tue.nl/~ecss/downloads/mpa-ares.pdf
41. Cremers, C.: The Scyther Tool: Verification, falsification, and analysis of security protocols. In: Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc. Lecture Notes in Computer Science, vol. 5123/2008, pp. 414–418. Springer (2008). https://doi.org/10.1007/978-3-540-70545-1_38
42. Cremers, C.: Key exchange in IPsec revisited: formal analysis of IKEv1 and IKEv2. In: Proceedings of the 16th European conference on Research in computer security. pp. 315–334. ESORICS, Springer-Verlag, Berlin, Heidelberg (2011)
43. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
44. Delaune, S., Kremer, S., Ryan, M.: Verifying privacy-type properties of electronic voting protocols. Journal of Computer Security **17**(4), 435–487 (2009)
45. Doghmi, S.F., Guttman, J.D., Thayer, F.J.: Searching for shapes in cryptographic protocols. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 523–537. Springer (2007)
46. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on information theory **29**(2), 198–208 (1983)
47. Donenfeld, J.A.: WireGuard: Next generation kernel network tunnel. In: Network and Distributed System Security Symposium (NDSS) (2017)

48. Donenfeld, J.A., Milner, K.: Formal verification of the WireGuard protocol. Tech. rep., Technical Report (2017)
49. Escobar, S., Meadows, C., Meseguer, J.: Maude-npa: Cryptographic protocol analysis modulo equational properties. In: Foundations of Security Analysis and Design V, pp. 1–50. Springer (2009)
50. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 8413, pp. 187–201 (2014)
51. Girol, G., Hirschi, L., Sasse, R., Jackson, D., Cremers, C., Basin, D.: A spectral analysis of noise: A comprehensive, automated, formal analysis of diffie-hellman protocols. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/girol
52. Hirschi, L., Baelde, D., Delaune, S.: A method for verifying privacy-type properties: the unbounded case. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 564–581. IEEE (2016)
53. Hoare, C.A.R.: Communicating sequential processes. In: The origin of concurrent programming, pp. 413–443. Springer (1978)
54. Jackson, D., Cremers, C., Cohn-Gordon, K., Sasse, R.: Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In: ACM CCS 2019 (2019)
55. Jakobsen, J., Orlandi, C.: On the cca (in) security of mtproto. In: Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices. pp. 113–116 (2016)
56. Kobeissi, N.: An analysis of the protonmail cryptographic architecture. IACR Cryptol. ePrint Arch. **2018**, 1121 (2018)
57. Kobeissi, N., Bhargavan, K., Blanchet, B.: Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In: IEEE European Symposium on Security and Privacy (EuroS&P). pp. 435–450. IEEE (2017)
58. Kobeissi, N., Nicolas, G., Bhargavan, K.: Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. In: IEEE European Symposium on Security and Privacy (EuroS&P) (2019)
59. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Advances in Cryptology (CRYPTO), pp. 631–648. IACR (2010)
60. Lafourcade, P., Puys, M.: Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In: International Symposium on Foundations and Practice of Security. pp. 137–155. Springer (2015)
61. Lapiha, O.: A cryptographic investigation of secure scuttlebutt. Tech. rep., École Normale Supérieure (2019)
62. Lee, J., Choi, R., Kim, S., Kim, K.: Security analysis of end-to-end encryption in telegram. In: Simposio en Criptografía Seguridad Informática, Naha, Japón. Disponible en https://bit.ly/36aX3TK (2017)
63. Lipp, B., Blanchet, B., Bhargavan, K.: A mechanised cryptographic proof of the WireGuard virtual private network protocol. In: IEEE European Symposium on Security and Privacy (EuroS&P) (2019)
64. Miculan, M., Urban, C.: Formal analysis of Facebook Connect single sign-on authentication protocol. In: SOFSEM. vol. 11, pp. 22–28. Citeseer (2011)
65. Millen, J.: A necessarily parallel attack. In: Workshop on Formal Methods and Security Protocols. Citeseer (1999)
66. Oak, P.: Kanto Regional Pokédex. Kanto Region Journal on Pokémon Research **19** (1996)
67. Percival, C., Josefsson, S.: The scrypt password-based key derivation function. IETF Draft URL: http://tools. ietf. org/html/josefsson-scrypt-kdf-00. txt (accessed: 30.11. 2012) (2016)

68. Pereira, O., Rochet, F., Wiedling, C.: Formal analysis of the FIDO 1. x protocol. In: International Symposium on Foundations and Practice of Security. pp. 68–82. Springer (2017)
69. Protzenko, J., Beurdouche, B., Merigoux, D., Bhargavan, K.: Formally verified cryptographic web applications in WebAssembly. In: IEEE Symposium on Security and Privacy (S&P). p. 0. IEEE (2019)
70. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., et al.: Verified low-level programming embedded in F. Proceedings of the ACM on Programming Languages **1**(ICFP), 17 (2017)
71. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., et al.: Verified low-level programming embedded in F. Proceedings of the ACM on Programming Languages **1**(ICFP), 17 (2017)
72. Schmidt, B., Meier, S., Cremers, C., Basin, D.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Chong, S. (ed.) IEEE Computer Security Foundations Symposium (CSF), Cambridge, MA, USA, June 25-27, 2012. pp. 78–94. IEEE (2012)
73. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
74. Steinbrecher, S., Köpsell, S.: Modelling unlinkability. In: International Workshop on Privacy Enhancing Technologies. pp. 32–47. Springer (2003)
75. Tronosco, C., et al.: Decentralized privacy-preserving proximity tracing (April 2020)
76. Woo-Sik, B.: Formal verification of an RFID authentication protocol based on hash function and secret code. Wireless personal communications **79**(4), 2595–2609 (2014)