Verifpal

Radu-Constantin Onutu
Faculty of Mathematics and Computer Science, University of Bucharest

Abstract

Verifpal is an automated symbolic verifier for cryptographic protocols, designed around an intuitive specification language and a small set of built-in cryptographic primitives to reduce modeling errors. The tool supports analysis under a passive or active attacker and provides query types that directly target common security goals such as confidentiality, authentication, and freshness. Following the project requirements, Verifpal is introduced and then used on toy examples covered in the course. In particular, the Denning–Sacco protocol and the Needham–Schroeder public-key protocol are modeled to illustrate how standard secrecy and authentication expectations can fail under an active network attacker and how simple fixes (e.g., binding identities to messages) change the verification outcome. A third small example based on an unauthenticated Diffie–Hellman exchange is included as an additional suggestive case for illustrating man-in-the-middle behavior and the effect of adding authentication.

1 Introduction

Security protocols are often short on paper, but their security guarantees can fail because of small design details (missing identity binding, replayable messages, or missing authentication of key material). For this reason, many courses study symbolic protocol verification, where protocol messages are modeled as abstract terms and the attacker is modeled by generic deduction rules (e.g., rules for composing/decomposing pairs and encrypting/decrypting when keys are known). In this setting, common goals are expressed as precise checks, such as secrecy ("can the attacker derive term M?") and authentication (often expressed via event-style correspondences in verification tools).

This report focuses on Verifpal, an automated verifier that aims to make protocol modeling closer to how protocols are described informally, while keeping the model precise. Verifpal reasons with explicit principals (each with its own state and knowledge) and uses a fixed set of built-in cryptographic primitives to reduce modeling errors caused by user-defined crypto. The tool also emphasizes readable analysis output and includes heuristics to avoid non-termination due to state-space explosion.

Following the project requirements, the report presents Verifpal and applies it to toy examples discussed in the course, plus an additional simple suggestive example. The selected course examples are the Denning–Sacco protocol (vulnerability and a standard fix by binding identities) and the Needham–Schroeder public-key protocol (a classic case for analyzing authentication under an active attacker).

2 Background

The course treats security protocols in a symbolic model: protocol messages are represented as terms built from names (nonces, keys, identities) and function symbols (pairing, encryption, signatures). An adversary is typically assumed to control the network (Dolev–Yao style) and can intercept, replay, and compose messages, while also being able to apply public constructors and known destructors. In the ProVerif course, attacker knowledge is captured by a predicate such as $att(t)$ ("the attacker knows term t"), and the attacker's abilities are encoded as deduction rules (e.g., building encryptions from known plaintext/key and extracting plaintext if the right key is known).

Two security goals appear repeatedly in the examples:

- Secrecy is typically checked by asking whether

a target term becomes derivable by the adversary. In ProVerif-style queries, this is expressed by a query of the form *query attacker*(M), meaning that the tool searches for a trace in which the attacker can obtain M.

- Authentication is often expressed as a correspondence between events inserted in the protocol roles. For example, a query of the form $event(end(...)) ==> event(begin(...))$ expresses that whenever a party finishes a run, the matching "begin" event must have occurred earlier; an injective variant additionally requires a one-to-one matching of sessions.

The course also introduces different verification backends and modeling styles. ProVerif [1] can translate protocols into Horn clauses and then reason about reachability of attacker facts (such as $att(secret)$) using resolution-like saturation techniques. In contrast, Tamarin models protocols as multiset rewriting rules over facts that represent global state and network interaction (e.g., $Out(t)$ for sending, $In(t)$ for receiving, and $K(t)$ for adversary knowledge), and it proves (or refutes) trace properties stated as lemmas.

3 Verifpal overview

Verifpal [2] is an automated symbolic verifier for cryptographic protocols, with a modeling style that aims to look close to an "informal" protocol description while still being precise enough for formal analysis. Its main goal is usability without dropping core symbolic verification features such as an active attacker model and unbounded sessions.

3.1 Language and modeling style

A Verifpal model usually has four parts: (1) the attacker type, (2) principals and their local computations, (3) network messages, and (4) security queries. The attacker is declared as either *attacker*[*passive*] or *attacker*[*active*], where the active attacker can intercept and modify network values. Protocol roles are written as *principal Alice*[...], *principal Bob*[...], etc., each with its own state and knowledge.

