# VERIFPAL: CRYPTOGRAPHIC PROTOCOL VERIFICATION FOR THE REAL WORLD

Nadim Kobeissi
Symbolic Software
nadim@symbolic.software

Georgio Nicolas
Symbolic Software
georgio@symbolic.software

Mukesh Tiwari
University of Melbourne
mukesh.tiwari@unimelb.edu.au

May 2, 2020

## Abstract

Verifpal is a new automated modeling framework and verifier for cryptographic protocols 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. 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.

## 1 Introduction

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

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

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

## 1.1 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 goals/features:

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

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

**Compatibility with the Coq theorem prover.** The Verifpal language and analysis methodology has recently been formalized within the Coq theorem prover [17]. Consequently, Verifpal models can be automatically translated and further analyzed 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. We 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.

Verifpal is able to verify the security of complex protocols, such as Signal, and query for complex attack scenarios such as post-compromise security and key compromise impersonation, across unbounded session executions of the protocol and with fresh values not being shared across sessions. By giving practitioners this powerful symbolic analysis paradigm in an intuitive package, Verifpal stands a chance at making symbolic formal verification a staple in the diet of any protocol designer.

## 1.2 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 [18, 19] protocol verifier, designed by Bruno Blanchet. It does not construct all terms out of Horn clauses [20] in the way that ProVerif does, and it does not use the applied pi-calculus [21] as its modeling language. However, its analysis logic is inspired by ProVerif and is similarly based on the Dolev-Yao model [22]. ProVerif's construction/deconstruction/rewrite logic is also mirrored in Verifpal's own design. ProVerif has been recently used to formally verify TLS 1.2 and TLS 1.3 [4], Let's Encrypt's ACME certificate issuance protocol [9], the Signal secure messaging protocol [3], the Noise protocol framework [10], the Plutus network filesystem [23], e-voting protocols [24, 25, 26, 27], FIDO [28] and many more use cases.

The Tamarin [29] protocol prover also works under the symbolic model, but derives the progeny of its analysis from principals' state transitions rather than from the viewpoint of an attacker observing and manipulating network messages. It is also different from ProVerif in its analysis style, and its modeling language is unique within the domain. Tamarin has been recently used to formally verify Scuttlebutt [7], TLS [30], WireGuard [31], 5G [5, 6], the Noise protocol framework [12, 11], multiple e-voting protocols [32, 33] and many more use cases.

Scyther[1] [35, 36], 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 [37] (used in IPSec), a large amount of Authenticated Key Exchange (AKE) protocols such as HMQV, UM and NAXOS [38], and to check for *"multi-protocol attacks"* [39]. Research focus seems to be moving towards Tamarin, but Scyther is still sometimes used.

AVISPA [40]'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 [41] which shares many of the same authors. In 2016, a new authentication protocol was designed and prototyped with AVISPA [42]. In 2011, Facebook's *Connect* single sign-on protocol was modeled with AVISPA [43].

FDR [44] is not specifically a protocol verifier, but rather a refinement and equivalence checker for processes written using the Communicating Sequential Processes language [45]. CSP can

---

[1] Not to be confused with the bug/flying-type Pokémon of the same name, which, despite its *"ninja-like agility and speed"* [34], does not appear to have published work in formal verification.

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 [46].

A performance analysis of symbolic formal verification tools by Lafourcade and Pus [47], conducted in 2015, as well as a preceding study by Cremers and Lafourcade in 2011 [48] found mixed results, with ProVerif coming out on top more often than not.

ProVerif and Tamarin appear to the the current titans of the symbolic verification space, and they tend to compliment each other due to diverging design decisions: for example, ProVerif does not require human assistance for verification, but sometimes may not terminate and may also sometimes find false attacks (although it is proven not to miss attacks.) Tamarin, on the other hand, 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 [10].

## 1.3 Formal Verification Paradigms

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

Traditionally, *symbolic* models are favored the security protocol verification community for ease of automated analysis. Cryptographers, on the other hand, prefer to use *computational* models and do their proofs by hand. A full comparison between these styles [49] 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 [50] verify computational models. The input languages for both types of tools can be similar. However, in the symbolic model, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). In ProVerif, for example, the attacker is an arbitrary process running in parallel with the protocol, which can read and write messages on public channels and can manipulate them symbolically.

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

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

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

Recently, the F$^\star$ programming language [51], which exports type definitions to the Z3 theorem prover [52], has been used to produce an implementation of the Signal secure messaging protocol that is formally verified for functional correctness at the *level of the implementation itself* [53]. Microsoft Research's Project Everest [54] is attempting to accomplish the same thing for HTTPS, also using F$^\star$ [55].

### 1.4 Contributions

We present the following contributions:

- In §1, we introduce Verifpal and provide a comparison against existing automated verification tools in the symbolic model (§1), as well as a recap of the current state of the art.
- In §2, we introduce the Verifpal modeling language complete with syntax and semantics and 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.
- In §4, we provide the first formal model of the DP-3T decentralized pandemic-tracing protocol [56], 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 are captured in the Coq theorem prover, as well as how Verifpal can translate arbitrary Verifpal models into Coq models for further analysis.

A discussion of future work follows before presenting our conclusion.

Verifpal is already 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.

## 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 latter 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.

⟨*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*⟩)* ‘`]`’ [⟨*queryOptions*⟩]

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

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

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

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

⟨*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 1: Verifpal regular language syntax.

**Simple Example Protocol**

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

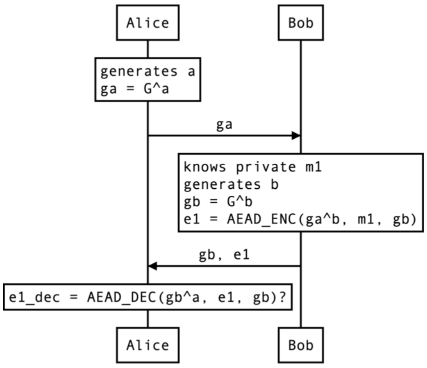

Figure 2: A complete example model of a simple protocol is shown on the left. On the right, a helpful diagram is provided to illustrate how modeling in Verifpal works.

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

## 2.1 Principals

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.

### 2.2.1 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).[2]
- **`generates`**: This allows a principal to describe a *"fresh"* value, i.e. a value that is regenerated 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. But remember: constants may not be assigned to other constants.

### 2.2.2 Primitives

In order to describe cryptographic protocols, we will of course need cryptographic 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, automatically reveal one of its inputs. (Given `DEC(k, c)` and `k`, reveal `c`).

---

[2] 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.

- RECOMPOSE. Given a defined subset of a primitive's outputs, automatically 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 into one value. *"Concatenation"* is a word often used in computer science to describe joining multiple strings or values together. For example, the concatenation of the strings `cat` and `dog` would be `catdog`.
- `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.

**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 [57]. Takes an arbitrary number of input arguments $\geq 1$, 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 [58]. Essentially, `HKDF` is used to extract more than one key out a single secret value. `salt` and `info` help contextualize derived keys. Produces an arbitrary number of outputs $\geq 1$.
- `PW_HASH(a): x`. Password hashing function, similar in practice to, for example, Scrypt [59] or Argon2 [60]. 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.

- `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.

**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, for example `a`.

- **`SIGNVERIF`**`(`**`G`**`^k, message, `**`SIGN`**`(k, m)): m`. Verifies if 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 and 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 key must be the private key of the actual signer, while the subsequent two keys must be the public keys of the other potential signers. Paired with **`RINGSIGNVERIF`**.

- **`BLIND`**`(k, m): m`. Message blinding primitive, useful for the implementation of blind signatures [61]. 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 [62], 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.

- **`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.

### 2.2.3 Equations

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

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

**Example Equations**

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

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

### 2.2.4 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 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.

**Simple Example Protocol: Queries**

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

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

For example: `SIGNVERIF(k, m, s)?` makes this instantiation of `SIGNVERIF` a *"checked"* primitive.

Phases allow Verifpal to reliably model post-compromise security properties such as forward secrecy or future secrecy. When modeling with an active attacker, a new phase can be declared thus:

**Example: Phases**

```
principal Alice[...]
principal Bob [...]
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.

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.

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.

## 2.3 Queries

In Figure 3, we see three different types of queries, from Verifpal's current four:

### 2.3.1 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.

### 2.3.2 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.

## 2.4 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.

## 2.5 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 [63, 64, 65], 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.

## 2.6 Query Options

Imagine that we want to check if Alice will only send some message to Alice 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

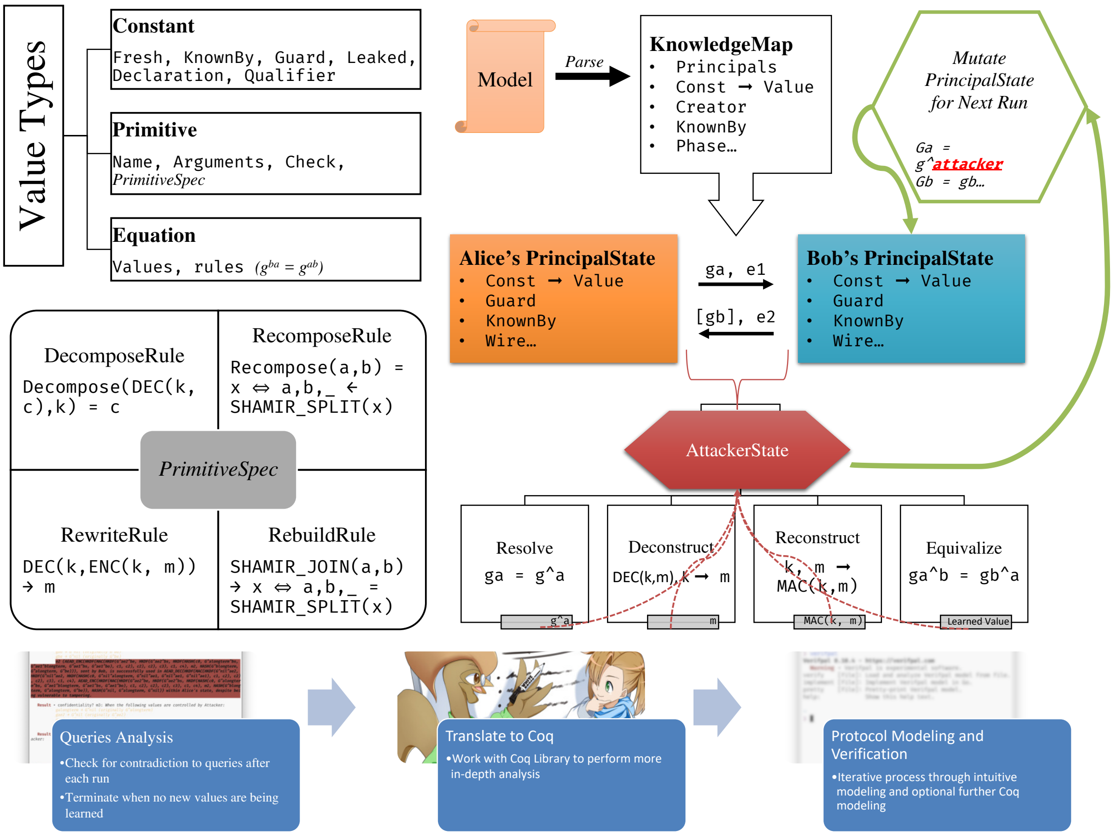

Figure 4: Verifpal analysis methodology. On the left, the three fundamental types usable in Verifpal models are illustrated. As noted in §2.2, all cryptographic primitives are defined via a standard PrimitiveSpec structure, which adapts a primitive's definition via a combination of four rules. On the right, a model analysis is illustrated: first, the Verifpal model first parsed and translated into a global immutable *"knowledge map"* structure. From that structure, 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 apply the four transformations discussed in §3. The attacker keeps doing this until it is unable to learn new values, at which point it mutates the model in each possible way while still following the optimization heuristics touched upon in §3.1.4. At the bottom, we see a description of the envisioned workflow implicit to using Verifpal in production.

Verifpal's active attacker analysis methodology follows a simple set of procedures and algorithms. The overall process is comprised of five steps (see Figure 4 in the Appendix for an illustration):

1. **Gather values.** Attacker passively observes a protocol execution and gathers all values shared publicly between principals.
2. **Insert learned values into attacker state.** Attacker's state ($\mathcal{V}_A$) obtains newly learned values.
3. **Apply transformations.** Attacker applies the four main transformations on all obtained values (these transformations are detailed below.)
4. **Prepare mutations for next session.** If the attacker has learned new values due to the transformations executed in the previous step, they create a combinatorial table of all possible value substitutions, and from that, derive a set of all possible value substitutions across future executions of the protocol on the network.
5. **Iterate across protocol mutations.** Attacker proceeds to execute the protocol across sessions, each time *"mutating"* the execution by mayor-in-the-middling a value. Attacker then returns to step 1 of this list. The process continues so long as the attacker keeps learning new values.

After each 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 deconstruct an equation, the attacker must similarly possess all but one private exponent. Executed on $\mathcal{V}_A$, the set of all values known by the attacker.
- RECONSTRUCT. Attempts to reconstruct primitives and equations given that the attacker possesses all of the component values. Executed on $\mathcal{V}_A$, the set of all values known by the attacker, as well as on $\mathcal{V}_P$, the values known by the principal whose state is currently being evaluated by the attacker.
- 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 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 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, such that we can say with a high degree of confidence that:

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

It is important to note that we do not currently explicitly seek to rule out false attacks (i.e. false positives.) Our central argument is that the analysis logic described in this section is sufficient in order to capture all possible confidentiality and authentication attacks within the language defined in Figure 1. We further buttress this claim with the formalization of Verifpal's semantics and analysis logic in Coq, as shown in §5.

### 3.1.1 Value Construction

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

### 3.1.2 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` → `Bob` → `Carol`.

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

### 3.1.3 Mutations and Guarded Constants

Except for guarded constants (see §2.2.4), 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.

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.

### 3.1.4 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 huge 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 to 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.

## 4 Case Study: Pandemic Contact Tracing in Verifpal

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 [56], 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 lifecycle of a pandemic.

### 4.1.1 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 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
```

