From provable models to provable implementations: translating Alice & Bob security protocols to F*

Rémi Garcia[1,2], Paolo Modesti[1], and Leo Freitas[3]

[1] Teesside University, Middlesbrough, United Kingdom
{r.garcia,p.modesti}@tees.ac.uk
[2] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
remi.garcia@inria.fr
[3] Newcastle University, Newcastle upon Tyne, United Kingdom
leo.freitas@ncl.ac.uk

Abstract. Deploying secure communication protocols remains a challenging task. To ensure that intended security properties hold, formal verification has become increasingly important. In this work, we present a translation from the design-oriented Alice & Bob language to the implementation-level language F*, leveraging the DY* library's verification capabilities. Our approach addresses the expressiveness gap between abstract specifications and concrete implementations by generating verifiable F* code that benefits from its dependent-type system. This integration of model-driven development with dependent types enables the specification and proof of security properties directly at the implementation level. As a result, users, even those without expertise in formal methods, can start from an intuitive Alice & Bob notation and obtain a formally backed implementation that can be enriched with features not expressible in Alice & Bob, while preserving correctness through re-verification. We demonstrate the effectiveness of this workflow through the generation and verification of several real-world security protocols, showcasing an end-to-end approach to verified protocol implementation.

Keywords: Security Protocols · Design and Verification · Code generation · Provable Implementation · Model-Driven Development · Dolev-Yao

1 Introduction

The design and implementation of security protocols lie at the core of modern network infrastructure. Properly verifying that a given protocol satisfies its security properties is a considerable challenge. Even when a design is verified to be secure, implementation defects can still undermine the security of a deployed application, as observed with the OpenSSL Heartbleed bug [16].

To define the threat scenario to be defended against, the Dolev-Yao attacker model [23] has become the standard for analysing secure communications. It assumes a ubiquitous adversary who can read, modify, or block messages in

transit, subject to the constraints of cryptographic rules, and who may impersonate honest agents. A modern and concrete-level solution for implementation verification is the Dolev-Yao* (DY*) framework [7], which enables symbolic type-based verification for security protocols in F* [43,44]. F* is an ML-style functional programming language with dependent-type system [44] for verification and supports user-defined effects for effectful code.

Symbolic verification, which employs high-level cryptographic abstractions, offers advantages in simplicity and generality [12,6,33]. The ability to prove detailed protocols at the implementation level is therefore crucial; however, producing a verified implementation requires expertise, skill and time. Model-Driven Development (MDD) is a methodology with a proven track record, in which the general approach is to automatically generate concrete code that reflects a specified design, as advocated in [5,35,3]. We follow this philosophy, focusing on *Alice & Bob* (*AnB*) models [34], using the *AnBx* specification language [13]. We have chosen this style of protocol narration for its simplicity, aligning with notations commonly used in cybersecurity technical documentation within professional contexts. Unlike provably correct code-generation approaches [1,15,4,40,27], we preserve provability in the generated implementation, which allows us to make modifications directly in the code and subsequently prove them correct.

In fact, working with an abstract protocol representation poses challenges that arise after code has been generated. High-level specification languages inherently trade expressivity for simplicity, and some implementation details are left unspecified, such as function bodies or additional data attached to ciphertexts as with AEAD encryption [10]. Hence, amendments may be needed.

Contribution The main contribution presented in this paper builds upon the *AnBx* Compiler [35], extending its code generation capabilities. This results in concrete implementations of *AnBx* protocols in F*, using the DY* library. We provide our source code and case studies for reproducibility [4].

A prevalent concern that emerged from an extensive survey on the limited adoption of formal methods [25] relates to "Applicability: develop more usable software tools", with 68.5% of participants identifying it as a priority. To address this, we advocate a top-down approach to code generation:

1. Specify security requirements in an intuitive, yet formal, high-level language.
2. Validate the design using appropriate verification tools.
3. Follow a MDD strategy, generating concrete code from abstract models.
4. Allow for implementation modifications if required, and re-validation of code to guarantee that the security properties are preserved, subject to the condition that security goals proof code is not changed.

Outline of the Paper We start by detailing the specification languages and the back-end tools in Section 2. Section 3 outlines our methodology. In Section 4, we present the code generation construction and features. Sections 5 and 6 are dedicated to our evaluation and discussion on the related work. Finally, Section 7 summarises our conclusions and future work around our approach.

[4] `https://paolo.science/anbx/`

2 Background

2.1 The *AnBx* Compiler and the *ExecNarr* representation

The *AnBx* Compiler [35] can translate *AnBx* specifications to an intermediate representation called *Typed Executable Narration* (*ExecNarr*), which specifies the concrete actions and checks that honest agents must perform to execute the protocol. This yields a narration of the protocol from the point of view of every role, in which each role has a fully typed protocol specification, serving as the final stage before concrete code generation.

Consider an *AnBx* protocol with a single action `A->B: {A,NA}pk(B),NB`, where *NA* and *NB* are nonces and *pk(B)* denotes the public key of role *B*. *A* and *B* are two roles, and concrete entities assuming those roles are called principals. The security goal is that the *NA* created by *A* must remain confidential. The protocol can be compiled into an *ExecNarr* as follows, with π as a projection operator, *i.e.* here an indexed access to a tuple:

```
A: new NA
A: new NB
A: send(B:Agent,<enc(<A,NA>,pk(B)),NB>:
                  [SealedPair [Agent,Nonce],Nonce])
B: R0 := receive()
B: eq(A,π1/2[dec(π1/2[R0],inv(pk(B)))])
A: secret(chgoal_Confidential_NA_AB,NA,[A,B])
```

Here, the public-key ciphertext *{A,NA}pk(B)* has a dedicated *SealedPair* type. Symmetric key ciphertexts are, on their end, given a *SealedObject* type. *R0* represents what *B* receives from an insecure channel, supposedly sent by *A*. *inv(pk(B))* is the private key corresponding to *pk(B)*. In this example, *B* performs an equality check against the first projection of the decryption of *R0*, which corresponds to *A* if no Dolev-Yao intruder has tampered with the payload.

2.2 The F⋆ Language

F⋆ [44] is a general-purpose, proof-oriented functional programming language, that supports effectful code, *i.e.* code with side effects. By default, F⋆ is compiled to OCaml and relies on SMT solving using the Z3 solver [37] for proof automation. Its proof system is based on dependent types, *i.e.*, types that are filtered by predicates defining the space of allowed values.

