Formalizing and Strengthening the Security Proof of NTOR

François Dupressoir[1], Kristian Gjøsteen[2], Cameron Low[1], and Charlotte Mylog[2]

[1] University of Bristol
[2] Norwegian University of Science and Technology (NTNU)

Abstract. We present a machine-checked security proof for the NTOR key exchange protocol, which is used to establish connections in the Tor onion routing system. It was previously studied by Goldberg *et al.* (DCC 2013) and the protocol ladder project, however there is no full proof including forward secrecy in the computational model. Our proof is formalized in EasyCrypt, adding to the still small set of cryptographic protocols verified using EasyCrypt. A key contribution is a systematic treatment of halting reductions involving failure events expressed as global properties of the execution. In the course of this work, we also contributed improvements to the EasyCrypt framework itself.
We prove NTOR secure in a model for unilaterally authenticated key exchange (UAKE) similar to the maximum exposure model by Fiedler *et al.* (Crypto 2025). It captures forward secrecy concretely, and is intentionally close to established bilaterally authenticated key exchange models (such as eCK) while being simple to formalize. By examining more carefully how identities and public keys are used in key exchange protocols, we obtain simpler formal arguments and introduce several variants of the UAKE security model, connected by general reductions that, in the case of NTOR, are also verified in EasyCrypt. This allows us to carry out the main proof in a simpler setting and then derive the desired security guarantee for NTOR via these reductions.

1 Introduction

Key exchange protocols are a cornerstone of modern cryptographic systems, enabling practical and efficient secure communication over untrusted channels. As a result, key exchange remains an active research field, driven by real-world protocols. The literature distinguishes two families of key exchange protocols. *Bilaterally authenticated key exchange* (BAKE) requires both participants to authenticate each other, whereas *unilaterally authenticated key exchange* (UAKE) authenticates only one side. In onion routing networks such as Tor [Tora], the service provider is authenticated while clients remain unauthenticated. TLS and QUIC likewise support deployment modes with unilateral authentication.

A concrete UAKE protocol is `ntor` [GSU13], which Tor uses to establish a shared secret between a client and a relay. It is designed to provide forward

secrecy, minimal server authentication, resistance to active attacks, and client anonymity. Despite its practical importance, `ntor` still lacks a rigorous computational security proof including forward secrecy. At the same time, there has been a growing shift toward *machine-checked* security proofs, which increase confidence in cryptographic arguments. A machine-checked proof for `ntor` including forward secrecy in the computational model is therefore highly desirable.

1.1 Related Work

Goldberg, Stebila, and Ustaoglu [GSU13], who introduced `ntor`, adapted the extended Canetti-Krawczyk (eCK) framework [LLM06,LLM07] to UAKE. Their model supports strong adversarial powers, but relies on somewhat complex queries that are not widely used. This is common in the key exchange literature, where real-world protocols are often accompanied by tailored security models.

Today, most analyses of modern key-exchange protocols, follow the *maximum-exposure* approach based on the model by Cohn-Gordon *et al.* [CCD+16] in their analysis of Signal's initial key exchange. The central idea is to maximize the adversary's corruption capabilities while excluding only attacks that let the adversary win trivially.

For the formalization of protocol proofs, existing tools generally fall into two categories: symbolic and computational. Symbolic tools offer high levels of automation which is desirable and has been leveraged for the formalization of protocol proofs [CHSvdM16,CHH+17,BDH+18,CHSW22]. However, they often lack the expressiveness needed to model complex cryptographic primitives, probabilistic behavior and computational assumptions. As a result, they may be unsuitable for analyzing certain AKE protocols and their proofs cannot always guarantee cryptographic security in the computational sense. Computational tools offer a more faithful model of cryptographic security, but typically provide less automation, making proofs more labor-intensive and harder to scale. Nevertheless, they have been used to study real-world protocols such as TLS [BBK17] and SSH [CB12].

To the best of our knowledge, there is no publication providing a mechanized security proof of `ntor`. The protocol was, however, previously considered in the *protocol ladder* project, which aims to formalize security proofs for a collection of protocols in different verification tools. The project provides publicly available code and a pen-and-paper reference proof [PL]. However, the pen-and-paper model allows adversaries to choose which instances to test before they complete and does not consider forward secrecy. The only computational proof of `ntor`, implemented in CryptoVerif, considers a model without corruptions and therefore does not support forward secrecy. There are two formal proofs in the symbolic model of which only the proof in Tamarin considers forward secrecy. Both of them establish "weak secrecy" of the input to the key derivation function, modelled as a random oracle: the input cannot be derived from the adversary's observations, but indistinguishability of the derived session key in a computational AKE sense is not shown. Establishing a general reduction from

such weak secrecy of the RO input to computational key indistinguishability for UAKE protocols therefore remains an interesting direction for future work.

In EasyCrypt, the work on proofs of protocol security remains limited due to their complexity. Barthe *et al.* [BCLS15] prove Naxos and Nets secure and also show how their security notion can be extended to the eCK model using the Naxos trick. However, they neither released their EasyCrypt code nor discussed insights from the formalization. Concurrently, Barbosa *et al.* [BDL+26] study a more systematic approach to formalizing protocol proofs in EasyCrypt, using a key exchange protocol with pre-shared keys as a case study. Their proof only considers session-key reveals, as the protocol would otherwise be insecure, which simplifies freshness notions in their model.

1.2 Our Contributions

We propose four versions of our model for UAKE security that balance expressiveness and simplicity for both paper-based and machine-checked proofs, and relate them via simple generic transformations. These relations enable a machine-checked security proof of a slightly simplified version of `ntor` in EasyCrypt. We summarize our contributions below.

Our Security Model. As discussed earlier, real-world key exchange protocols are often accompanied by tailored security models that balance readability, proof convenience, and model strength. We take the same approach, adapting the eCK model to UAKE with a focus on formalization. The resulting model resembles the maximum-exposure approach similar to [FGPZ25], but is tailored to UAKE by allowing asymmetry between clients and servers, as described in Section 2. Our main motivation for introducing this model is the gap between the GSU model and what is convenient to formalize in EasyCrypt.

Comparison with the Goldberg, Stebila and Ustaoglu Model. The `ntor` protocol was accompanied by a security model for UAKE [GSU13] which successfully adapts eCK to the unilateral setting. However, the GSU model introduces additional complexity that we aim to avoid in our proposed model.

More importantly, the GSU model does not track the timing of server corruptions relative to the acceptance of corresponding client instances. Consequently, it excludes cases where the adversary learns both the server's long-term key and ephemeral secret, provided the long-term key is revealed only after the client accepts. This case is necessary to be considered when arguing about forward secrecy and can be reduced to solving the underlying assumption relying on the authentication tag. The GSU analysis is therefore missing a formal proof of forward secrecy. To address this and strengthen the security proof, we introduce local session state to our model that records whether corruption occurred before the client's acceptance, similarly to Brzuska *et al.* [BFWW11], and we adapt the freshness notion accordingly.

Another distinction concerns the adversary's ability to reveal instance state. The GSU model introduces a RevealNext query, allowing the adversary to access

public values before they are bound to specific sessions (and, using the Partner query, to obtain the associated secret before it is used). We instead model randomness reveals analogously to ephemeral-key reveals in eCK [LLM07], allowing them for completed sessions so that the server can be modelled using a single algorithm. The models thus differ in adaptivity: unlike GSU, our adversary cannot *predict* randomness. We are unaware of a realistic setting where this matters.[3]

Names vs. Public Keys. During our analysis of `ntor`, we identified two subtle sources of proof complexity. The first concerns the server's identifier. The `ntor` protocol uses names as identifiers for servers and includes them, together with a DDH secret derived from the server's long-term key, as input to the session-key derivation, modeled as a random oracle. Consequently, the proof must relate server names to their corresponding public keys and rely on this mapping in the reduction step. This additional bookkeeping can be avoided by using public keys directly as server identifiers, which also simplifies the use of the DDH oracle in the reduction, particularly in the formalization.

The second concerns whether the adversary is allowed to register dishonest server parties. Although it is unsurprising that, in the unilateral setting, dishonest party registration provides no additional advantage to the adversary, we include a formal proof of this result because it illustrates how a seemingly trivial modeling decision can significantly complicate a mechanized proof. This complexity again arises in the reduction, in particular when simulating random-oracle queries. We emphasize that this observation does not extend to BAKE.

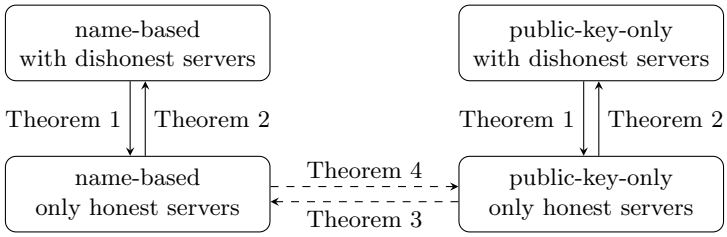


Fig. 1. Relations between models, with the relevant theorems indicated. The dashed arrows indicate that the theorems apply to a restricted class of protocols.

[3] The typical case is the failure or compromise of a (pseudo-)random generator, revealing to an adversary *future* generator outputs, and thereby future ephemeral keys. One failure mode (*e.g.* CVE-2008-0166) results in low-entropy pseudo-randomness, which could among other things cause two instances to use the same randomness, which is not well-modelled by the RevealNext query. One mode of compromise (*e.g.* Dual-EC-DRBG) results in the ability to use some output from a generator to predict future outputs of that generator instance, which is not well-modelled by the RevealNext query. Also, we are not aware of any non-contrived AKE scheme where the ability to see an ephemeral key *before* interacting with a session using that emphemeral key, not just after interaction, results in further useful attacks.

To systematically investigate these issues, we introduce four variants of our UAKE security model, illustrated in Fig. 1. The horizontal axis distinguishes how servers are referenced—by name or by public key—while the vertical axis captures the adversary's capabilities, distinguishing between models that do and do not permit dishonest server registration.

All relationships shown in Fig. 1 are established generically for a broad class of protocols, as detailed in Section 2.3. In EasyCrypt we then instantiate these results for the `ntor` protocol, as described in Section 2.5. Our focus there is on the theorems required to compose a proof that moves from public-key-only security with only honest servers to name-based security with dishonest servers.

Game-based Security Proof for `ntor`. In our work, we exploit the relations between models and prove the security of a modified version `m_ntor`, introduced in Section 3.2, which uses public keys rather than names as identifiers. We also consider security with only honest servers. This yields a cleaner, more modular proof, which is particularly beneficial for the formalization.