### 4.1.2 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
BackendServer -> SmartphoneB: infectedPatients0
BackendServer -> SmartphoneC: infectedPatients0
```

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 `EphID`s just like before.

**DP-3T: EphID Generation**

```
principal SmartphoneA[
  SK1A = HASH(SK0A)
  EphID10A, EphID11A, EphID12A = HKDF(nil, SK1A, BroadcastKey)
]
principal SmartphoneB[
  SK1B = HASH(SK0B)
  EphID10B, EphID11B, EphID12B = HKDF(nil, SK1B, BroadcastKey)
]
principal SmartphoneC[
  SK1C = HASH(SK0C)
  EphID10C, EphID11C, EphID12C = HKDF(nil, SK1C, BroadcastKey)
]
```

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

### 4.1.3 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.

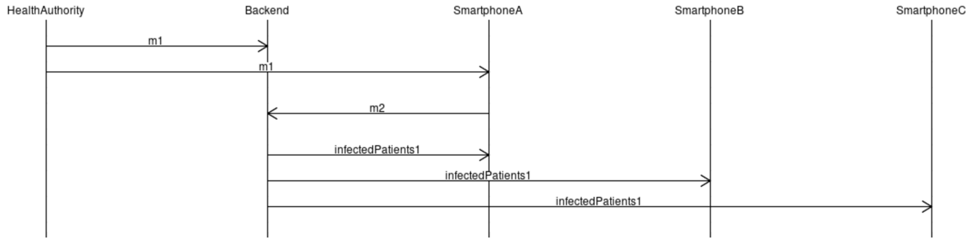

Figure 5: A summary of the parties and network exchanges involved in Day 15 of our Verifpal model of the DP-3T protocol.

**DP-3T: EphID Generation**

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

### 4.1.4 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 §2.2.4) in our protocol model:

**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 private key encryption 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
BackendServer -> SmartphoneB: infectedPatients1
BackendServer -> SmartphoneC: infectedPatients1
```

Everything that happened in Day 15 can be summarized in Figure 5.

## 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 them 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[
  // Would someone who shared a value 15 days
  // before they got tested get flagged?
  // ie in phase[0], before phase[1]
  confidentiality? EphID02A
  // Will people who cross Alice be able to compute
  // all of Alice's EphIDs starting from Day 1?
  confidentiality? EphID10A
  confidentiality? EphID11A
  confidentiality? EphID12A
  confidentiality? EphID20A
  confidentiality? EphID21A
  confidentiality? 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 own security conditions with 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, Verifpalradically speeds up this process by allowing for the automated translation of easy-to-write Verifpalmodels to full-fat Coq and ProVerif models, as discussed in §5.

## 5 Verifpal in Coq

Verifpal's core verification logic and semantics can be captured in Coq via 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. This is sufficient to allow for automated translations of Verifpal models into representations in Coq for further analysis. We have included a utility that when input with a protocol file, automatically generates Coq code that uses the high level functions from our library in order to perform analysis in Coq's powerful paradigm of constructive logic. Once executed, this code would yield results for the queries defined in the protocol model.

**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 6: A simple Verifpal model used in order to illustrate the Coq Library.

## 5.1 Verifpal Semantics in Coq

To formalize the execution of this protocol, we define several types in our library such as `constant`, `Principal`, and `knowledgemap`. For every principal defined in the model, there exists an element of type `Principal` which contains a list of items of knowledge, also known as `constants`. Every time a `constant` is declared, generated, assigned or received in a message by a principal, it would be added to the `Principal`'s knowledge. In order to send a `constant` from one `Principal` to another, we model `knowledgemap`, a type which wraps a list of `Principal` elements.

The latest `knowledgemap` before Alice sent `c` to Bob would contain an object containing Alice's knowledge: `a`, `ma`, `ka`, and `c`, and another one containing Bob's knowledge of `a`. By applying the `send_message` function on that `knowledgemap`, we could send the constant `c` from Alice to any other principal included in the `knowledgemap` and obtain an updated `knowledgemap`. There, we notice that Alice's knowledge is still the same, but Bob's knowledge now contains `a` and `c`, which is the effect of sending the message `c` from Alice to Bob. Alice and Bob perform several primitive operations in the blocks defined above such as **`HASH`**`(a)` and **`ENC`**`(ka, ma)`. All of the primitives supported by Verifpal are formally specified in our Coq library. Outputs of 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 decidably proves equality between elements of type `constant`, one of the cornerstones of our Coq library:

**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`. This behavior is defined as follows in our **`DEC`** function:

**Coq: Modeling Decryption**

```
Definition DEC(key ciphertext: constant): constant :=
match ciphertext with
| ENC_c k m => match k =? key with
| true => m
| false => ciphertext
end
| _ => ciphertext
end.
```

We provide additional lemmas to prove that our model satisfies the behavior expected from primitives. In this example, we prove that `ENC(k, DEC(k,m))` would be equal to `m`.

**Coq: ENC/DEC Lemma**

```
Lemma 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.
```

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 proofs based on which our primitives are defined by simply running the included proofs that are written using the Ltac tactics language supported by Coq.

## 5.2 Verifpal Analysis in Coq

The passive attacker methodology in Verifpal is defined in the following way:

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 tries to apply transformations on the values learned. These transformations are pre-defined and independently provable.
3. This process is repeated so long as the attacker was able to learn new values.

We formalize this methodology using an `Attacker` type which is and a `constant_meta` type. An instance of type `Attacker` type would contain a list of `constant` values that are known by the attacker. `constant_meta` acts as a wrapper type for `constant` with elements of metadata and is defined with some helper types 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)...
```

Whenever a constant is constructed by a `Principal`, it is wrapped in an element of type `constant_meta` with metadata corresponding to the way in which this constant was defined in the Verifpal model. `constant_meta` objects are stored inside the `Principal` data structure and constitute the principal knowledge. Whenever a value is sent over the wire, it is also sent with its corresponding metadata as type `constant_meta`.

### 5.2.1 Example Verifpal Analysis in Coq

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 is equal to `leaked` or if its `(q: qualifier)` is equal to **`public`**, otherwise the value is simply ignored.

At the end `phase[0]` of the protocol illustrated in §5.1, 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 is able to construct `HASH1 a` after learning `a` then consequently attempt **`DEC`** `(HASH1 a) c`. As discussed before, the **`DEC`** operation would reveal the plaintext if the key provided is equivalent to the encryption key. Developing further we obtain **`DEC`** `(HASH1 a)` `(`**`ENC`** `ka ma)` then `DEC (HASH1 a) (ENC (HASH1 a) ma)`, the attacker would then automatically apply the `enc_dec` lemma to deduce `ma` and add it to its knowledge. It is worth noting that all transformations that can be applied by the attacker are accompanied with independently provable lemmas, just like the `enc_dec`.

### 5.2.2 Example Verifpal Query in Coq

Verifpal queries are analogous to decidable processes and help us reason about protocols. The confidentiality query defined in the protocol in (part 1) 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`; 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 in the previous section. Generating a Coq implementation of the protocol discussed will yield an identical result, and could allow the user to independently verify the soundness of this result by checking the proofs included in the code.

## 6 Discussion and Conclusion

Aside from its more formal aspects, Verifpal's focus on prioritizing usability has led it to obtain a substantially high performance benchmark while analyzing complex protocols, largely due to it being implemented in the Go programming language and by taking advantage of the excellent multi-threading support that it provides.

Verifpal also ships with a Visual Studio Code extension that turns into 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's focus on prioritizing usability leads it to 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 indifferentiability or observational equivalence [66, 67] and other advanced features.

Verifpal is also fully capable of supporting a more nuanced definition of primitives recently seen in other symbolic verifiers — for example, recent, more precise models for signature schemes [8] in Tamarin can be fully integrated into Verifpal's design. We also plan to add support for more primitives as these are suggested by the Verifpal user community. We 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 and macOS, along with a user manual that goes more in-depth into the Verifpal language and analysis methodology, at `https://verifpal.com`.

## Acknowledgements

Verifpal is fundamentally inspired by Bruno Blanchet's decades of research into automated formal verification, and would not exist without his work. Funding was provided through the NGI0 PET Fund, a fund established by NLnet with financial support from the European Commission's Next Generation Internet program, under the aegis of DG Communications Networks, Content and Technology under grant agreement 825310.

## References

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

[2] Andreas Straub. OMEMO encryption. 2018.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[18] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[56] Carmela Tronosco et al. Decentralized privacy-preserving proximity tracing, April 2020.

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

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

[59] Colin Percival and Simon Josefsson. 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.

[60] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[61] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[62] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[63] Sandra Steinbrecher and Stefan Köpsell. Modelling unlinkability. In *International Workshop on Privacy Enhancing Technologies*, pages 32–47. Springer, 2003.

[64] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for verifying privacy-type properties: the unbounded case. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 564–581. IEEE, 2016.

[65] David Baelde, Stéphanie Delaune, and Solène Moreau. *A Method for Proving Unlinkability of Stateful Protocols*. PhD thesis, Irisa, 2020.

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

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

# A Full DP-3T Verifpal Model Automatic Coq Translation

```
Require Import PeanoNat String Coq.Numbers.DecimalString Decimal.
Local Open Scope nat_scope.

Inductive generator : Type :=
| G.

Inductive constant : Type :=
| Nil
| value (s: string)
| pub_key_c (G: generator) (exp: constant)
| DH_c (G: generator) (exp1 exp2: constant)
| ENC_c (key message : constant)
| AEAD_ENC_c (key message ad: constant)
| PKE_ENC_c (gk message: constant)
| CONCAT2_c (a b: constant)
| CONCAT3_c (a b c: constant)
| CONCAT4_c (a b c d: constant)
| CONCAT5_c (a b c d e: constant)
| HASH1_c (x: constant)
| HASH2_c (x1 x2: constant)
| HASH3_c (x1 x2 x3: constant)
| HASH4_c (x1 x2 x3 x4: constant)
| HASH5_c (x1 x2 x3 x4 x5: constant)
| MAC_c (key message: constant)
| HKDF1_c (salt ikm info: constant)
| HKDF2_c (salt ikm info: constant)
| HKDF3_c (salt ikm info: constant)
| HKDF4_c (salt ikm info: constant)
| HKDF5_c (salt ikm info: constant)
| PW_HASH_c (x: constant)
| SIGN_c (k m: constant)
| RINGSIGN_c (ka gkb gkc message: constant)
| SHAMIR_SPLIT1_c (k: constant)
| SHAMIR_SPLIT2_c (k: constant)
| SHAMIR_SPLIT3_c (k: constant)
| SHAMIR_JOIN_c (sa sb: constant)
| INVALID (s: string)
| NOT_FOUND
| UNSUCCESSFUL
| VALID.

