A ladder of cryptographic problems to illustrate formal verification

Cas Cremers François Dupressoir Charlie Jacomme Aurora Naska

February 25, 2025

This document and its companion repository[1] provide a set of cryptographic problems dedicated to protocols. It aims to help understand how the multiple approaches fit and compare within the formal verification landscape. This document specifies the problems and expected format of solutions, while the repository aims to collect the solutions and provide nice summary table.

This initiative was pushed forward for the HACS workshop. See here [2] for acknowledgments and how to cite this work.

1 Protocol problem set

This document outlines several cryptographic problems that we recommend for the crypto ladder. The proposed protocols are listed in roughly increasing order of proof difficulty within the crypto world:

- **Basic-Hash** (Section 2.1): a RFID tag/reader access protocol, meant to provide unlinkability (a strong notion of anonymity) between the tags. This is the base pattern for electronic passport authentication.
- **Signed DH** (Section 2.2): a simple key-exchange providing secrecy and authentication, which is the historic pattern for SSH/TLS.
- **Signed KEM** (Section 2.3): a bilateral authenticated variant relying on KEMs, illustrating subtle attacks against key-exchanges.
- **Signed DH+KEM** (Section 2.4): a hybrid variant with both KEM and DH following the pattern for the ongoing hybridization of SSH/TLS.
- **Simplified NTOR** protocol (Section 2.5): a DH key-exchange using long-term DH keys for authentication instead of signatures, which is the base pattern for Wireguard and PQXDH.
- **Simplified ACME** protocol (Section 2.6): a simplified signature-based challenge-response protocol to allow a website owner to prove ownership of their domain[1].

The rationale behind this set is the following:

- provide very simple examples, that increase in difficulty;
- show a variety of security properties (secrecy, authentication, privacy) and threat models;
- include examples linked to widely deployed real-world applications and relevant real-world attacks;
- showcase the scope and cutting edge between different methodologies.

Multi-party protocols and e-voting protocols are not included in this set, as verification for those problems is for the moment scarce.

We note that advanced users can use the tools mentioned for much more intricate problems – sometimes orders of magnitude more complex – than the ones shown here. For the purpose of presenting problems meaningfully within this document, we have chosen relatively simple problems that can serve to highlight the core elements.

[1] `https://datatracker.ietf.org/doc/html/rfc8555`

1.1 Notation

In this document, secret keys and key pairs are denoted by using the subscripts sk/pk to distinguish between secret/public parts. We rely on usual algorithms for KEMs, signatures and DH. We use a generic hash function Hash without a fixed arity: modelers may use the most canonical way to represent such computations in their tools.

1.2 Modeling protocols

Protocols are asynchronous programs executed between several parties over an often untrusted network. Given the wide range of potential modeling choices for both execution models and security properties, we do not commit to any single option in this presentation. Each problem will be specified at a high level by giving a message sequence chart of an expected run between two (or more) honest agents. An agent can be characterized by a role and some long-term material. Typically, for a key-exchange setting we can denote by $S(s_{sk}, s_{pk})$ a server owning the signing key s_{sk} and $C(s_{pk})$ a client that already knows the corresponding public key, and thus intends to communicate to this specific server.

Each problem may be "solved" in a variety of ways based on many different modeling choices. We detail some of the main those choices below.

Execution scenario The simplest execution scenario for each protocol involves a single party for each role, with each party participating in only one session. However, we typically want to consider scenarios closer to real world executions, involving an unbounded number of sessions and potentially parties. We categorize some of these modeling choices as follows:

- **Number of sessions:** ***bounded vs unbounded*** - Scenarios with a bounded number of sessions corresponds to some $C(s_{pk})$ interacting with at most n instances –a bounded fixed number of times– of the corresponding partner $S(s_{sk}, s_{pk})$. However, we may want to allow the two agents, C and S, to execute between themselves the protocol an unbounded number of time, ∞, corresponding to the real-world setting where we might not know how many instances are running.
- **Number of agents:** ***bounded vs unbounded*** - In basic scenarios, we may only consider a bounded number of possible agents, and typically consider that there is a single server. However, realistically, we may also need to consider the scenario where there is an unbounded (i.e., arbitrarily large) number of possible agents/identities for each role. This means that there typically would be a set of possible public keys $\{s^i_{pk}\}$ for all the possible identities.

Threat model We consider that the attacker can to some extent control network communications, with their capabilities varying as follows:

- **Attacker:** ***passive vs active*** - A passive attacker can intercept and read all messages sent over the network, but it cannot tamper with them. An active attacker can modify and drop any message sent over the network and additionally inject their own.
- **Compromise:** ***none vs ephemeral and/or long-term material*** - The attacker can perform a targeted attack on some users, and partially or fully compromise their cryptographic material. In the basic scenario, the attacker capabilities are very limited and unable to execute any such compromise. The attacker may then be able to corrupt an agent's long-term materials, typically the s_{sk} of a server (note that in some key exchange scenarios, C may not possess any long-term secret material, because they do not wish to authenticate themselves to the server). Additionally, the attacker can compromise the ephemeral material of a specific session, effectively exposing the ephemeral secret keys for that session. In the more complex scenario, the attacker is able to compromise both.

Modeling of primitives For protocol analysis, primitives are most of the time abstracted away as abstract function calls or primitive assumptions.