To establish the security of `m_ntor`, we follow the overall structure of the original `ntor` security argument, summarized in Section 3.3. We refine this argument by introducing a detailed game sequence illustrated in Figure 2 which explicitly accounts for forward secrecy. An overview explaining each step can be found in Section 4.1. Under the multi-instance strong computational Diffie–Hellman (St-CDH) assumption with reveals, as described in Section 4.3, we obtain a security guarantee formalized in Theorem 5 in Section 4.5.

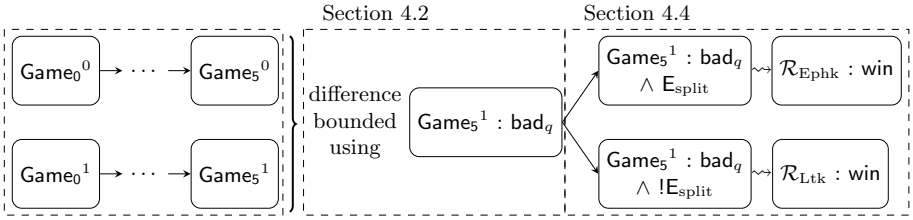


Fig. 2. Intuition for full game-based security proof for the modified `ntor` protocol. We start at the left with the real and ideal version of Game_0 that captures public-key-only security with only honest servers of the modified `ntor` protocol. Applying the depicted transformations leads to two reductions that win against the hardness assumption.

Formalization of security proof. The main objective of this project is to implement a full game-based security proof for `ntor` in the computational model. To formalize, we chose EasyCrypt [EC,BGHZB11,BDG+12], a mature tool for computational cryptography that follows the code-based game-playing paradigm. Our choice of tool was influenced in part by the structure of the original proof,

which relies on an indexed family of reductions.[4] Such indexed reductions are not directly supported in tools such as CryptoVerif [Bla08]. Although protocol proofs in EasyCrypt are relatively rare due to the effort required for formalization, we aim to extend its use to AKE and share our insights. In particular, we formalize the security proof of the modified `ntor` version using only public keys and present it alongside the proof on paper in Section 4 and following.

Another advantage of using EasyCrypt is its comprehensive library of reusable theories, which includes standard techniques and lemmas needed for protocol proofs. In addition to demonstrating these techniques in practice, our work also provides two new observations at the tool level.

The first is a more systematic approach to simulations or reductions that halt the experiment. Although they are often used on paper to simplify the treatment of failure events, they are somewhat complex to capture formally: the precise semantics of an oracle signalling to the experiment that it should halt with an adversary loss is more complex to handle than it appears—as pointed out by Stepanovs [Ste25], among others. Most proofs formalized in EasyCrypt so far get away with a lightweight treatment of failure events: when they occur in a non-interactive setting (or when they can be decided immediately as they happen), the reduction can typically recover somewhat gracefully. This proof, however, requires us to consider failure events that are expressed as global properties of the execution, and could be triggered *after* the reduction has made an irrecoverable decision. In those instances, it is necessary for the reduction to halt.

The second is a change to the tool itself, allowing games to be more easily defined as a modification of a previous game. This is described more in detail in Section 2.4. Beyond the simple benefits of not repeating definitions, this makes proofs a lot more robust to changes made in early games. (This currently comes at a cost in the readability of definitions, which we hope can be alleviated in future by better interactive proving tools.)

A more detailed discussion of the insights gained from formalizing these arguments, along with a comparison to concurrent work, is provided in Section 5. Sections 4.2 and 4.4 highlight the proof obligations arising in the more complex steps, including proof invariants. An invariant captures properties of the game state and its relations that hold independently of the adversary's queries, and can also relate the states of two games.

2 Security Model for UAKE

There are several approaches to modeling security of unilaterally authenticated key exchange (UAKE) that include different frameworks, different security goals, and varying levels of adversarial power. The most common modelling technique nowadays follows the maximum-exposure approach which is mostly used for multi-stage protocols that are more complex than `ntor`.

[4] The final result presented in this work is based on a multi-instance assumption and therefore no longer requires indexed reductions. This decision was made during the formalization process rather than prior to selecting the tool.

The models we propose are based on the eCK framework of [LLM07][5], adapted to the unilateral setting. We allow all non-trivial compromises as in the maximum-exposure approach and account for forward secrecy. In contrast to existing maximum-exposure models, we specialize our model to the `ntor` setting by considering only two-message protocols with a long-term server key and ephemeral keys or randomness for both the client and the server. An earlier version of our work included a more general UAKE model; however, this did not correspond to the model formalized in EasyCrypt, where expressing such generic constructions is considerably more difficult.

In this section, we start by giving a formal definition of UAKE protocols before introducing four variants of our model. These variants differ along two dimensions: whether servers are identified by names or only by public keys, and whether the adversary can register dishonest server parties or not. We then establish formal relationships between these models. Before concluding the section with a discussion of the formalized proofs of the relations, we go on a brief technical detour introducing a new feature of EasyCrypt used in our formalization.

2.1 UAKE protocols

Key Exchange protocols are used to establish a shared key between two parties that can be used for further communication. We refer to the parties as client and server instead of initiator and responder since this is the usual setting for UAKE. Here the server is the authenticated party of the protocol whereas the client stays unauthenticated. As in the bilaterally authenticated setting, we define a UAKE two-message protocol Π to have the following algorithms:

- $\Pi.\mathsf{KeyGen}() \to (pk_S, sk_S)$: This algorithm generates and outputs long-term key pairs which are used for server parties.
- $\Pi.\mathsf{Start}(s) \to (st, m)$: We consider the starting algorithm only for the client side. It takes an identifier s of the server it is supposed to communicate with as input and returns a protocol state of this client instance and a potential message. The algorithm outputs $\perp$ to signal a failure.
- $\Pi.\mathsf{SProcess}(st, m) \to (st', m', k)$: The server's processing algorithm takes a protocol state st and a message m as input. The protocol state may include the long-term key, since we do not have a starting algorithm. The algorithm outputs an optional state st', a message m' and a key k. If the entire output is empty or $\perp$ is returned the execution of this step failed.
- $\Pi.\mathsf{CProcess}(st, m) \to (st', k)$: The processing algorithm for client is defined similarly except for not outputting a message.

Many UAKE definitions introduce an explicit starting algorithm for the server to separate randomness generation from protocol execution. This design also enables ephemeral keys or randomness to be revealed before an instance

[5] There are two versions of this paper, which differ substantially. The presented work relies on the later version.

completes, after which such state would typically be deleted. In practice, however, real-world protocols with one-round of messages commonly use a single execution for a server instance that samples and computes all required values at once. We therefore decided to model the server execution without separate starting algorithm.

We implement the algorithms in two module types, shown in Listing 1.1, which define the procedure interfaces in EasyCrypt. All procedures assume full, well-typed inputs, enforced at the model level. Both the server and client access a key-derivation function modelled as random oracle (RO), passed as a parameter in EasyCrypt. On paper, we omit the RO from the model because in our setting it is not necessary for understanding the model and would only add formalism without providing additional insight. Nevertheless, in other settings, it may be useful to include the RO explicitly in the definitions.

```
1  module type Server (H : RO) = {
2    proc keygen() : pkey * skey
3    proc respond_session(st: s_state, m: msg1) :
     (s_state * msg2 * key) option
4  }.
5
6  module type Client (H : RO) = {
7    proc new_session(s: s_id, pk: pkey) : c_state * msg1
8    proc complete_session(st: c_state, m: msg2) :
     (c_state * key) option
9  }.
```

Listing 1.1. Definition for one-round UAKE protocols

2.2 Our Security Models

As in standard key exchange models, we assume an unauthenticated network and grant the adversary full control over all communications. In particular, the adversary is permitted to redirect, delete, and forge messages. Furthermore, they initiate all key exchange sessions and determine the order in which these instances are executed. A detailed description of the adversarial queries is provided below and is illustrated in Figure 3.

Parties and Instances. We consider a setting where we distinguish between server and client instances. Our model allows for multiple server parties, each having multiple instances. On the other hand, we do not use parties for the clients, as our analysis does not consider privacy properties such as unlinkability.

Therefore, the instance identifier ID is asymmetric and is either an integer i denoting the ith client instance or a tuple (s, j) for the jth instance of server s. In the name-based model s represents the server's name n whereas in the public-key-only model we use the public key pk as identifier instead. This causes issues in the

