A Tutorial-Style Introduction to DY⋆

Karthikeyan Bhargavan[1], Abhishek Bichhawat[2], Quoc Huy Do[3,4], Pedram Hosseyni[3(✉)], Ralf Küsters[3], Guido Schmitz[3,5], and Tim Würtele[3]

[1] INRIA, Paris, France karthikeyan.bhargavan@inria.fr
[2] IIT Gandhinagar, Gandhinagar, Gujarat, India abhishek.b@iitgn.ac.in
[3] University of Stuttgart, Stuttgart, Germany {quoc-huy.do, pedram.hosseyni, ralf.kuesters, guido.schmitz, tim.wuertele}@sec.uni-stuttgart.de
[4] GLIWA GmbH, Weilheim i.OB., Germany
[5] Royal Holloway University of London, Egham, Surrey, UK

Abstract. DY⋆ is a recently proposed formal verification framework for the symbolic security analysis of cryptographic protocol code written in the F⋆ programming language. Unlike automated symbolic provers, DY⋆ accounts for advanced protocol features like unbounded loops and mutable recursive data structures as well as low-level implementation details like protocol state machines and message formats, which are often at the root of real-world attacks. Protocols modeled in DY⋆ can be executed, and hence, tested, and they can even interoperate with real-world counterparts. DY⋆ extends a long line of research on using dependent type systems but takes a fundamentally new approach by explicitly modeling the global trace-based semantics within the framework, hence bridging the gap between trace-based and type-based protocol analyses. With this, one can uniformly, precisely, and soundly model, for the first time using dependent types, long-lived mutable protocol state, equational theories, fine-grained dynamic corruption, and trace-based security properties like forward secrecy and post-compromise security.
In this paper, we provide a tutorial-style introduction to DY⋆: We illustrate how to model and prove the security of the ISO-DH protocol, a simple key exchange protocol based on Diffie-Hellman.

Keywords: Cryptographic Protocols · Protocol Analysis · Mechanized Proofs · Formal Methods · F⋆

1 Introduction

Since the proposal of the authentication protocol by Needham and Schroeder [26], the security of such cryptographic protocols has become a continuous field of study for the research community. The first formalization for symbolic protocol

This is the author's own version of the paper. It was originally published in *Protocols, Logic, and Strands: Essays Dedicated to Joshua Guttman on the Occasion of His 66.66 Birthday*, Springer LNCS 13066, pp. 1–21, 2021. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-91631-2_4. Only for personal use, do not redistribute.

analysis has been proposed by Dolev and Yao in [13]. Still, a severe protocol flaw in the public-key authentication protocol (NS-PK) proposed by Needham and Schroeder remained undiscovered for 17 years: In [23], Lowe presented an attack that breaks the security of the NS-PK protocol by mixing two concurrent protocol sessions. Lowe also proposed a fix and showed that this fix is indeed sufficient using the symbolic tool FDR [24].

Since then, the research community has developed several formal analysis techniques and (semi-)automated tools to verify cryptographic protocols (see [2,9] for detailed surveys). The approaches can be divided into two categories: i) *computational* approaches, where proofs are built on precise probabilistic assumptions of the underlying cryptographic primitives, and ii) *symbolic* approaches which build upon a simpler, abstract notion of these primitives. While computational analyses are more precise, they require significantly more effort, and even with the aid of mechanized verification tools, it becomes infeasible to cover all protocol features and attack vectors for large protocols. In contrast, symbolic analyses scale much better, but with less precision regarding cryptographic details.

The symbolic approach has also been in the research focus of Joshua Guttman for a long time: For example, in [31], Thayer, Herzog, and Guttman have proposed a framework that formalizes possible executions of a protocol together with possible actions of an adversary into so-called *strand spaces* and enables precise proofs w.r.t. different kinds of attackers. This approach is extended, exercised, and refined in several papers by Guttman (and others), e.g., (1) to prove independence of sub-protocols' security goals when combining protocols with shared cryptographic material [18]; (2) by introducing *authentication tests* [19] to prove certain authentication properties more easily and using those to not only analyze, but also to design security protocols [17]. Other contributions include work on an algebra for symbolic Diffie-Hellman protocol analysis [14] and reasoning on participant's state [27].

By now, the field has matured a lot. Many real-world protocols have been analyzed using symbolic methods, which often are based on Joshua's work. For example, important protocols like TLS [7,12], Signal [11,21], IKEv2 [1], OAuth [15], and OpenID Connect [16] have undergone symbolic analysis, often revealing severe flaws. In many cases, such protocols have been analyzed using automated provers for symbolic protocol analyses, such as AVISPA [1], ProVerif [10], and Tamarin [25], with Tamarin being based on the concept of strand spaces. These tools can quickly analyze all possible execution traces of protocols and find attacks like Lowe's and much more sophisticated ones in a matter of seconds.

Still, existing symbolic analysis tools, such as ProVerif and Tamarin, have many limitations: (1) These tools do not scale well for complex protocols, as they always perform *whole protocol analysis* and cannot break the analysis into smaller (re-usable) modules. (2) Protocols with unbounded loops, for which a proof typically needs inductive reasoning, as well as protocols that use recursive, unbounded data structures are very challenging for these tools. (3) Models for these tools are far from actual implementations that take low-level protocol details into account.

Existing symbolic analysis frameworks based on dependent-type systems (see, e.g., [3,8]) mitigate some of these limitations as they focus on implementations and modular analysis. However, they come with other restrictions: For example, these works do not model global trace-based runtime semantics and rely on external security arguments (typically proven by hand); security goals such as forward secrecy and post-compromise security are difficult to express and prove; they do not model cryptographic primitives like Diffie-Hellman or XOR that require equational theories and they only have limited support for stateful code with mutable data structures.

A recent approach, the DY⋆ framework [5], tries to combine the best of both worlds. It allows for *modeling protocols in detail*, including implementation features, such as state management, that are usually left out in other approaches. Moreover, the models are *executable*, and hence, testable using (say) test vectors from protocol specifications. Protocols can even be implemented in DY⋆ to a level of detail that yields implementations that *interoperate with real-world counterparts* [6]. DY⋆ is based on the full-fledged programming language F⋆ [29,30], which provides an advanced dependent type system and a powerful proof environment. The F⋆ type-checker can prove that programs meet their specifications using a combination of SMT solving and interactive proofs. With F⋆'s type system and proof environment, DY⋆ is also able to build and verify protocols in a *modular* way, use *induction-based proofs* and capture *unbounded and recursive protocols* with *complex data structures*.