- **Approach: *computational vs symbolic*** - In the computational approach, hardness assumptions are made over the primitives, e.g., IND-CCA for encryption, and the attacker is restricted by what it cannot do w.r.t. to the primitives. In the symbolic approach, primitives are idealized, and typically, an encryption perfectly hides its content and the attacker cannot learn its plaintext unless it know the corresponding decryption keys [2].
- **Primitive characterization** - Each primitive might be modeled in a variety of way, with weaker or stronger assumptions over it. In the computational approach, we may model the encryption as IND-CPA or IND-CCA1 or IND-CCA2. The symbolic approach may consider a signature under different security assumptions: perfect, leaking the signed message, suffering from non-exclusive ownership attacks, etc.

Security properties Finally, security properties can themselves be expressed in a variety of way, without always a clear consensus in the community. Rather than commit to a specific notion, we detail below some classes of attacks that are relevant in our setting.

Some very generic properties are:

- **Protection from Replay Attacks** - A message sent by a party *cannot* be replayed successfully several times to another party to establish a connection. (Note that this is only meaningful in a multiple session scenario.)
- **Data agreement** - When an agent A completes a session, apparently with honest peer B, both parties are expected to agree on a set of data, such as keys, transcripts, and identities.

In the problem set, we will have a particular focus on key-exchanges, which have a variety of properties/attacks:

- **Authentication** - Whenever a party expects to be talking to a given honest agent, it is indeed the case. Authentication might be *unilateral* (at the end of the key-exchange the client knows it is talking to the server owning s_{pk} but the server has no guarantee about the client), or *bilateral* (the client also owns some long-term key, and the server obtains guarantees about the identity of the client).

  In addition, authentication might be *injective or non-injective*, where it is injective if for each session of a client believing to be talking with some server, there is a single and distinct corresponding session on the server. If the authentication is non-injective, a single server session may validate authentication for several sessions of the client, which can typically happen in case of replay attacks.

  Moreover, authentication can also require agreement on data, or on complete transcripts (e.g., variants of "matching conversations" or "synchronization").

- **Unknown Key-Share (UKS) attacks** - This class of attacks occurs when two honest agents derive the same key, but did not expect to be sharing a session key, typically because they do not have the same expectations on who the respective peer is.

  UKS attacks come in several flavors. Traditionally, a UKS attack targets an honest party trying to communicate with an honest peer. Concrete attack scenarios typically involve honest agents A and B and a dishonest agent C, where A aims to talk to B, and the attacker manages to get B to compute the same key as A while B thinks they are talking to C. Such UKS attacks can be viewed as a violation of (implicit) agreement on the identities involved in the communication. In the context of a larger protocol, UKS may cause mis-attribution of received messages or sending messages to the wrong agent.

[2] In more details, equations define their only possible behaviors

A second type of UKS attack is one in which two honest agents compute the same session key, but where each of them is trying to communicate with a dishonest peer. This is not a violation of classical agreement or authentication properties, since they only provide guarantees when trying to communicate with honest peers. This variant of a UKS attack can be avoided.[3]

- **Secrecy** - The key derived at the end of a session of a honest agent trying to communicate with an honest agent should be secret. A stronger variant is **Forward Secrecy (FS)**, where even if at some point the long-term material of an agent is compromised, this does not affect the secrecy of all previously completed exchanges.
- **Session independence** - For a given agent, compromising the ephemeral material of all sessions except for one should not affect the security of the uncompromised session.
- **Key-Compromise Impersonation (KCI)** - in a scenario with a bilateral authenticated key exchange, when we compromise the long-term material of a given agent C, the attacker can typically impersonate C to some honest server S. However, we additionally want to guarantee that this does not later allow the adversary to impersonate as an honest part (e.g., S) towards C. A typical scenario where KCI can occur is when two parties share a symmetric pre-shared (PSK) key used for authentication: leaking this long-term material on the side of C can allow the adversary to impersonate as S to C in the future. In contrast, for protocols based on long-term signatures, this is typically not possible.

1.3 Problem solutions

We invite tool developers and users to contribute to this project. For each tool we thus encourage developers to have a dedicated public repository containing in whatever format they wish, some full or partial solutions, along with tutorials or any other desired material.

In addition, we hope to aggregate in the companion repository [1] the full solutions file for each problem and tool, to enable for a quick browsing of the files, and to provide for each problem a table summarizing all the existing contributions and giving some comparison points. To be able to build such a table, we ask that each solution for a particular problem comes with a standardized description summarizing the features of the model and analysis. See the README of the repository for the template description, as well as the tables.

2 Protocol problems

2.1 Basic-Hash

The protocol is illustrated in Fig. 1.

Description For a set TAGS of valid identities, a RFID reader has a database of valid secret keys $\{t_{sk}\}_{T \in \text{TAGS}}$. A RFID tag with identity T and key t_{sk} authenticates to the reader by sampling a fresh challenge n, and sending the pair $(n, \text{Hash}(n, t_{sk}))$ to the reader, which looks into its database to see if the hash can be mapped to a valid identity and then answers true or false.

Properties This protocol should provide two guarantees against an active attacker:

- *unilateral non-injective authentication* - if the reader accepts some message as coming from some tag, the corresponding tag did send this message at some point in the past.
- *unlinkability* - it should be impossible to decide whether two sessions of the protocol correspond to the same tag or not. That is, it should be impossible to distinguish a scenario where there is a single tag executing n times the protocol in sequence with a scenario where n distinct tags all execute the protocol each a single time.

[3] However, there is always a similar attack possible for an active network attacker, who can establish and learn two different keys with each of the parties, allowing it to actively relay traffic if required.