F⋆ was developed to meet the need for verifiable implementations of critical software. It has successfully demonstrated this by providing a verified implementation of TLS [20]. For cryptographic operations, F⋆ notably relies on the *High-Assurance Cryptographic Library* (HACL⋆) [47].

2.3 The DY⋆ framework

DY⋆ [7] is a Dolev-Yao-based symbolic verification framework written in F⋆, targeting F⋆ implementations. To this end, it defines symbolically proven primitives, which can be called directly by the F⋆ code. Among its notable features,

it allows the modelling of mutable data structures, loops, and low-level implementation details like message formats. The models can be extracted to OCaml. Verification is based on dependent type checking over a global mutable trace.

Principals perform operations using modules that provide APIs for cryptography, state management, and other services. These modules impose the constraints that verification relies upon.

DY* has been enriched with *Comparse* [45], a library that automatically generates verified and interoperable binary parsers and serialisers. Parsing and serialisation are implemented with meta-programming tactics, encoding data of defined algebraic types into a binary format.

3 Methodology

3.1 Compiler Construction: Integration with the *AnBx* Compiler

To move from design to implementation, we rely on the *AnBx* Compiler and its ability to translate *AnB* narrations into an intermediate *Typed Executable Narration*, which encodes the steps of each role, as explained in Section 2.

In line with current code-generation practices in the *AnBx* Compiler, we use the HStringTemplate [18] library. It can process properties, conditionals, and lists, thereby enabling the generation of fine-grained, structured code.

We then extend the *AnBx* Compiler to generate implementations of a protocol from *ExecNarr*, its common base structure, to several languages (Figure 1).

DY* is a modular library, and we follow its main principle: to split the generated code into several modules, each serving a specific purpose. Consequently, each role's code resides in its own module, which depends on the *Messages*, *Sessions*, and *Functions* modules for message and session state structures and predicates. The security goals of a protocol are enforced through the checks included in the *Security Properties* module. These modules are described in detail in Section 4.

3.2 Formal Verification Methodology in DY*

Here, we present the core aspects of protocol verification in DY*, focusing on the concrete code artefacts that are manipulated during code generation.

Dependent types. Since DY* and F* generally rely on dependent types to prove specification properties, our security properties are expressed as type-refinement predicates on given terms. For example, if a term t must be a nonce secret between principals with roles a and b, we must enforce, through our predicates, an identities *label* such that `get_label t == (readers [P a; P b])`. If the value must be ephemeral, *i.e.* local to a session, we can restrict this label to `(readers [S a sa; S b sb])` with *sa* and *sb* session numbers. Further restriction can be applied with `(readers [V a sa vi; V b sb vi])`, with *vi* a protocol version number.

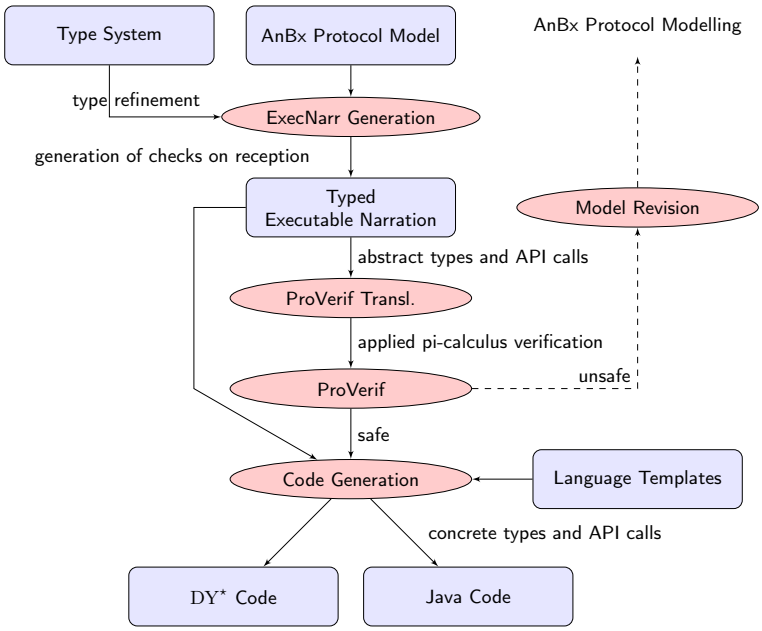


Fig. 1: *AnBx* Compiler MDD workflow (- - - - - manual ——— automatic)

Those type restrictions are typically checked in DY⋆ when using cryptographic or network primitives. Encrypting a term t requires that t satisfies labelling lemmas regarding principals' validity. Similarly, the verification of a send operation on t would fail if t is not labelled as publishable.

The nature of the term, here a nonce, can be defined at creation time, as a usage restriction. We define a custom string for our protocol, and make use of DY⋆ constructors, in this instance: `(nonce_usage protocolname)`. As with the labels, predicates of the form `get_usage t == (nonce_usage protocolname)` will have to be satisfied.

Trace-based reasoning. The defined predicates are checked as trace properties in DY⋆. We remind that trace properties are a combination of checks over program states. For example, the absence of bad behaviours like leaking secrets, as well as the presence of good behaviours like type invariants holding. Concretely, a protocol records every effectful operation in a global trace, such as term creation, event triggering, send and receive operations, etc. Any trace must then satisfy our protocol-dependent predicates, as well as type invariants.

For example, for the dependent type invariants, DY⋆ expresses the well-formedness of an arbitrary term t, as `is_valid i t`, with i as a timestamp established by the length of protocol traces, with the additional stability lemma $\forall$`j.(is_valid i t` $\wedge$ `j>i)` $\implies$ `(is_valid j t)`, ensuring that `is_valid` re-

mains true at all times. Validity here is defined as the set of predicates that honest participants must satisfy when creating a term. For example, root-level messages sent and received over the network must be public.

Core proof constructs. The proof workflow consists in the combination of checks applied when an effectful operation is done, with `is_valid` invariants stability, to satisfy the final high-level theorems, proving the security properties intended in the *AnBx* specification. This is compiled in Figure 2, where we detail specific predicates proved against F* code modules within the generated implementations. In it, the labelling checks are associated with secrecy properties, while event occurrence predicates are used for proving authentication through corresponding events. Intuitively, secrecy, defined as "secret between" some principals, means that, if one of the term t's holders knows t has a certain label l, then all principals referenced by l must know t by the end of the protocol, and prove that it carries label l. Authentication on t must ensure that, if a responder acknowledges t, then it knows that a specific initiator has transmitted t. Those final conclusions use intermediate facts mid-protocol to be established.