This new approach has already been used to analyze complex protocols: In [5], we have analyzed the Signal protocol, which is used in many popular messaging systems and makes heavy use of Diffie-Hellman exponentiation, signatures, key derivation functions, symmetric encryption, and MACs. Signal employs multiple layers of recursive sub-protocols, which we have modeled/implemented and analyzed in detail in DY⋆. Our work on Signal is the first type-based formulation and proof of post-compromise security for any protocol. In [6], we analyzed the ACME protocol, which is used by certification authorities, such as Let's Encrypt, to verify domain ownership and issue certificates. Our model of ACME enjoys an unprecedented level of detail, sufficient to be interoperable with real-world implementations; it, among others, can interact with the Let's Encrypt server. Our model and proof of ACME totals more than 16,000 lines of F⋆ code and is one of the largest and most in-depth analyses of a cryptographic protocol standard in the literature. Again, in the analysis we precisely handle recursive sub-protocols and implementation loops.

In this paper, we provide a tutorial-style introduction to DY⋆ using the relatively simple ISO-DH authentication and key establishment protocol [20] as a running example. In Section 2, we first give an overview of DY⋆ itself. The ISO-DH protocol is briefly described in Section 3. We show how to model ISO-DH in DY⋆ in Section 4, with the analysis of this protocol in DY⋆ presented in Section 5. The code of our analysis is available in [4]. We conclude in Section 6. We refer the reader to [5,6] for a detailed introduction of DY⋆, more complex case studies,

and a more comprehensive discussion of related work. More information on the umbrella project *REPROSEC* can be found in [28].

2 The DY⋆ Framework

The DY⋆ framework has been proposed in [5]. In this section, we give a brief overview of this framework following the descriptions of the original publication. For full details, we refer the reader to [5].

The DY⋆ framework is meant to model a distributed system that consists of *principals* executing protocol code and exchanging messages over an *untrusted network* which is under the control of a *Dolev-Yao adversary*.

A central component of our framework is the *global (execution) trace*. Among others, it records the history of the states of all principals at any time throughout the run of a system as well as all messages sent over the network by principals. A principal's state may contain arbitrary information. For example, it can contain long-lived keys, such as the principal's public and private keys. Also, principals may be involved in an unbounded number of sessions at the same time. Hence, a principal's state also contains the current session state for all of its sessions.

In the simplest case, at each protocol step a principal first retrieves its current state from the global trace, possibly reads a message from the network (and hence, from the trace), performs its computation, sends messages back to the network (and hence, to the trace), and at the end of the invocation saves its new state in the global trace. Being based on a fully-fledged programming language, DY⋆, of course, does not restrict protocol implementations to follow that pattern and it allows for arbitrary computations, including, for example, loops within protocol steps as well as iterations of subprotocols.

The trace also records the nonces generated by principals and documents whether principals or their sessions (even versions of sessions, see below) are corrupted by the adversary, who can corrupt principals dynamically in a fine-grained way.

The trace determines the attacker's knowledge at any point in a run: the attacker knows all messages that have been sent on the network thus far as well as the state of corrupted principals or corrupted sessions of principals. This knowledge in turn determines which messages the attacker can send to (sessions of) principals. An attacker can only construct and send messages it can derive from its knowledge. In particular, it cannot simply guess secrets.

For principals to interact with the trace, we provide a set of *modules* (containing APIs). These modules are *layered*: At the bottom is the *symbolic runtime layer*, which allows principals to access and manipulate the trace in a straightforward way. On top of this layer, we construct the *labeled layer*, which factors out generic security abstractions and invariants, all of which are mechanically proven sound in F⋆ w.r.t. to the lower-level trace-based runtime semantics of the symbolic runtime layer.

We typically prove security properties in DY⋆ with the help of the labeled layer: At the heart of our methodology is a security-oriented coding discipline

for protocol code written in terms of secrecy labels and usage constraints. Labels allow us to proactively track knowledge of secrets. Whenever some secret is generated (e.g., a nonce), we annotate this secret with a label that states who is allowed to know this secret.

Usage constraints complement labeling: We annotate key material with a usage, for example: a key may only be used for signing but not for encryption (which rules out decryption oracles). Moreover, the annotation can also express that a key may only be used for cryptographic operations with certain payloads, e.g., that some key is only ever used to sign specific messages. This allows us to (by local type-checking) even reason about the behavior of other honest principals.

The labeling layer contains a generic *trace invariant* that describes a *valid* trace. For example, in a valid trace, messages sent to the network must always be publishable (according to their labeling) and principals only store terms in their states that they are allowed to know. This means that, e.g., code modeling a protocol in which principals send secrets unprotected via the public network will violate our valid trace invariant. DY⋆ comes with generic security lemmas, proven in F⋆ in the framework itself, which show that the labeling of messages is actually sound w.r.t. the symbolic runtime layer.

The global trace also allows us to naturally and explicitly express security properties, such as secrecy properties and authentication/integrity properties, involving features like (long-lived) mutable state, dynamic compromise, forward secrecy, and post-compromise security that require reasoning about the adversary's knowledge and the precise order of events in the global trace. As mentioned in Section 1, all this was lacking in previous dependent type based approaches. In order to prove such properties, we typically formulate global invariants over the global trace which the code of every principal has to preserve. The invariants must be strong enough to then imply the security properties we care about.

States and Corruption. As mentioned above, principals' states are recorded in the global trace. Every time a principal stores its state, a new (immutable) entry is appended to the trace that contains the principal's identity and the whole principal's state. This state is grouped in so-called *sessions*, each annotated with a *version* identifier. Sessions can store long-term keys, such as a principal's public and private keys, but also, as the name suggests, the principal's states of arbitrary many ongoing protocol sessions. An adversary can at any time *compromise* a specific version of a session of some principal. Such a compromise is recorded in the trace and the adversary can use all information stored in that state session. This particularly fine-grained notion of compromise allows an attacker to dynamically compromise both long-term keys and ephemeral protocol states. By this, we can model that only a subset of data stored in a principal's state leaks to the adversary, allowing us, for instance, to analyze forward secrecy and post-compromise security. The attacker, however, is not restricted by this model as it can corrupt as many versions and sessions as it likes.

Equational Theories. DY⋆ builds upon an abstract type of byte strings called bytes and defines a series of (abstract) conversion and cryptographic functions for constructing and parsing byte strings. One can also define equational theories on

bytes to capture algebraic properties. For example, for Diffie-Hellman, we have the functions dh_pk and dh, where dh_pk reflects the generation of the public key (g^y) from a private key (y), and dh reflects the computation of the shared secret (g^{xy}) from a private key (x) and public key (g^y).