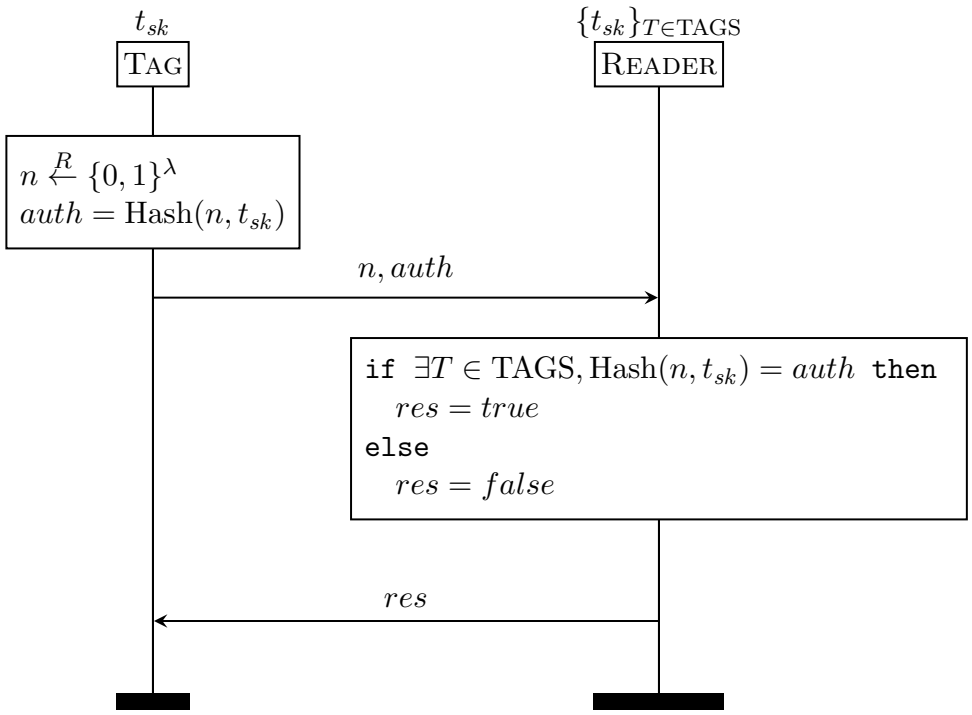

Figure 1: Basic Hash

Scenarios The core scenario we wish to target is one reader and many tags with many sessions for each tag. Multiple readers do not really increase the relevance of the model, as readers never process anything from them. Compromises of material is not of particular interest here.

Bonus Show that *injective authentication* does not hold due to a replay attack. The corresponding attack trace is illustrated in Appendix A, Fig. 7.

2.2 Signed DH key exchange

The protocol is illustrated in Fig. 2.

Description The server has a long-term signature keypair s_{sk}, s_{pk}. This key is used to authenticate the server, while the client and the server exchange fresh DH values to derive a shared secret key.

Properties This protocol should provide two guarantees against an active attacker:

- *unilateral client-side injective authentication* - if a client derives a key with some parameters (the set of public keys), a corresponding server session derived the same key with the same parameters.
- *forward secrecy* - nobody except the two matching sessions can derive the key of this session, even if the long-term key of the server is compromised in the future.

Scenarios Scenarios as strong as possible are relevant for this problem. We hope to see an unbounded number of distinct servers and clients, with an unbounded number of sessions. Compromise of long-term keys is needed to enable verification of Forward Secrecy.

2.3 Signed KEM key exchange

The protocol is illustrated in Fig. 3.