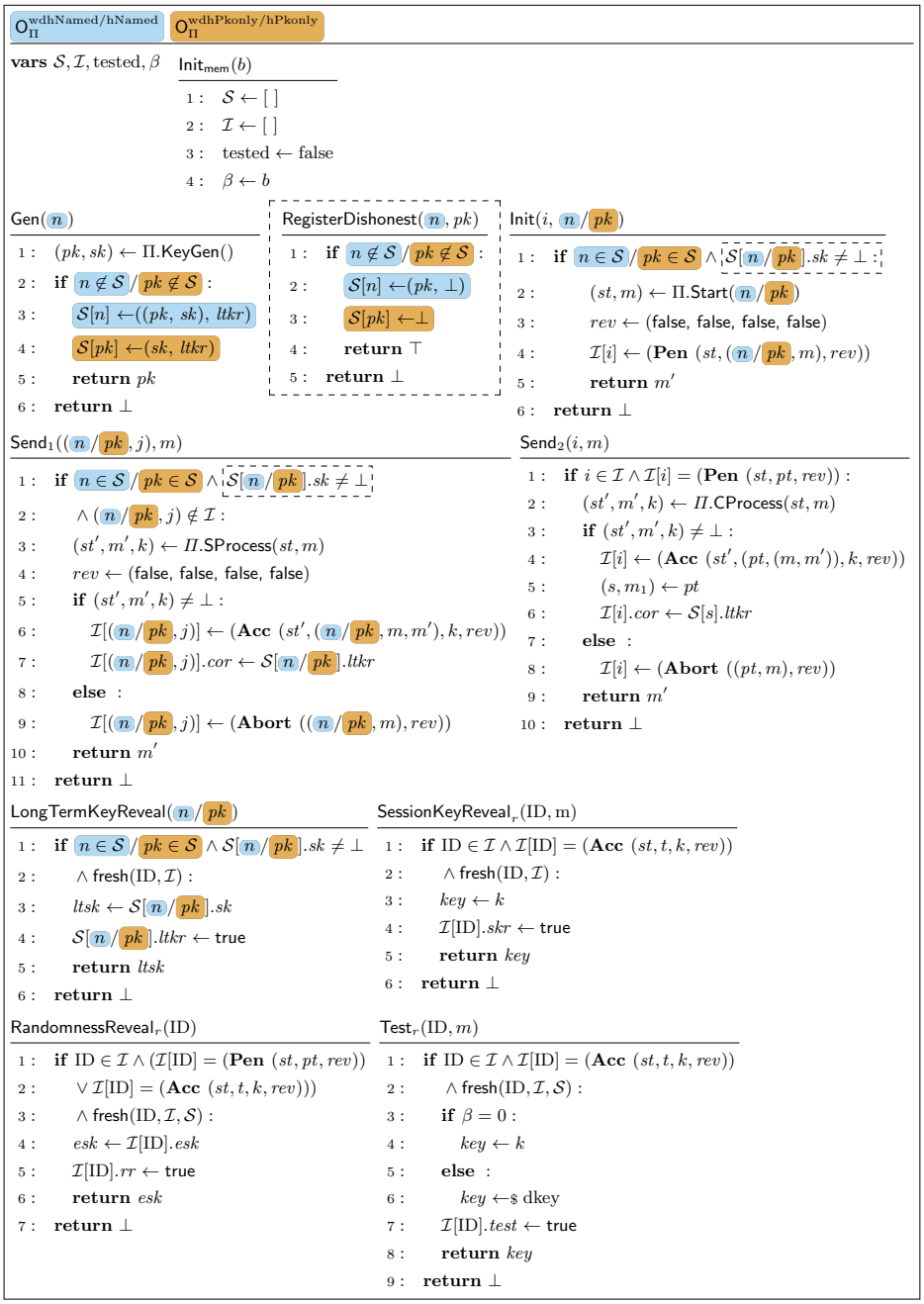


Fig. 3. Oracles for the name-based and public-key-only UAKE security model. Dashed-boxed lines are included only in the version with dishonest (wdh) servers; removing them gives the version with only honest (h) servers. Combining these options yields four models: wdhNamed, hNamed, wdhPkonly, and hPkonly. All queries indexed by r can be executed for each role $r \in \{\text{client}, \text{server}\}$.

case of collisions of sampled public keys since then the identifiers are not referring to a unique party. However, collisions only appear with negligible probability. The concrete relation between these models is discussed in Section 2.3.

All server parties are recorded in a finite map $\mathcal{S}$ mapping the instance ID to a key pair (pk, sk) in the name-based model and only the secret key sk in the public-key-only model. The secret key sk is left empty for dishonest servers. Additionally, $\mathcal{S}$ stores a record $ltkr$ which indicates if the secret long-term key has been corrupted. $\mathcal{S}.ltkr$ is initialized as 0 and set to 1 in the long-term key reveal. All server and client instances are kept track of using the game state $\mathcal{I}$. In our game each instance can be in one of three states: *pending* when it is a running instance, *accepted* if the instance completed and holds a key, or *aborted* for instances that received invalid inputs. We keep track of the model state of an instance and all values necessary for later queries with the following constructors:

- **Pen** (st, pt, rev): A pending instance with protocol state st including secret values of that instance, partial transcript pt and reveal status rev.
- **Acc** (st, t, k, rev): An accepted instance with protocol state st, full transcript of messages received and sent by this instance t, obtained shared key k and reveal status rev.
- **Abort** (pt, rev): An aborted instance with partial trace pt and reveal status rev which is used to evaluate freshness of potential partners.

The protocol state st includes the ephemeral secret key esk of the instance and for clients the identifier s of its intended partner. In the reveal status rev the model keeps track of the following flags where the last one is crucial to reason about forward secrecy.

- $rr \in \{\mathsf{true}, \mathsf{false}\}$ indicates whether the randomness or ephemeral key of the instance is revealed.
- $skr \in \{\mathsf{true}, \mathsf{false}\}$ indicates whether the session key of the instance is revealed.
- $test \in \{\mathsf{true}, \mathsf{false}\}$ indicates whether the instance is tested.
- $cor \in \{\mathsf{true}, \mathsf{false}\}$ indicates whether the instance accepted after the corresponding server party was corrupted.

Servers and Long-term Keys. Before initiating a session, the adversary has to register the long-term keys of the involved server party. In addition to honest keys, the adversary may register dishonest keys (no possession of the secret key required) in the versions that include dishonest servers. For the public-key-only versions we disallow registering and generating the same public key which would make the adversary win trivially. In all models the adversary can reveal server keys and corrupt its instances. The registration may occur at any time and is not required to be completed before any protocol instances are started. The adversary is given access to the following queries, each of which (like all subsequent queries) validates its inputs and returns $\perp$ if any check fails.

– Register Honest$(n) \to pk$: Samples a long-term key pair and connects it to the honest server with name n. For the public-key-only model the input to this query is empty, and the returned public key pk is used as identifier.
– Register Dishonest$(n, pk) \to \top$: Sets pk as the long-term key of server n that is registered as dishonest. For the public-key-only model with dishonest parties the input is only a public key pk.
– Long-Term Key Reveal$(s) \to ltsk$: Reveals the long-term secret key of a server party with identifier s which then is registered as revealed.

Protocol Queries and Adversarial Powers. Besides the registration and revealing of long-term keys, the adversary has access to queries for different protocol and game actions. The latter capture the revealing powers including the test query that can be performed once.

– Init$(\mathrm{ID}, s) \to m'$: Starts either a client instance with identifier ID that aims to establish a shared key with the server party with identifier s or a server instance with identifier ID and initializes instance state.
– Send$_*(\mathrm{ID}, m) \to m'$: Sends the message m to the instance with identifier ID. Since we restrict this model to two-message protocols, we use a Send$_1$ and a Send$_2$ query.
– Session Key Reveal$(\mathrm{ID}) \to key$: Reveals the established session key of the completed instance with identifier ID.
– Randomness Reveal$(\mathrm{ID}) \to esk$: Reveals the randomness or ephemeral secret key of the instance identified by ID. This instance may be incomplete.
– Test$(\mathrm{ID}) \to key$: Returns the same value as the Session Key Reveal(ID) query if the distinguishing bit $\beta = 0$. Otherwise, it returns a randomly sampled key. It fails if it is queried on incomplete or non-fresh sessions.

Partnering Notion. We adopt matching conversations as our partnering notion and identify sessions via transcripts including two instances of opposite role. Each transcript includes all messages sent or received by the instance and the identifier of the server party. While this inclusion is not strictly necessary for protocols that send or cryptographically bind the server's identity (e.g., through tags), we choose to retain it in the model.

In addition, we define a pre-partnering notion that collects all instances of the opposite role that agree on the server identity and the first message; this is needed to evaluate freshness for queries that occur before both partners have completed. We call any instance satisfying the pre-partnering notion an origin.

Where needed, we define operators that combine the partnering notion with additional constraints on the state of the partners following the modelling in EasyCrypt. For example, the operator that checks whether an instance has untested partners is defined as follows:

$$\mathsf{untested_partner}(\mathsf{ID}, \mathcal{I}) = \begin{cases} \bot, & \text{if ID has no partner} \\ \text{true}, & \text{if ID has at least one untested partner} \\ \text{false}, & \text{otherwise} \end{cases}$$

Similarly, we define the operators fresh_partner and tested_origins. We also introduce an operator get_partners, which returns a set of all partner indices.

Freshness Conditions. We introduce a freshness notion to rule out adversarial behavior that would lead to trivial distinguishing attacks and thus break protocol security. An instance ID is called *not* fresh in any of the following cases:

- The adversary reveals the session key of the instance ID.
- The adversary compromises the instance ID. In our case, a server instance is compromised if both its long-term and randomness are revealed. A client instance can be impersonated by revealing only the randomness or ephemeral key.
- Instance ID has one or more partners, but none of them is fresh with respect to the first two points.
- Instance ID is a server instance without partner or a client instance without partner and the intended server party was corrupted before ID accepted. In the models with dishonest parties this includes dishonest servers.

To ensure that all sessions remain fresh, we prevent all of these situations from arising by following the filtering approach of [BCJ+24] and silencing all misbehaving queries. Since we do not assume any particular order of queries and cannot guarantee that the test oracle is queried last, we must impose restrictions on all relevant queries. The freshness conditions fresh(ID, $\mathcal{I}$, $\mathcal{S}$) and fresh(ID, $\mathcal{I}$) for each query are given formally in Figure 4. An instance is fresh in a specific query if it satisfies all listed conditions. A server party is fresh in the long-term key reveal if all its instances satisfy the condition.

	Client i	Server (s, j)
LTKeyRev		• $\neg\, \mathcal{I}[(s,j)].rr \vee (\neg\, \mathcal{I}[(s,j)].test \wedge$ untested_partner $((s,j), \mathcal{I}) \neq$ false)
SesKeyRev	• $\neg\, \mathcal{I}[i].test$ • untested_partner $(i, \mathcal{I}) \neq$ false	• $\neg\, \mathcal{I}[(s,j)].test$ • untested_partner $((s,j), \mathcal{I}) \neq$ false
EphKeyRev	• $\neg\, \mathcal{I}[i].test$ • tested_origins $(i, \mathcal{I}) \neq$ true	• $\neg\, \mathcal{S}[s].ltkr \vee (\neg\, \mathcal{I}[(s,j)].test \wedge$ untested_partner $((s,j), \mathcal{I}) \neq$ false)
Test	• $\neg\, \mathcal{I}[i].skr$ • $\neg\, \mathcal{I}[i].rr$ • fresh_partner $(i, \mathcal{I}) \neq$ false • ($\lvert$get_partners $(i, \mathcal{I})\rvert \neq 0 \vee$ $\neg\, \mathcal{I}[i].cor$) • $\mathcal{S}[s].sk \neq \bot$	• $\neg\, \mathcal{I}[(s,j)].skr$ • $\neg\, \mathcal{I}[(s,j)].rr \vee \neg\, \mathcal{S}[s].ltkr$ • fresh_partner $((s,j), \mathcal{I}) \neq$ true

Fig. 4. Freshness conditions for each role and query as phrased in EasyCrypt.

Security Definition. Our security notion is formulated as an indistinguishability property between a real and an ideal game, both embedded in a single experiment. The experiment Exp^b, denoting the real game when $b = 0$ and the ideal game when $b = 1$, initializes the game state by invoking $\mathsf{Init}_{\mathsf{mem}}(b)$. It then provides the adversary $\mathcal{A}$ with the oracle $\mathsf{O}_{\Pi,\mathrm{H}}$, which includes all queries defined above except $\mathsf{Init}_{\mathsf{mem}}$, and access to the random oracle H. The oracle may be instantiated according to any of the four models. At the end, the experiment outputs the adversary's guessing bit b'. Within this framework, security for UAKE protocols is defined as follows.