Scheme Equality for constant.

Lemma string_equality: forall n m : string, (string_beq n m) = true ↔ n = m.
Proof.
intros; split.
apply internal_string_dec_bl.
apply internal_string_dec_lb.
Qed.

Axiom diffie_Hellman_commute : forall (G: generator) (a b: constant), DH_c G a b = DH_c G b a.

Axiom shamir_join : forall (a b c key : constant),
a = SHAMIR_SPLIT1_c key →
b = SHAMIR_SPLIT2_c key →
c = SHAMIR_SPLIT3_c key →
SHAMIR_JOIN_c a b = SHAMIR_JOIN_c b a ∧
SHAMIR_JOIN_c a c = SHAMIR_JOIN_c c a ∧
SHAMIR_JOIN_c b c = SHAMIR_JOIN_c c b.

Ltac bool_destruct_simp :=
intros a b c; destruct a, b, c;
simpl; try auto.

Lemma bool_commutative2: forall a b c : bool,
```

```
a = true → orb (b || a) c = true.
Proof.
bool_destruct_simp.
Qed.

Lemma bool_commutative3: forall a b c : bool,
a = true → orb (c || b) a = true.
Proof.
bool_destruct_simp.
Qed.

Definition public_key(secret: constant) : constant := pub_key_c G secret.

Notation " G^( c )" := (public_key c) (at level 30, right associativity).
Notation "x =? y" := (constant_beq x y) (at level 70) : nat_scope.


Theorem pub_key: forall x: constant, G^( x ) = pub_key_c G x.
Proof.
auto.
Qed.

Theorem pub_key_eq: forall x y: constant,
x = y → G^( x ) = G^( y ).
Proof.
intros x y H.
subst; auto.
Qed.

Lemma equality_generator :
forall (x : generator), generator_beq x x = true.
Proof.
destruct x; auto.
Qed.

Lemma equal_constant_true : forall (c : constant),
c =? c = true.
Proof.
induction c; simpl; try firstorder.
apply string_equality. reflexivity.
rewrite IHc. rewrite equality_generator; auto.
rewrite equality_generator, IHc1, IHc2; auto.
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.

Definition DH (c1 c2: constant): constant := DH_c G c1 c2.

Lemma DH_commute :
forall x y, DH x y = DH y x.
Proof.
apply diffie_Hellman_commute.
Qed.

(* Encryption Primitives *)
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.

Theorem enc_dec_2: forall k m c: constant, c = ENC k m → m = DEC k c.
Proof.
intros k m c H.
rewrite → H.
rewrite → enc_dec.
reflexivity.
Qed.

Definition AEAD_ENC(key plaintext ad: constant): constant :=
AEAD_ENC_c key plaintext ad.

Definition AEAD_DEC(key ciphertext ad: constant) : constant :=
match ciphertext with
| AEAD_ENC_c k m ad’ ⇒ match ad =? ad’ with
| true ⇒ match key =? k with
| true ⇒ m
| false ⇒ ciphertext
end
| false ⇒ INVALID "AEAD_DEC_fail_ad_mismatch"
end
| _ ⇒ ciphertext
end.

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.

Definition PKE_ENC(gkey plaintext: constant) : constant :=
PKE_ENC_c gkey plaintext.

Definition PKE_DEC(key ciphertext: constant) : constant :=
match ciphertext with
| PKE_ENC_c gkey plaintext ⇒
match (G^( key )) =? gkey with
| true ⇒ plaintext
| false ⇒ ciphertext
end
| _ ⇒ ciphertext
end.

Theorem pke_enc_dec: forall k m: constant,
PKE_DEC k (PKE_ENC (G^( k )) m) = m.
Proof.
unfold PKE_ENC, PKE_DEC.
intros k m; rewrite equal_constant_true; reflexivity.
```

```
Qed.

Theorem pke_enc_dec_2: forall k m c: constant,
c = PKE_ENC (G^( k )) m → m = PKE_DEC k c.
Proof.
intros k m c H.
rewrite → H.
rewrite → pke_enc_dec.
reflexivity.
Qed.


(* Hashing Primitives *)
Definition HASH1(a: constant) : constant := HASH1_c a.
Definition HASH2(a b : constant) : constant := HASH2_c a b.
Definition HASH3(a b c : constant) : constant := HASH3_c a b c.
Definition HASH4(a b c d : constant) : constant := HASH4_c a b c d.
Definition HASH5(a b c d e : constant) : constant := HASH5_c a b c d e.
Definition MAC(key message: constant) : constant := MAC_c key message.
Definition PW_HASH(a: constant) : constant := PW_HASH_c a.
Definition HKDF1 (salt ikm info: constant) := HKDF1_c salt ikm info.
Definition HKDF2 (salt ikm info: constant) := HKDF2_c salt ikm info.
Definition HKDF3 (salt ikm info: constant) := HKDF3_c salt ikm info.
Definition HKDF4 (salt ikm info: constant) := HKDF4_c salt ikm info.
Definition HKDF5 (salt ikm info: constant) := HKDF5_c salt ikm info.

(* Signature Primitives *)
Definition SIGN(key message: constant) : constant := SIGN_c key message.

Definition SIGNVERIF(gkey message signature: constant) : constant :=
match gkey, signature with
| pub_key_c _ exp, SIGN_c key m ⇒
match andb (exp =? key) (message =? m) with
| true ⇒ message
| false ⇒ INVALID "SIGNVERIF_fail"
end
| _, _ ⇒ signature
end.

Definition RINGSIGN(key_a gkey_b gkey_c message: constant) : constant :=
RINGSIGN_c key_a gkey_b gkey_c message.

Definition RINGSIGNVERIF(ga gb gc m signature: constant): constant :=
match signature with
| RINGSIGN_c key_a b c message ⇒ match ga, gb, gc with
| pub_key_c _ exp_a, pub_key_c _ exp_b, pub_key_c _ exp_c ⇒
match orb (( exp_a =? key_a) || (exp_b =? key_a))( exp_c =? key_a) with
| true ⇒ m
| false ⇒ INVALID "RINGSIGNVERIF_fail_unable_to_auth"
end
| _, _, _ ⇒ INVALID "RINGSIGNVERIF_fail_key_type_mismatch"
end
| _ ⇒ signature
end.

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.

Theorem ringsignverif_order_verif2: forall a b c m: constant,
m = RINGSIGNVERIF (G^( b )) (G^( a )) (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. rewrite bool_commutative2.
reflexivity. reflexivity.
Qed.

Theorem ringsignverif_order_verif3: forall a b c m: constant,
m = RINGSIGNVERIF (G^( b )) (G^( c )) (G^( a )) m (
RINGSIGN a (G^( c )) (G^( b )) m).
Proof.
unfold RINGSIGN, RINGSIGNVERIF.
intros a b c m.
simpl. rewrite equal_constant_true. simpl. rewrite bool_commutative3.
reflexivity. reflexivity.
Qed.

(* Secret Sharing Primitives *)
Definition SHAMIR_SPLIT1 (k: constant) : constant := SHAMIR_SPLIT1_c k.
Definition SHAMIR_SPLIT2 (k: constant) : constant := SHAMIR_SPLIT2_c k.
Definition SHAMIR_SPLIT3 (k: constant) : constant := SHAMIR_SPLIT3_c k.

Definition SHAMIR_JOIN (sa sb: constant) : constant :=
match sa, sb with
| SHAMIR_SPLIT1_c ka, SHAMIR_SPLIT2_c kb ⇒ match ka =? kb with
| true ⇒ ka
| false ⇒ SHAMIR_JOIN_c sa sb
end
| SHAMIR_SPLIT1_c ka, SHAMIR_SPLIT3_c kb ⇒ match ka =? kb with
| true ⇒ ka
| false ⇒ SHAMIR_JOIN_c sa sb
end
| SHAMIR_SPLIT2_c ka, SHAMIR_SPLIT1_c kb ⇒ match ka =? kb with
| true ⇒ ka
| false ⇒ SHAMIR_JOIN_c sa sb
end
| SHAMIR_SPLIT2_c ka, SHAMIR_SPLIT3_c kb ⇒ match ka =? kb with
| true ⇒ ka
| false ⇒ SHAMIR_JOIN_c sa sb
end
| SHAMIR_SPLIT3_c ka, SHAMIR_SPLIT1_c kb ⇒ match ka =? kb with
| true ⇒ ka
| false ⇒ SHAMIR_JOIN_c sa sb
end
| SHAMIR_SPLIT3_c ka, SHAMIR_SPLIT2_c kb ⇒ match ka =? kb with
| true ⇒ ka
| false ⇒ SHAMIR_JOIN_c sa sb
end
| _, _ ⇒ SHAMIR_JOIN_c sa sb
end.

(* Lemma shamir_join_commute : forall (a b : constant), SHAMIR_JOIN_c a b = SHAMIR_JOIN_c b a.
Proof.

Qed. *)

(* Core Primitives *)
Definition ASSERT (c1 c2: constant) : constant :=
```

```
match c1 =? c2 with
| true ⇒ VALID
| false ⇒ INVALID "ASSERT_fail"
end.

Definition CONCAT2 (c1 c2: constant) : constant := CONCAT2_c c1 c2.
Definition CONCAT3 (c1 c2 c3: constant) : constant := CONCAT3_c c1 c2 c3.
Definition CONCAT4 (c1 c2 c3 c4: constant) : constant := CONCAT4_c c1 c2 c3 c4.
Definition CONCAT5 (c1 c2 c3 c4 c5: constant) : constant := CONCAT5_c c1 c2 c3 c4 c5.

Definition SPLIT1 (c: constant) : constant :=
match c with
| CONCAT2_c c’ _ ⇒ c’
| CONCAT3_c c’ _ _ ⇒ c’
| CONCAT4_c c’ _ _ _ ⇒ c’
| CONCAT5_c c’ _ _ _ _ ⇒ c’
| _ ⇒ INVALID("Attempting to use SPLIT1 with an incompatible argument")
end.

Definition SPLIT2 (c: constant) : constant :=
match c with
| CONCAT2_c _ c’ ⇒ c’
| CONCAT3_c _ c’ _ ⇒ c’
| CONCAT4_c _ c’ _ _ ⇒ c’
| CONCAT5_c _ c’ _ _ _ ⇒ c’
| _ ⇒ INVALID("Attempting to use SPLIT2 with an incompatible argument")
end.

Definition SPLIT3 (c: constant) : constant :=
match c with
| CONCAT3_c _ _ c’ ⇒ c’
| CONCAT4_c _ _ c’ _ ⇒ c’
| CONCAT5_c _ _ c’ _ _ ⇒ c’
| _ ⇒ INVALID("Attempting to use SPLIT3 with an incompatible argument")
end.

Definition SPLIT4 (c: constant) : constant :=
match c with
| CONCAT4_c _ _ _ c’ ⇒ c’
| CONCAT5_c _ _ _ c’ _ ⇒ c’
| _ ⇒ INVALID("Attempting to use SPLIT4 with an incompatible argument")
end.

Definition SPLIT5 (c: constant) : constant :=
match c with
| CONCAT5_c _ _ _ _ c’ ⇒ c’
| _ ⇒ INVALID("Attempting to use SPLIT5 with an incompatible argument")
end.

(*end of primitives*)
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_invalid (code: string).

Fixpoint constant_meta_constructor (c: constant) (d: declaration)
(q: qualifier) (created_by name: string) :=
match eqb created_by "", eqb name "" with
| true, true ⇒ constant_meta_invalid
"constant_meta must have an non empty value for created_by and name."
| true, false ⇒ constant_meta_invalid
"constant_meta must have an non empty value for created_by."
| false, true ⇒ constant_meta_invalid
"constant_meta must have an non empty value for name."
| false, false ⇒ constant_meta_c c d q created_by name not_leaked
end.

Fixpoint get_name_constant_meta (c: constant_meta) : string :=
match c with
| constant_meta_invalid code ⇒ code
| constant_meta_c _ _ _ _ name _ ⇒ name
end.

Fixpoint equal_constant_meta (a b: constant_meta) : bool :=
match a, b with
| constant_meta_c c1 _ _ _ _ _, constant_meta_c c2 _ _ _ _ _ ⇒ c1 =? c2
| _, _ ⇒ false
end.