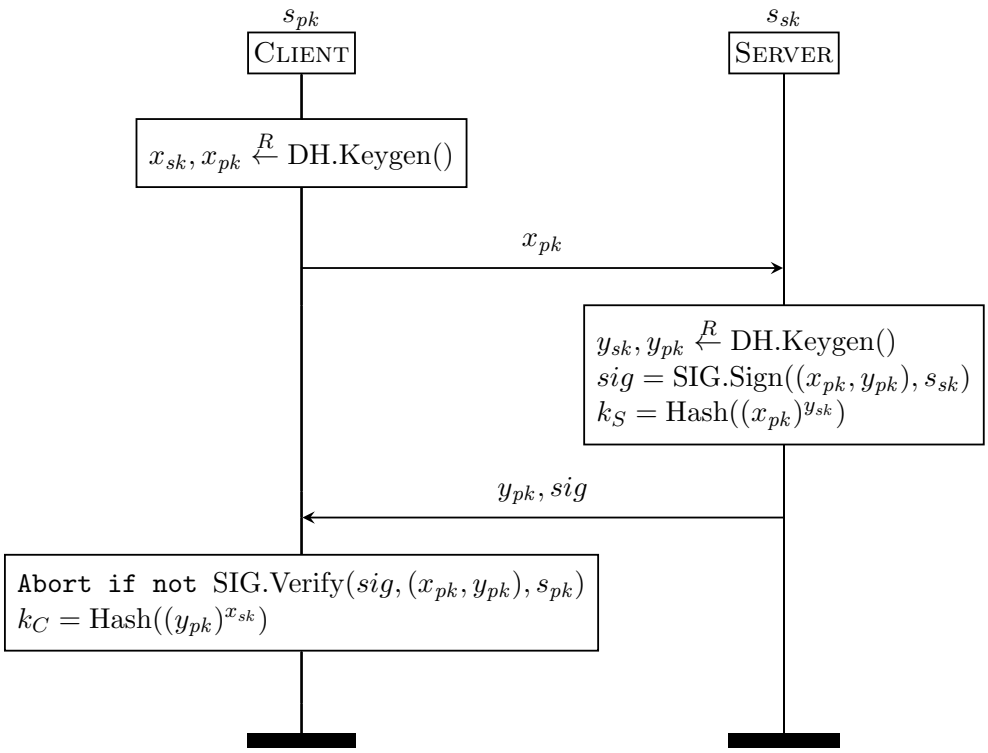

Figure 2: Signed Diffie-Hellman key exchange

Description The client has a long-term signature keypair c_{sk}, c_{pk}. This key is used to authenticate the client, while the client and the server exchange fresh KEM values to derive a shared secret key.

Properties This protocol should provide two guarantees against an active machine-in-the-middle attacker:

- *implicit authentication* - if both client and server derive the same key with the same parameters (the set of public keys), and they are talking to honest partners, then they are indeed talking to one another.
- *forward secrecy* - nobody except the two matching sessions can derive the key of this session, even if the long-term key of the client is compromised in the future. Note that compromising the server keys breaks the forward secrecy of half of the key.

Scenario The target scenario is unbounded parties and unbounded session, with compromise of long-term keys. We want to consider in this problem: a KEM that is just IND-CCA, i.e., the KEM may be implemented using an asymmetric encryption function KEM.AEnc, and with:

$$\text{KEM.Encaps}(x_{pk}) := ss \xleftarrow{R} \{0,1\}^{\lambda}; ct \xleftarrow{R} \text{KEM.AEnc}(ss, x_{pk}); \text{return } (ct, ss)$$

Bonus

1. If the attacker can register their own long-term keypair, this protocol suffers from an Unknown Key-Share (UKS) attack. The corresponding attack trace is illustrated in Appendix A, Fig. 8.
2. Under compromise of ephemeral keys x_{sk}, we should on a good protocol still be able to prove that whenever a client session and a server session derive the same secret key, then if the corresponding client session x_{sk} was not compromised, the key is fully secret. This corresponds to *session independence*, where allowing to compromise the material of other sessions does not impact the

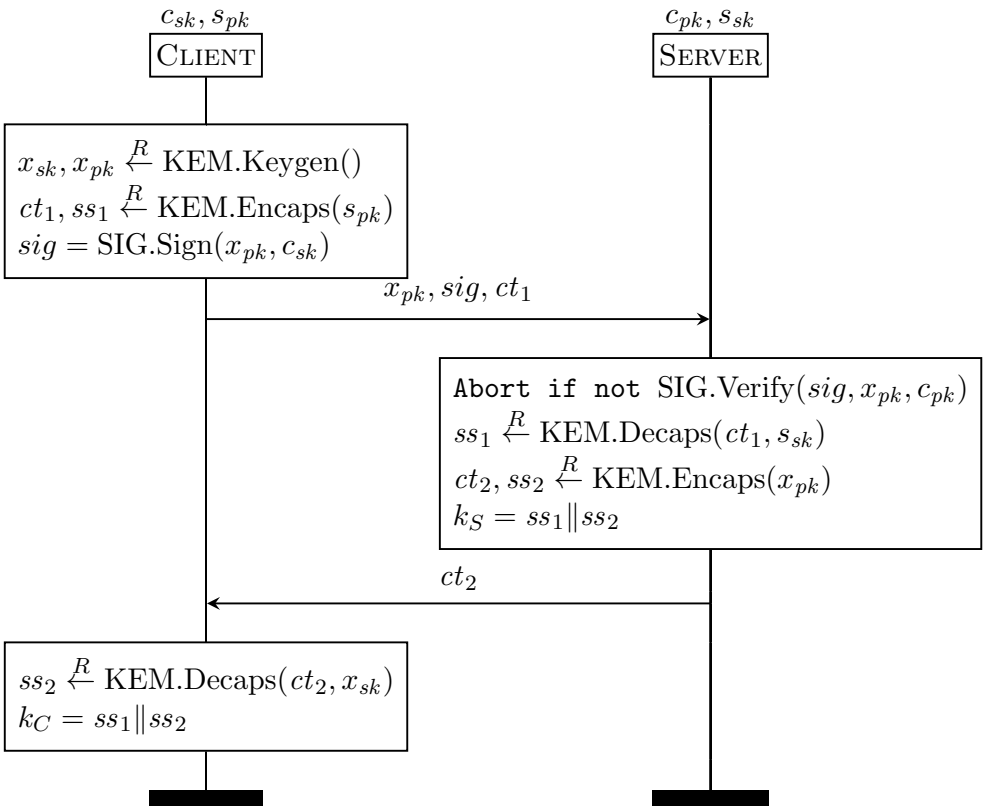

Figure 3: Signed KEM key exchange

security of uncompromised sessions. The corresponding attack trace is illustrated in Appendix A, Fig. 9.

2.4 Signed DH+KEM key exchange

The protocol is illustrated in Fig. 4.

Description The server has a long-term signature keypair s_{sk}, s_{pk}. This key is used to authenticate the server, while the client and the server exchange fresh DH and KEM values to derive a shared secret key.

Properties This protocol should provide two guarantees against an active attacker:

- *unilateral injective client-side authentication* - if a client derives a key with some parameters (the set of public keys), a corresponding server session derived the same key with the same parameters.
- *forward secrecy* - nobody except the two matching sessions can derive the key of this session, even if the long-term key of the server is compromised in the future.