To reduce modeling mistakes, Verifpal enforces a disciplined style:

- crypto operations are built-in primitives (users do not define custom primitives), which is meant to avoid incorrect crypto definitions in models;
- constants are immutable and live in a global namespace, so values cannot be silently "reused" or redefined in confusing ways.

Principals typically declare what they know (*knows*), what they freshly generate per session (*generates*), what they intentionally leak (*leaks*), and how new values are computed using primitives or equations.

3.2 Queries

Verifpal supports several query types; the most relevant for course-style toy examples are:

- Confidentiality queries: ask whether the attacker can obtain a given sensitive value.
- Authentication queries: rely on "checked primitives" (e.g., signature verification or authenticated decryption). Intuitively, authentication is violated if a principal successfully performs a checked operation using a value that did not originate from the claimed sender.
- Freshness queries: target replay behavior across sessions by checking whether values stay fresh between protocol runs (and whether an active attacker can force non-freshness).

Verifpal also provides phases for modeling time/order constraints (useful for post-compromise scenarios).

3.3 Analysis methodology

Internally, Verifpal parses a model into a global "knowledge map" and derives a state for each principal. From the attacker's point of view, analysis proceeds by collecting network-visible values and repeatedly applying four main transformations on attacker knowledge: **Resolve**, **Deconstruct**, **Reconstruct**, and **Equivalize**.

For an active attacker, Verifpal additionally runs mutated sessions where unguarded values can be

substituted by other known or attacker-crafted values, iterating across sessions while new information is still being learned. After each run, queries are checked for contradictions (i.e., possible attacks).

Because unbounded-session symbolic analysis can explode in size, Verifpal uses heuristics intended to improve termination, such as analyzing in stages and restricting how "explosive" internal values of primitives are mutated.

The tool explicitly prioritizes termination and usability over producing full proof objects, meaning the design accepts compromises in completeness in exchange for practical analysis behavior.

3.4 Output and tool support

A practical advantage is that Verifpal tries to present results in a readable way, linking a violated query to a concrete attack scenario (instead of only showing low-level deduction chains). In addition, Verifpal can translate models into Coq (with a formalized semantics for the passive-attacker setting) and can also translate models to ProVerif for parallel checking, which increases confidence by cross-validating results in another framework.

4 Examples from the course in Verifpal

4.1 Denning–Sacco

The Denning–Sacco example in the course is a clean demonstration that encrypting a signed session key is not automatically enough if the session key is not bound to the intended peer (identity confusion / key substitution). The ProVerif run in the course shows $secretB$ becomes reachable by the attacker.

In the vulnerable variant, A signs only k, encrypts it to a public key received from the network ($pk2$), and then B uses the recovered k to protect $secretB$. The Horn clause trace shows that if the attacker supplies $pk(y)$ instead of Bob's key, the attacker can learn k and then decrypt $secretB$.

```
attacker[active]

principal Alice[
  generates k
]

// The vulnerability: Alice encrypts
// for 'someone' (attacker) thinking it is
// the intended recipient.
Alice -> Bob: ENC(pkB, SIGN(k, skA))

principal Bob[
  knows private secretB
]

// Bob reuses the compromised k
Bob -> Alice: ENC(k, secretB)

queries[
  confidentiality? secretB
]
```

The corrected Denning–Sacco version signs a tuple that includes both endpoints and the session key (e.g., (pkA, pkB, k)), so that a key intended for Bob cannot be replayed/retargeted to someone else without detection.

4.2 Needham–Schroeder Public Key (NSPK)

The course presents NSPK with the standard 3-message flow:

$A \to B : \{A, na\}_{pk(B)}$
$B \to A : \{na, nb\}_{pk(A)}$
$A \to B : \{nb\}_{pk(B)}$

ProVerif's summary in the slides shows a typical outcome: secrecy for values tied to Bob's view of the run fails ($secretBNa$, $secretBNb$ become reachable), while Alice's secrecy checks succeed in that model. The slides also explain the common trick: instead of directly querying secrecy of nonces, use each nonce as a symmetric key to encrypt a private "marker" secret, and query secrecy of that marker.

```
queries[
  confidentiality? secretANa
  confidentiality? secretANb
  confidentiality? secretBNa
  confidentiality? secretBNb
]
```

In the course, NSPK authentication is expressed using ProVerif correspondence events.