4 Code generation from *AnBx*

The *AnBx* Compiler constructs a *Typed Executable Narration*, which we denote as $P^{T\mathcal{X}}$ (introduced in Section 2). All subsequent DY* code artefacts are obtained through syntax-driven translation from $P^{T\mathcal{X}}$.

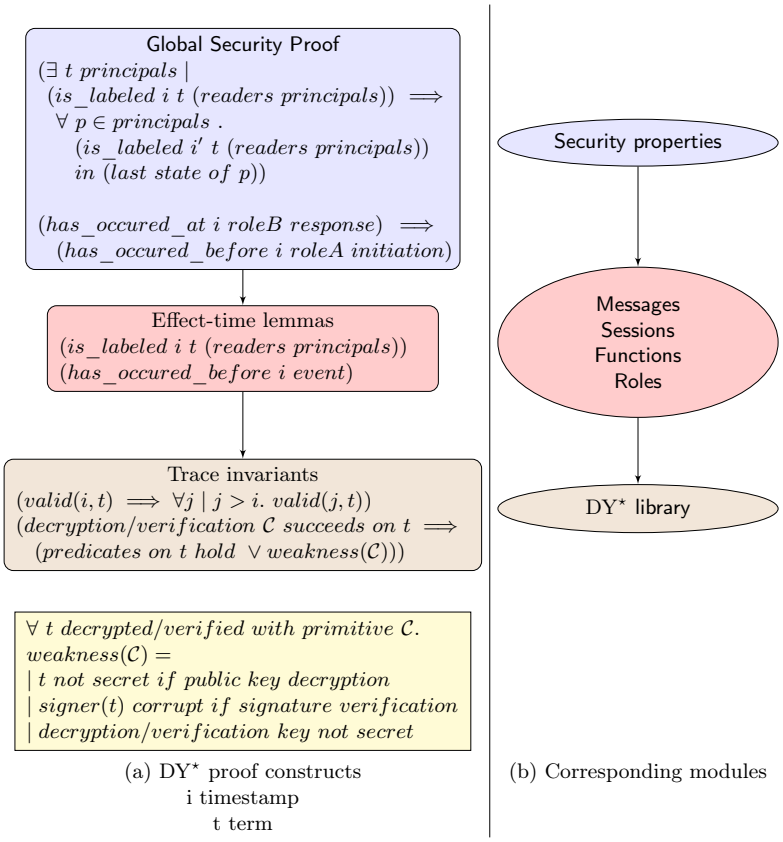
The remainder of this section presents the type system employed in the translation, and the organisation and responsibilities of each generated module, in line with the proof-oriented architecture illustrated in Figure 2.

4.1 Overview of code generation from *ExecNarr* to DY*

As the *ExecNarr* specifies cryptographic operations and security properties, DY* verifies them in a symbolic fashion. DY* checks *ExecNarr* typing properties, in contrast to just state-space exploration. We map the core operations of an *ExecNarr* to their corresponding DY* constructs (Table 1). These include term generation, assignments, message sending and receiving, sanity checks, and event recording. Each operation in DY* is associated with a logical timestamp, derived from the global trace length.

4.2 Type system

F* is a statically typed language, and DY* provides specialised types that can be attached to specific values within a given protocol. We adapt the *AnBx* type system, as refined in the *ExecNarr*, to DY*, thereby capturing constraints on freshness and designated usage contexts. Table 2 illustrates this with selected value–type correspondences, where i and l respectively denote a timestamp and a label, *i.e.*, the set of agents authorised to learn the term.



(a) DY⋆ proof constructs
i timestamp
t term

(b) Corresponding modules

Fig. 2: Proof architecture in the generated DY⋆ code (⟶ = depends on)

ExecNarr action	DY⋆ translation
A : `new` $x : \tau$	`let` $(\|idx, x\|)$ = `rand_gen` $label(secrecy\_goal\_x)$ $usg(\tau)$ `in`
A : `send`(ch, B, E)	`let` idx = `send` A B E `in`
$A : x :=$ `receive`(ch, B)	`let` $(\|idx', sender, x\|) = receive\_i$ idx A `in`
$A : x := E$	`let` $x = E$ `in`
A : `eq`(E, F)	`if` $E = F$ `then`
A : `wff`(E)	`print_string`$(A$ ^ "`knows `" ^ `print_msg` $E)$
A : `event`$(id, E, [A, B])$	`trigger_event A (event_id A B E)`

rand_gen: random labelled term generation	*label*: list of agent roles
secrecy_goal_x: *ExecNarr* goal identifier	$usg(\tau)$: enforce type τ
event $\in \{witness, request, wrequest\}$	*idx*: timestamp

Table 1: Translation of *ExecNarr* into DY⋆ (where *ch* denotes a plain channel).

ExecNarr term: type	DY* term: type
a: Agent	a: string
n: Nonce	n: bytes{was_rand_generated_before i n l (nonce_usage "protocol")}
k: SymmetricKey	k: bytes{is_publishable i k ∨ is_secret i k l (aead_usage "protocol")
k: PrivateKey <PKE>	k: bytes{is_secret i k l (pke_usage "protocol")}
k: PublicKey <PKE>	k: bytes{is_publishable i k ∧ (∃ sk. is_secret i sk l (pke_usage "protocol") ∧ k == pk sk)}
k: PrivateKey <SIG>	k: bytes{is_secret i k l (sig_usage "protocol")}
k: PublicKey <SIG>	k: bytes{is_publishable i k ∧ (∃ sk. is_secret i sk l (sig_usage "protocol") ∧ k == vk sk)}
b: SealedPair	b: bytes{is_publishable i b}
b: SealedObject	b: bytes{is_publishable i b}
s: DHSecret	s: bytes{is_secret i s l (dh_usage "protocol")}
k: DHPubKey	k: bytes{is_publishable i k ∧ (∃ s. is_secret i s l (dh_usage "protocol") ∧ k == dh_pk s)}
k: DHSecKey	k: bytes{is_secret i s l (aead_usage "protocol") ∧ local_to_session l}

i, l timestamp and secrecy requirement labels
is_secret i s l u $\iff$ (is_labeled i s l ∧ has_usage i s u)
local_to_session l $\iff$ l only allows one session for each of its principals
DH types are intended for Diffie-Hellman key exchange

Table 2: Type mappings between *ExecNarr* and DY* for base terms

In Table 2, we omit predicates for well-formedness and protocol-specific global usage parameters, for the sake of brevity. The base types in DY* that are supported for serialisation are *bytes*, *nat* and *string*, with agents systematically of type *string*, for inclusion in labels. The intended use of asymmetric keys is denoted as PKE and SIG respectively, for public-key encryption and digital signature. The existentially quantified formulae in Table 2 are easily discharged thanks to DY*'s PKE infrastructure available within the global trace.