Scenarios Scenarios as strong as possible are relevant for this problem. We hope to see an unbounded number of distinct servers and clients, with an unbounded number of sessions. Compromise of long-term keys is needed to enable verification of Forward Secrecy.

Bonus Provide proofs for the secrecy where the signature is secure, but only either the KEM or the DH part is assumed secure, while the other primitive is completely broken.

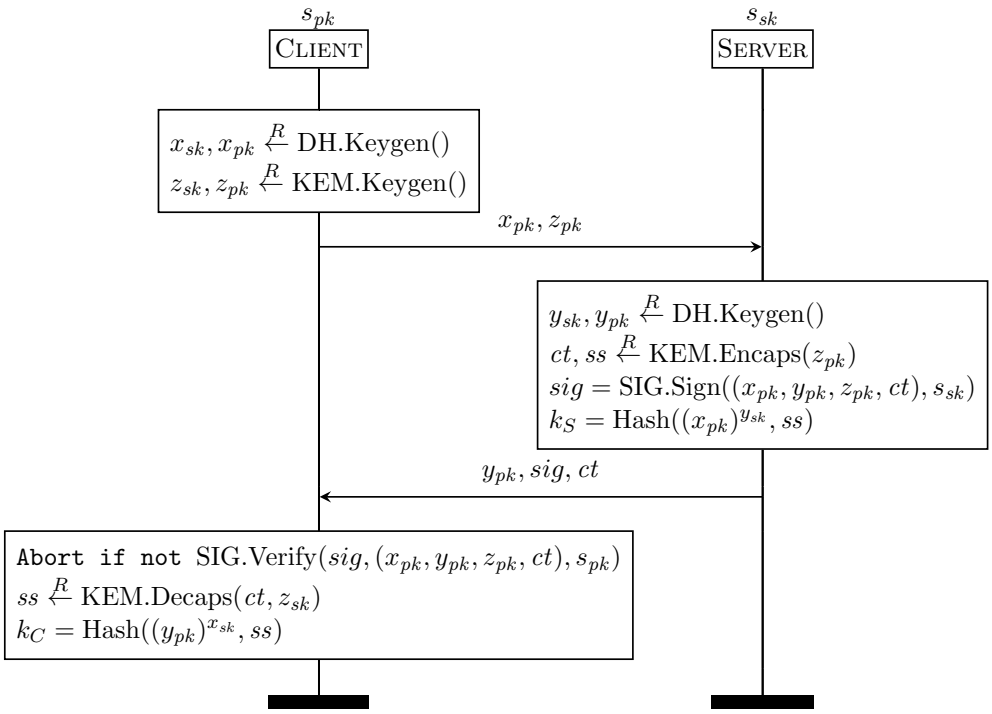

Figure 4: Hybridization of a signed Diffie-Hellman key exchange

2.5 Simplified NTOR

The protocol is illustrated in Fig. 5.

Description The server has a long-term DH keypair s_{sk}, s_{pk}. This key is used to authenticate the server, while the client and the server exchange fresh DH values to derive a shared secret key.

Properties This protocol should provide two guarantees against an active machine-in-the-middle attacker:

- *unilateral injective authentication* - if a client derives a key with some parameters (the set of public keys), a corresponding server session derived the same key with the same parameters.
- *secrecy* - nobody except the two matching sessions can derive the key. The key might be proven to be indistinguishable from a random value.

Scenarios Scenarios as strong as possible are relevant for this problem. We hope to see an unbounded number of distinct servers and clients, with an unbounded number of sessions. Compromise of long-term keys is needed to enable verification of Forward Secrecy.

2.6 Simplified ACME Protocol

The protocol is illustrated in Fig. 6.

Description The website owner has a long-term keypair o_{sk}, o_{pk} of their website certificate. They want to claim ownership of their website and distribute the new certificate to the DNS Sever and Let's Encrypt (LE) server. To this end, the owner starts a challenge response protocol with the LE server, signs a token using the certificate key, and sends the certificate to the DNS Server. The LE server retrieves the certificate of the website from the DNS server and upon successful verification of