Fixpoint leak_constant_meta (cm: constant_meta) : constant_meta :=
match cm with
| constant_meta_invalid code ⇒ constant_meta_invalid (
"Attempting to leak invalid constant_meta; " ++ code)
| constant_meta_c c d q created_by name _
⇒ constant_meta_c c d q created_by name leaked
end.

Inductive principal_knowledge: Type :=
| principal_knowledge_empty
| principal_knowledge_invalid (code: string)
| principal_knowledge_c (c: constant_meta) (next: principal_knowledge).

Fixpoint principal_knowledge_constructor (cm: constant_meta)
(next: principal_knowledge) : principal_knowledge :=
match cm with
| constant_meta_invalid code ⇒ principal_knowledge_invalid
"Attempting to construct principal_knowledge using invalid constant_meta"
| constant_meta_c _ _ _ _ _ _ ⇒ match next with
| principal_knowledge_invalid code ⇒ principal_knowledge_invalid
"invalid provided next principal_knowledge"
| _ ⇒ principal_knowledge_c cm next
end
end.

Fixpoint push_pk (pk: principal_knowledge)
(cm: constant_meta) : principal_knowledge :=
match pk with
| principal_knowledge_invalid code ⇒ principal_knowledge_invalid (
"Attempting to push constant_meta to invalid principal_knowledge; " ++ code)
| _ ⇒ principal_knowledge_constructor cm pk
end.

Fixpoint get_constant_meta_by_name_pk (pk: principal_knowledge)
(name: string) : constant_meta :=
match pk with
| principal_knowledge_invalid code ⇒ constant_meta_invalid (
```

```
"Attempting to get constant_meta from invalid principal_knowledge; " ++code)
| principal_knowledge_empty ⇒ constant_meta_invalid "Value not found"
| principal_knowledge_c c next ⇒ match eqb name "" with
| true ⇒ constant_meta_invalid
"Attempting to get a constant_meta with an empty string as its name"
| false ⇒ match eqb (get_name_constant_meta c) name with
| true ⇒ c
| false ⇒ get_constant_meta_by_name_pk next name
end
end
end.

Fixpoint search_constant_meta_by_name_pk (pk: principal_knowledge)
(name: string) : bool :=
match pk with
| principal_knowledge_invalid code ⇒ false
| principal_knowledge_empty ⇒ false
| principal_knowledge_c c next ⇒ match eqb name "" with
| true ⇒ false
| false ⇒ match eqb (get_name_constant_meta c) name with
| true ⇒ true
| false ⇒ search_constant_meta_by_name_pk next name
end
end
end.

Fixpoint remove_constant_meta_pk (pk: principal_knowledge)
(name: string) : principal_knowledge :=
match pk with
| principal_knowledge_empty ⇒ pk
| principal_knowledge_invalid code ⇒ principal_knowledge_invalid (
"Attempting to remove constant_meta from invalid principal_knowledge; " ++code)
| principal_knowledge_c cm next ⇒ match eqb name "" with
| true ⇒ principal_knowledge_invalid
"Attempting to remove a constant_meta with an empty string as its name"
| false ⇒ match eqb name (get_name_constant_meta cm) with
| true ⇒ next
| false ⇒ principal_knowledge_constructor cm (
remove_constant_meta_pk next name)
end
end
end.

Fixpoint update_constant_meta_pk (pk: principal_knowledge)
(cm: constant_meta): principal_knowledge :=
match pk with
| principal_knowledge_invalid code ⇒ principal_knowledge_invalid (
"Attempting to update a constant_meta in an invalid principal_knowledge; " ++code)
| principal_knowledge_empty ⇒ principal_knowledge_invalid
"constant_meta not found"
| principal_knowledge_c _ _ ⇒ match cm with
| constant_meta_invalid _ ⇒ principal_knowledge_invalid
"Attempting to update a constant_meta using an invalid principal"
| constant_meta_c _ _ _ _ _ _ _ ⇒ principal_knowledge_constructor cm (
remove_constant_meta_pk pk (get_name_constant_meta cm))
end
end.

Fixpoint leak_constant_meta_pk (pk: principal_knowledge)
(name: string) : principal_knowledge :=
match pk with
| principal_knowledge_invalid code ⇒ principal_knowledge_invalid (
"Attempting to leak constant_meta in invalid principal_knowledge; " ++code)
| principal_knowledge_empty ⇒ principal_knowledge_invalid
"Attempting to leak constant_meta in empty principal_knowledge"
| principal_knowledge_c _ _ ⇒ update_constant_meta_pk pk (
leak_constant_meta(get_constant_meta_by_name_pk pk name))
```

```
end.

Inductive principal : Type :=
| principal_invalid (code: string)
| principal_c (name: string) (pk: principal_knowledge).

Fixpoint principal_constructor (name: string)
(pk: principal_knowledge) : principal :=
match eqb name "" with
| true ⇒ principal_invalid
"Attempt to construct a principal without a name."
| false ⇒ principal_c name pk
end.

Fixpoint teach_principal (p: principal) (cm: constant_meta) : principal :=
match p with
| principal_invalid _ ⇒ p
| principal_c name knowledge ⇒ principal_constructor name (
push_pk knowledge cm)
end.

Fixpoint generate_value (p: principal) (s: string) : principal :=
match eqb "" s with
| true ⇒ principal_invalid
"Generated value must have a non empty string as its name."
| false ⇒ match p with
| principal_invalid _ ⇒ p
| principal_c name _ ⇒ teach_principal p (
constant_meta_constructor (value s) generates private name s)
end
end.

Fixpoint know_value (p: principal)
(s: string) (q: qualifier) : principal :=
match eqb "" s with
| true ⇒ principal_invalid
"Value to be known must have a non empty string as its name."
| false ⇒ match p with
| principal_invalid _ ⇒ p
| principal_c name _ ⇒ teach_principal p (
constant_meta_constructor (value s) knows q name s)
end
end.

Fixpoint assign_value (p: principal)
(c: constant) (s: string) : principal :=
match eqb "" s with
| true ⇒ principal_invalid
"Assigned value must have a non empty string as its name."
| false ⇒ match p with
| principal_invalid code ⇒ p
| principal_c name _ ⇒ teach_principal p (
constant_meta_constructor c assignment private name s)
end
end.

Fixpoint get_name_principal (p: principal) : string :=
match p with
| principal_invalid code ⇒ code
| principal_c name _ ⇒ name
end.

Fixpoint get_constant_meta_by_name_principal (p: principal)
(name: string) : constant_meta :=
match eqb "" name with
| true ⇒ constant_meta_invalid
"Attempting to look for a value with an empty string as its name"
```

```
| false ⇒ match p with
| principal_invalid _ ⇒ constant_meta_invalid "Value not found."
| principal_c _ k ⇒ get_constant_meta_by_name_pk k name
end
end.

Fixpoint leak_value (p: principal) (value_name: string) : principal :=
match eqb "" value_name with
| true ⇒ principal_invalid
"Attepmting to leak a value with an invalid name."
| false ⇒ match p with
| principal_invalid code ⇒ principal_invalid (
"Attempting to leak a value in an invalid principal; " ++code)
| principal_c principal_name pk ⇒ principal_constructor principal_name (
leak_constant_meta_pk pk value_name)
end
end.

Fixpoint get (p: principal) (name: string) : constant :=
match (get_constant_meta_by_name_principal p name) with
| constant_meta_invalid code ⇒ INVALID code
| constant_meta_c c' _ _ _ _ _ ⇒ c'
end.

Inductive principal_list : Type :=
| principal_list_invalid (code: string)
| principal_list_empty
| principal_list_c (p: principal) (next: principal_list).

Fixpoint principal_list_constructor (p: principal)
(next: principal_list) : principal_list :=
match p with
| principal_invalid code ⇒ principal_list_invalid (
"Cannot construct principal_list using invalid principal; " ++code)
| principal_c _ _ ⇒ match next with
| principal_list_invalid code ⇒ principal_list_invalid (
"Cannot construct principal_list using invalid next principal_list; " ++code)
| _ ⇒ principal_list_c p next
end
end.

Fixpoint add_principal (list: principal_list)
(p: principal) : principal_list :=
match list with
| principal_list_invalid code ⇒ principal_list_invalid (
"Cannot add principal to invalid list; " ++code)
| principal_list_empty ⇒ principal_list_constructor p list
| principal_list_c _ next ⇒ principal_list_constructor p list
end.

Fixpoint remove_principal (list: principal_list)
(name: string) : principal_list :=
match list with
| principal_list_invalid code ⇒ principal_list_invalid (
"Attempting to remove a principal from an invalid principal_list; " ++code)
| principal_list_empty ⇒ principal_list_invalid
"Principal not found"
| principal_list_c p next ⇒ match eqb name "" with
| true ⇒ principal_list_invalid
"Attempting to remove a principal with an empty string as its name"
| false ⇒ match eqb name (get_name_principal p) with
| true ⇒ next
| false ⇒ principal_list_constructor p (
remove_principal next name)
end
end
end.
```

```
Fixpoint update_principal (list: principal_list)
(p: principal): principal_list :=
match list with
| principal_list_invalid code ⇒ principal_list_invalid (
"Attempting to update a principal in an invalid principal_list; " ++code)
| principal_list_empty ⇒ principal_list_invalid "Principal not found"
| principal_list_c _ _ ⇒ match p with
| principal_invalid _ ⇒ principal_list_invalid
"Attempting to update a principal_list using an invalid principal"
| principal_c _ _ ⇒ principal_list_constructor p (
remove_principal list (get_name_principal p))
end
end.

Fixpoint get_principal_by_name_principal_list (list: principal_list)
(name: string) : principal :=
match list with
| principal_list_invalid code ⇒ principal_invalid (
"Attempting to get a principal from an invalid principal_list; " ++code)
| principal_list_empty ⇒ principal_invalid "Principal not found"
| principal_list_c p list' ⇒ match eqb name "" with
| true ⇒ principal_invalid
"The provided name for the principal cannot be empty"
| false ⇒ match eqb (get_name_principal p) name with
| true ⇒ p
| false ⇒ get_principal_by_name_principal_list list' name
end
end
end.

Fixpoint teach_principal_principal_list (list: principal_list)
(principal_name: string) (cm: constant_meta) : principal_list :=
match cm with
| constant_meta_invalid code ⇒ principal_list_invalid (
"Attempting to teach an invalid constant_meta to a principal; " ++code)
| constant_meta_c _ _ _ _ _ _ _ ⇒ match eqb principal_name "" with
| true ⇒ principal_list_invalid
"The provided name for the principal cannot be empty"
| false ⇒ match list with
| principal_list_invalid code ⇒ principal_list_invalid (
"Attempting to teach a principal in an invalid principal_list; " ++code)
| principal_list_empty ⇒ add_principal list (
teach_principal (
principal_constructor principal_name principal_knowledge_empty)
cm)
| principal_list_c p list' ⇒ update_principal list (
teach_principal (
get_principal_by_name_principal_list list principal_name)
cm)
end
end
end.

Fixpoint get_constant_meta_by_name_principal_list (list: principal_list) (name: string) : constant_meta :=
match eqb "" name with
| true ⇒ constant_meta_invalid "Name provided to get_constant_meta_by_name_principal_list can not be empty
      "
| false ⇒ match list with
| principal_list_invalid code ⇒ constant_meta_invalid "Attempting to
      get_constant_meta_by_name_principal_list from an in valid principal list"
| principal_list_empty ⇒ constant_meta_invalid ("Constant: " ++name ++" not Found;
      get_constant_meta_by_name_principal_list")
| principal_list_c principal next ⇒ match get_constant_meta_by_name_principal principal name with
| constant_meta_c _ _ _ _ _ _ _ ⇒ get_constant_meta_by_name_principal principal name
| _ ⇒ get_constant_meta_by_name_principal_list next name
end
end
```

```
end.

Inductive message : Type :=
| message_c (from to value_name: string) (g:  guard_state)
| message_invalid (code: string).

Fixpoint message_constructor (from to value_name: string) (g: guard_state) :=
match eqb "" from, eqb "" to, eqb "" value_name with
| true, _, _ ⇒ message_invalid "The value of from cannot be empty"
| _, true, _ ⇒ message_invalid "The value of to cannot be empty"
| _, _, true ⇒ message_invalid "The value of value_name cannot be empty"
| false, false, false ⇒ message_c from to value_name g
end.

Inductive message_list : Type :=
| message_list_invalid (code: string)
| message_list_empty
| message_list_c (m: message) (next:  message_list).