4.3 Main module and Makefiles

To orchestrate the execution of a protocol, we first establish the initial knowledge of each role. In particular, we install the public and private keys in their local initial states, as specified in *AnBx*. Private keys are generated as follows, and the #-prefixed parameters are optional ones that could be inferred by F*:

```
gen_private_key #myprot #t role_A PKE my_pke_usg
```

Here, *myprot* is a protocol-specific package that carries runtime predicates, while t is a timestamp. The key is installed in A's knowledge and assigned the *PKE* type for public-key encryption, with a usage that may be key-specific.

Once an agent possesses a private key, the corresponding public key must be included in the knowledge of other agents. For example, to make A's public key available to role B, we install it as follows:

```
install_public_key #myprot #t role_B role_A PKE my_pke_usg
```

Long-term symmetric keys are created in the same module and distributed to their intended holders for inclusion in their respective session states. For instance, a symmetric key shared between roles A and B can be specified as:

```
(|_, shk|) = rand_gen #myprot (readers [P role_A; P role_B])
                              (aead_usage my_symenc_usg)
```

Once all long-term keys are initialised, the *Main* file orchestrates the execution of every protocol step. The first action executed by an agent returns both a session index and a timestamp, while subsequent actions return only a timestamp since the session is already created, and its number known. The session index is used by the initiating agent, whereas the timestamps capture the ordering of execution, as shown below:

```
let (idx_s_B, t1) = role_B_send_msg_1 t0 role_A role_B shk in
let (idx_s_A, t2) = role_A_recv_msg_1 t1 role_A role_B shk in
let t3 = role_A_send_msg_2 idx_s_A t2 in
let t4 = role_B_recv_msg_2 idx_s_B t3 in ...
```

The additional parameters in each role's first action are its initial knowledge. Here, roles A and B know each other and share a symmetric key. Verification and execution of the protocol is managed through *Makefiles* that invoke F* and DY* libraries. If all files type-check successfully, the specification is extracted to OCaml, compiled, and executed to produce a trace. Two forms of traces can be obtained via *make* commands: (i) a symbolic trace that prints the execution using symbolic operators defined in DY*, and (ii) a concrete trace that relies on HACL* cryptographic primitives, producing byte-level outputs.

4.4 Messages module

Messages are parsed and serialised using *Comparse* utility functions, providing brevity, convenience, and robustness. Dedicated tactics generate the parser, serialiser, and associated well-formedness lemmas. For each unencrypted payload, a type structure must be created to store the message. For example, given the *AnBx* action `A -> B: NA,NB` the corresponding payload structure is:

```
Msg0: na: bytes -> nb: bytes -> message_generic bytes
```

Now suppose that we add encryption and identities with `A -> B: {A,NA}pk(B),NB`. In this case, a structure must exist for both the encrypted component and its ciphertext. Structure names describe the positions of payloads within a channel message. In this example, *NA* appears in the first projection, which is a public-key encryption. We therefore obtain two structures:

```
Msg0: enc_a__na__pk_b:bytes -> nb:bytes -> message_generic bytes
Msg0_1_inPKE: a:principal -> na:bytes -> message_generic bytes
```

The *Messages* file also contains core predicates that capture the properties of encrypted messages. The underlying principle is that a message must satisfy

certain predicates at encryption time, which we can then rely on during parsing. For example, to ensure that NA is fresh and secret between A and B, we may specify a predicate (*ppred*) restricting the conditions under which public-key encryption can be applied at time i, with an encryption key pk given a usage string u, for a serialised message m:

```
let ppred i u pk m: prop =
  exists p. get_sk_label key_usages pk == readers [P p] /\
  match parse_message_raw m with
  | Success (Msg0_1_inPKenc role_A na) ->
      let role_B = p in
        was_rand_generated_before i na (readers [P role_A; P role_B])
                                       (nonce_usage my_nonce_usg)
  | _ -> False
```

Analogous predicates exist for digital signature, symmetric encryption and MACing. Note that identities relate to each principal role.

4.5 Sessions module

As with *Messages*, the *Sessions* module leverages *Comparse* parsers and serialisers, but for session states. These states capture relevant information at each protocol step for every participating principal, including received terms, generated terms, and variable assignments. For each recorded term, a set of properties is enforced through the *valid_session* predicate.

The facts required to prove security properties are embedded in *valid_session*. Term labelling and event occurrences are checked conjunctively. A property holds if, and only if, the expected labelling or events occur, or if one of the involved principals is corrupt. In the latter case, the adversary has full control over the compromised principal, including access to its knowledge.

Returning to the fresh nonce example from Section 4.4, the secrecy of NA for B is expressed as follows, with i denoting a timestamp. Here, we enforce strict checks on labelling and usage. If NA is also established as authentic, then its secrecy condition is satisfied. In this example, however, no signature or challenge–response mechanism is present to ensure authenticity; therefore, the check fails and the proof does not hold.

```
corrupt_id i (P a) \/ corrupt_id i (P b)
\/ was_rand_generated_before i na (readers [P a; P b])
                                  (nonce_usage my_nonce_usg)
```

4.6 Functions module

In *AnBx*, functions can be declared with a signature and a visibility. Public functions are generated in an imported file, whereas private functions reside in the role file of their respective agents, thus inaccessible to the intruder.

In *AnBx*, all functions are treated as injective symbolic functions. This formalism does not allow the specification of the body of a declared function. Consequently, implementations would require either reserved function names or equational theories for predefined behaviour.

We generate function-stubs, while preserving injectivity: each distinct input produces a distinct result. This is achieved by DY*'s handling of symbolic terms and the concatenation of the parameters provided.

4.7 Role modules

Every declared agent has its own role module specifying its behaviour. At the end of each principal's action, all observed terms are registered in a session state, where their predicates are required.