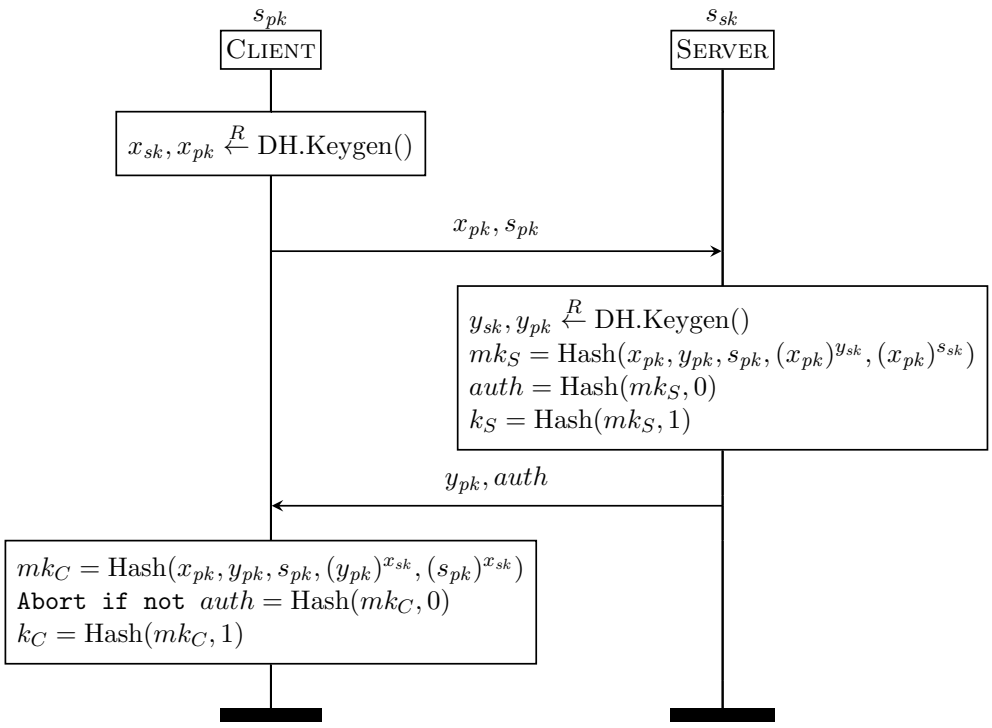

Figure 5: Simplified NTOR

signature it concludes that the owner controls the private key of the certificate and the DNS record for the website address.

Properties This protocol should provide the guarantee that if a certificate is accepted, then the owner owns the private key of the certificate and registered website.

Scenarios Scenarios as strong as possible are relevant for this problem. We hope to see an unbounded number of distinct servers and clients, with an unbounded number of sessions.

2.7 Showcase

To give an idea of the specific capabilities of their tools, we finally invite for this problem set submissions of interesting case-studies verified by the individual tools. Those solutions do not need to come with the model files nor follow the full template, as comparison will be less relevant. This can typically be based on recently published verification work. The format should simply be the name of the case-study, a short description with the salient point, and eventually a link to a paper or models.

A Attack traces

References

[1] Formal Verification Ladder Protocol repository, 2025. `https://github.com/charlie-j/fm-crypto-lib`.

[2] Formal Verification Ladder Protocol repository acknowledgements, 2025. `https://github.com/charlie-j/fm-crypto-lib/blob/main/README.md#acknowledgments`.

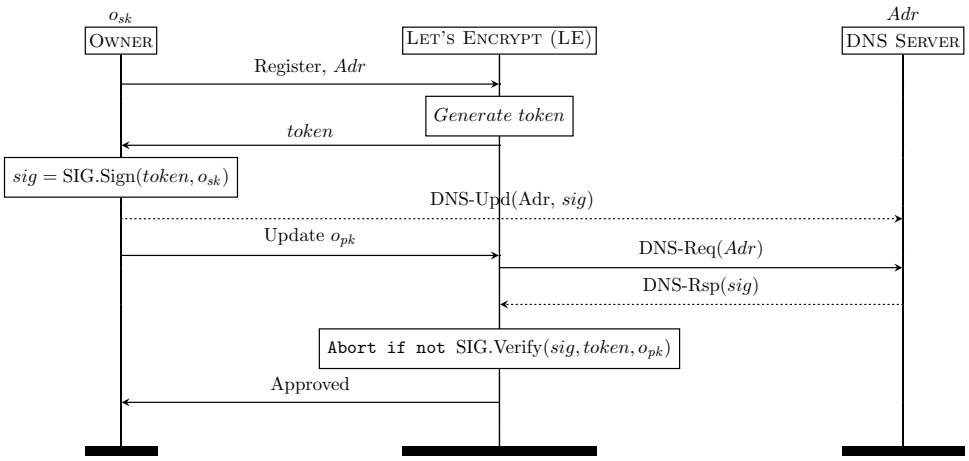

Figure 6: Simplified ACME Protocol. The dotted arrows indicate an authenticated channel.

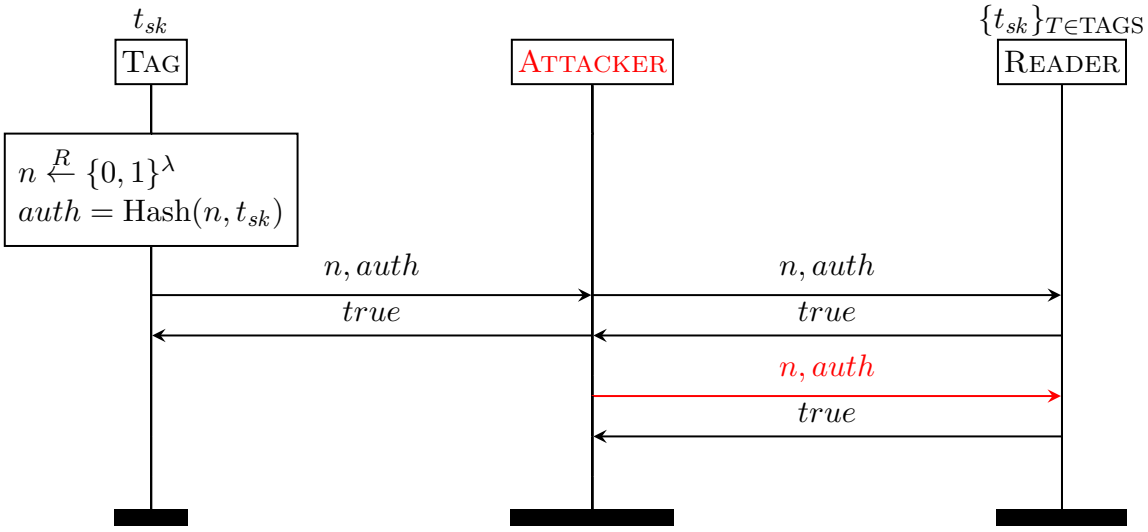

Figure 7: Basic Hash - Basic replay attack breaking *injective authentication*.