Fixpoint message_list_constructor (m: message) : message_list :=
match m with
| message_invalid _ ⇒ message_list_invalid
"Attempting to construct message_list using an invalid message"
| message_c _ _ _ _ ⇒ message_list_c m message_list_empty
end.

Fixpoint add_message_to_list (list: message_list)
(m:  message) :  message_list :=
match m with
| message_invalid _ ⇒ message_list_invalid
"Attempting to add invalid message to list"
| message_c _ _ _ _ ⇒ match list with
| message_list_invalid _ ⇒ message_list_invalid
"Attempting to add message to invalid message_list"
| message_list_empty ⇒ message_list_constructor m
| message_list_c _ next ⇒ add_message_to_list next m
end
end.

Inductive knowledgemap : Type :=
| knowledgemap_invalid (code: string)
| knowledgemap_c (list: principal_list) (messages: message_list).

Fixpoint knowledgemap_constructor (principal_name: string) : knowledgemap :=
match eqb principal_name "" with
| true ⇒ knowledgemap_invalid
"Attempting to construct knowledge map with empty principal name"
| false ⇒ knowledgemap_c (principal_list_constructor (
principal_constructor principal_name principal_knowledge_empty)
principal_list_empty) message_list_empty
end.

Fixpoint knowledgemap_constructor_alternative (pl: principal_list)
(ml:  message_list) :  knowledgemap :=
match pl with
| principal_list_invalid code ⇒ knowledgemap_invalid (
"Attempting to contruct knowledgemap using invalid principal_list" ++ code)
| _ ⇒ match ml with
| message_list_invalid code ⇒ knowledgemap_invalid (
"Attempting to contruct knowledgemap using invalid message_list" ++ code)
| _ ⇒ knowledgemap_c pl ml
end
end.

Fixpoint add_principal_knowledgemap (k: knowledgemap)
(name:  string) :  knowledgemap :=
match k with
```

```
| knowledgemap_invalid code => knowledgemap_invalid (
"Attempting to add principal to invalid knowledgemap; " ++code)
| knowledgemap_c list m => knowledgemap_c (add_principal list (
principal_constructor name principal_knowledge_empty)) m
end.

Fixpoint get_principal_knowledgemap (k: knowledgemap)
(name: string) : principal :=
match k with
| knowledgemap_invalid code => principal_invalid (
"Attempting to get principal from invalid knowledgemap; " ++code)
| knowledgemap_c list _ => get_principal_by_name_principal_list list name
end.

Fixpoint get_principal_knowledge_knowledgemap (k: knowledgemap)
(name: string) : principal_knowledge :=
match get_principal_knowledgemap k name with
| principal_invalid code => principal_knowledge_invalid (
"Attempting to get principal_knowledge from invalid principal; " ++code)
| principal_c _ pk => pk
end.

Fixpoint get_constant_meta_from_principal_by_name_knowledgemap (k: knowledgemap)
(principal_name constant_name: string) : constant_meta :=
match eqb "" principal_name, eqb "" constant_name with
| true, true => constant_meta_invalid
"Invalid principal_name and constant_name provided to get_constant_meta_from_principal_by_name"
| true, false => constant_meta_invalid
"Invalid principal_name provided to get_constant_meta_from_principal_by_name"
| false, true => constant_meta_invalid
"Invalid constant_name provided to get_constant_meta_from_principal_by_name"
| false, false => get_constant_meta_by_name_pk (
get_principal_knowledge_knowledgemap k principal_name) constant_name
end.

Fixpoint get_constant_meta_by_name_knowledgemap (k: knowledgemap)
(name: string) : constant_meta :=
match eqb "" name with
| true => constant_meta_invalid
"Invalid constant_name provided to get_constant_meta_from_principal_by_name"
| false => match k with
| knowledgemap_invalid code => constant_meta_invalid (
"Attempting to get constant_meta from invalid knowledgemap; " ++code)
| knowledgemap_c pl _ => get_constant_meta_by_name_principal_list pl name
end
end.

Fixpoint update_principal_knowledgemap (k: knowledgemap)
(p: principal) : knowledgemap :=
match k with
| knowledgemap_invalid code => knowledgemap_invalid (
"Attempting to update principal in invalid knowledgemap; " ++code)
| knowledgemap_c list m => knowledgemap_c (update_principal list p) m
end.

Fixpoint add_message_knowledgemap (k: knowledgemap)
(m: message) : knowledgemap :=
match k with
| knowledgemap_invalid code => knowledgemap_invalid (
"Attempting to add message to invalid knowledgemap; " ++code)
| knowledgemap_c list messages => knowledgemap_c list (
add_message_to_list messages m)
end.

Fixpoint send_message (s: knowledgemap): knowledgemap :=
match s with
| knowledgemap_invalid _ => knowledgemap_invalid
```

```
"Attempting to send a message using an invalid knowledgemap"
| knowledgemap_c list messages ⇒ match messages with
| message_list_invalid _ ⇒ knowledgemap_invalid
"Invalid message list"
| message_list_empty ⇒ s
| message_list_c m next ⇒ match m with
| message_invalid _ ⇒ knowledgemap_invalid
"Attempting to send an invalid message"
| message_c from to value_name g ⇒
match get_principal_by_name_principal_list list from with
| principal_invalid code ⇒ knowledgemap_invalid (
"The sender provided is not valid; " ++code)
| principal_c _ sender_pk ⇒
match get_constant_meta_by_name_pk sender_pk value_name with
| constant_meta_invalid code ⇒ knowledgemap_invalid (
"The sender does now list know the value being sent; " ++code)
| constant_meta_c _ _ _ _ _ _ _
⇒ match get_principal_by_name_principal_list list to with
  | principal_invalid code ⇒ knowledgemap_invalid (
    "The recipient provided is not valid; " ++code)
  | principal_c _ recipient_pk ⇒ knowledgemap_c (
    teach_principal_principal_list list to (
    get_constant_meta_by_name_pk sender_pk value_name)
    ) next
end
end
end
end
end
end.

Inductive attacker_type : Type :=
| passive
| active.

Inductive mutability : Type :=
| mutable
| immutable.

Inductive attacker_knowledge : Type :=
| attacker_knowledge_invalid (code: string)
| attacker_knowledge_empty
| attacker_knowledge_c (cm: constant_meta)
(m: mutability) (next: attacker_knowledge).

Fixpoint attacker_knowledge_constructor (cm: constant_meta)
(m: mutability) (next: attacker_knowledge) : attacker_knowledge :=
match cm with
| constant_meta_invalid code ⇒ attacker_knowledge_invalid (
"Attempting to construct attacker_knowledge using invalid constant_meta; " ++code)
| constant_meta_c _ _ _ _ _ _ _ ⇒ match next with
| attacker_knowledge_invalid code ⇒ attacker_knowledge_invalid
"invalid provided next attacker_knowledge"
| _ ⇒ attacker_knowledge_c cm m next
end
end.

Fixpoint push_ak (ak: attacker_knowledge)
(cm: constant_meta) (m: mutability) : attacker_knowledge :=
match ak with
| attacker_knowledge_invalid code ⇒ attacker_knowledge_invalid (
"Attempting to push constant_meta to invalid attacker_knowledge; " ++code)
| _ ⇒ attacker_knowledge_constructor cm m ak
end.

Fixpoint get_constant_meta_by_name_ak (ak: attacker_knowledge)
(name: string) : constant_meta :=
```

```
match ak with
| attacker_knowledge_invalid code ⇒ constant_meta_invalid (
"Attempting to get constant_meta from invalid attacker_knowledge; " ++code)
| attacker_knowledge_empty ⇒ constant_meta_invalid "Value not found"
| attacker_knowledge_c c _ next ⇒ match eqb name "" with
| true ⇒ constant_meta_invalid
"Attempting to get a constant_meta with an empty string as its name"
| false ⇒ match eqb (get_name_constant_meta c) name with
| true ⇒ c
| false ⇒ get_constant_meta_by_name_ak next name
end
end
end.

Fixpoint search_constant_meta_by_name_ak (ak: attacker_knowledge)
(name: string) : bool :=
match ak with
| attacker_knowledge_invalid code ⇒ false
| attacker_knowledge_empty ⇒ false
| attacker_knowledge_c cm _ next ⇒ match eqb name "" with
| true ⇒ false
| false ⇒ match eqb (get_name_constant_meta cm) name with
| true ⇒ true
| false ⇒ search_constant_meta_by_name_ak next name
end
end
end.

Fixpoint get_equivalent_constant_ak (ak: attacker_knowledge) (c: constant) : constant :=
match ak with
| attacker_knowledge_invalid _ ⇒ INVALID "Attempting to get equivalent constant in invalid attacker"
| attacker_knowledge_empty ⇒ NOT_FOUND
| attacker_knowledge_c cm _ next ⇒ match cm with
| constant_meta_invalid _ ⇒ INVALID "Attempting to get equivalent constant in invalid attacker"
| constant_meta_c const _ _ _ _ _ ⇒ match const =? c with
| true ⇒ const
| false ⇒ get_equivalent_constant_ak next c
end
end
end.

Fixpoint can_mutate_ak (ak: attacker_knowledge) (name: string) : bool :=
match ak with
| attacker_knowledge_invalid code ⇒ false
| attacker_knowledge_empty ⇒ false
| attacker_knowledge_c c m next ⇒ match eqb name "" with
| true ⇒ false
| false ⇒ match eqb (get_name_constant_meta c) name with
| false ⇒ search_constant_meta_by_name_ak next name
| true ⇒ match m with
| mutable ⇒ true
| immutable ⇒ false
end
end
end
end.

Fixpoint length_ak (ak: attacker_knowledge) : nat :=
match ak with
| attacker_knowledge_c _ _ next ⇒ S (length_ak next)
| _ ⇒ 0
end.

Inductive attacker : Type :=
| attacker_invalid (code: string)
| attacker_c (t: attacker_type) (learn_counter: uint) (ak: attacker_knowledge).
```

```
Fixpoint attacker_constructor (type: attacker_type) (learn_counter: uint)
(knowledge: attacker_knowledge) : attacker := attacker_c type learn_counter knowledge.

Fixpoint search_cm_attacker (a: attacker) (cm: constant_meta) : bool :=
match a with
| attacker_invalid _ ⇒ false
| attacker_c _ _ ak ⇒ search_constant_meta_by_name_ak ak (
get_name_constant_meta cm)
end.

Fixpoint search_by_name_attacker (a: attacker) (name: string) : bool :=
match a with
| attacker_invalid _ ⇒ false
| attacker_c _ _ ak ⇒ search_constant_meta_by_name_ak ak name
end.

Fixpoint get_equivalent_constant_attacker (a: attacker) (c: constant) : constant :=
match a with
| attacker_invalid code ⇒ INVALID (
"Attempting to get_equivalent_constant_attacker from an invalid attacker; " ++ code)
| attacker_c _ _ ak ⇒ get_equivalent_constant_ak ak c
end.

Fixpoint can_learn_attacker (a: attacker) (cm: constant_meta) : bool :=
match a with
| attacker_invalid _ ⇒ false
| attacker_c _ _ ak ⇒ match search_cm_attacker a cm with
| true ⇒ false
| false ⇒ match cm with
| constant_meta_invalid _ ⇒ false
| constant_meta_c _ _ q _ _ l ⇒ match l, q with
| leaked, _ ⇒ true
| _, public ⇒ true
| _, _ ⇒ false
end
end
end
end.

Fixpoint absorb_constant_meta_attacker (a: attacker)
(cm: constant_meta) (m: mutability) : attacker :=
match a with
| attacker_invalid _ ⇒ attacker_invalid
"Attempting to teach an invalid Attacker"
| attacker_c t lc ak ⇒ attacker_constructor t lc (push_ak ak cm m)
end.

Fixpoint absorb_principal_knowledge_attacker (a: attacker)
(pk: principal_knowledge) : attacker :=
match a with
| attacker_invalid _ ⇒ attacker_invalid
"Attempting to teach an invalid Attacker"
| attacker_c _ _ ak ⇒ match pk with
| principal_knowledge_invalid _ ⇒ attacker_invalid
"Attempting to teach invalid principal knowledge to attacker"
| principal_knowledge_empty ⇒ a
| principal_knowledge_c cm pk’ ⇒ match can_learn_attacker a cm with
| true ⇒ absorb_principal_knowledge_attacker (
absorb_constant_meta_attacker a cm immutable) pk’
| false ⇒ absorb_principal_knowledge_attacker a pk’
end
end
end.