We technically represent equational rules as F* lemmas. For example, the equational rule $(g^x)^y = (g^y)^x$ is expressed as follows:[7]

```
val dh_shared_secret_lemma: x:bytes → y:bytes →
    Lemma ((dh x (dh_pk y)) == (dh y (dh_pk x)))
```

DY* already provides a large set of typical equational rules, which —if needed— can easily be extended (for DH, XOR, signatures, etc.). For example, in [6] we add a property called *non-destructive exclusive ownership* for signatures.

Modeling Adversarial Behavior. We model an active network attacker, in the tradition of Dolev and Yao [13]. The attacker can intercept, modify, and block all messages sent on the network. The attacker can compromise any session state and can call any cryptographic function (using messages/principal's states it already knows), and can schedule any part of a protocol, i.e., functions that model the behavior of honest parties (see below). By using these capabilities, the attacker grows its knowledge as the global trace is extended and can try to break the protocol's security goals. Essential for an attacker's behavior is its knowledge. The *attacker's knowledge* at each index i in the global trace is logically characterized using a set of derivation rules. For example, the attacker can immediately derive literals, read any message sent on the network, read previously compromised states, and decrypt ciphertexts for which it knows the corresponding keys. To verify the correctness and expressiveness of our attacker model, we implement a (typed) API for the attacker with commonly expected operations like sending and receiving messages or corrupting principals. By typechecking this attacker API, we prove that our trace invariants do not restrict the adversary in unexpected ways, a property called *attacker typability*.

Labeled Crypto API. The core of the labeled layer is a labeled crypto API that provides labeled wrappers for all the crypto functions on the symbolic runtime layer and internally enforces labeling and usage rules. Each byte in bytes is assigned a unique label that indicates who may know it. For example, a label CanRead [P p1; P p2] indicates a secret that only the principals p1 and p2 may know, whereas the label public indicates that anyone may know it. Literals are always labeled public, nonces are assigned a label when they are generated.

Secrecy labels form a lattice, where can_flow i l1 l2 says that the label l1 is equal or less strict than the label l2 at trace index i. In particular, public flows to all other labels, and CanRead [P p] can flow to public at index i if Compromised p sid v (for some session sid and some version v) occurs in the trace at or before i.

The labeled APIs enforce a labeling discipline that ensures that secret values never flow to public channels. In particular, we require that the labels of all

[7] Note that we format all F* code in this paper using a pretty-printer, i.e., some syntactic constructs are displayed using well-known mathematical symbols for readability, such as $\rightarrow$, $\forall$, $\exists$, and λ, instead of their textual representations.

network messages must flow to public. If a secret value has to be sent over the network, it must first be encrypted with a key whose label is at least as strict as the message's label. We refer the reader to [4] for the full set of labeling rules.

In addition to secrecy labels, the labeled APIs also enforce usage pre-conditions. Each key is assigned an intended usage. For example, a signature key cannot be used as an encryption key. Furthermore, we define a global usage predicate controlling what kinds of messages a given key can encrypt/sign. Of course, these restrictions only apply to honest principals. For example, the labeled API for the signature and verification functions is as follows:[8]

```
val vk: sk:bytes → pk:bytes{is_labeled_public pk}
val sign: #p:global_usage → #i:timestamp → #l:label → #l':label →
    k:bytes{∃ s. is_signing_key p i k l s} → nonce:bytes →
    m:bytes{get_label m == l' ∧ ∀s. is_signing_key p i k l s
      ⟹ p.usage_preds.can_sign i s (vk k) m} →
    tag:bytes{can_flow i (get_label tag) l'}
val verify: pk:{is_labeled_public pk} → m:bytes → tag:bytes → bool
val verify_lemma: #p:global_usage → #i:timestamp → #l1:label → #l2:label →
    pk:bytes → m:msg p i l1 → tag:msg p i l2 →
    Lemma (if verify pk m tag then (∀ l s. is_verification_key p i pk l s
          ⟹ (can_flow i l public ∨ (∃ j . j ≤ i ∧ p.usage_preds.can_sign j s pk m)))
          else (C.verify pk m s = false))
```

Signing keys are supposed to be secrets (typically labeled with CanRead [P prin] to model that they should be known only to some principal prin) and marked to be used as signing keys (along with some string s that we can use to tag such keys in order to track them in our proofs). The corresponding verification keys (generated with vk) are always labeled public. For each protocol, we define a (global) predicate can_sign i s k m (part of the global usage data structure p) that indicates if at some timestamp i the private key corresponding to the public key k (tagged with the string s) may sign the message m. This predicate is then used as a pre-condition for sign, ensuring that protocol code does not accidentally call sign with a message that does not conform to can_sign. Conversely, if verify succeeds, then the API guarantees that the signature must be valid and the signed message must satisfy can_sign, unless the signing key can be known by the attacker (see verify_lemma above); in this lemma, l indicates the label of the *private* key of pk.

For Diffie-Hellman, each DH private key has the type dh_private_key p i l s indicating that it has a secrecy label l and that the *shared secret* generated from this private key should have the usage defined by the function dh_usage

[8] Note that the code excerpts we show in this paper are a bit simplified for presentation purposes (see [4] for the full code). Further note that we here use so-called *refinement types* provided by F⋆ to further restrict types. For example, the result pk of the function vk is of type bytes, which is —by refinement— further required to satisfy the predicate is_labeled_public, which states that the byte string pk must be labeled as public. We also make use of so-called *implicit* arguments, which are marked by #. In many cases, these parameters can be dropped when calling the function, as F⋆ can derive them from the context.

that takes as parameter the string s. The corresponding public keys have type dh_public_key p i l s. The declarations in F* are as follows:

```
val dh_usage: string → usage
val dh_pk: #p:global_usage → #i:nat → #l:label → #s:string →
    dh_private_key p i l s → dh_public_key p i l s
val dh: #p:global_usage → #i:nat → #l1:label → #l2:label → #s:string →
    dh_private_key p i l1 s → dh_public_key p i l2 s →
    b:bytes{has_label i b (join l1 l2) ∧ has_usage i b (dh_usage s)}
```

The function dh takes a private key with type dh_private_key p i l1 s and a public key with type dh_public_key p i l2 s to compute a shared secret with label join l1 l2 and usage defined by dh_usage given the string s. The label join l1 l2 means that the shared secret may be used in any session covered by l1 or l2. We define several other variants of the dh function, including for cases where the peer's public key is untrusted.

The types for the rest of the cryptographic API are similar. In each construction, the arguments must satisfy some protocol-specific usage predicate (can_aead_encrypt, can_mac, ...), and in all encryption functions, we ask that messages must flow to the labels of the decryption keys.