Definition 1 (UAKE Security). *A protocol Π is said to be UAKE secure in model α, where $\alpha \in \{\text{wdhNamed, hNamed, wdhPkonly, hPkonly}\}$, if the following distinguishing advantage of any adversary $\mathcal{A}$ is small:*

$$\mathsf{Adv}^{\Pi}(\mathcal{A}) := |\Pr\left[\mathrm{Exp}^0(\mathsf{O}^{\alpha}_{\Pi,\mathrm{H}}, \mathcal{A}) = 1\right] - \Pr\left[\mathrm{Exp}^1(\mathsf{O}^{\alpha}_{\Pi,\mathrm{H}}, \mathcal{A}) = 1\right]|$$

2.3 Relations between Security Models

We have sketched four different UAKE models, differing along two distinct axes: whether parties are identified by names or solely by public keys, and whether honest clients are allowed to interact with dishonest servers.

With vs without dishonest servers. We deal with the simpler issue first and show equality of the versions with and without dishonest registration for both the name-based and the public-key-only model. Any adversary that is prohibited from initiating honest clients with dishonest servers can be transformed, at no cost, into an adversary that is allowed to make such initiations, but refrains from doing so. Since a forbidden query will not change the state of the experiment or the evaluation of the win conditions, we can modify the adversary to detect forbidden queries and then not make them, simulating any error message if needed. This yields a trivial reduction in one direction.

Theorem 1. *Let $\mathcal{A}$ be an adversary against Π in the model without dishonest servers. Then there exists an adversary $\mathcal{B}$ against Π in the model with dishonest servers, with essentially the same runtime as $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\Pi}(\mathcal{A}) = \mathsf{Adv}^{\Pi}(\mathcal{B}).$$

It remains to consider the other direction. In general, an honest client talking to a dishonest server is not so interesting, since it can never be fresh with respect to authentication or confidentiality. However, in ordinary BAKE, having an honest client talk to a dishonest server could still be valuable, since the honest client will use some secret key material, and it is this usage that the adversary in principle could leverage to attack the server. This fact becomes apparent when we do security proofs, in the sense that we cannot in general simulate clients that use secret key material.

This proof problem disappears in the unilateral setting, where the client does not have any long-term key material. Simulating honest clients talking to dishonest servers is then easy and can be done by the adversary. Moreover, as mentioned, no honest client talking to a dishonest server affects the win condition, so there is no difference if these clients are run by the experiment or simulated by the adversary. This gives us the required reduction in the other direction, and we find that for UAKE, allowing or disallowing honest clients talking to dishonest servers is equivalent.

Theorem 2. *Let $\mathcal{A}$ be an adversary against Π in the model with dishonest servers. Then there exists an adversary $\mathcal{B}$ against Π in the model with only honest servers, with essentially the same runtime as $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\Pi}(\mathcal{A}) = \mathsf{Adv}^{\Pi}(\mathcal{B}).$$

Names vs public keys. We show that for a particular class of protocols that uses server names in a particular way is equivalent to protocols that use only public keys. We therefore focus on the setting with only honest servers and rely on our composition theorems to lift the results to the case with dishonest servers. In the class we consider, server names are used only as inputs to a particular function, which we model as a random oracle.

This class is quite large and contains many practical protocols in their unilateral form[6], including `ntor`. Intuitively, the result should extend to an even broader class: for instance, merely including names in messages to add them to the transcript should be straightforward to handle.[7] However, if a protocol applies more complex transformations to names that cannot be well modeled by a random oracle, then simulating such behavior becomes difficult. In that case, translating a protocol with names into one using only public keys is hard, and aligning their executions is correspondingly challenging.

Let U and W be sets. We define a "base protocol" where the algorithms have access to an oracle for a function from U to W. In particular, the client start algorithm takes a server long-term public key as input and has access to an oracle from U to W. It outputs a protocol state and a message. The process algorithm takes a state, a server long-term public key and a message as input and has access to an oracle from U to W. It outputs a state, a message and a key, any of which can be empty. We denote the execution of these algorithms with access to an oracle for a particular function $f : U \to W$ by

$$\begin{aligned}(st_c, m) &\leftarrow \Pi.\mathsf{Start}^f(pk)\\ (st'_c, m', k) &\leftarrow \Pi.\mathsf{CProcess}^f(st_c, pk, m)\\ (st'_s, m', k) &\leftarrow \Pi.\mathsf{SProcess}^f(st_s, pk, sk, m)\end{aligned}$$

[6] Examples can be found in the chapter *Key Agreement Protocols* by Boyd *et al.* [BMS19] starting from the one-pass HMQV and the Naxos protocol.

[7] In fact, the original specification of `ntor` [GSU13] does include a name in a message, but this is a simple case and handled by including names in the partnering notion.

In this way, the "base protocol" has no access to the server name, except2 through the function f. This captures the essential property of our class of protocols.

Let V be a set and $f : U \times V \to W$. For any $v \in V$, we can now define a function $f^v : U \to W$ by

$$f^v(u) = f(u, v).$$

With this notation, we can now use the base protocol to instantiate a name-based variant $\Pi_N = (\Pi.\mathsf{KeyGen}, \Pi_n.\mathsf{Start}, \Pi_N.\mathsf{Process})$ and a public-key-only variant $\Pi_P = (\Pi.\mathsf{KeyGen}, \Pi_p.\mathsf{Start}, \Pi_P.\mathsf{Process})$ as follows. Let $f_N : U \times \mathcal{N} \to W$ and let $f_p : U \times \mathcal{P} \to W$ be functions. The name-based protocol is defined by

$$\begin{aligned}
\Pi_n.\mathsf{Start}(n, pk) &= \Pi.\mathsf{Start}^{f_N^n}(pk) \\
\Pi_n.\mathsf{CProcess}(st_c, n, pk, m) &= \Pi.\mathsf{CProcess}^{f_N^n}(st_c, pk, m) \\
\Pi_n.\mathsf{SProcess}(st_s, n, pk, sk, m) &= \Pi.\mathsf{SProcess}^{f_N^n}(st_s, pk, sk, m)
\end{aligned}$$

The public-key-based protocol is defined by

$$\begin{aligned}
\Pi_p.\mathsf{Start}(pk) &= \Pi.\mathsf{Start}^{f_P^{pk}}(pk) \\
\Pi_p.\mathsf{CProcess}(st_c, pk, m) &= \Pi.\mathsf{CProcess}^{f_P^{pk}}(st_c, pk, m) \\
\Pi_p.\mathsf{SProcess}(st_s, pk, sk, m) &= \Pi.\mathsf{SProcess}^{f_P^{pk}}(st_s, pk, sk, m)
\end{aligned}$$

Under the additional assumption that the underlying functions f_N and f_P are modeled as random oracles, we can now prove the equivalence of the two models. Note, that we only consider the models without dishonest servers to simplify the theorem and proof. The same argument for the versions with dishonest servers can be retrieved by composition.

Theorem 3. *Let $\mathcal{A}$ be an adversary against Π_N in the name-based model. Then there exists an adversary $\mathcal{B}$ against Π_P in the public-key-only model, with essentially the same runtime as $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\Pi_N}(\mathcal{A}) \leq \mathsf{Adv}^{\Pi_P}(\mathcal{B}) + \epsilon_1 + \epsilon_2,$$

where ϵ_1 is the probability that collisions in sampling occur, and ϵ_2 is the probability that the adversary predicts a message before sampled by an honest party.

Proof. The proof proceeds as a sequence of games. The first game is the adversary interacting with the name-based model experiment.

First we modify the game so that whenever a collision occurs among the honestly generated public keys, the experiment ignores the sampling and returns $\perp$. Such a collision appears with probability ϵ_1, so the advantage of $\mathcal{A}$ in this hop is reduced by at most the probability ϵ_1. Now there is a bijection π between the set of registered names and the set of registered public keys.

Next, we replace the RO for the function f_N^s with an oracle for $f_P^{\pi(s)}$. For server names that have not been registered, we must query f_P. This means that

we must also make sure that adversarial queries are consistent before and after server names are registered. We do not ensure this consistency for queries made by honest parties. This introduces a possible bad event, where an adversarial query involving a server name s is made, the server name is later registered, and then an honest party repeats the adversarial query. This happens only of the adversary can predict an honest message. This event happens with probability ϵ_2. Since we model both f_N and f_P as random oracles and π is a bijection, this change is unobservable unless the bad event happens.

Finally, we construct a simulator *Sim* between the adversary $\mathcal{A}$ and the public-key-only experiment translating all oracle queries. Throughout its execution, it maintains the bijection π. *Sim* also simulates an f_N oracle, keeping track of its responses. When $\mathcal{A}$ makes a new query (s, t) to its f_N oracle with s in the domain of π, *Sim* makes a $(\pi(s), t)$ query to its f_P oracle and forwards the response to $\mathcal{A}$. Otherwise, *Sim* responds using its f_N oracle simulation.

When $\mathcal{A}$ makes a RegisterHonest query with name n, *Sim* makes the same query to its experiment (without the name), to which it receives a public key. The simulator updates the bijection π and forwards the public key to $\mathcal{A}$. When $\mathcal{A}$ makes an Gen with name n, *Sim* makes an Gen query with public key $\pi(n)$, forwarding any response. Long-term key reveal queries are handled in the same way, while the other queries are simply forwarded unchanged.

The public-key-only experiment, together with *Sim* and the random oracle for f_P, exactly simulates the name-based experiment (with rejection of duplicate public keys) and the random oracle for f_N. The theorem therefore follows. □

The same approach will also give us the opposite result, but since we can hardly assign meaningful names to the public keys, it is unclear how useful the reduction is. The essential step is to introduce a simulator that assigns arbitrary names to RegisterHonest queries and keeps track of the bijection π. Again, it translates public keys to names in the other queries. We state the result, but leave the detailed proof to the interested reader.

Theorem 4. *Let $\mathcal{A}$ be an adversary against Π_P in the public-key-only model. Then there exists an adversary $\mathcal{B}$ against Π_N in the name-based model, with essentially the same runtime as $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\Pi_P}(\mathcal{A}) \leq \mathsf{Adv}^{\Pi_N}(\mathcal{B}) + \epsilon_1 + \epsilon_2,$$

where ϵ_1 is the probability that honestly generated public keys collide and ϵ_2 is the probability that the adversary predicts some honestly generated public key.