Fixpoint absorb_message_attacker (a: attacker)
(m: message) (k: knowledgemap) : attacker :=
match a with
```

```
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to teach invalid attacker; " ++code)
| attacker_c type _ ak ⇒ match m with
| message_invalid code ⇒ attacker_invalid (
"Attempting to absord an invalid message" ++code)
| message_c from _ value_name g ⇒ match k with
| knowledgemap_invalid code ⇒ attacker_invalid (
"Attempting to send a message using an invalid knowledgemap" ++code)
| knowledgemap_c _ _ ⇒ match type, g with
| active, unguarded ⇒ absorb_constant_meta_attacker a (
get_constant_meta_from_principal_by_name_knowledgemap
k from value_name)
mutable
| _, _ ⇒ absorb_constant_meta_attacker a (
get_constant_meta_from_principal_by_name_knowledgemap
k from value_name)
immutable
end
end
end
end.

Fixpoint absorb_principal_list_attacker (a: attacker)
(pl: principal_list) : attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to teach invalid attacker; " ++code)
| attacker_c t _ ak ⇒ match pl with
| principal_list_invalid code ⇒ attacker_invalid (
"Attempting to teach attacker using invalid principal_list; " ++code)
| principal_list_empty ⇒ a
| principal_list_c principal next ⇒ match principal with
| principal_invalid code ⇒ attacker_invalid (
"Attempting to teach attacker using invalid principal; " ++code)
| principal_c _ pk ⇒ absorb_principal_list_attacker (
absorb_principal_knowledge_attacker a pk) next
end
end
end.

Fixpoint absorb_message_list_attacker (a: attacker)
(ml: message_list) (k: knowledgemap) : attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to teach invalid attacker; " ++code)
| attacker_c t _ ak ⇒ match ml with
| message_list_invalid code ⇒ attacker_invalid (
"Attempting to teach attacker using invalid message_list; " ++code)
| message_list_empty ⇒ a
| message_list_c message next ⇒ match message with
| message_invalid code ⇒ attacker_invalid (
"Attempting to teach attacker an invalid message; " ++code)
| message_c _ _ _ _ ⇒ absorb_message_list_attacker (
absorb_message_attacker a message k) next k
end
end
end.

Fixpoint absorb_knowledgemap_attacker (a: attacker)
(k: knowledgemap) : attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to teach invalid attacker; " ++code)
| attacker_c t _ ak ⇒ match k with
| knowledgemap_invalid code ⇒ attacker_invalid (
"Attempting to absorb invalid knowledgemap; " ++code)
| knowledgemap_c pl ml ⇒ absorb_message_list_attacker (
```

```
absorb_principal_list_attacker a pl) ml k
end
end.

Fixpoint learn_constant (a: attacker) (c: constant) : attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to learn_constant to using invalid attacker; " ++code)
| attacker_c type count ak ⇒ match ak with
| attacker_knowledge_invalid code ⇒ attacker_invalid (
"Attempting to learn_consting using an attacker that has an invalid attacker knowledge; " ++code)
| _ ⇒ match get_equivalent_constant_ak ak c with
| NOT_FOUND ⇒ a
| INVALID code ⇒ attacker_invalid(
"get_equivalent_constant_ak returned invalid in learn_constant; " ++code)
| _ ⇒ attacker_constructor type (Little.succ count) (
push_ak ak (constant_meta_constructor c knows public "Attacker" (
append ("unnamed_") ((NilEmpty.string_of_uint(Little.succ count))%string)))
immutable)
end
end
end.

Fixpoint learn_concat (a: attacker) (concat: constant): attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to apply learn_concat on invalid attacker; " ++code)
| attacker_c _ _ _ ⇒ match concat with
| CONCAT2_c _ _ ⇒ learn_constant (learn_constant a (SPLIT1 concat)) (SPLIT2 concat)
| CONCAT3_c _ _ _ ⇒ learn_constant (learn_constant (learn_constant a (SPLIT1 concat)) (SPLIT2 concat)) (SPLIT3 concat)
| CONCAT4_c _ _ _ _ ⇒ learn_constant (learn_constant (learn_constant (learn_constant a (SPLIT1 concat)) (SPLIT2 concat)) (SPLIT3 concat)) (SPLIT4 concat)
| CONCAT5_c _ _ _ _ _ ⇒ learn_constant (learn_constant (learn_constant (learn_constant (learn_constant a (SPLIT1 concat)) (SPLIT2 concat)) (SPLIT3 concat)) (SPLIT4 concat)) (SPLIT5 concat)
| _ ⇒ attacker_invalid "Attempting to apply learn_concat on attacker using constant that isnt of type CONCAT"
end
end.

Fixpoint learn_enc (a: attacker) (c: constant): attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to apply learn_concat on invalid attacker; " ++code)
| attacker_c _ _ ak ⇒ match c with
| ENC_c k m ⇒ learn_constant a (DEC (get_equivalent_constant_ak ak k) c)
| _ ⇒ attacker_invalid "Attempting to apply learn_concat on attacker using constant that isnt of subtype ENC_c"
end
end.

Fixpoint learn_aead_enc (a: attacker) (c: constant): attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to apply learn_concat on invalid attacker; " ++code)
| attacker_c _ _ ak ⇒ match c with
| AEAD_ENC_c k m ad ⇒ learn_constant a (AEAD_DEC (get_equivalent_constant_ak ak k) c (get_equivalent_constant_ak ak ad))
| _ ⇒ attacker_invalid "Attempting to apply learn_aead_enc on attacker using constant that isnt of subtype AEAD_ENC_c"
end
end.

Fixpoint learn_pke_enc (a: attacker) (c: constant): attacker :=
match a with
| attacker_invalid code ⇒ attacker_invalid (
"Attempting to apply learn_concat on invalid attacker; " ++code)
```

```
| attacker_c _ _ ak ⇒ match c with
| PKE_ENC_c k m ⇒ match k with
| pub_key_c _ sk ⇒ learn_constant a (PKE_DEC sk c)
| _ ⇒ attacker_invalid "Attempting to apply learn_concat on attacker using PKE_ENC_c that was not
    encrypted with a public_key_c object"
end
| _ ⇒ attacker_invalid "Attempting to apply learn_concat on attacker using constant that isnt of subtype
    PKE_ENC_c"
end
end.


Fixpoint reconstruct_into_c (a: attacker) (c: constant) : constant :=
match a with
| attacker_invalid code ⇒ INVALID (
"Attempting to reconstruct_into_c using an invalid attacker" ++code)
| attacker_c type lc ak ⇒ match ak with
| attacker_knowledge_invalid code ⇒ INVALID (
"Attempting to reconstruct_into_c using an invalid attacker_knowledge" ++code)
| attacker_knowledge_empty ⇒ UNSUCCESSFUL
| attacker_knowledge_c cm m next ⇒ match c with
| value n ⇒ match get_equivalent_constant_attacker a c with
| INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| NOT_FOUND ⇒ match get_equivalent_constant_attacker a (SHAMIR_SPLIT1_c c), get_equivalent_constant_attacker
    a (SHAMIR_SPLIT2_c c), get_equivalent_constant_attacker a (SHAMIR_SPLIT3_c c) with
| INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, NOT_FOUND, NOT_FOUND ⇒ UNSUCCESSFUL
| NOT_FOUND, _, NOT_FOUND ⇒ UNSUCCESSFUL
| NOT_FOUND, NOT_FOUND, _ ⇒ UNSUCCESSFUL
| _, _, NOT_FOUND ⇒ SHAMIR_JOIN (get_equivalent_constant_attacker a (SHAMIR_SPLIT1_c c)) (
    get_equivalent_constant_attacker a (SHAMIR_SPLIT2_c c))
| _, NOT_FOUND, _ ⇒ SHAMIR_JOIN (get_equivalent_constant_attacker a (SHAMIR_SPLIT1_c c)) (
    get_equivalent_constant_attacker a (SHAMIR_SPLIT3_c c))
| NOT_FOUND, _, _ ⇒ SHAMIR_JOIN (get_equivalent_constant_attacker a (SHAMIR_SPLIT2_c c)) (
    get_equivalent_constant_attacker a (SHAMIR_SPLIT3_c c))
| _, _, _ ⇒ c
end
| _ ⇒ c
end
| pub_key_c _ exp ⇒ match reconstruct_into_c a exp with
| UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _ ⇒ c
end
| DH_c _ exp1 exp2 ⇒ match reconstruct_into_c a exp1, reconstruct_into_c a exp2 with
| INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL ⇒ match reconstruct_into_c a (pub_key_c G exp2) with
| INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _ ⇒ c
end
| UNSUCCESSFUL, _ ⇒ match reconstruct_into_c a (pub_key_c G exp1) with
| INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _ ⇒ c
end
```

```
| _, _ ⇒ c
end
| HASH1_c c1 ⇒ match reconstruct_into_c a c1 with
| INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _ ⇒ c
end
| HASH2_c c1 c2 ⇒ match reconstruct_into_c a c1, reconstruct_into_c a c2 with
| INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c1) c
| UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a c2) c
| _, _ ⇒ c
end
| HASH3_c c1 c2 c3 ⇒ match reconstruct_into_c a c1, reconstruct_into_c a c2, reconstruct_into_c a c3 with
| INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c1) c
| UNSUCCESSFUL, _, UNSUCCESSFUL⇒ reconstruct_into_c (learn_constant a c2) c
| UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a c3) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c2) c
| _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c3) c
(* | _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c2) c3) c *)
| _, _, _ ⇒ c
end
| HASH4_c c1 c2 c3 c4 ⇒ match reconstruct_into_c a c1, reconstruct_into_c a c2, reconstruct_into_c a c3,
     reconstruct_into_c a c4 with
| INVALID code, _, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c1) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c2) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c4) c
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a c3) c
| _, _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c2) c
| _, UNSUCCESSFUL, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c3) c
| _, UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c4) c
| UNSUCCESSFUL, _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c2) c3) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c2) c4) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c3) c4) c
| _, _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c2) c3) c
| _, _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c2) c4) c
| _, UNSUCCESSFUL, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c3) c4) c
| UNSUCCESSFUL, _, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c2) c3) c4) c
| _, _, _, _ ⇒ c
end
| HASH5_c c1 c2 c3 c4 c5 ⇒ match reconstruct_into_c a c1, reconstruct_into_c a c2, reconstruct_into_c a c3,
     reconstruct_into_c a c4, reconstruct_into_c a c5 with
| INVALID code, _, _, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
```

```
| _, _, INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c1) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c2) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c3) c
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a c4) c
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a c5) c
| _, _, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c2) c
| _, UNSUCCESSFUL, _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c3) c
| _, UNSUCCESSFUL, UNSUCCESSFUL, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c4) c
| _, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c1) c5) c
| UNSUCCESSFUL, _, _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c2) c3) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c2) c4) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c2) c5) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a c3) c4) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c3) c5) c
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a c4) c5) c
| _, _, _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c2) c3) c
| _, _, UNSUCCESSFUL, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c2) c4) c
| _, _, UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c2) c5) c
| _, UNSUCCESSFUL, _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c3) c5) c
| _, UNSUCCESSFUL, _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c3) c4) c
| _, UNSUCCESSFUL, UNSUCCESSFUL, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c1) c4) c5) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c2) c4) c5) c
| UNSUCCESSFUL, _, _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c2) c3) c5) c
| UNSUCCESSFUL, _, _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c2) c3) c4) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant a c3) c4) c5) c
| _, _, _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant (learn_constant a c1) c2) c3) c4) c
| _, _, _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant (learn_constant a c1) c2) c3) c5) c
| _, _, UNSUCCESSFUL, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant (learn_constant a c1) c2) c4) c5) c
| _, UNSUCCESSFUL, _, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant (learn_constant a c1) c3) c4) c5) c
| UNSUCCESSFUL, _, _, _, _ ⇒ reconstruct_into_c (learn_constant (learn_constant (learn_constant (learn_constant a c2) c3) c4) c5) c
| _, _, _, _, _ ⇒ c
end
| MAC_c key message ⇒ match reconstruct_into_c a key, reconstruct_into_c a message with
| INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a message) c
```