We summarise the overall flow for the example *ExecNarr* from Section 2.1. Role *A* generates fresh nonces *NA* and *NB*, serialises the inner payload, and encrypts it using role *B*'s public key to produce a well-formed timestamped message (*Msg0*) for transmission. Upon reception, *B* first parses the outer message to identify *Msg0*, then extracts and decrypts the inner encrypted component (*Msg0_1_inPKE*) using its private key. The decrypted content is then parsed and checked against expected tags, identities, and cryptographic predicates. The *ExecNarr* actions that correspond to the DY* operations are included as comments at operation time, and expression names are abbreviated for readability. The *AnBx* action here is `A -> B: {A,NA}pk(B),NB`, with *NA* confidential.

4.7.1 Sender's side The code for principal role *A* follows the flow described above. *A* labels *NA* using its own view of roles *A* and *B*.

```
//A: new NA
//A: secret(chgoal_Confidential_NA_AB,NA,[A,B])
let (|_,na|) = rand_gen #myprot (readers [P a; P b])
                  (nonce_usage my_nonce_usg) in
//A: new NB
let (|_,nb|) = rand_gen #myprot public (nonce_usage my_nonce_usg) in

let tsrz = global_timestamp () in
let srz_Msg0_1_inPKE = serialize_msg tsrz (Msg0_1_inPKE a na)
                         (readers [P a; P b]) in

let tenc = global_timestamp () in
let pk_b = get_public_key #myprot #tenc a b PKE my_pke_usg in
let (|tPKNonce,n_pke|) = rand_gen #myprot (readers [P a])
                            (nonce_usage "PKE_NONCE") in
let enc_a__na__pk_b = pke_enc #my_global_usage #tPKNonce
                       #(get_label myprot_key_usages n_pke)
                       pk_b n_pke srz_Msg0_1_inPKE in

let tsrz = global_timestamp () in
let srz_Msg0 = serialize_msg tsrz (Msg0 enc_a__na__pk_b nb) public in

let tBeforeSend = global_timestamp () in
//A: send(B:Agent,<enc(<A,NA>,pk(B)),NB>: [SealedPair [Agent,Nonce],Nonce])
let sendtime = send #myprot #tBeforeSend a b srz_Msg0 in ...
```

This code demonstrates nonce generation, public-key encryption, and final message serialization prior to sending. Timestamps and labels ensure traceability and enforce secrecy predicates, as discussed in Sections 4.4 and 4.5.

4.7.2 Receiver's side.

On B's side, the steps are detailed as:

```
//B: R0 := receive()
let (|trecv,_,r0|) = receive_i #myprot _tr_idx b in
let (Msg0 enc_a_na nb) = get_Msg0 trecv r0 in

let tdec = global_timestamp () in
let ((Msg0_1_inPKE a' na),_) = get_Msg0_1_inPKE b tdec enc_a_na in
//B: eq(A,π1/2[dec(π1/2[R0],inv(pk(B)))])
if not (a = a') then
  error ("EQCHECK Failed between A and a'. Values are:\n" ^ a ^ "\n" ^ a')
else ...
```

The dedicated decomposition helpers handle reception and decryption. *t0* and *t1* are the two versions of the global trace, respectively existing before and after applying the function.

```
let get_Msg0 (i:timestamp) (r:msg i public) :
              LCrypto (m:message{Msg0? m}) (pki myprot)
 (requires (fun t0 -> i == trace_len t0))
 (ensures (fun t0 m t1 -> trace_len t0 == trace_len t1
                          /\ receive_postcond m i)) =
   match parse_msg #i #(get_label myprot_key_usages r) r with
    | Success m -> if (Msg0? m) then m
                   else error ("Wrong tag for " ^ print_msg r
                               ^ "\n expected Msg0")
    | _ -> error ("Cannot parse " ^ print_msg r)
```

```
let get_Msg0_1_inPKE (b:principal) (_i:timestamp)
                     (enc_a_na:_tmsg _i public) :
    LCrypto (_m:message{Msg0_1_inPKE? _m} *
             _k:private_dec_key my_global_usage _i
                 (readers [P b]) my_pke_usg) (pki myprot)
 (requires (fun t0 -> _i == trace_len t0))
 (ensures (fun t0 (_m,_k) t1 -> trace_len t0 == trace_len t1
                  /\ pkdec_postcond _m _i b enc_a_na _k)) =
 let (|_,inv_pk_b|) = get_private_key #myprot #_i b PKE my_pke_usg in
 match pke_dec #my_global_usage #_i #(readers [P b]) inv_pk_b enc_a_na with
  | Success a_na ->
    (match parse_msg #_i #(get_label myprot_key_usages a_na) a_na with
     | Success _m -> if (Msg0_1_inPKE? _m) then (_m,inv_pk_b)
                     else error ("Wrong tag for " ^ print_msg a_na ^
                                 "\n expected Msg0_1_inPKE")
     | _ -> error ("Cannot parse " ^ print_msg a_na))
  | _ -> error ("PKE decryption failed for: " ^ print_msg enc_a_na ^
                "\nwith key: " ^ print_msg inv_pk_b)
```

This code ensures that B correctly decrypts and parses the message, verifies tags, and enforces all cryptographic properties. Here, *pkdec_postcond* captures the lemmas of a successful *pke_dec* operation, with *pke_dec* the public-key decyption primitive defined in DY*. *receive_postcond* checks that root-level payloads over the network are public.

4.8 Security Properties module

The *Security Properties* module reflects the *ExecNarr* secrecy and authentication annotations. Its purpose is to check that the derived predicates hold in the session states. This module should not be modified, otherwise the correctness of the intended *AnBx* properties could not be guaranteed. We provide illustrative correspondences below.

Secrecy Properties. For each principal p holding a given term t, the *ExecNarr* specifies the expression that p considers as being t. For instance, principal A knows NA in our running example (Section 2.1). If it must be secret between A and B, the annotation for A would be:

```
A: secret(NA_AB,NA,[A,B])
```

Here, NA_AB is a goal identifier. More generally, a secrecy annotation has the form: `holder: secret(goal id, expression, allowed holders)`