Specifying Protocols. A protocol is written as a set of functions, each of which defines one protocol step performed by a principal. These functions can be called by the adversary in arbitrary order. The parameters of these functions allow the adversary to specify which session of the protocol is to be invoked and which message the principal is supposed to read from the network. In particular, we have no restrictions on the number of principals or sessions in a protocol run.

When called, a function typically parses the principal's state as well as the network message to some semantically rich data type (we provide protocol-dependent parsing and serializing functions). Next, it performs the computation of the respective protocol step, serializes its results (a new state for this principal and possibly new network messages), and places these results on the trace (by storing the new state and sending the network messages). Since with F* we have a full-fledged functional programming language at our disposal, the functions can perform arbitrary computation and, in combination with global traces, easily deal with recursive, mutable, and long-lived state, unlike previous approaches.

The protocol code for each principal cannot directly read or write to the trace, but instead must use the labeled API that enforces an append-only discipline on the global trace using a custom computational and stateful (monadic) effect called LCrypto. Recall that effects (and so-called monads) are common in functional programming languages, for example, to implement stateful functions. LCrypto allows the function to use and modify the global trace, without providing the global trace as a parameter to the function. The effect also captures trace invariants (see below). Functions annotated with the LCrypto effect are total (i.e., they always terminate) but can return errors, which are automatically propagated by LCrypto.

The labeled API provides functions to generate new nonces, send and receive messages, store and retrieve states, and log security events. Using these functions,

and a library of functions for cryptography and bytes manipulations, we can build stateful implementations of protocols.

The LCrypto effect enforces the global trace invariant valid_trace. Functions in the trace API and with the LCrypto effect take valid_trace as both pre- and post-condition. Hence, this generic trace invariant must hold in all global traces generated by protocol code that follows the labeling rules. The invariant consists of several components, some generic and some that have to be defined for each protocol. The following F* code specifies the generic parts with the protocol-specific invariants/predicates given in the argument pr:

```
1 let valid_trace (pr:preds) (tr:trace) =
2   (∀ (i:timestamp) (t:bytes) (s:principal) (r:principal). i < trace_len tr ⟹
3     (was_message_sent_at i s r t ⟹ (is_publishable pr.global_usage i t))) ∧
4   (∀ (i:timestamp) (p:principal) (v:version_vec) (s:state_vec). i < trace_len tr ⟹
5     (state_was_set_at i p v s ⟹ ((Seq.length v = Seq.length s) ∧
6       (∀ j. j < Seq.length v ⟹ pr.trace_preds.session_st_inv i p s[j] v[j] )))) ∧
7   (∀ (i:timestamp) (p:principal) (e:event). i < trace_len tr ⟹
8     (did_event_occur_at i p e ⟹ (pr.trace_preds.can_trigger_event i s e)))
```

The invariant states that i) (Lines 2–3) any message t that is sent on the network (at index i by the sender s to the intended receiver r) must have a label that can flow to public; ii) (Lines 4–6) any state (with sessions s and corresponding versions v)[9] that is stored by an honest principal p at index i must satisfy the protocol-specific state invariant session_st_inv i p s' v' contained in pr for each session s' (in s) and their corresponding version identifier v' (in v); iii) (Lines 7–8) any event e logged by principal p at index i must satisfy the protocol-specific event predicate can_trigger_event i p e in pr. We also prove that all functions in the attacker API preserve valid_trace (regardless of protocol-specific predicates), i.e., the attacker is not restricted by this invariant.

For a protocol model, we define the above-mentioned protocol-specific invariants pr and provide pr to the effect LCrypto as an argument. As valid_trace is parameterized by pr, the effect can then instantiate this invariant for a concrete protocol. Note that pr also contains usage predicates for cryptographic functions, such as can_sign mentioned above. Hence, these predicates are propagated in the same way as valid_trace.

Protocol-specific invariants can be parameterized as well. This way, we can easily define re-usable modular layers, such as a generic PKI layer (which we also provide). This PKI layer, for example, provides key material to each principal stored in distinguished sessions. To enable layering, the protocol-specific invariant of this layer takes another (higher-layer) protocol-specific invariant pr as an argument and combines both to pki pr, where pki maps pr to the richer invariant.

Specifying Security Goals. The labeled layer of the DY* framework allows us to specify security goals in several ways: i) we can use labeling to specify "simple" goals such as the secrecy of certain terms; ii) we can use the state invariant

[9] Sessions and versions are stored in two separate sequences s and v (of the same length). For each session s' that is stored at index j in s, the corresponding version identifier v' is stored at the same index j in v.

and event predicate from valid_trace to specify conditions under which a certain principal may reach a certain state/record an event; iii) we can specify more complex properties independently and show that these are implied by valid_trace. In the latter case, we have to define the state invariant and event predicate such that they reflect sufficient properties of the protocol to prove the security goal.

Symbolic Execution. To enable debugging and testing protocol models, we provide a symbolic implementation of all abstract parts of the symbolic runtime layer. In particular, we provide an algebraic model for our basic data type bytes and all conversion and cryptographic functions of this layer. We emphasize that this model is mechanically proven to satisfy the equational theory, i.e., all lemmas describing this theory must hold true for the implementation.

For each protocol that we model in DY⋆, we can write a scheduler function which calls the protocol functions in the expected order. This scheduler essentially describes a run of the protocol and can be seen as a test case. We can then compile the scheduler along with the DY⋆ framework and the protocol implementation to OCaml and execute this code to print out a symbolic trace of a protocol run. This way, we can inspect symbolic runs and check our model for errors, something not possible in tools like Tamarin and ProVerif. We can also implement further test cases and also implement and check known attacks for unfixed protocol code.

3 The ISO-DH Protocol

The ISO-DH protocol is a variant of the Diffie-Hellman protocol for authenticated key exchanges. More precisely, it extends the Diffie-Hellman protocol by adding an authentication mechanism as defined in [20]. The protocol is depicted in Figure 1 with an initiator I and receiver R. For computing and verifying signatures, the protocol requires both parties to have a key pair and know the corresponding public key of the other party. We denote the private signing key of P by sk_P and the corresponding public key by pk_P.

Security Goals. The primary goal of the protocol is to provide secrecy of the generated shared key (g^{xy}) if the protocol is run between an honest initiator and an honest responder. More precisely, the protocol aims to achieve forward secrecy in the presence of an active network attacker. That is, even if the attacker corrupts the long-term secrets of the principals (the signature keys) after a shared key has been established, the attacker is not able to obtain the shared key. The protocol also aims to provide mutual authentication by means of the signatures added to the Diffie-Hellman protocol.