2.4 Modelling Game Changes in EasyCrypt

Before presenting any proofs in EasyCrypt, we take a brief detour to examine some EasyCrypt-specific technicalities. The issue we focus on arises whenever we formalize a sequence of games—or even a single intermediate step—in a proof, including when establishing the relationships between models described above.

In any game-based proof one has to translate the individual games of the proof into the corresponding EasyCrypt code. On paper, we usually describe intermediate games by applying transformations to the previous game in the sequence. These transformations, generally, specify small changes to oracles such as dropping or adding a guard, or the introduction of state that the oracles may access. In Figure 5, we illustrate two common types of game changes: introducing a bad event, and adding new state to oracles.

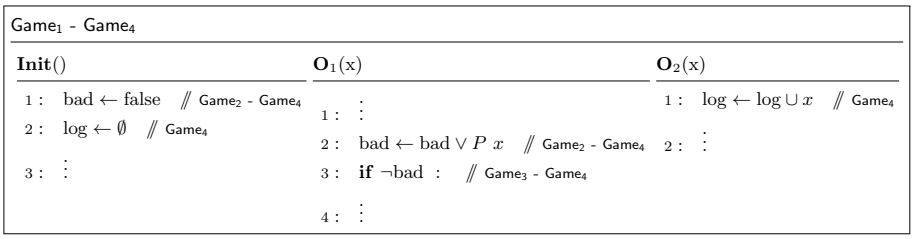


Fig. 5. Examples of how games can be described in pen-and-paper proofs.

```
1  (* Initial game *)
2  module Game1 = {
3      proc Init_mem() = { (* ... *) }
4      proc O1(x) = { (* ... *) }
5      proc O2(x) = { (* ... *) }
6  }.
7  (* Add bad event *)
8  module Game2 = Game1 with {
9      var bad : bool
10     proc Init_mem [ 1 + { bad <- false; } ]
11     proc O1 [ n + { bad <- bad \/ P x; } ]
12 }.
13 (* Use bad event *)
14 module Game3 = Game2 with {
15     proc O1 [ [n .. m] + (!bad) ]
16 }.
17 (* Introduce a log *)
18 module Game4 = Game3 with {
19     var log : t fset
20     proc Init_mem [ 1 + { log <- fset0; } ]
21     proc O2 [ 1 + { log <- log `|` fset1 x; } ]
22 }.
```

Listing 1.2. EasyCrypt module tweaks for Figure 5

Before this work, translating games into EasyCrypt required writing each game separately and manually propagating changes across all subsequent definitions. This is error-prone and makes it difficult to see which parts of the games

differ. To address this, we developed *Module Tweaks*, a new EasyCrypt feature that allows games to be specified as on paper. It lets users derive a module (EasyCrypt's grouping of oracles) from an existing one by applying syntactic transformations, such as adding, removing, or modifying expressions and conditions. Listing 1.2 shows how this feature can be used to translate Figure 5.

We use this new feature especially in the security proof presented later to define all games in the sequence and reductions, which can be found in the artifact file Games.ec.

2.5 Proving Relations between Models in EasyCrypt

We first formalize the models in EasyCrypt in the artifact files UAKE_name.ec and UAKE_pkonly.ec, where each oracle is a module and each query is a procedure. The main difference to the model described on paper lies in splitting up the game state $\mathcal{I}$ into clients and servers and using role-specific queries, which is primarily due to using different types for the identifier ID on each side.

The need for types in EasyCrypt also prevents us from formalizing the model relations as generic as stated on paper. Instead we prove a concrete instantiations of the theorems for the `ntor` protocol, which we introduce in Section 3 and implement in the NTOR_name.ec and NTOR_pkonly.ec files. This still yields a somewhat generic result: the types for the random-oracle inputs and the protocol algorithms can be easily adapted to other one-round protocols, and the proof in EasyCrypt then follows the same structure and can be reused to a large extent. Extending this approach to multi-round protocols is more involved, but our proof can serve as a blueprint for such generalizations.

In our formalization we focus on the relations that are relevant for the security proof. Concretely, we show equivalence between the security in the name-based model with and without dishonest servers, and that public-key-only security implies name-based model security in the setting with only honest servers.

With vs without dishonest servers. As in the paper proof, the main obligation is to show that, when removing dishonest servers, we can simulate all dishonest servers and all client instances interacting with them. In EasyCrypt, we realize this via an intermediate step using the Module Tweak feature: we first forbid communication with dishonest servers, and only in a subsequent step remove the corresponding query altogether. In the first step, we split the client state into instances that communicate with honest servers, which are handled within the experiment, and instances that communicate with dishonest servers, which are controlled directly by the adversary. The key argument is that such simulated client instances can never be partnered with an honest server. The second step is simpler: we keep track of all names registered as dishonest and ensure that no dishonest and honest server party share the same name.

Names vs public keys. In this step we reduce a name-based adversary to a public-key-only adversary in the setting with only honest servers. As discussed above these notions are equivalent upto the case that either there occurs a collision

in the sampling of long-term keys or the adversary manages to predict any randomness that is later sampled by the protocol. As is standard in EasyCrypt, we introduce two flags in the reduction that capture these bad events. To simplify the use of the builtin up to bad reasoning provided in EasyCrypt, the reduction is made to halt upon the triggering of the bad event. This frees us from having to think (and formally reason) about what the reduction should do *after* the bad event has happened. We apply this strategy also in the following security proof. To accomplish a halting reduction, each oracle in the reduction must be wrapped with a guard that checks whether the event has been set, and if it has, it immediately returns. Effectively, all oracles are silenced as discussed in the introduction.

For equivalence up to the triggering of the bad events, we need different invariants that express equality of states up to a function that replaces or removes names. For instance, the instance state for clients in the public-key-only game is the same as the corresponding state in the name-based game after replacing the name with the public key in the protocol state and removing the name of the corresponding server from the transcript. Furthermore, we require the correspondence between names and public keys to be correct and ensure that corruptions are tracked consistently. We know that the public (and secret) keys associated with a name are unique, since otherwise the bad event is triggered. Finally, we bound both bad events in EasyCrypt to ensure that they are defined correctly.

3 The `ntor` Protocol

Goldberg, Stebila, and Ustaoglu (GSU) introduced the `ntor` protocol [GSU13], a unilaterally authenticated key exchange (UAKE) protocol designed to provide client anonymity specifically for the use in the Tor network. For this work, we focus on its security and do not prove anonymity.

In this section, we first present the version of the protocol by GSU and its implementation in the Tor onion routing system, before we describe the simplifications and modifications we make to the protocol in our work. Finally, we summarize the security argument provided by GSU.

3.1 Original Protocol and Application in the Tor Network

The `ntor` protocol is illustrated in the flow diagram in Figure 6. The participating parties are divided into clients, who remain unauthenticated, and servers, which are authenticated. For clarity and visual purposes, the diagram includes only the cryptographically relevant inputs to the hash function queries. The first part describes the generation of long-term keys for servers, where each server sets its certificate to bind its name to its public key. The second part illustrates one session of the protocol.

The protocol uses Diffie-Hellman (DH) secrets which are elements of a cyclic group $\mathbb{G}$ of prime order p with generator g. The session key and MAC key are

derived from a hash function $\mathsf{H} : \mathbb{G} \times \mathbb{G} \times \mathcal{N} \times \mathbb{G} \times \mathbb{G} \to \{0,1\}^k \times \{0,1\}^k$ where $\mathcal{N}$ denotes the set of server names. The MAC key is used by a hash functions $\mathsf{H}_{mac} : \{0,1\}^k \times \mathcal{N} \times \mathbb{G} \times \mathbb{G} \to \{0,1\}^k$ to compute an authentication tag. Additionally, the `ntor` uses H_{sid} to generate session identifiers which we omit in the figure.

`ntor` is specifically designed for the use in onion routing systems, and it was adopted as handshake protocol in the third generation of the Tor network. It replaces the older TAP protocol with a more efficient and secure design and is used to establish a shared seed that serves as input to a key derivation function. The implementation of `ntor` can be found in the Tor project proposal #216 [Torb].

The key difference between this implementation and the `ntor` protocol lies in the input to the key derivation function. In the original protocol, the input consists only of the server's name, the ephemeral public keys, and the two DH shared secrets. In contrast, the Tor implementation additionally includes the server's long-term public key. This modification avoids a subtle issue in the proof: since servers are identified by names while the security ultimately relies on their public keys, the proof requires establishing the correspondence between a server name and its public key before verifying the validity of the DH shared secret involving the long-term key. Including the long-term public key directly in the key derivation input eliminates this additional matching step. However, the resulting design still couples server names with the protocol's security properties, which we consider conceptually undesirable. Therefore, we introduce a variant of the protocol that relies exclusively on public keys, as described below.

3.2 Subjects of this Work

In this work we consider two protocols-one slightly simplified version `s_ntor` of the `ntor` protocol by GSU and one modified protocol `m_ntor` using public keys instead of names to identify servers. Both are illustrated in Figure 6 and differ from `ntor` in the following points.