```
| _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a key) c
| _, _ ⇒ c
end
| HKDF1_c salt ikm info ⇒ match reconstruct_into_c a salt, reconstruct_into_c a ikm, reconstruct_into_c a info
     with
| INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a salt) c
| UNSUCCESSFUL, _, UNSUCCESSFUL⇒ reconstruct_into_c (learn_constant a ikm) c
| UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a info) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a salt) ikm) c
| _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a salt) info) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a ikm) info) c
| _, _, _ ⇒ c
end
| HKDF2_c salt ikm info ⇒ match reconstruct_into_c a salt, reconstruct_into_c a ikm, reconstruct_into_c a info
     with
| INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a salt) c
| UNSUCCESSFUL, _, UNSUCCESSFUL⇒ reconstruct_into_c (learn_constant a ikm) c
| UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a info) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a salt) ikm) c
| _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a salt) info) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a ikm) info) c
| _, _, _ ⇒ c
end
| HKDF3_c salt ikm info ⇒ match reconstruct_into_c a salt, reconstruct_into_c a ikm, reconstruct_into_c a info
     with
| INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a salt) c
| UNSUCCESSFUL, _, UNSUCCESSFUL⇒ reconstruct_into_c (learn_constant a ikm) c
| UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a info) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a salt) ikm) c
| _, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant (learn_constant a salt) info) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a ikm) info) c
| _, _, _ ⇒ c
end
| HKDF4_c salt ikm info ⇒ match reconstruct_into_c a salt, reconstruct_into_c a ikm, reconstruct_into_c a info
     with
| INVALID code, _, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant a salt) c
| UNSUCCESSFUL, _, UNSUCCESSFUL⇒ reconstruct_into_c (learn_constant a ikm) c
| UNSUCCESSFUL, UNSUCCESSFUL, _ ⇒ reconstruct_into_c (learn_constant a info) c
| _, _, UNSUCCESSFUL ⇒ reconstruct_into_c (learn_constant (learn_constant a salt) ikm) c
```

```
| _, UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant (learn_constant a salt) info) c
| _, _, UNSUCCESSFUL => reconstruct_into_c (learn_constant (learn_constant a ikm) info) c
| _, _, _ => c
end
| HKDF5_c salt ikm info => match reconstruct_into_c a salt, reconstruct_into_c a ikm, reconstruct_into_c a info with
| INVALID code, _, _ => INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _ => INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code => INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL => UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL => reconstruct_into_c (learn_constant a salt) c
| UNSUCCESSFUL, _, UNSUCCESSFUL=> reconstruct_into_c (learn_constant a ikm) c
| UNSUCCESSFUL, UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant a info) c
| _, _, UNSUCCESSFUL => reconstruct_into_c (learn_constant (learn_constant a salt) ikm) c
| _, UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant (learn_constant a salt) info) c
| _, _, UNSUCCESSFUL => reconstruct_into_c (learn_constant (learn_constant a ikm) info) c
| _, _, _ => c
end
| PW_HASH_c x => match reconstruct_into_c a x with
| INVALID code => INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL => UNSUCCESSFUL
| _ => c
end
| SIGN_c k m => match reconstruct_into_c a k, reconstruct_into_c a m with
| INVALID code, _ => INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code => INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL => UNSUCCESSFUL
| _, UNSUCCESSFUL => reconstruct_into_c (learn_constant a k) c
| UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant a m) c
| _, _ => c
end
| RINGSIGN_c ka gkb gkc m => match reconstruct_into_c a ka, reconstruct_into_c a gkb, reconstruct_into_c a gkc, reconstruct_into_c a m with
| INVALID code, _, _, _ => INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code, _, _ => INVALID (
"Found invalid constant in attacker" ++code)
| _, _, INVALID code, _ => INVALID (
"Found invalid constant in attacker" ++code)
| _, _, _, INVALID code => INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL => UNSUCCESSFUL
| _, UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL => reconstruct_into_c (learn_constant a ka) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, UNSUCCESSFUL => reconstruct_into_c (learn_constant a gkb) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, UNSUCCESSFUL => reconstruct_into_c (learn_constant a m) c
| UNSUCCESSFUL, UNSUCCESSFUL, UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant a gkc) c
| _, _, UNSUCCESSFUL, UNSUCCESSFUL => reconstruct_into_c (learn_constant (learn_constant a ka) gkb) c
| _, UNSUCCESSFUL, _, UNSUCCESSFUL => reconstruct_into_c (learn_constant (learn_constant a ka) gkc) c
| _, UNSUCCESSFUL, UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant (learn_constant a ka) m) c
| UNSUCCESSFUL, _, _, UNSUCCESSFUL => reconstruct_into_c (learn_constant (learn_constant a gkb) gkc) c
| UNSUCCESSFUL, _, UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant (learn_constant a gkb) m) c
| UNSUCCESSFUL, UNSUCCESSFUL, _, _ => reconstruct_into_c (learn_constant (learn_constant a gkc) m) c
| _, _, _, UNSUCCESSFUL => reconstruct_into_c (learn_constant (learn_constant (learn_constant a ka) gkb) gkc) c
| _, _, UNSUCCESSFUL, _ => reconstruct_into_c (learn_constant (learn_constant (learn_constant a ka) gkb) m) c
| _, UNSUCCESSFUL, _, _ => reconstruct_into_c (learn_constant (learn_constant (learn_constant a ka) gkc) m) c
| UNSUCCESSFUL, _, _, _ => reconstruct_into_c (learn_constant (learn_constant (learn_constant a gkb) gkc) m) c
| _, _, _, _ => c
end
| SHAMIR_SPLIT1_c k => match reconstruct_into_c a k with
| INVALID code => INVALID (
"Found invalid constant in attacker" ++code)
```

```
| UNSUCCESSFUL ⇒ match reconstruct_into_c a (SHAMIR_SPLIT2_c k), reconstruct_into_c a (SHAMIR_SPLIT3_c k) with
| INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL ⇒ c
| UNSUCCESSFUL, _ ⇒ c
| _, _ ⇒ c
end
end
| SHAMIR_SPLIT2_c k ⇒ match reconstruct_into_c a k with
| INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL ⇒ match reconstruct_into_c a (SHAMIR_SPLIT1_c k), reconstruct_into_c a (SHAMIR_SPLIT3_c k) with
| INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL ⇒ c
| UNSUCCESSFUL, _ ⇒ c
| _, _ ⇒ c
end
end
| SHAMIR_SPLIT3_c k ⇒ match reconstruct_into_c a k with
| INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL ⇒ match reconstruct_into_c a (SHAMIR_SPLIT1_c k), reconstruct_into_c a (SHAMIR_SPLIT2_c k) with
| INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| UNSUCCESSFUL, UNSUCCESSFUL ⇒ UNSUCCESSFUL
| _, UNSUCCESSFUL ⇒ c
| UNSUCCESSFUL, _ ⇒ c
| _, _ ⇒ c
end
end
(* | SHAMIR_JOIN_c sa sb ⇒ match sa, sb with
| INVALID code, _ ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| _, INVALID code ⇒ INVALID (
"Found invalid constant in attacker" ++code)
| SHAMIR_SPLIT1_c ka, SHAMIR_SPLIT2_c kb ⇒ match reconstruct_into_c a ka, reconstruct_into_c a kb with
|
|
end
| SHAMIR_SPLIT1_c ka, SHAMIR_SPLIT3_c kb ⇒ match reconstruct_into_c a ka, reconstruct_into_c a kb with
|
|
end
| SHAMIR_SPLIT2_c ka, SHAMIR_SPLIT3_c kb ⇒ match reconstruct_into_c a ka, reconstruct_into_c a kb with
|
|
end
end *)
| _ ⇒ c
end end end.




(* Fixpoint deduce_passive (ak: attacker_knowledge) : attacker_knowledge :=
match ak with
| attacker_knowledge_invalid code ⇒ attacker_knowledge_invalid (
"Provided invalide attacker_knowledge to deduce_passive" ++code)
```

```
| attacker_knowledge_c cm m next => match

*)
(* Fixpoint deduce (a: attacker) : attacker :=
match a with
| attacker_invalid code => attacker_invalid (
"Attempting to deduce using an invalid attacker; " ++code)
| attacker_c type lc ak => match type with
| passive => attacker_constructor type lc (deduce_passive ak)
| active => attacker_constructor type lc (deduce_active ak)
end
end.
*)


(* Fixpoint query_confidentiality (a: attacker) (knowledgemap: k) (name: string) : string :=
match getconstan k with
match search_by_name_attacker a name with
| false => "confidentiality ? " ++name ++": PASS"
| true => "confidentiality ? " ++name ++": FAIL"
end. *)

Fixpoint query_confidentiality (a: attacker) (name: string) : string :=
match search_by_name_attacker a name with
| false => "confidentiality ? " ++name ++": PASS"
| true => "confidentiality ? " ++name ++": FAIL"
end.



(* Protocol: lc-dp-3t.vp *)

(* Phase 0: *)

Definition attacker_0 := attacker_constructor active 0 attacker_knowledge_empty.
Definition kmap_0 := knowledgemap_constructor "Smartphonea".
Definition kmap_1 := add_principal_knowledgemap kmap_0 "Smartphonea".
Definition kmap_2 := add_principal_knowledgemap kmap_1 "Smartphoneb".
Definition kmap_3 := add_principal_knowledgemap kmap_2 "Smartphonec".
Definition kmap_4 := add_principal_knowledgemap kmap_3 "Backendserver".
Definition kmap_5 := add_principal_knowledgemap kmap_4 "Healthcareauthority".
Definition principal_smartphonea_0 := get_principal_knowledgemap kmap_5 "Smartphonea".
Definition principal_smartphonea_1 := know_value principal_smartphonea_0 "broadcastkey" public.
Definition principal_smartphonea_2 := generate_value principal_smartphonea_1 "sk0a".
Definition principal_smartphonea_3 := assign_value principal_smartphonea_2 (HKDF1 (get principal_smartphonea_2
      "nil") (get principal_smartphonea_2 "sk0a") (get principal_smartphonea_2 "broadcastkey")) "ephid00a".
Definition principal_smartphonea_4 := assign_value principal_smartphonea_3 (HKDF2 (get principal_smartphonea_3
      "nil") (get principal_smartphonea_3 "sk0a") (get principal_smartphonea_3 "broadcastkey")) "ephid01a".
Definition principal_smartphonea_5 := assign_value principal_smartphonea_4 (HKDF3 (get principal_smartphonea_4
      "nil") (get principal_smartphonea_4 "sk0a") (get principal_smartphonea_4 "broadcastkey")) "ephid02a".
Definition kmap_6 := update_principal_knowledgemap kmap_5 principal_smartphonea_5.
Definition attacker_1 := absorb_knowledgemap_attacker attacker_0 kmap_6.
Definition principal_smartphoneb_0 := get_principal_knowledgemap kmap_6 "Smartphoneb".
Definition principal_smartphoneb_1 := know_value principal_smartphoneb_0 "broadcastkey" public.
Definition principal_smartphoneb_2 := generate_value principal_smartphoneb_1 "sk0b".
Definition principal_smartphoneb_3 := assign_value principal_smartphoneb_2 (HKDF1 (get principal_smartphoneb_2
      "nil") (get principal_smartphoneb_2 "sk0b") (get principal_smartphoneb_2 "broadcastkey")) "ephid00b".
Definition principal_smartphoneb_4 := assign_value principal_smartphoneb_3 (HKDF2 (get principal_smartphoneb_3
      "nil") (get principal_smartphoneb_3 "sk0b") (get principal_smartphoneb_3 "broadcastkey")) "ephid01b".
Definition principal_smartphoneb_5 := assign_value principal_smartphoneb_4 (HKDF3 (get principal_smartphoneb_4
      "nil") (get principal_smartphoneb_4 "sk0b") (get principal_smartphoneb_4 "broadcastkey")) "ephid02b".
Definition kmap_7 := update_principal_knowledgemap kmap_6 principal_smartphoneb_5.
Definition attacker_2 := absorb_knowledgemap_attacker attacker_1 kmap_7.
Definition principal_smartphonec_0 := get_principal_knowledgemap kmap_7 "Smartphonec".
Definition principal_smartphonec_1 := know_value principal_smartphonec_0 "broadcastkey" public.
Definition principal_smartphonec_2 := generate_value principal_smartphonec_1 "sk0c".
Definition principal_smartphonec_3 := assign_value principal_smartphonec_2 (HKDF1 (get principal_smartphonec_2
```