4 Modeling ISO-DH in DY⋆

We now illustrate the overall modeling process in DY⋆ using the ISO-DH protocol as an example. This section presents the model of the first two steps of this protocol in detail (up to the point where the responder processes the first message and sends the second message) and gives a brief overview of the remaining steps.

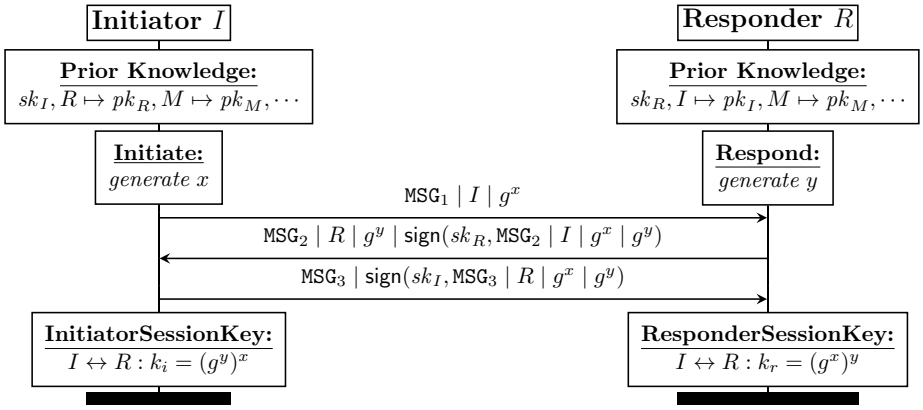

Fig. 1. Signed Diffie-Hellman Protocol (ISO-DH) [22]. We use message tags to avoid reflection and type confusion attacks. This figure is taken from [5].

The full DY* implementation of this protocol can be found in [4]. We note that the analysis of ISO-DH has first been conducted as a case study in [5] but was only briefly sketched there. Here, we go into much more detail regarding both the formal model as well as the security analysis.

Initiator: Send First Message. To model the first step of the protocol, we define a function initiator_send_msg_1 which chooses a fresh nonce x for a principal a (the initiator), sends the first protocol message to a principal b (the responder), and stores the relevant values in the principal state of a. The interface and the implementation of this function are shown in Figure 2. As specified by the interface in Lines 2 to 6 of Figure 2, the function has two principals a and b as input parameters. The return values are idx_msg and idx_session, where idx_msg is the trace index at which the global trace records the sent message and idx_session is the index in the trace at which the new state of the initiator is recorded. Furthermore, the function has the LCrypto effect, parametrized by pki isodh, with isodh being the protocol-specific predicates (see also Section 2).

Before explaining the function in detail, we note that the LCrypto effect allows us to specify pre- and post-conditions using the requires and ensures clauses. The initiator_send_msg_1 has no additional pre-conditions except those required by the effect, i.e., that the (implicit) input trace is valid (see Line 5). In Line 6, the post-condition of the function specifies a condition on the input trace t0, the return values (i, si), and the output trace t1. More precisely, the length of the output trace must be larger than the length of the input trace, and the message index i must point to the last trace entry of the output trace t1. Recall that the LCrypto effect (implicitly) also stipulates the validity of the output trace. Users do not have to state this.

```
1  // Interface of the first protocol step
2  val initiator_send_msg_1:
3    a:principal → b:principal →
4    LCrypto (idx_msg:timestamp × idx_session:nat) (pki isodh)
5    (requires (λ _→⊤))
6    (ensures (λ t0 (i,si) t1 → trace_len t1 > trace_len t0 ∧ i == trace_len t1 − 1))
7
8  // Implementation of the first protocol step
9  let initiator_send_msg_1 a b =
10   let si = new_session_number a in
11   let (|t0,x|) = rand_gen (readers [V a si 0]) (dh_usage "ISODH.dh_key") in
12   let gx = dh_pk x in
13
14   let ev = initiate a b gx in
15   trigger_event a ev;
16
17   let t1 = global_timestamp () in
18   let new_ss_st = InitiatorSentMsg1 b x in
19   let new_ss = serialize_valid_session_st t1 a si 0 new_ss_st in
20   new_session a si 0 new_ss;
21
22   let t2 = global_timestamp () in
23   let msg1 = Msg1 a gx in
24   let w_msg1 = serialize_msg t2 msg1 in
25   let i = send a b w_msg1 in
26   i, si
```

Fig. 2. Interface and implementation of the first protocol step. See module `ISODH.Protocol` in [4].

The implementation of initiator_send_msg_1 starts with choosing a new state session index in Line 10, which is used to store information related to one protocol session. The function new_session_number is provided by DY* and returns the next available session index for the current principal state of a.

Next, the function generates a fresh nonce x using the DY* function rand_gen in Line 11. The label of this nonce is (readers [V a si 0]), indicating that only version 0 of the state session si of principal a may read the nonce. Furthermore, the usage of the nonce is specified as (dh_usage "ISODH.dh_key"). By calling the DY* function dh_pk, the DH public key g^x is calculated.

In Lines 14 and 15, an application-specific event is created and added to the trace, stating that a protocol flow is initiated with initiator a, responder b, and initiator public value gx.

The relevant information about the protocol flow is saved in the principal state of a in Lines 18 to 20. In particular, the intended responder b and the nonce x are stored in an application-specific session type InitiatorSentMsg1 and this state session is serialized (i.e., turned into a value of type bytes). In Line 20, the serialized state session is appended to the current principal state of a and

the new principal state is then stored in the global trace. Recall that a principal p may only store state labeled to be readable by p. This is a time-dependent property, which is why the timestamp (Line 17) is needed.

In Lines 22 to 25, the function once more acquires a new timestamp (i.e., the current length of the trace, which is not equal to t1, as the state of principal a was updated in Line 20) and creates an application-specific message Msg1 containing the identity of the initiator and the DH public key gx. This message is serialized in Line 24 and sent in Line 25. The send function appends the message to the global trace and returns the message's trace index. (Note that the message has to be publishable, a time-dependent property.)

The function returns two indices, as already mentioned above. With these values, it is possible to write a scheduler for symbolic test cases as described in Section 2.

Responder: Receive Message and Send Reply. The steps performed by the receiver are shown in Figure 3. As mentioned, DY⋆ supports a high degree of modularity. In particular, we can split up large functions into small helper functions, as shown below: The main function modeling the second step is responder_send_msg_2 (see module ISODH.Protocol in [4]), which we split up into two helper functions, one for receiving the first protocol message and one for sending the second protocol message (for brevity, we omit the responder_send_msg_2 function, as this function simply uses the two helper functions shown in Figure 3). The security, i.e., non-violation of the trace invariant, is proven for each (helper) function independently, modularizing proof obligations on a fine-grained level.