- We do not manage session state within the protocol itself, but instead delegate that responsibility to the model. In particular, we can remove any checks if the instance has the necessary state for a given round of the protocol or that the messages have the correct form. Moreover, we do not compute any session identifiers as part of the protocol and discard the hash function H_{sid} originally used since our model does not rely on session identifiers.
- Since our UAKE model already includes the name n in the transcript used for partnering, we omit it from the first protocol message. The server is able to determine its own identity.
- Instead of computing a MAC with a separate hash function H_{mac} we use the first part of the output of H–used as the MAC key in the design by GSU–as the tag itself, simplifying the proof in EasyCrypt. Since the MAC key is never revealed, the adversary can distinguish a single random function from

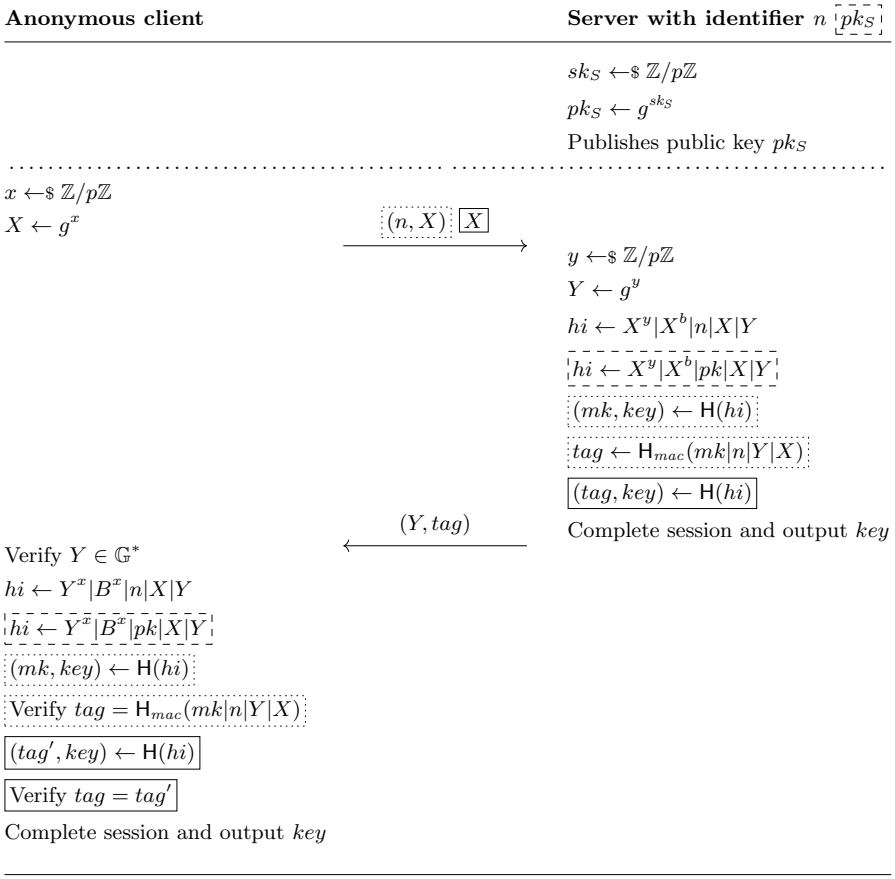


Fig. 6. The `ntor` protocol as introduced in [GSU13] with changes applied to achieve a simplified and a modified version. To get from `ntor` to the simplified version `s_ntor` we replace the dotted boxes by the solid boxes. In the modified protocol `m_ntor`, we additionally use the dashed box instead of the line above it. Additionally, we change the identifier of the server and the first message according to the dashed boxes.

the composition of two random functions[8] only by exploiting collisions or by querying the specific input directly. The latter case is ruled out in our proof, while the former occurs only with negligible probability.

– For the modified variant, the hash function H has domain $\mathbb{G} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G}$, where the third element is the server's long-term public key.

It is easy to check that the simplified and modified versions both are the instantiations of a corresponding base protocol, as described in Section 2.3, allowing us to use the results in that section to simplify our analysis.

[8] We assume the MAC is a PRF, as is standard in real-world protocols. In particular, Tor uses HMAC, which is a PRF.

3.3 Original Security Argument

The security argument for `ntor`, introduced by Goldberg *et al.* [GSU13], sketches how its security relies on the Gap Diffie-Hellman (GDH) assumption. The proof is given in the random oracle model (ROM), and its core idea is that, an adversary can distinguish the test session key from random only if it queries the random oracle (RO) on exactly the same input as the test session, which has the form

$$(\mathrm{CDH}(X, Y), \mathrm{CDH}(X, pk_S), n, X, Y).$$

Further, for sessions between honest parties, the public components of the hash input are sufficient to ensure uniqueness across sessions, except with negligible probability due to collisions of sampled values. The adversary could trivially compute the correct hash input if they revealed either the client's ephemeral secret or both the server's ephemeral and long-term secrets. The freshness condition therefore forbids these combinations of secret revelations.

Assuming an adversary $\mathcal{A}$ that distinguishes a fresh session key from random, we can construct a reduction that guesses the test session and its public values in advance and embeds a computational Diffie-Hellman (CDH) challenge (U, V). The embedding depends on which secrets $\mathcal{A}$ corrupts: if the server's ephemeral key for the test session is revealed or controlled by $\mathcal{A}$, the challenge value V is embedded in the long-term key; otherwise, it is embedded in the ephemeral key. The value U is always embedded in the tested client's ephemeral key. This yields two reductions, $\mathcal{B}_1$ and $\mathcal{B}_2$, each equipped with a DDH oracle to answer $\mathcal{A}$'s random-oracle queries consistently. A successful $\mathcal{A}$ then implies that either $\mathcal{B}_1$ or $\mathcal{B}_2$ solves CDH, contradicting the GDH assumption.

4 Game-based Proof for the Modified `ntor` Protocol

The original analysis of `ntor` in [GSU13] presents an informal sketch showing how security could be argued under the GDH assumption. However, a full game-based proof including forward secrecy and a formalization of it in the computational model are still missing. In this section, we address these omissions.

We begin by outlining the sequence of game transformations that constitute the proof, highlighting the purpose of each transition. We then examine the final two game hops in greater detail, as these are the most technically involved in the formalization in EasyCrypt. Since the final hop contains the reduction to the underlying hardness assumption which we modify and replace by the Strong CDH assumption, we first introduce the necessary notation before presenting the reduction. Finally, we combine all steps to obtain our security theorem which implies name-based security of `s_ntor` using the theorems from Section 2.3.

4.1 Proof overview

We start with the experiment parametrized by the distinguishing bit b using the restricted public-key-only oracle hPkonly with Π equal to `m_ntor` and the

random oracle H modelling the hash function $\mathsf{H} : \mathbb{G} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \to \{0,1\}^k \times \{0,1\}^k$. The sequence of game transformations presented below is applied identically to both the real and ideal experiments. The objective is to gradually transform the experiment into a game whose advantage can be bounded by a reduction to the Strong Computational Diffie–Hellman (St-CDH) assumption.

Game $\mathbf{G_0^b}$: Since the protocol is defined as the external modules `Server` and `Client` in EasyCrypt, we introduce a new starting game that inlines the protocol calls to improve readability in later hops. The game's behavior remains unchanged, and its equivalence to the security experiment is trivial.

$$\Pr\Big[\mathrm{Exp}^b(\mathsf{O}^{\mathrm{hPkonly}}_{\mathtt{m_ntor},\mathrm{H}}, \mathcal{A}) = 1\Big] = \Pr\Big[\mathrm{Exp}^b(\mathrm{G}_0, \mathcal{A}) = 1\Big]$$

Game G_0^b and all subsequent games are defined in the artifact file Games.ec, while the proofs of the individual game hops, including the corresponding reductions, are contained in Security.ec.

Game $\mathbf{G_1^b}$: In this game, we introduce the flag bad_1 to capture a sampling collision. The flag is set whenever a freshly generated public key, either ephemeral or long-term, has already been generated previously. Upon such a collision, the adversary is made to win by stopping the response to the triggering query and subsequently silence all oracle interactions. This modification resembles a halting reduction and simplifies the subsequent proof steps by eliminating the need to establish invariants in executions where bad1 has occurred. The probability of setting bad1 is bounded by the following birthday bound where $q_1 = q_{\mathrm{gen}} + q_{\mathrm{init}} + q_{\mathrm{msg1}}$, summing the number of queries made to the Gen, Init and Send_Msg1 oracle, respectively, and p is the group order of $\mathbb{G}$:

$$\begin{aligned}\Big|\Pr\Big[\mathrm{Exp}^b(\mathrm{G}_0, \mathcal{A}) = 1\Big] - \Pr\Big[\mathrm{Exp}^b(\mathrm{G}_1, \mathcal{A}) = 1\Big]\Big| &\leq \Pr\Big[\mathrm{Exp}^b(\mathrm{G}_1, \mathcal{A}) : \mathsf{bad}_1\Big] \\ &\leq \frac{q_1(q_1-1)}{2p}\end{aligned}$$

In EasyCrypt, this game hop follows standard proof techniques, combining up-to-bad reasoning with a bound on the probability of the bad event `bad1` via the `fel` (failure event lemma) tactic, both of which are well established.

Game $\mathbf{G_2^b}$: Similar to the previous game, we introduce a flag bad2, which is set whenever a newly generated public key has already appeared as part of a random oracle (RO) query by the adversary. This rules out the case in which the adversary successfully predicts a public key that is later generated by an honest party. The probability of the event bad2 is again bounded by a birthday argument, where $q_2 = q\mathrm{H} + q\mathrm{gen} + q_{\mathrm{init}} + q_{\mathrm{msg1}}$, with q_{H} denoting the number of RO queries and the remaining terms defined as in the previous game.

$$\left|\Pr\left[\mathrm{Exp}^b(\mathrm{G}_1, \mathcal{A}) = 1\right] - \Pr\left[\mathrm{Exp}^b(\mathrm{G}_2, \mathcal{A}) = 1\right]\right| \leq \frac{q_2(q_2 - 1)}{2p}$$

Game $\mathbf{G}_3^b$: In the third game, we delay sampling the session key until it is exposed to the adversary. Concretely, the RO calls used to derive the session key are removed from Send_Msg1 and Send_Msg2 and are instead performed in the Reveal and Test oracles. This hop prepares the subsequent reduction step by ensuring that protocol execution does not reveal any information about the session key, but is usually left implicit in pen-and-paper proofs.

Since both the authentication tag and the session key are derived from the same RO query, we first split this into two independent random oracles with identical domains: one for sampling tags and one for sampling session keys. To improve the readability of both the EasyCrypt formalization, we introduce an intermediate game that performs this separation before moving the session-key queries. We then prove equivalence between all three games.

$$\Pr\left[\mathrm{Exp}^b(\mathrm{G}_2, \mathcal{A}) = 1\right] = \Pr\left[\mathrm{Exp}^b(\mathrm{G}_{2.5}, \mathcal{A}) = 1\right] = \Pr\left[\mathrm{Exp}^b(\mathrm{G}_3, \mathcal{A}) = 1\right]$$

In EasyCrypt, the first part of the hop is trivially proven equivalent using the theorem on splitting the codomain of a RO which is formalized in the `PROM` theory. The second part relies on the argument on lazy and eager sampling of random oracles that is also provided as a theorem in the `PROM` theory. Specifically, Game $\mathrm{G}_{2.5}$ still uses eager sampling, whereas Game G_3 adopts the corresponding lazy-sampling implementation.

The second part of this hop also includes a proof obligation specific to EasyCrypt. Since session keys are no longer sampled before an instance accepts, they cannot be stored in the instance state. This affects the partnering operators, such as $\mathsf{untested_partner}(ID, \mathcal{I})$, which are defined over the entire instance state even though their behavior depends only on the protocol transcript. Consequently, the formalization requires proving that these operators produce identical results on states that differ only in the session keys. As these operators are invoked by multiple oracles, we establish these equivalence lemmas separately. Their implementation begins at line 1506 of the artifact file `Security.ec`.