The verification process is twofold. First, we determine whether a given agent knows a value at the end of its protocol run, *i.e.* in its final session state. Second, this value is checked to be unknown to the intruder via API functions, unless one of the honest principals in its label is corrupt, as explained in Section 4.5. Type0 here is a fundamental type, and level 0 is a universe level meaning Type0 belongs in all sets of types in F*. Consider NA created by principal A. Its secrecy is checked as follows:

```
val is_na_in_A_state (idx_state idx_sess:nat) (v_A:version_vec)
                     (st_A:state_vec) (a b:principal) (na':bytes) : Type0
let is_na_in_A_state idx_state idx_sess v_A st_A a' b' na' =
  state_was_set_at idx_state a' v_A st_A /\
  state_inv myprot idx_state a' v_A st_A /\
  idx_sess < Seq.length st_A /\
  (match parse_session_st (st_A.[idx_sess]) with
   | Success (Role_A_Send_Msg0 a b na nb) -> a'==a /\ b'==b /\ na'==na
   | _ -> False)

val na_in_A_state_is_secret (idx_state idx_sess:nat) (v_A:version_vec)
                            (st_A:state_vec) (a b:principal) (na:bytes)
                            : LCrypto unit (pki myprot)
 (requires fun t0 -> idx_state <= trace_len t0 /\
                     is_na_in_A_state idx_state idx_sess v_A st_A a b na)
 (ensures fun t0 _ t1 -> t0==t1 /\
         (is_unknown_to_attacker_at (trace_len t0) na
          \/ corrupt_role (trace_len t0) a \/ corrupt_role (trace_len t0) b))
let na_in_A_state_is_secret idx_state _ _ _ _ _ na =
  let _len_t0 = global_timestamp () in
  assert (later_than _len_t0 idx_state);
  secrecy_lemma #(pki myprot) na
```

Analogous checks are performed for B.

Authentication Properties. Authentication properties are checked via corresponding events. Following the convention adopted by provers, such as OFMC and ProVerif, we define authentication as agreements between agents on given terms [32].

We denote by *witness* and *request* the events recording the initiator's and responder's sides, respectively. If the responder triggers a *request* event acknowledging a term, then the initiator must have triggered a matching *witness* event, barring corruption. Given B authenticating A on NA as a security requirement and the actions of 2.1, the events in *ExecNarr* are:

```
A: witness(auth_NA_BA,NA,[A,B])
B: request(auth_NA_BA,π1/2dec(π1/2[R0],inv(pk(B))),[A,B])
```

Here, the request references the expression that B sees as NA after extracting the received payloads. In DY*, the request-witness correspondence modulo corruption is encoded as:

```
val lemma_auth_NA_BA (i:timestamp): LCrypto unit (pki myprot)
 (requires fun t0 -> i < trace_len t0)
 (ensures fun t0 _ t1 -> t0 == t1 /\
  (forall a b na .
   did_event_occur_at i b (request_auth_NA_BA a b na)
    ==> (corrupt_role i a \/ corrupt_role i b \/
          did_event_occur_before i a
            (witness_auth_NA_BA a b na))))
let lemma_auth_NA_BA i = ()
```

The distinction between weak and strong (non-injective vs injective) authentication lies in the freshness of the *request*-event terms. Injectivity requires that the *request* terms are fresh relative to prior events. However, we inherit limitations that exist in DY* at the moment: if A triggers the *witness* event and B handles the *request*, B would need a challenge-response mechanism with fresh values from B to ensure that the values coming from A are fresh. Still, even this would not suffice, as DY* enforces predicates persistence across session states: once satisfied at a time i, they hold for all later times. Consequently, predicates such as "*request(N) never happened before*" could not be guaranteed for future states. Thus, strong (injective) authentication cannot be proven at the moment, and with it the general absence of replay attacks.

5 Verifications results and performance

In this section, we present the results of translating *AnBx* executable narrations to DY* and evaluating the generated implementations. We first discuss the validation of our translation by comparing the verification outcomes with established tools such as OFMC and ProVerif, including experiments that introduce intentional faults to test the detection capabilities. Subsequently, we report on the performance of the *AnBx* Compiler in generating DY* code and the time required for symbolic verification and concrete execution of the protocols, highlighting practical feasibility and efficiency in a range of case studies.

5.1 Translation empirical validation

The translation from *ExecNarr* to DY* is implemented in Haskell, and at the time of writing we do not provide a proof of translation correctness. Nevertheless, we present empirical evidence of consistency in the satisfaction of security properties when verification is performed in DY* as compared with OFMC and ProVerif, based on 10 protocols, subject to the limitation on strong authentication verification discussed in 4.8. These protocols employ fundamental crypto primitives like symmetric/asymmetric encryption, key agreement, hashing and Hmac schemes. Such experiments include secure and known vulnerable versions of these protocols, when available, notably with the original Needham-Schroeder protocol or the non-authenticated Diffie-Hellman protocol. Otherwise, as a sanity check, we introduced modified terms at certain steps (action indexes), as shown in Table 3, that lead to failure of the security goals.

Those changes enable various intruder exploits, because of mechanisms such as lack of signature, role confusion, etc. Having all our DY* implementations fail to verify under those conditions gives us empirical confidence on our translation layer.

This provides supporting evidence our translation with various protocol features, such as symmetric and asymmetric encryption, signing, fresh and long-term keys, Diffie-Hellman key exchange, hashing, and Hmac-ing, is rigorous.

Protocol	*Step*	*Secure version term*	*Insecure version term*
AndrewSecureRPC [31]	1	A,NA	A,NA,KAB
Denning-Sacco [21]	2,3	{|A,B,KAB,Ns|}shk(B,s)	{|B,KAB,Ns|}shk(B,s)
Diffie-Hellman [22]	1	{B,exp(g,XxX)}inv(sk(A))	B,exp(g,XxX)
ISOCCF1PassUniAuth [41]	1	{|NA,A,B,Text1|}shk(A,B)	{|NA,A,B|}shk(A,B)
ISOsym1PassUniAuth [42]	1	{|NA,A,B,Text1|}shk(A,B)	{|NA,A,B|}shk(A,B),Text1
ISOsym2PassUniAuth [42]	2	{|NB,A,B,Text1|}shk(A,B)	{|NB,A,B|}shk(A,B),Text1
NSL [30]	2	{B,NA,NB}pk(A)	{NA,NB}pk(A)
PGP_Auth [46]	1	{B,hash(Msg)}inv(sk(A))	{hash(Msg)}inv(sk(A))
TLS_HmacTranscripts [39]	1	hmac((A,s,Ns),shk(A,s))	hmac((A,s),shk(A,s))
Wide-Mouth Frog [14]	1	{|A,B,KAB|}shk(A,s)	{|A,KAB|}shk(A,s)

Table 3: Secure/Insecure version terms tested with DY⋆, OFMC, and ProVerif

5.2 DY⋆ Code generation and verification performance

In this section, we give performance numbers for the generation part with the *AnBx* Compiler, as well as the time taken to symbolically verify and concretely execute the protocols' implementations in DY⋆ through HACL⋆ primitives. We remind that the latter part depends on many factors independent of this work, such as DY⋆ itself, but also the F⋆ and Z3 versions used. Notably, we observed considerable time and memory usage increase with the number of steps, generating parsers and serialisers with Comparse. This benchmark was single-threaded, carried out with F⋆ 2025.09.04, Z3 4.13.3, and on a laptop Intel Core i5-1235U CPU with 16GB RAM. It can be observed that generation time is always below 100ms, execution time is reasonably fast, and the speed is basically bounded by concrete HACL⋆ cryptographic operations (ChaCha20-Poly1305, EdDSA-Ed25519, X25519 and SHA2-256). Memory utilisation for verification varies but never exceeds 5 GB.