The function that models receiving the first message, receive_msg_1_helper, is shown in Lines 1 to 12 of Figure 3. In Line 8, the helper function calls the receive_i function of DY⋆, which is given the trace index at which the message to be read is stored and the receiver's name. However, the receive function does not provide any guarantees on authenticity or confidentiality of the received message and only guarantees that a message was sent at the given trace index. (The operation might fail and the failure is propagated by the LCrypto effect.) The received message is then parsed in Lines 9 to 12.

The second helper function send_msg_2_helper is similar to the initiator function shown above. The responder creates the values y and g^y in Lines 23 and 24, stores all relevant values in a new session in its state in Lines 26, 27, and 31, and generates an event in Line 30. The responder creates the second protocol message in Line 38, serializes it in Line 39, and sends the message in Line 40.

For computing the signature contained in the second protocol message, the responder first retrieves its private key in Line 21 using the get_private_key function. This function is provided by the PKI layer of DY⋆ and returns the key of b of the specified key type (here, the key type is SIG, hence the function returns a signing key). The responder creates the signature in Lines 33 to 36. The function sigval_msg2 (used in Line 34) is a helper function that creates the payload for the signature, i.e., a concatenation of the identifier a, the values gx and gy, and a string "msg2" (as a tag).

The assert clause in the code states a simple property that facilitates the F* proofs (see also Section 5).

Remaining protocol steps. For details on the remaining protocol steps, we refer to the reader to [4]. The model implements the remaining steps from Figure 1 similarly to the functions presented in this section. Upon finishing the protocol run, the initiator and responder each write events to the trace indicating that they completed a protocol run, and in these events include their names, the values gx, gy, and the shared key k. In the following, we briefly explain the verification of the signature by the initiator, as this step is crucial for the proof outlined in Section 5.

When the initiator receives the second message, it verifies the signature contained in the message using the helper function shown in Figure 4. The initiator calls the function with the following arguments: the current trace length i, the session and version indices si and vi at which the initiator manages the values of the protocol flow, the principal names a and b, the public key pkb of b (for verifying the signature), the values gx and gy of the current protocol flow, and the signature sig contained in the second protocol message.

First, the initiator creates the message for which the signature should be valid in Line 2 of Figure 4. As described previously, the sigval_msg2 function essentially concatenates the input arguments. Next, the helper function tries to verify the signature in Line 3. If the verification is successful, the initiator calculates and returns the shared key k in Lines 5 and 7. The remaining code is needed to prove the security properties and will be described in Section 5.

5 Security Analysis

In this section, we describe in detail how security properties can be proven within the DY* framework, illustrated by the ISO-DH protocol. In particular, we show how security properties can be stated as F* lemmas, encoded in trace invariants, and how these invariants are enforced on the application code layer.

5.1 Forward Secrecy

A central security property of the ISO-DH protocol is the secrecy of the resulting shared key, even if long-term secrets used by the initiator and responder become corrupted. We formalize this forward secrecy property, which was already outlined in Section 3, as an F* lemma as shown in Figure 5.

The lemma is formulated as a function of the LCrypto effect, but without a return value (i.e., the type of the return value is unit). The pre-condition of the lemma requires that the initiator a has finished the flow (modeled by a finishI event) at a trace index i. If this is the case, then the lemma ensures that either b has been corrupted at or before i, or the key has a join label (containing the specific session and version identifiers at which a and b store the key) and cannot be derived by the attacker unless it compromises one of these sessions (with the respective version). In particular, as long as the specific sessions at which a

```
val receive_msg_1_helper:
  b:principal → idx_msg:timestamp →
  LCrypto (now:timestamp & a:principal & gx:msg now public) (pki isodh)
  (requires (λ _ → ⊤))
  (ensures (λ t0 (|now, _, _|) t1 → t0 == t1 ∧ now == trace_len t0))

let receive_msg_1_helper b idx_msg =
  let (|now,_,w_msg1|) = receive_i idx_msg b in
  let msg1 = parse_msg w_msg1 in
  match msg1 with
   | Success (Msg1 a gx) → (|now,a,gx|)
   | _ → error "responder_send_msg_2: not a msg1"

val send_msg_2_helper: #idx:timestamp → b:principal → a:principal →
    gx:msg idx public → LCrypto (timestamp × nat) (pki isodh)
  (requires (λ t0 → later_than (trace_len t0) idx))
  (ensures (λ t0 (i,si) t1 → trace_len t1 > trace_len t0 ∧ i == trace_len t1 − 1))

let send_msg_2_helper #idx b a gx =
  let si = new_session_number b in
  let (|_, skb|) = get_private_key b SIG sig_key_label in

  let (|t1, y|) = rand_gen (readers [V b si 0]) (dh_usage "ISODH.dh_key") in
  let gy = dh_pk y in

  let new_ss_st = (ResponderSentMsg2 a gx gy y) in
  let new_ss = serialize_valid_session_st t1 b si 0 new_ss_st in

  assert (is_eph_priv_key (t1+1) y b si 0);
  trigger_event b (respond a b gx gy y);
  new_session b si 0 new_ss;

  let t2 = global_timestamp () in
  let sv: msg t2 public = sigval_msg2 a gx gy in
  let (|t3,n_sig|) = rand_gen (readers [P b]) (nonce_usage "SIG_NONCE") in
  let sg = sign skb n_sig sv in

  let msg2 = Msg2 b gy sg in
  let w_msg2 = serialize_msg t3 msg2 in
  let i = send b a w_msg2 in
  i,si
```

Fig. 3. Interface and Implementation of the Second Protocol Step. See module `ISODH.Protocol` in [4] for full details. Note that we marked proof-related code with a gray background (see also Section 5).

```
let initiator_verify_signature i si vi a b pkb x gx gy sig =
  let sv = sigval_msg2 a gx gy in
  if verify pkb sv sig then (
    can_flow_to_public_implies_corruption i (P b);
    let k = dh x gy in
    dh_key_label_lemma isodh_global_usage i gy;
    k
  ) else (error "sig verification failed")
```

Fig. 4. Helper function for verifying the signature of the second message and – if the signature is valid – calculating the shared DH key. See module `ISODH.Protocol` in [4].

```
val initiator_forward_secrecy_lemma: i:timestamp → a:principal → b:principal →
    gx:bytes → gy:bytes → k:bytes → LCrypto unit (pki isodh)
  (requires (λ t0 → i < trace_len t0 ∧ did_event_occur_at i a (finishI a b gx gy k)))
  (ensures (λ t0 _t1 → t0 == t1 ∧ (corrupt_at i (P b) ∨ (
    ∃si sj vi vj. is_labeled isodh_global_usage i k (join (readers [V a si vi])
                                                          (readers [V b sj vj])) ∧
    (corrupt_at (trace_len t0) (V a si vi) ∨ corrupt_at (trace_len t0) (V b sj vj) ∨
      is_unknown_to_attacker_at (trace_len t0) k)
  ))))
```

Fig. 5. Forward Secrecy Theorem. See module `ISODH.SecurityLemmas` in [4].

and b store their key is not corrupted, the key stays secret even if the attacker corrupts the long-term signing keys of a or b after the initiator has finished the protocol run. We formulate a similar lemma for an event type indicating that the responder has finished the protocol flow.