Game $\mathbf{G}_4^b$: The fourth and final game prepares the proof for the forward secrecy argument. To this end, we introduce the flag bad_3, which captures the event that a client accepts a valid authentication tag that is not output by the RO. If this event occurs, the game again halts all oracle interactions and declares the adversary the winner. As in the previous game hops, this stopping condition eliminates the need to consider executions in which the adversary successfully guesses a valid tag. The probability of the event bad_3 is bounded by the number of accepting client instances divided by the size of the tag space $\mathcal{T}$.

$$\left|\Pr\left[\mathrm{Exp}^{b}(\mathrm{G}_3, \mathcal{A}) = 1\right] - \Pr\left[\mathrm{Exp}^{b}(\mathrm{G}_4, \mathcal{A}) = 1\right]\right| \leq \frac{q_{\mathrm{msg2}}}{|\mathcal{T}|}$$

As in the previous game hops, the proof uses up-to-bad reasoning together with the `fel` tactic.

Reduction: Before defining the reduction, we first transform the decisional problem—namely, distinguishing between the real and ideal versions of Game G_4—into a computational one. This is necessary because our security proof ultimately reduces to the computational St-CDH assumption introduced below in Section 4.3. This transformation formalizes precisely the claim that the adversary can only distinguish the games when they query the RO on exactly the same input as the test session. We capture this event using the flag bad_q, which is set either when the adversary makes such a RO query or, conversely, when the test query is issued after the corresponding input has already appeared in one of the adversary's RO queries.

$$\left|\Pr\left[\mathrm{Exp}^{0}(\mathrm{G}_4, \mathcal{A}) = 1\right] - \Pr\left[\mathrm{Exp}^{1}(\mathrm{G}_4, \mathcal{A}) = 1\right]\right| \leq \Pr\left[\mathrm{Exp}^{1}(\mathrm{G}_4, \mathcal{A}) : \mathsf{bad}_q\right]$$

We next show that the occurrence of this bad query can be leveraged to construct a reduction that solves an St-CDH challenge. Specifically, we define two reductions, $\mathcal{B}_1$ and $\mathcal{B}_2$, which embed the challenge values into the protocol messages using different strategies depending on the reveal status of the test session. The probability of the event bad_q can then be decomposed according to these reveal cases and bounded by the success probabilities of the corresponding reductions against the St-CDH assumption.

$$\Pr\left[\mathrm{Exp}^{1}(\mathrm{G}_4, \mathcal{A}) : \mathsf{bad}_q\right] \leq$$
$$\Pr\left[\mathrm{Exp}_{\text{St-CDH}}(\mathsf{O}^{\mathrm{St-CDH}}, \mathcal{B}_1) : \mathsf{win}\right] + \Pr\left[\mathrm{Exp}_{\text{St-CDH}}(\mathsf{O}^{\mathrm{St-CDH}}, \mathcal{B}_2) : \mathsf{win}\right]$$

Since the reduction constitutes the most technically involved part of the proof and introduces several challenges specific to the EasyCrypt formalization, we describe it in greater detail in the following sections, including the strategies used to address these challenges.

4.2 From Indistinguishability to the Probability of bad_q

In Game G_4, the sampling of session keys is already delayed to the test query. The only remaining difference between the real and ideal sides is that, on the ideal side, the session key is sampled twice rather than once, so the key returned by the test query does not coincide with the key that would be returned by the RO. However, this difference is only observable by the adversary in the event that they query the RO on the same input as the test session.

Using an up-to-bad argument, we show that this event, captured by bad_q, is the only way the adversary can distinguish the games. The key observation is that two honest, unpartnered sessions never query the RO on the same input. Hence, only partnered sessions possess the required information, and freshness guarantees that it remains secret. In particular, at the time of the test query, the corresponding input cannot already appear in the RO map.

In EasyCrypt, this argument corresponds to removing the guard in the test query that checks whether the handle is not already present in the RO map before sampling a fresh session key. The guard must always hold: otherwise, either (1) the session key was revealed, violating freshness; (2) the adversary queried the RO on the correct input, triggering bad_q; or (3) the session had already been tested, which is excluded by our single-test-query model.

While this reasoning is straightforward on paper, formalizing it in EasyCrypt requires an invariant relating client and server instance state to the RO map, thereby recording the information needed to prove the guard in the test query. This invariant captures temporal properties of the execution and depends on several auxiliary invariants. The most important is the injectivity of the instance state maps: two client instances producing the same first message, or two server instances producing the same second message, must be identical. We also maintain that the RO maps agree on both sides except for the test session, whose entry is never relevant, since any adversarial query requiring it would immediately trigger `badq`.

Most of these invariants are one-sided rather than relational (between the real and ideal side) and can therefore be proved independently for one game in EasyCrypt. The game-hop proof then assumes these one-sided invariants for each side and establishes only the remaining relational properties, substantially reducing the proof obligations of the hop. The one-sided invariant for Game G_4 appears in line 2146 of the artifact file `Security.ec`.

4.3 Strong CDH Assumption

Instead of the GDH assumption used in the GSU analysis, we rely on the weaker St-CDH assumption, which considers an adversary with access only to a restricted DDH oracle, as described below. More precisely, we consider a multi-instance variant with corruptions, substantially simplifying the mechanized proof in EasyCrypt. Using the standard single-instance assumption requires hybrid arguments to identify the session in which the challenge is embedded, leading to a quadratic loss in the security bound which cannot be avoided via random self-reduction since our reduction must reveal exponents. Although our approach appears to provide a tight reduction, this is only because we leave the reduction from the multi-instance assumption with corruptions to the standard single-instance assumption implicit. That reduction incurs the same quadratic tightness loss as a security proof based directly on the single-instance GDH assumption.

The St-CDH assumption used throughout this work is shown in Figure 7. In this game, the adversary $\mathcal{B}$ adaptively generates two sets of challenge values, X and Y, whose elements lie in the prime-order group $\mathbb{G}$ with generator g. They

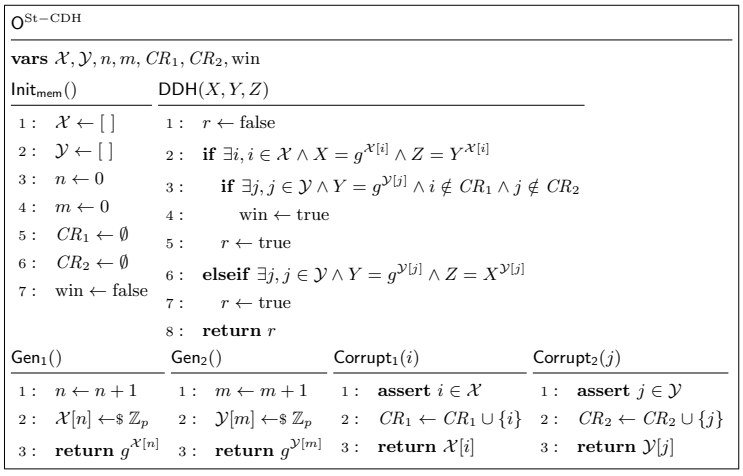


Fig. 7. Oracles for the St-CDH game.

are further given access to corruption oracles for both sets and to a restricted DDH oracle. In the strong variant, the DDH oracle accepts only queries involving challenge values that were previously generated through the corresponding oracles. The adversary succeeds if it queries the DDH oracle with uncorrupted values from X and Y together with a value Z that forms a valid Diffie–Hellman tuple. Success is recorded by the global game flag win. We define the experiment ExpSt-CDH to capture the adversary's advantage which initializes the game state and then grants $\mathcal{B}$ access to all oracle queries except Initmem.

$$\mathsf{Adv}^{\text{St-CDH}}(\mathcal{B}) := \Pr\left[\mathrm{Exp}_{\text{St-CDH}}(\mathsf{O}^{\mathrm{St-CDH}}, \mathcal{B}) : \mathsf{win}\right]$$

The implementation of this hardness assumption in EasyCrypt is given starting at line 161 of the artifact file Security.ec.

4.4 Embedding of the St-CDH Challenges

Before showing that the event bad_q implies the existence of a successful St-CDH attacker, we split the corresponding probability into the case where we embed into the long-term key and the case where we embed into the server's ephemeral key. Unlike the proof sketched by GSU, we must also account for the case where the client in the test session has no partner and the server's long-term key is corrupted only after the client has accepted, ensuring forward secrecy. In this case we still embed into the long-term key (since there is no ephemeral key). This is feasible because the adversary must have queried the RO before the client accepted in order to obtain the correct tag, implying bad_q, because we already removed the case where the adversary correctly guesses the tag.

Formally, we use the event $\mathsf{E}_{\text{p\&ltkr}}$ as the splitting condition. This event captures precisely the case where the client in the test session has a *p*artner and the corresponding server's *l*ong-*t*erm *k*ey is *r*evealed. In that case, we embed

into the server's ephemeral key. Otherwise, if the client has no partner or the long-term key is not revealed, we embed into the long-term key. These two cases form a disjoint union, since the occurrence of the test session is guaranteed, and the probability splits exactly into these two parts.

$$\Pr\big[\mathrm{Exp}^1(\mathrm{G}_4, \mathcal{A}) : \mathsf{bad}_q\big] =$$
$$\Pr\big[\mathrm{Exp}^1(\mathrm{G}_4, \mathcal{A}) : \mathsf{bad}_q \wedge \mathsf{E}_{\mathrm{p\&ltkr}}\big] + \Pr\big[\mathrm{Exp}^1(\mathrm{G}_4, \mathcal{A}) : \mathsf{bad}_q \wedge !\mathsf{E}_{\mathrm{p\&ltkr}}\big]$$

Both reductions to St-CDH are constructed similarly, and we only present the reasoning for the first setting where we embed into the server's ephemeral key. For the embedding, the reduction $\mathcal{B}_1$ calls the Gen_1 oracle of the St-CDH game whenever a client instance is executed and Gen_2 when a server receives a first message. The ephemeral key reveals for clients and servers can then be answered by querying $\mathsf{Corrupt}_1$ and $\mathsf{Corrupt}_2$, respectively. As in the proof by GSU, the reduction has access to the DDH oracle to ensure correct answers to the RO queries. Since we know that the adversary queries the RO on the same input as the test session (bad_q), and that the conditions in line 2 and 3 of the DDH oracle in Figure 7 hold for that input, this implies the win flag is set to true. We achieve the following inequality:

$$\Pr\big[\mathrm{Exp}^1(\mathrm{G}_4, \mathcal{A}) : \mathsf{bad}_q \wedge \mathsf{E}_{\mathrm{p\&ltkr}}\big] \leq \Pr\big[\mathrm{Exp}_{\text{St-CDH}}(\mathsf{O}^{\mathrm{St-CDH}}, \mathcal{B}_1) : \mathrm{win}\big]$$

The first challenge in proving this step in EasyCrypt is that its program logic does not allow reasoning directly about the outcome of an execution. Consequently, at the point where the server's ephemeral key is revealed, it is not yet possible to conclude that the query is invalid because the server's long-term key will later be revealed, together violating freshness. To address this, we first introduce a halting game that aborts whenever the test server's ephemeral key is revealed or the test client has no partner, as either case leads to an invalid execution. Otherwise, the game is identical to Game G_4, and equivalence follows by an up-to-bad argument using the splitting event as bad event.