Protocol	*Generation Time*	*Verification Time*	*Execution Time*
AndrewSecureRPC [31]	0.042s	6m33s	0.019s
Denning-Sacco [21]	0.043s	5m04s	0.022s
Diffie-Hellman [22]	0.072s	15m22s	0.081s
ISOCCF1PassUniAuth [41]	0.043s	2m27s	0.016s
ISOsym1PassUniAuth [42]	0.042s	1m14s	0.015s
ISOsym2PassUniAuth [42]	0.042s	2m40s	0.017s
NSL [30]	0.083s	4m51s	0.224s
PGP_Auth [46]	0.042s	1m03s	0.052s
TLS_HmacTranscripts [39]	0.052s	2m02s	0.049s
Wide-Mouth Frog [14]	0.043s	2m52s	0.018s

Table 4: Generation, verification, and execution times for DY⋆ implementations

6 Related Work

Using abstract primitives to enable higher levels of proof automation and capture increased complexity is not a new topic. On the other hand, concrete implementations that can still prove original goals after amendments are less common.

Implementation generation. Generating implementations after verifying an abstract model has been studied extensively by the research community. In [15], a computational model is proposed to leverage CryptoVerif [11] for model proof, and then to generate a runnable OCaml implementation that reflects it. That is, security properties in OCaml can be violated with the same probability as in the CryptoVerif model. In addition, the PSPSP tool [27] allows the specification and proof of security protocols with the Isabelle/HOL environment [38], relying on its Haskell code generation.

Another approach with symbolic models for close-to-implementation code generation is in [4]: Tamarin protocol-models are translated to I/O specifications. Those reflect the abstract model's features and can be used for later implementation verification, when encoded to a prover that supports separation logic.

Implementation verification. With an inferred model from a concrete specification, one can apply symbolic verification to general-purpose languages like C [2] or F# [8]. This inference can even be used to incorporate annotations in existing implementations. For example, in [36], authors generate postconditions for Java functions, executing methods on bounded exhaustive test sets depending on the used datatypes.

Direct verification, on the other hand, can be done in any language that supports proof mechanisms. Projects like Frama-C [19] have been useful to prove behavioural properties in C through annotations, along with dedicated tools for security protocols [24,17]. Another popular option based on Hoare logic is Dafny [29], with its statically-typed language, compiling to C#, Java, JavaScript, Go, and Python, while being close to their level of abstraction. F* supports invariants, pre and post conditions, and relies on the Z3 solver. Bhargavan *et al.* recently introduced a methodology to verify protocols written in Rust [9]: annotations are manually added to Rust specifications, then leveraged by multiple provers like ProVerif or F*, each suited to a specific task.

7 Conclusion and Future Work

In this contribution, we introduce a code generation approach from abstract *AnBx* models [13] to implementation-level F* [44], leveraging the DY* library [7]. We allow Dolev-Yao [23] style protocols to be formally defined and verified against user-defined security goals of secrecy and authenticity. The workflow we propose uses the *AnBx* Compiler, enriched to generate DY* code from its own intermediate *ExecNarr* representation of *AnBx* protocols. The benefit of this approach is to maintain security protocol provability at both ends of the development effort: a design is provable, as well as its implementation. In doing so, concrete implementations, proven correct, can be amended to fit specific deployment requirements not expressible in *Alice & Bob*, and those amendments can be checked as compatible with the specified security properties.

We present a detailed overview of correspondences from *AnBx* to DY* with ten case studies on well-known protocols, covering asymmetric and symmetric encryption, long term keys, hash and Hmac. After verifying a DY* implementation, both symbolic and concrete executions are possible.

In the future, we plan to refine our code generator to fit more complex protocols, support equational theories, and explore the interoperability possibilities with existing implementations, such as protocols of the Noise framework [28]. In the spirit of accessibility, we will also integrate it to the AnBx IDE [26], for further user support.

Acknowledgements We would like to thank Théophile Wallez for his help, advice and clarifications on DY* throughout this work. For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

References

1. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The Spi calculus. In: CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997. pp. 36–47. ACM (1997). https://doi.org/10.1145/266420.266432
2. Aizatulin, M., Gordon, A.D., Jürjens, J.: Extracting and verifying cryptographic models from C protocol code by symbolic execution. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011. pp. 331–340. ACM (2011). https://doi.org/10.1145/2046707.2046745
3. Almousa, O., Mödersheim, S., Viganò, L.: Alice and Bob: Reconciling formal models and implementation. In: Bodei, C., Ferrari, G., Priami, C. (eds.) Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 9465, pp. 66–85. Springer (2015). https://doi.org/10.1007/978-3-319-25527-9_7
4. Arquint, L., Wolf, F.A., Lallemand, J., Sasse, R., Sprenger, C., Wiesner, S.N., Basin, D.A., Müller, P.: Sound verification of security protocols: From design to interoperable implementations. In: 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. pp. 1077–1093. IEEE (2023). https://doi.org/10.1109/SP46215.2023.10179325
5. Avalle, M., Pironti, A., Pozza, D., Sisto, R.: JavaSPI: A framework for security protocol implementation. International Journal of Secure Software Engineering **2**(4), 34–48 (2011). https://doi.org/10.4018/jsse.2011100103
6. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. International Journal of Information Security **4**(3), 181–208 (2005). https://doi.org/10.1007/s10207-004-0055-7
7. 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 European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021. pp. 523–542. IEEE (2021). https://doi.org/10.1109/EUROSP51992.2021.00042
8. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(1), 5 (2008). https://doi.org/10.1145/1452044.1452049
9. Bhargavan, K., Hansen, L.L., Kiefer, F., Schneider-Bensch, J., Spitters, B.: Formal security and functional verification of cryptographic protocol implementations in rust. IACR Cryptol. ePrint Arch. p. 980 (2025), https://eprint.iacr.org/2025/980
10. Black, J.: Authenticated encryption. In: van Tilborg, H.C.A. (ed.) Encyclopedia of Cryptography and Security. Springer (2005). https://doi.org/10.1007/0-387-23483-7_15