As explained in Section 2, the security properties are proven by an appropriate instantiation of the valid_trace invariant from which the security properties should follow. We show how a suitable valid_trace can be specified and how to utilize the signature and event predicates in DY⋆ to prove that the protocol code preserves valid_trace.

Specifying valid_trace. In brief, we encode in valid_trace that, whenever a (finishI a b gx gy k) event occurs on the trace, then the key k must have the label (join (readers [V a si vi]) (readers [V b sj vj])) (for some values si, vi, sj, vj) if b is not corrupted at i. Using a generic security lemma for labels provided by DY⋆, we can then prove the secrecy of the key (see below). As described in Section 2, DY⋆ provides a straightforward way to define predicates on events using the parameter of the LCrypto effect. The effect parameter used in this analysis is (pki isodh), i.e., event predicates on the finishI event can be defined in the isodh invariants (at the application layer). For this purpose, we first define a predicate is_dh_shared_key as follows (see module ISODH.Sessions):

```
let is_dh_shared_key (i:timestamp) (key:bytes) (a:principal) (b:principal) =
  (∃ si sj vi vj. is_aead_key isodh_global_usage i key
    (join (readers [V a si vi]) (readers [V b sj vj])) "ISODH.aead_key")
```

By using the is_aead_key predicate as shown, we require the key to be labeled (join (readers [V a si vi]) (readers [V b sj vj])) (and to be used as an AEAD encryption key); see module Labeled.CryptoAPI in [4].

For (finishI a b gx gy k) events, we now require (using the event predicate) that (corrupt_id i (P b) ∨ is_dh_shared_key i k a b) must hold true. With such a predicate on finishI events, we can easily infer that the label of a shared key is (join (readers [V a si vi]) (readers [V b sj vj])) (for some values si, vi, sj, vj) as long as principal b is not corrupted at or before trace index i. Next, we show how we ensure that the protocol implementation fulfills this event predicate and why is_dh_shared_key is true if b is not corrupted at or before i.

Implementation Fulfills valid_trace. Recall that by using the LCrypto effect for protocol code, each function must ensure the validity of the resulting trace after the function call, e.g., whenever an initiator creates a finishI event, it must ensure that is_dh_shared_key is true if the responder is not corrupted.

Before the initiator of our model finishes the protocol run, it checks the signature contained in the second message and computes the shared key (see Figure 4). The post-condition of the function in Figure 4 looks as follows:

```
1 λ t0 k t1 → trace_len t0 == trace_len t1 ∧ k == CryptoLib.dh x gy ∧
2   is_msg isodh_global_usage i k (readers [V a si vi]) ∧ (
3     corrupt_id i (P b) ∨
4     (∃ y. k == CryptoLib.dh y gx ∧ is_dh_shared_key i k a b ∧
5           did_event_occur_before i b (respond a b gx gy y)))
```

Hence, when the initiator calls the helper function, it gets the shared key k and the guarantees on k needed for the validity of the trace. (The did_event_occur_before predicate is used only for authentication, see Section 5.2.) In the following, we show why the helper function yields this post-condition.

As described in Section 4, the helper function tries to verify the signature in Line 3 of Figure 4 using a verification key belonging to b (required by the function type, not shown here). As described in Section 2, a successful verification guarantees that either the signature predicate can_sign holds true or the signing key is known to the attacker (see verify_lemma in Section 2). In the latter case, b, the principal owning the signing key, must be corrupted, which is deduced by the (generic) lemma can_flow_to_public_implies_corruption called in Line 4 of Figure 4.

To determine which guarantees the signature predicate can_sign needs to provide, we first notice that the shared key k is calculated in Line 5 of Figure 4 using the dh function. The label of k is the join of the label of the initiator's secret key x, i.e., (readers [V a si vi]), and the label of the responder's secret key y. The connection between gy and the label of the corresponding secret key y is established using the lemma dh_key_label_lemma called in Line 6. Therefore, to show that is_dh_shared_key i k a b holds true, the signature predicate needs to imply that the label of the private key of gy is equal to (readers [V b sj vj]), for some session sj and version vj.

The idea for connecting the successful signature verification to the label of the private key y is as follows: We formulate the signature predicate such that the successful signature verification of the second message implies that, at a

previous trace index, the responder created an event, and define a predicate on this event enforcing the required label on y. Following this roadmap, we construct the application-specific signature predicate as follows:

```
1 match parse_sigval m with // parse the signature payload
2   | Success (SigMsg2 a gx gy) →
3     (∃ y. gy == (dh_pk y) ∧ did_event_occur_before i p (respond a p gx gy y))
4   ...
```

That is, the initiator code, after successful verification of the signature, can use the fact that a respond event was created for the private key y. For (respond a b gx gy y) events, we require (within the event predicate) that (∃si vi. is_eph_priv_key i y b si vi) must be true, where the is_eph_priv_key predicate enforces, amongst others, that the label of y is (readers [V b si vi]).

Overall, when the signature verification is successful, the event predicate implies that there is a private key y labeled with (readers [V b si vi]), and thus, the key k returned by initiator_verify_signature has the label join (readers [V a si vi]) (readers [V b sj vj])) (for some values si, vi, sj, vj) if b is not corrupted at i.

We highlight that every function that triggers a respond event, in particular, the responder function presented in Section 4, needs to ensure this property. This can be automatically done by explicitly asserting is_eph_priv_key i y b si 0 in Line 29, a statement then proven by F⋆.