We then use the halting game to embed the St-CDH challenge. The main difficulty is simulating RO queries: since the ephemeral secrets are stored in the St-CDH state rather than the reduction, honest protocol instances cannot access them to compute the correct DDH values. We therefore replace the unavailable RO inputs by $\perp$: for the client and any server instance receiving a message from an honest client, both g^{xy} and $g^{x \cdot sk_s}$ are omitted, while for other server instances only g^{xy} is removed. The same behavior must be reproduced for adversarial RO queries using guards on the query input.

Formalizing this simulation substantially increased the proof effort in EasyCrypt. It required many additional invariants to distinguish the relevant cases and to maintain consistency between the reduction state and the St-CDH game. In particular, these invariants ensure that triggering `badq` also sets the `win` flag, allowing the final reduction to follow by an up-to-bad argument.

4.5 Summing Up

By combining the established game hops and bounding steps, we obtain a proof of the following security theorem. Note that each bound is incurred twice (once for the real and once for the ideal side), resulting in a loss of tightness in the overall result. Our bounds can be improved on paper to be tighter, however, our focus is the simplification of the formalization.

Theorem 5. *Let $\mathcal{A}$ be an adversary against the* `m_ntor` *protocol in the restricted public-key-only model. Then there exists two adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ against the St-CDH problem, with essentially the same runtime as $\mathcal{A}$, such that*

$$\begin{aligned}
\mathsf{Adv}^{\texttt{m_ntor}}(\mathcal{A}) &:= \Big|\Pr\Big[\mathrm{Exp}^0(\mathsf{O}^{hPkonly}_{\texttt{m_ntor},\mathrm{H}}, \mathcal{A}) = 1\Big] - \Pr\Big[\mathrm{Exp}^1(\mathsf{O}^{hPkonly}_{\texttt{m_ntor},\mathrm{H}}, \mathcal{A}) = 1\Big]\Big| \\
&\leq \Pr\big[\mathrm{Exp}_{\text{St-CDH}}(\mathsf{O}^{\mathrm{St-CDH}}, \mathcal{B}_1) : \mathrm{win}\big] \\
&\quad + \Pr\big[\mathrm{Exp}_{\text{St-CDH}}(\mathsf{O}^{\mathrm{St-CDH}}, \mathcal{B}_2) : \mathrm{win}\big] + \frac{2q_{\mathrm{msg2}}}{|\mathcal{T}|} \\
&\quad + \frac{(q_{\mathrm{H}} + q_{\mathrm{gen}} + q_{\mathrm{init}} + q_{\mathrm{msg1}})(q_{\mathrm{H}} + q_{\mathrm{gen}} + q_{\mathrm{init}} + q_{\mathrm{msg1}} - 1)}{p} \\
&\quad + \frac{(q_{gen} + q_{init} + q_{msg1})(q_{gen} + q_{init} + q_{msg1} - 1)}{p},
\end{aligned}$$

where p is the order of the group, q_* the number of queries to the respective oracle and $\mathcal{T}$ denotes the tag space.

5 Discussion

We conclude the paper with a final discussion on insights and a short comparison to other observations on protocol proof complexity in EasyCrypt. While many of these observations are specific to the tool and may inform the development of future protocol proofs in EasyCrypt, some also provide guidance for protocol design and security analyses more generally.

Module tweak feature: The tool introduced in this work for implementing game transformations and reductions by explicitly specifying changes relative to previous games brings the formal development process closer to the game-based proof methodology used on paper. It also improves the maintainability and stability of the EasyCrypt proof, as changes to earlier games, such as updates to the security model, can automatically propagate to subsequent games.

Adapting assumptions: A further important lesson is the benefit of selecting assumptions that closely align with the formalization. In our case, replacing a single-instance with a multi-instance assumption avoids needing the test session-guessing argument directly in the security proof. Specifically, we identified an alternative standard assumption that provides equivalent security guarantees while simplifying the formal proof.

Removing names: As discussed previously, we argue that removing names from the protocol and treating the binding between names and public keys as a separate concern leads to a cleaner design and a simpler formalization. For future UAKE protocol proofs, our proof can serve as a blueprint for handling this issue in EasyCrypt. Alternatively, avoiding names altogether may be considered as a protocol design choice that reduces complexity in formal analyses.

Halting reductions: There is a significant distinction between strong aborts, which halt all oracle interactions, and weak aborts, which only suppress the query that caused the failure. Strong aborts and halting reductions not only eliminate the need to prove invariants after the failure event but also play an important role in reduction proofs. In particular, the program logic of EasyCrypt does not allow direct reasoning about the final outcome of an execution. This becomes challenging in the reduction step where the simulator's embedding strategy depends on future adversarial behavior, such as whether the adversary reveals the test session. Using halting reductions enables us to transfer reasoning about the execution outcome into the execution itself.

Comparison to other insights: The only work on protocol security in EasyCrypt that also discusses lessons learned is the concurrent work by Barbosa *et al.* [BDL+26]. They also employ standard techniques such as eliminating collisions in sampling and emphasize separating game hops and invariants where possible, e.g., by proving one-sided invariants independently. Their proof, however, differs substantially from ours in both security model and complexity. Barbosa *et al.* consider only session-key reveals, as their construction is insecure under stronger corruption models, and therefore do not establish forward secrecy. Moreover, their security relies on decisional assumptions for the underlying primitives. In contrast, proving the security of `ntor` requires reducing an indistinguishability notion to a computational hardness assumption. Moreover, this reduction requires reasoning about the trace—the complete sequence of interactions between the adversary and experiment in a single game execution—which, as Barbosa *et al.* observe, is particularly challenging in EasyCrypt.

Acknowledgements We would especially like to thank our shepherd their very detailed feedback, as well as the insightful discussions that followed. These greatly benefited the final presentation of this paper. F. Dupressoir's work was supported, in part, by Innovate UK through project 10065634 (SCHEME: Safety Critical Harsh Environment Micro-processing Evolution). C. Low is fully funded by EPSRC Doctoral Training Partnership 2765784.

References

BBK17. Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candi-

date. In *2017 IEEE Symposium on Security and Privacy*, pages 483–502. IEEE Computer Society Press, May 2017.

BCJ+24. Chris Brzuska, Cas Cremers, Håkon Jacobsen, Douglas Stebila, and Bogdan Warinschi. Falsifiability, composability, and comparability of game-based security models for key exchange protocols. Cryptology ePrint Archive, Report 2024/1215, 2024.

BCLS15. Gilles Barthe, Juan Manuel Crespo, Yassine Lakhnech, and Benedikt Schmidt. Mind the gap: Modular machine-checked proofs of one-round key exchange protocols. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 689–718. Springer, Berlin, Heidelberg, April 2015.

BDG+12. Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. *International School on Foundations of Security Analysis and Design*, pages 146–166, 2012.

BDH+18. David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1383–1396, 2018.

BDL+26. Manuel Barbosa, François Dupressoir, Cameron Low, Charlotte Mylog, Sabine Oechsner, and Théo Vignon. Principles for prhl proofs of cryptographic protocols: How to convince easycrypt that your protocol is secure. *Cryptology ePrint Archive*, 2026.

BFG+22. Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. Post-quantum asynchronous deniable key exchange and the Signal handshake. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *PKC 2022, Part II*, volume 13178 of *LNCS*, pages 3–34. Springer, Cham, March 2022.

BFWW11. Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 51–62. ACM Press, October 2011.

BGHZB11. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*, pages 71–90. Springer, 2011.

Bla08. Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

BMS19. Colin Boyd, Anish Mathuria, and Douglas Stebila. Key agreement protocols. In *Protocols for Authentication and Key Establishment*, pages 195–240. Springer, 2019.

CB12. David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *2012 Seventh International Conference on Availability, Reliability and Security*, pages 65–74. IEEE, 2012.

CCD+16. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016.

CHH+17. Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla Van Der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1773–1788, 2017.

CHSvdM16. Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy*, pages 470–485. IEEE Computer Society Press, May 2016.

CHSW22. Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. A tale of two models: Formal verification of KEMTLS via Tamarin. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part III*, volume 13556 of *LNCS*, pages 63–83. Springer, Cham, September 2022.

DFGS15. Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the tls 1.3 handshake protocol candidates. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1197–1210, 2015.

EC. Easycrypt. `https://easycrypt.info`.

FG14. Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of google's quic protocol. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1193–1204, 2014.

FG25. Rune Fiedler and Felix Günther. Security analysis of Signal's PQXDH handshake. In *PKC 2025, Part II*, LNCS, pages 137–169. Springer, Cham, May 2025.

FGPZ25. Rune Fiedler, Felix Günther, Jiaxin Pan, and Runzhi Zeng. XHMQV: Better efficiency and stronger security for signal's initial handshake based on HMQV. In *CRYPTO 2025, Part VIII*, LNCS, pages 109–140. Springer, Cham, August 2025.

GSU13. Ian Goldberg, Douglas Stebila, and Berkant Ustaoglu. Anonymity and one-way authentication in key exchange protocols. *DCC*, 67(2):245–269, 2013.

KPW13. Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 429–448. Springer, Berlin, Heidelberg, August 2013.

LLM06. Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. Cryptology ePrint Archive, Report 2006/073, 2006.

LLM07. Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007*, volume 4784 of *LNCS*, pages 1–16. Springer, Berlin, Heidelberg, November 2007.

PL. Protocol ladder. `https://github.com/proof-ladders/protocol-ladder.git`.

Ste25. Igors Stepanovs. The power of halting in security games. ProTeCS workshop, 2025. `https://igors.org/talks/2025-abort-false-in-security-games.pdf`.

Tora. Tor project. `https://www.torproject.org`.

Torb. Tor project, proposal 216. `https://spec.torproject.org/proposals/216-ntor-handshake.html`.