```
"nil") (get principal_smartphonec_2 "sk0c") (get principal_smartphonec_2 "broadcastkey")) "ephid00c".
Definition principal_smartphonec_4 := assign_value principal_smartphonec_3 (HKDF2 (get principal_smartphonec_3 "nil") (get principal_smartphonec_3 "sk0c") (get principal_smartphonec_3 "broadcastkey")) "ephid01c".
Definition principal_smartphonec_5 := assign_value principal_smartphonec_4 (HKDF3 (get principal_smartphonec_4 "nil") (get principal_smartphonec_4 "sk0c") (get principal_smartphonec_4 "broadcastkey")) "ephid02c".
Definition kmap_8 := update_principal_knowledgemap kmap_7 principal_smartphonec_5.
Definition attacker_3 := absorb_knowledgemap_attacker attacker_2 kmap_8.
Definition kmap_9 := add_message_knowledgemap kmap_8 (message_constructor "Smartphonea" "Smartphoneb" "ephid00a" unguarded).
Definition attacker_4 := absorb_knowledgemap_attacker attacker_3 kmap_9.
Definition kmap_10 := send_message kmap_9.
Definition kmap_11 := add_message_knowledgemap kmap_10 (message_constructor "Smartphoneb" "Smartphonea" "ephid00b" unguarded).
Definition attacker_5 := absorb_knowledgemap_attacker attacker_4 kmap_11.
Definition kmap_12 := send_message kmap_11.
Definition kmap_13 := add_message_knowledgemap kmap_12 (message_constructor "Smartphonec" "Smartphoneb" "ephid01c" unguarded).
Definition attacker_6 := absorb_knowledgemap_attacker attacker_5 kmap_13.
Definition kmap_14 := send_message kmap_13.
Definition kmap_15 := add_message_knowledgemap kmap_14 (message_constructor "Smartphoneb" "Smartphonec" "ephid01b" unguarded).
Definition attacker_7 := absorb_knowledgemap_attacker attacker_6 kmap_15.
Definition kmap_16 := send_message kmap_15.
Definition principal_backendserver_0 := get_principal_knowledgemap kmap_16 "Backendserver".
Definition principal_backendserver_1 := know_value principal_backendserver_0 "infectedpatients0" private.
Definition kmap_17 := update_principal_knowledgemap kmap_16 principal_backendserver_1.
Definition attacker_8 := absorb_knowledgemap_attacker attacker_7 kmap_17.
Definition kmap_18 := add_message_knowledgemap kmap_17 (message_constructor "Backendserver" "Smartphonea" "infectedpatients0" unguarded).
Definition attacker_9 := absorb_knowledgemap_attacker attacker_8 kmap_18.
Definition kmap_19 := send_message kmap_18.
Definition kmap_20 := add_message_knowledgemap kmap_19 (message_constructor "Backendserver" "Smartphoneb" "infectedpatients0" unguarded).
Definition attacker_10 := absorb_knowledgemap_attacker attacker_9 kmap_20.
Definition kmap_21 := send_message kmap_20.
Definition kmap_22 := add_message_knowledgemap kmap_21 (message_constructor "Backendserver" "Smartphonec" "infectedpatients0" unguarded).
Definition attacker_11 := absorb_knowledgemap_attacker attacker_10 kmap_22.
Definition kmap_23 := send_message kmap_22.
Definition principal_smartphonea_6 := get_principal_knowledgemap kmap_23 "Smartphonea".
Definition principal_smartphonea_7 := assign_value principal_smartphonea_6 (HASH1 (get principal_smartphonea_6 "sk0a")) "sk1a".
Definition principal_smartphonea_8 := assign_value principal_smartphonea_7 (HKDF1 (get principal_smartphonea_7 "nil") (get principal_smartphonea_7 "sk1a") (get principal_smartphonea_7 "broadcastkey")) "ephid10a".
Definition principal_smartphonea_9 := assign_value principal_smartphonea_8 (HKDF2 (get principal_smartphonea_8 "nil") (get principal_smartphonea_8 "sk1a") (get principal_smartphonea_8 "broadcastkey")) "ephid11a".
Definition principal_smartphonea_10 := assign_value principal_smartphonea_9 (HKDF3 (get principal_smartphonea_9 "nil") (get principal_smartphonea_9 "sk1a") (get principal_smartphonea_9 "broadcastkey")) "ephid12a".
Definition kmap_24 := update_principal_knowledgemap kmap_23 principal_smartphonea_10.
Definition attacker_12 := absorb_knowledgemap_attacker attacker_11 kmap_24.
Definition principal_smartphoneb_6 := get_principal_knowledgemap kmap_24 "Smartphoneb".
Definition principal_smartphoneb_7 := assign_value principal_smartphoneb_6 (HASH1 (get principal_smartphoneb_6 "sk0b")) "sk1b".
Definition principal_smartphoneb_8 := assign_value principal_smartphoneb_7 (HKDF1 (get principal_smartphoneb_7 "nil") (get principal_smartphoneb_7 "sk1b") (get principal_smartphoneb_7 "broadcastkey")) "ephid10b".
Definition principal_smartphoneb_9 := assign_value principal_smartphoneb_8 (HKDF2 (get principal_smartphoneb_8 "nil") (get principal_smartphoneb_8 "sk1b") (get principal_smartphoneb_8 "broadcastkey")) "ephid11b".
Definition principal_smartphoneb_10 := assign_value principal_smartphoneb_9 (HKDF3 (get principal_smartphoneb_9 "nil") (get principal_smartphoneb_9 "sk1b") (get principal_smartphoneb_9 "broadcastkey")) "ephid12b".
Definition kmap_25 := update_principal_knowledgemap kmap_24 principal_smartphoneb_10.
Definition attacker_13 := absorb_knowledgemap_attacker attacker_12 kmap_25.
Definition principal_smartphonec_6 := get_principal_knowledgemap kmap_25 "Smartphonec".
Definition principal_smartphonec_7 := assign_value principal_smartphonec_6 (HASH1 (get principal_smartphonec_6 "sk0c")) "sk1c".
Definition principal_smartphonec_8 := assign_value principal_smartphonec_7 (HKDF1 (get principal_smartphonec_7
```

```
       "nil") (get principal_smartphonec_7 "sk1c") (get principal_smartphonec_7 "broadcastkey")) "ephid10c".
Definition principal_smartphonec_9 := assign_value principal_smartphonec_8 (HKDF2 (get principal_smartphonec_8
       "nil") (get principal_smartphonec_8 "sk1c") (get principal_smartphonec_8 "broadcastkey")) "ephid11c".
Definition principal_smartphonec_10 := assign_value principal_smartphonec_9 (HKDF3 (get
      principal_smartphonec_9 "nil") (get principal_smartphonec_9 "sk1c") (get principal_smartphonec_9 "
      broadcastkey")) "ephid12c".
Definition kmap_26 := update_principal_knowledgemap kmap_25 principal_smartphonec_10.
Definition attacker_14 := absorb_knowledgemap_attacker attacker_13 kmap_26.
Definition principal_smartphonea_11 := get_principal_knowledgemap kmap_26 "Smartphonea".
Definition principal_smartphonea_12 := assign_value principal_smartphonea_11 (HASH1 (get
      principal_smartphonea_11 "sk1a")) "sk2a".
Definition principal_smartphonea_13 := assign_value principal_smartphonea_12 (HKDF1 (get
      principal_smartphonea_12 "nil") (get principal_smartphonea_12 "sk2a") (get principal_smartphonea_12 "
      broadcastkey")) "ephid20a".
Definition principal_smartphonea_14 := assign_value principal_smartphonea_13 (HKDF2 (get
      principal_smartphonea_13 "nil") (get principal_smartphonea_13 "sk2a") (get principal_smartphonea_13 "
      broadcastkey")) "ephid21a".
Definition principal_smartphonea_15 := assign_value principal_smartphonea_14 (HKDF3 (get
      principal_smartphonea_14 "nil") (get principal_smartphonea_14 "sk2a") (get principal_smartphonea_14 "
      broadcastkey")) "ephid22a".
Definition kmap_27 := update_principal_knowledgemap kmap_26 principal_smartphonea_15.
Definition attacker_15 := absorb_knowledgemap_attacker attacker_14 kmap_27.
Definition principal_healthcareauthority_0 := get_principal_knowledgemap kmap_27 "Healthcareauthority".
Definition principal_healthcareauthority_1 := generate_value principal_healthcareauthority_0 "triggertoken".
Definition principal_healthcareauthority_2 := know_value principal_healthcareauthority_1 "ephemeral_sk"
      private.
Definition principal_healthcareauthority_3 := assign_value principal_healthcareauthority_2 (ENC (get
      principal_healthcareauthority_2 "ephemeral_sk") (get principal_healthcareauthority_2 "triggertoken")) "
      m1".
Definition kmap_28 := update_principal_knowledgemap kmap_27 principal_healthcareauthority_3.
Definition attacker_16 := absorb_knowledgemap_attacker attacker_15 kmap_28.
Definition kmap_29 := add_message_knowledgemap kmap_28 (message_constructor "Healthcareauthority" "
      Backendserver" "m1" guarded).
Definition attacker_17 := absorb_knowledgemap_attacker attacker_16 kmap_29.
Definition kmap_30 := send_message kmap_29.
Definition kmap_31 := add_message_knowledgemap kmap_30 (message_constructor "Healthcareauthority" "
      Smartphonea" "m1" unguarded).
Definition attacker_18 := absorb_knowledgemap_attacker attacker_17 kmap_31.
Definition kmap_32 := send_message kmap_31.
Definition principal_smartphonea_16 := get_principal_knowledgemap kmap_32 "Smartphonea".
Definition principal_smartphonea_17 := know_value principal_smartphonea_16 "ephemeral_sk" private.
Definition principal_smartphonea_18 := assign_value principal_smartphonea_17 (DEC (get
      principal_smartphonea_17 "ephemeral_sk") (get principal_smartphonea_17 "m1")) "m1_dec".
Definition principal_smartphonea_19 := assign_value principal_smartphonea_18 (ENC (get
      principal_smartphonea_18 "ephemeral_sk") (get principal_smartphonea_18 "sk1a")) "m2".
Definition kmap_33 := update_principal_knowledgemap kmap_32 principal_smartphonea_19.
Definition attacker_19 := absorb_knowledgemap_attacker attacker_18 kmap_33.
Definition kmap_34 := add_message_knowledgemap kmap_33 (message_constructor "Smartphonea" "Backendserver" "m2"
       unguarded).
Definition attacker_20 := absorb_knowledgemap_attacker attacker_19 kmap_34.
Definition kmap_35 := send_message kmap_34.
Definition principal_backendserver_2 := get_principal_knowledgemap kmap_35 "Backendserver".
Definition principal_backendserver_3 := know_value principal_backendserver_2 "ephemeral_sk" private.
Definition principal_backendserver_4 := assign_value principal_backendserver_3 (DEC (get
      principal_backendserver_3 "ephemeral_sk") (get principal_backendserver_3 "m2")) "m2_dec".
Definition principal_backendserver_5 := assign_value principal_backendserver_4 (CONCAT2 (get
      principal_backendserver_4 "infectedpatients0") (get principal_backendserver_4 "m2_dec")) "
      infectedpatients1".
Definition kmap_36 := update_principal_knowledgemap kmap_35 principal_backendserver_5.
Definition attacker_21 := absorb_knowledgemap_attacker attacker_20 kmap_36.
Definition kmap_37 := add_message_knowledgemap kmap_36 (message_constructor "Backendserver" "Smartphonea" "
      infectedpatients1" unguarded).
Definition attacker_22 := absorb_knowledgemap_attacker attacker_21 kmap_37.
Definition kmap_38 := send_message kmap_37.
Definition kmap_39 := add_message_knowledgemap kmap_38 (message_constructor "Backendserver" "Smartphoneb" "
      infectedpatients1" unguarded).
Definition attacker_23 := absorb_knowledgemap_attacker attacker_22 kmap_39.
```

```
Definition kmap_40 := send_message kmap_39.
Definition kmap_41 := add_message_knowledgemap kmap_40 (message_constructor "Backendserver" "Smartphonec" "
    infectedpatients1" unguarded).
Definition attacker_24 := absorb_knowledgemap_attacker attacker_23 kmap_41.
Definition kmap_42 := send_message kmap_41.

(* Phase 0 queries *)
Compute(query_confidentiality attacker_24 "ephid02a").
```