Proving the Secrecy Lemma. The proof of the initiator_forward_secrecy_lemma (Figure 5) essentially follows from valid_trace, in particular, from the label of the shared key shown above, i.e., whenever a finishI event occurs, the label of the shared key is (join (readers [V a si vi]) (readers [V b sj vj])) (for some values si, vi, sj, vj) unless b is corrupted. Given this label, we can use the generic security lemma secrecy_join_label_lemma provided by DY⋆, which states that if both ids of the join label of the key are uncorrupted, then the attacker cannot derive the key. The F⋆ proof of initiator_forward_secrecy_lemma is now performed automatically by F⋆. It only needs to be hinted at secrecy_join_label_lemma:

```
1 let initiator_forward_secrecy_lemma i a b gx gy k =
2   secrecy_join_label_lemma k // generic lemma from DY⋆
```

5.2 Authentication Properties

Besides the key secrecy property, we formulate and prove authentication properties. Here, we give a brief overview of these properties and refer to the module ISODH.SecurityLemmas in [4] for their formal statements and proofs.

The authentication properties state that, after finishing a protocol flow, both the initiator and responder agree on all session parameters. Hence, we formulate two properties, one from the initiator's perspective and one from the responder's perspective. The property from the initiator's perspective states that, whenever the initiator a finishes the flow and creates an event indicating that it finished the run with b using the session parameters gx, gy, and the shared key k, then either the responder has previously created an event indicating that it sent the second

protocol message to a with the same values gx, gy, and the private key y such that k = (dh y gx), or the responder is corrupted. The authentication property from the responder's point of view is analogous.

6 Conclusion

DY⋆ is a recently proposed framework for formal protocol analysis and verification, a field which was shaped significantly by Joshua's work, e.g., in [14,17,18,19,27,31].

In this paper, we have given a tutorial-style introduction to DY⋆ to help potential users of the framework to get started. DY⋆ provides many more features than what we have been able to show in this paper, such as reasoning on unbounded loops, recursive data structures, low-level implementation aspects like data encoding, and interoperability. As discussed in [5,6], we plan to enrich DY⋆ with even more features, including support for equivalence-based properties.

Acknowledgments

This work was partially supported by the *Deutsche Forschungsgemeinschaft (DFG)* through Grants KU 1434/10-2 and KU 1434/12-1, the *European Research Council (ERC)* through Grant CIRCUS-683032, and the *Office of Naval Research (ONR)* through Grant N000141812618.

References

1. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., et al.: The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In: CAV. LNCS, vol. 3576, pp. 281–285. Springer (2005)
2. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: SoK: Computer-Aided Cryptography. In: IEEE S&P. pp. 777–795 (2021)
3. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM TOPLAS **33**(2), 8:1–8:45 (2011)
4. Bhargavan, K., Bichhawat, A., Do, Q.H., Hosseyni, P., Küsters, R., Schmitz, G., Würtele, T.: DY⋆ Code Repository, https://github.com/REPROSEC/dolev-yao-star/tree/festschrift-guttman
5. Bhargavan, K., Bichhawat, A., Do, Q.H., Hosseyni, P., Küsters, R., Schmitz, G., Würtele, T.: DY⋆: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In: IEEE EuroS&P '21. pp. 523–542 (2021)
6. Bhargavan, K., Bichhawat, A., Do, Q.H., Hosseyni, P., Küsters, R., Schmitz, G., Würtele, T.: An In-Depth Symbolic Security Analysis of the ACME Standard. In: ACM CCS '21 (2021), to appear
7. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In: IEEE S&P. pp. 483–502 (2017)
8. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: ACM POPL. pp. 445–456 (2010)
9. Blanchet, B.: Security protocol verification: Symbolic and computational models. In: POST. pp. 3–29 (2012)

10. Blanchet, B.: Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. Found. Trends Priv. Secur. **1(1-2)**(1-2), 1–135 (2016)
11. Cohn-Gordon, K., Cremers, C.J.F., Dowling, B., Garratt, L., Stebila, D.: A Formal Security Analysis of the Signal Messaging Protocol. In: IEEE EuroS&P. pp. 451–466 (2017)
12. Cremers, C., Horvat, M., Hoyland, J., Scott, S., van der Merwe, T.: A Comprehensive Symbolic Analysis of TLS 1.3. In: ACM CCS. pp. 1773–1788 (2017)
13. Dolev, D., Yao, A.C.: On the Security of Public Key Protocols. IEEE Trans. Inf. Theor. **29**(2), 198–208 (1983)
14. Dougherty, D.J., Guttman, J.D.: An Algebra for Symbolic Diffie-Hellman Protocol Analysis. In: TGC. LNCS, vol. 8191, pp. 164–181. Springer (2012)
15. Fett, D., Küsters, R., Schmitz, G.: A Comprehensive Formal Security Analysis of OAuth 2.0. In: ACM CCS. pp. 1204–1215 (2016)
16. Fett, D., Küsters, R., Schmitz, G.: The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines. In: IEEE CSF. pp. 189–202 (2017)
17. Guttman, J.: Security protocol design via authentication tests. In: IEEE CSFW. pp. 92–103 (2002)
18. Guttman, J., Thayer, F.: Protocol independence through disjoint encryption. In: IEEE CSFW. pp. 24–34 (2000)
19. Guttman, J.D., Thayer, F.J.: Authentication tests and the structure of bundles. Theor. Comput. Sci. **283**(2), 333–380 (2002)
20. ISO/IEC 9798-3:2019(E): IT Security techniques — Entity authentication — Part 3: Mechanisms using digital signature techniques. Tech. rep. (Jan 2019)
21. Kobeissi, N., Bhargavan, K., Blanchet, B.: Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In: IEEE EuroS&P. pp. 435–450 (2017)
22. Krawczyk, H.: SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols. In: CRYPTO. pp. 400–425 (2003)
23. Lowe, G.: An Attack on the Needham-Schroeder Public-Key Authentication Protocol. Inf. Process. Lett. **56**(3), 131–133 (1995)
24. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In: TACAS. pp. 147–166 (1996)
25. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In: CAV. LNCS, vol. 8044, pp. 696–701. Springer (2013)
26. Needham, R.M., Schroeder, M.D.: Using Encryption for Authentication in Large Networks of Computers. Communications of the ACM **21**(12), 993–999 (Dec 1978)
27. Ramsdell, J.D., Dougherty, D.J., Guttman, J.D., Rowe, P.D.: A Hybrid Analysis for Security Protocols with State. In: IFM. pp. 272–287 (2014)
28. REPROSEC: REPROSEC Project (2021), https://reprosec.org/
29. Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. J. Funct. Program. **23**(4), 402–451 (2013)
30. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., et al.: Dependent types and multi-monadic effects in $\mathsf{F}^\star$. In: ACM POPL. pp. 256–270 (2016)
31. Thayer, F.J., Herzog, J.C., Guttman, J.D.: Strand Spaces: Proving Security Protocols Correct. J. Comput. Secur. **7**(1), 191–230 (1999)