In Verifpal, authentication queries are phrased differently: they focus on whether a party can successfully use some value inside a checked operation even though it did not really come from the claimed sender. So, for NSPK, the cleanest "minimal" use is secrecy-style confidentiality queries; authentication-style checks usually become more meaningful when the protocol includes explicit checked steps (signature verification, AEAD decryption, MAC verification, etc.).

5 Other example

This section explores a standard Diffie–Hellman (DH) key exchange. This example is chosen to illustrate how Verifpal models algebraic properties (like modular exponentiation) and how it detects Man-in-the-Middle (MITM) attacks when authentication is missing.

5.1 Unauthenticated Exchange

In the basic version, Alice and Bob generate private exponents (a and b) and exchange public components (G^a and G^b). They then compute a shared secret (G^{ab}). Verifpal models DH exponentiation using a built-in equation syntax where $G^\wedge a$ represents the generator raised to the power of a.

```
attacker[active]

principal Alice[
  generates a
  ga = G^a
]

Alice -> Bob: ga

principal Bob[
  generates b
  gb = G^b
  gab = ga^b
]

Bob -> Alice: gb

principal Alice[
  gba = gb^a
]

queries[
  confidentiality? gab
]
```

When analyzed under an active attacker, Verifpal identifies a vulnerability. Because the public keys ga and gb are sent over the network without protection, the attacker intercepts ga, replaces it with their own value G^e, and forwards it to Bob. The attacker does the same for gb. Consequently, Alice computes a key shared with the attacker (G^{ae}) thinking it is Bob, and Bob computes a key shared with the attacker (G^{be}) thinking it is Alice. The confidentiality query fails because the attacker can compute the keys used by the principals.

5.2 Authenticated Exchange (Guarded)

To prevent this attack, the exchange must be authenticated. Verifpal allows modeling "guarded constants" using brackets (e.g., $[ga]$). This indicates that the value is transmitted over a channel where the attacker can read the message but cannot modify it, effectively simulating a pre-authenticated channel or a successful signature verification.

```
// ... principal definitions
// same as before ...

// Guarded transmission
Alice -> Bob: [ga]
Bob -> Alice: [gb]

// ... calculations same as before ...

queries[
  confidentiality? gab
]
```

In this variation, the active attacker cannot inject a malicious generator value. The analysis concludes that the confidentiality of the shared secret gab is preserved, confirming that authentication is required to secure a Diffie–Hellman exchange against active adversaries.

6 Discussion and limitations

Verifpal distinguishes itself by prioritizing ease of use and readability over the exhaustive manual control found in other formal verification tools. The modeling language is designed to resemble informal protocol descriptions, making it accessible for students and engineers. However, this design choice introduces specific limitations compared to tools like ProVerif or Tamarin.

One significant limitation is the inability to define custom cryptographic primitives. Unlike ProVerif, where users can define custom equational theories to model specific algebraic properties, Verifpal restricts users to a fixed set of built-in primitives. While this strictness aims to prevent modeling errors, it limits the tool's flexibility for analyzing protocols that rely on novel or non-standard cryptographic constructions.

Another trade-off involves analysis completeness. To avoid the state space explosion problem often encountered in symbolic verification, Verifpal employs heuristics during its analysis. While these heuristics allow the analysis to terminate in a reasonable time for complex protocols, they theoretically preclude the tool from generating full proofs of correctness in the same strict mathematical sense as theorem provers.

To mitigate these limitations, Verifpal supports exporting models to ProVerif and Coq. This feature allows users to benefit from Verifpal's intuitive modeling language while retaining the option to perform cross-verification using more established or rigorous academic frameworks.

7 Conclusion

This report presented an overview and practical application of Verifpal, a symbolic verification tool designed for usability and educational clarity. By modeling standard course examples, the Denning-Sacco and Needham-Schroeder Public Key protocols,it was demonstrated that Verifpal can effectively detect classical vulnerabilities such as identity confusion and lack of freshness. The additional analysis of the Diffie-Hellman exchange further illustrated how the tool captures algebraic properties and man-in-the-middle attacks.

The experiments confirm that Verifpal's strict language design and built-in cryptographic primitives successfully reduce the complexity often associated with formal verification. While it lacks the extensibility of tools like ProVerif or Tamarin regarding custom equational theories, its accessible syntax and readable output make it a valuable instrument for students and practitioners to prototype and verify security protocols efficiently.

References

[1] Bruno Blanchet et al. 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.

[2] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world. In *International Conference on cryptology in India*, pages 151–202. Springer, 2020.