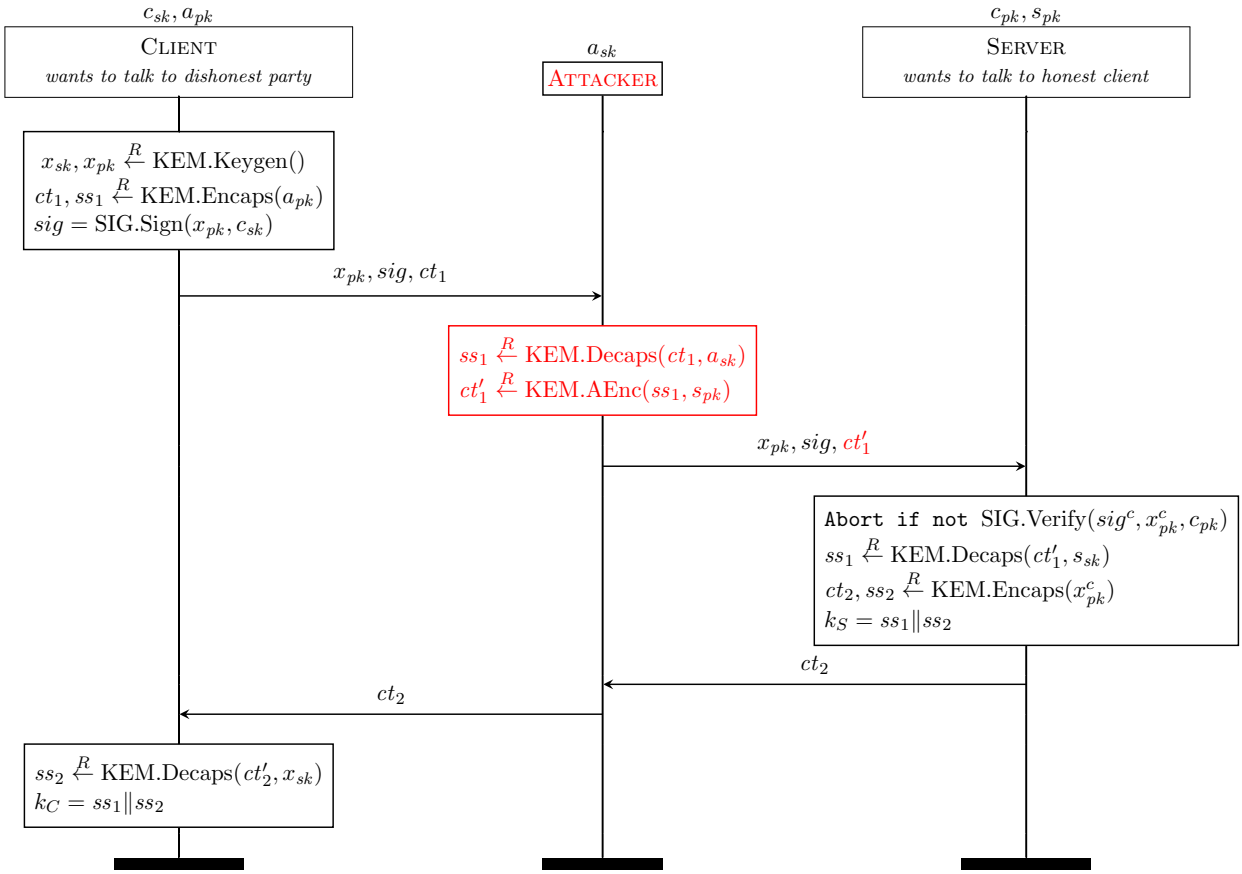

Figure 8: Signed KEM key exchange - *Unknown Key-Share (UKS)* attack through re-encapsulation attack. The honest server, trying to talk to the honest client, derives the same key as the client, who does not believe they are talking to each other.