11. Blanchet, B.: A computationally sound mechanized prover for security protocols. IEEE Transactions on Dependable and Secure Computing **5**(4), 193–207 (2008). https://doi.org/10.1109/TDSC.2007.1005
12. Blanchet, B., et al.: Modeling and verifying security protocols with the applied pi calculus and proverif. Foundations and Trends in Privacy and Security **1**(1-2), 1–135 (2016). https://doi.org/10.1561/9781680832075
13. Bugliesi, M., Calzavara, S., Mödersheim, S., Modesti, P.: Security protocol specification and verification with AnBx. Journal of Information Security and Applications **30**, 46–63 (2016). https://doi.org/10.1016/j.jisa.2016.05.004
14. Burrows, M., Abadi, M., Needham, R.M.: A logic of authentication. ACM Transactions on Computer Systems **8**(1), 18–36 (1990). https://doi.org/10.1145/77648.77649
15. Cadé, D., Blanchet, B.: Proved generation of implementations from computationally secure protocol specifications. J. Comput. Secur. **23**(3), 331–402 (2015). https://doi.org/10.3233/JCS-150524
16. Cassidy, S.: Existential type crisis : Diagnosis of the OpenSSL Heartbleed Bug (2014), http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html
17. Chaki, S., Datta, A.: ASPIER: an automated framework for verifying security protocol implementations. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009. pp. 172–185. IEEE Computer Society (2009). https://doi.org/10.1109/CSF.2009.20
18. Clover, S.: HStringTemplate, http://www.haskell.org/haskellwiki/HStringTemplate
19. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c - A software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
20. Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Protzenko, J., Rastogi, A., Swamy, N., Béguelin, S.Z., Bhargavan, K., Pan, J., Zinzindohoue, J.K.: Implementing and proving the TLS 1.3 record layer. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 463–482. IEEE Computer Society (2017). https://doi.org/10.1109/SP.2017.58
21. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. Commun. ACM **24**(8), 533–536 (Aug 1981). https://doi.org/10.1145/358722.358740
22. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6), 644–654 (1976). https://doi.org/10.1109/TIT.1976.1055638
23. Dolev, D., Yao, A.: On the security of public-key protocols. IEEE Transactions on Information Theory **2**(29) (1983). https://doi.org/10.1109/tit.1983.1056650
24. Dupressoir, F., Gordon, A.D., Jürjens, J., Naumann, D.A.: Guiding a general-purpose C verifier to prove cryptographic protocols. J. Comput. Secur. **22**(5), 823–866 (2014). https://doi.org/10.3233/JCS-140508
25. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Nickovic, D. (eds.) Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12327, pp. 3–69. Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1

26. Garcia, R., Modesti, P.: A practical approach to formal methods: An Eclipse Integrated Development Environment (IDE) for security protocols. Electronics **13**(23) (2024). https://doi.org/10.3390/electronics13234660
27. Hess, A.V., Mödersheim, S., Brucker, A.D., Schlichtkrull, A.: Performing security proofs of stateful protocols. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00006
28. Ho, S., Protzenko, J., Bichhawat, A., Bhargavan, K.: Noise*: A library of verified high-performance secure channel protocol implementations. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022. pp. 107–124. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833621
29. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
30. Lowe, G.: An attack on the needham-schroeder public-key authentication protocol. Information Processing Letters **56**(3), 131–133 (1995). https://doi.org/10.1016/0020-0190(95)00144-2
31. Lowe, G.: Some new attacks upon security protocols. In: Ninth IEEE Computer Security Foundations Workshop, March 10 - 12, 1996, Dromquinna Manor, Kenmare, County Kerry, Ireland. pp. 162–169. IEEE Computer Society (1996). https://doi.org/10.1109/CSFW.1996.503701
32. Lowe, G.: A hierarchy of authentication specifications. In: CSFW'97, pp. 31–43. IEEE Computer Society Press (1997)
33. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 696–701. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_48
34. Mödersheim, S.: Algebraic properties in alice and bob notation. In: Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan. pp. 433–440. IEEE Computer Society (2009). https://doi.org/10.1109/ARES.2009.95
35. Modesti, P.: AnBx: Automatic generation and verification of security protocols implementations. In: García-Alfaro, J., Kranakis, E., Bonfante, G. (eds.) Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9482, pp. 156–173. Springer (2015). https://doi.org/10.1007/978-3-319-30303-1_10
36. Molina, F., Ponzio, P., Aguirre, N., Frias, M.F.: Evospex: An evolutionary algorithm for learning postconditions. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 1223–1235. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00112
37. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008,

Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
38. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
39. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Aug 2018). https://doi.org/10.17487/RFC8446, https://www.rfc-editor.org/info/rfc8446
40. Sprenger, C., Klenze, T., Eilers, M., Wolf, F.A., Müller, P., Clochard, M., Basin, D.A.: Igloo: soundly linking compositional refinement and separation logic for distributed system verification. Proc. ACM Program. Lang. **4**(OOPSLA), 152:1–152:31 (2020). https://doi.org/10.1145/3428220
41. International Organization for Standardization, Genève, S.: ISO/IEC 9798-4:1999, Information technology – Security techniques – Entity Authentication – Part 3: Mechanisms using a cryptographic check function, Second edition (1999)
42. International Organization for Standardization, Genève, S.: ISO/IEC 9798-2:2008, Information technology – Security techniques – Entity Authentication – Part 2: Mechanisms using symmetric encipherment algorithms, Third edition (2008)
43. 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). https://doi.org/10.1017/S0956796813000142
44. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 256–270. ACM (2016). https://doi.org/10.1145/2837614.2837655
45. Wallez, T., Protzenko, J., Bhargavan, K.: Comparse: Provably secure formats for cryptographic protocols. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023. pp. 564–578. ACM (2023). https://doi.org/10.1145/3576915.3623201
46. Zimmermann, P.R.: The official PGP user's guide. MIT Press, Cambridge, MA, USA (1995)
47. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: Hacl*: A verified modern cryptographic library. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1789–1806. ACM (2017). https://doi.org/10.1145/3133956.3134043