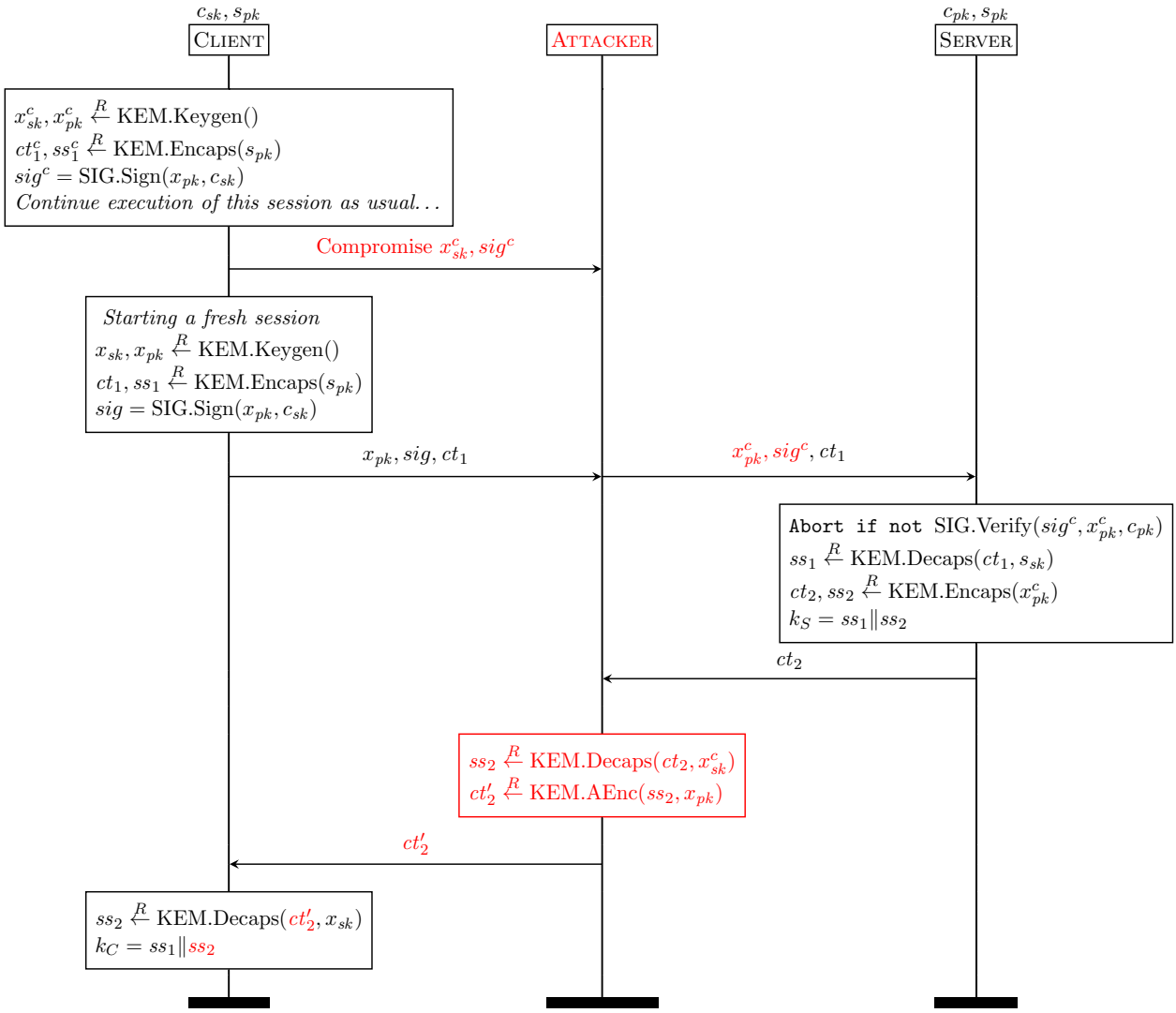

Figure 9: Signed KEM key exchange - Reencapsulation attack against session independence. The attacker learns half of the key of a fresh session by compromising material from an independent session.

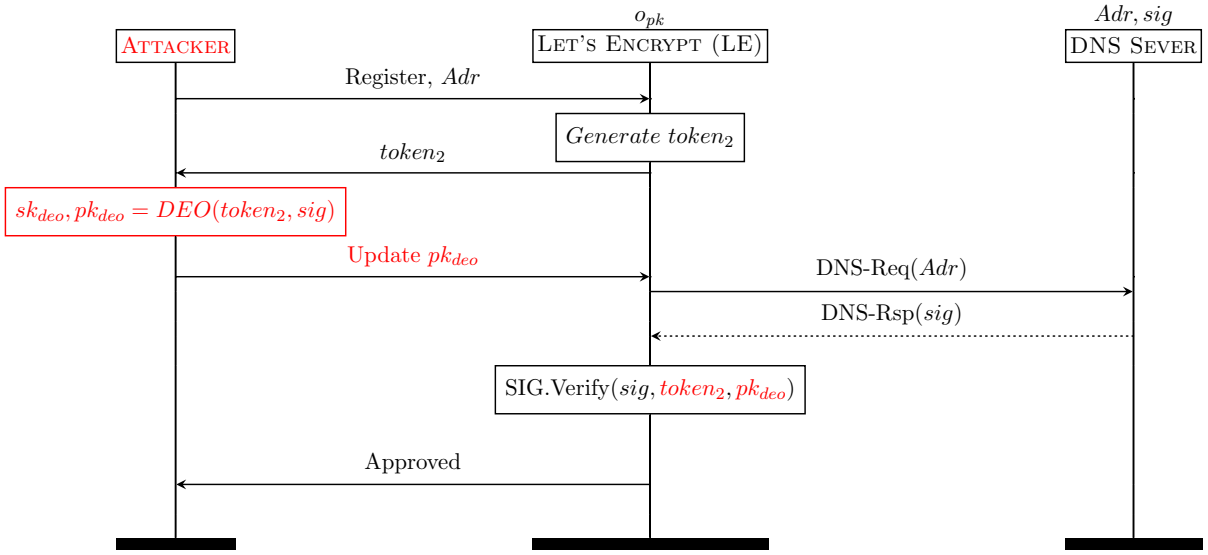

Figure 10: Attack on ACME Protocol. The owner of Adr has completed an execution of ACME and placed the public key o_{pk} on the LE server, and signature sig for address Adr on the DNS server. The attacker starts a new instance of the protocol to claim ownership of the victim's website, and substitutes the public key using a DEO (*Destructive Exclusive Ownership*) attack on the signature. I.e, The attacker can generate a public key pk_{deo} for signature sig, such that the signature verifies under another plaintext.