A Comprehensive Formal Security Analysis of OPC UA[*]

Vincent Diemunsch
ANSSI & Université de Lorraine, CNRS, Inria, LORIA, France

Lucca Hirschi
Université de Lorraine, CNRS, Inria, LORIA, France

Steve Kremer
Université de Lorraine, CNRS, Inria, LORIA, France

v1.0[1] — January 31, 2025

Abstract

OPC UA is a standardized Industrial Control System (ICS) protocol, deployed in critical infrastructures, that aims to ensure security. The forthcoming version 1.05 includes major changes in the underlying cryptographic design, including a Diffie-Hellmann based key exchange, as opposed to the previous RSA based version. Version 1.05 is supposed to offer stronger security, including Perfect Forward Secrecy (PFS).

We perform a formal security analysis of the security protocols specified in OPC UA v1.05 and v1.04, for the RSA-based and the new DH-based mode, using the state-of-the-art symbolic protocol verifier ProVerif. Compared to previous studies, our model is much more comprehensive, including the new protocol version, combination of the different sub-protocols for establishing secure channels, sessions and their management, covering a large range of possible configurations. This results in one of the largest models ever studied in ProVerif raising many challenges related to its verification mainly due to the complexity of the state machine. We discuss how we mitigated this complexity to obtain meaningful analysis results. Our analysis uncovered several new vulnerabilities, that have been reported to and acknowledged by the OPC Foundation. We designed and proposed provably secure fixes, most of which are included in the upcoming version of the standard.

Contents

1 Introduction **3**

2 OPC UA Protocol **4**
2.1 Overview 4
2.2 Protocol Description 5
2.2.1 UA Secure Conversation 5
2.2.2 Sessions 6
2.3 Security Goals 7
2.3.1 Threat Model 7
2.3.2 Security Properties 8

3 OPC UA Formal Model **8**
3.1 Proverif Background 8
3.2 Modeling in the Applied Pi Calculus 9
3.3 Formal Security Properties 10

[*]An extended abstract of this paper appears at USENIX Security 2025; this is the full version.
[1]A list of changes since the initial version can be found in Appendix E.

4 Proof Methodology **11**
4.1 Protocol Configurations and Modularity . . . 11
4.2 Systematically Exploring Configurations . . . 11
4.3 Advanced proof techniques . . . 12
4.3.1 Advanced features . . . 12
4.3.2 A novel proof methodology . . . 12

5 Analysis Results **13**
5.1 Race Condition for User Contexts . . . 14
5.2 Client Impersonation in ECC . . . 14
5.3 KCI: User Impersonation . . . 17
5.4 Session Hijack by Reopening or Switching . . . 18
5.5 KCI: Session and User Confusion . . . 19
5.6 Other Findings . . . 19
5.7 Lessons Learned . . . 20

6 Related Work **20**

7 Conclusion **21**

8 Ethics Considerations and Compliance with the Open Science Policy **21**
8.1 Ethics Considerations . . . 21
8.2 Open Science . . . 21

A Weakened Properties **24**

B OPC UA Protocol Description **24**
B.1 On Reliability Constraints . . . 24
B.2 Secure Communication . . . 25
B.3 Sessions . . . 26
B.4 OPC UA Model and Security Goals . . . 27
B.4.1 Tables in ProVerif . . . 27
B.4.2 Threat Model . . . 27
B.4.3 Security Properties . . . 28

C Additional Details on Attacks **29**
C.1 Race Condition for User Contexts . . . 29
C.2 Client Impersonation in ECC . . . 29
C.3 Session Hijack Attacks . . . 29
C.4 Additional Findings . . . 30
C.4.1 Downgrade of Password Secrecy . . . 30
C.4.2 Risk of Signature Oracle . . . 31
C.4.3 Other Weaknesses . . . 32

D Security proofs **32**
D.1 short . . . 32
D.2 A novel proof methodology . . . 32
D.2.1 Background on ProVerif . . . 33
D.2.2 Step 1: Identify the loops . . . 33
D.2.3 Step 2: Adding noselect statements to break the loops . . . 33
D.2.4 Step 3: Collecting false assumptions . . . 34
D.2.5 Step 4: Design invariants contradicting the false assumptions . . . 34
D.2.6 Step 5: Proving the invariants . . . 35
D.2.7 Optional: Chaining invariants . . . 35
D.2.8 Soundness . . . 36

D.3 Full Results 36

E List of Changes **36**

1 Introduction

Industrial Control Systems (ICS) manage critical infrastructure facilities, such as power plants, and are part of Operational Technology (OT), *i.e.,* covering devices that interact with the physical environment [27]. As opposed to classical Information Technology (IT) systems, OT historically prioritizes safety and reliability, above other security properties such as confidentiality. Moreover, the life cycle of OT systems is much longer, as updates require heavy and costly qualification processes. Severe security incidents, *e.g.,* the Stuxnet [1] worm in 2010, led to adding a security layer (such as Internet Protocol Security (IPsec), or more often Transport Layer Security (TLS) [21]) to existing ICS protocols. However, these additional security layers are tailored for the Internet and are poorly suited to OT, motivating secure ICS protocols.

Open Platform Communication Unified Architecture (OPC UA) is a general purpose ICS protocol between Supervisory Control and Data Acquisition (SCADA) systems – including operators' workstations – and automation devices, that regulate an industrial process through sensors and actuators. It is developed by the OPC Foundation and standardized as IEC 62541 (currently in v1.04) [22]. OPC UA has undergone several security analyses by state agencies, such as the German BSI (two analyses in 2017 and 2022 [13, 30]), the French ANSSI (development of the open source implementation S2OPC [28]), as well as by OT security companies that analyzed implementations [8, 9, 29].

OPC UA follows a modular and layered design, where functionalities are activated through profiles. We focus on the OT profile "UA-TCP UA-SC UA-Binary", which is by far the most commonly used [30, § 6.3.4]. This so called OPC UA Binary profile mainly relies on the UA Secure Conversation (UASC) sub-protocol. The protocol is parameterized by a *security policy*: policies based on Elliptic Curve Cryptography (ECC) have been introduced in version 1.05 of the OPC UA specification. The use of ECC based Diffie-Hellman is supposed to provide stronger confidentiality guarantees, namely PFS, compared to RSA which was the only option previously. In addition, ECC allows for improved performance through shorter keys, which can be crucial for embedded systems. Once a secure channel has been opened, sessions can be created inside this channel. Users can then activate sessions using their credentials and send user requests. OPC UA also allows to re-open existing channels (for key rotation), and re-activate sessions (for transferring to another channel or user).

Contributions. In this paper, we perform the first comprehensive, formal security analysis of OPC UA. We leveraged the ProVerif protocol prover [6] to conduct this automated analysis. Our main contributions are as follows.

We provide a *detailed description* of the security sub-protocols of the OPC UA Binary specification version 1.05 (Sections 2.1 and 2.2), their expected security goals, and the considered threat model (Section 2.3). The relevant security aspects of the specification [22] span 5 documents (out of more than 20), totaling 528 pages. We have extracted what we believe to be a specification suitable for a protocol security analysis. This description is concise, yet comprehensive as it covers both ECC and RSA security profiles (also covering v1.04), sub-protocols for both channel and session establishment, and management, under different configurations (considering different channel modes, session security modes, and user authentication means).

In Section 3, we present our *formal model of OPC UA* in the applied pi calculus (the input language of ProVerif) which in particular encompasses the following features: *(i)* the reopening of secure channels, *(ii)* the reactivation of sessions using passwords as well as user certificates, *(iii)* the switching of sessions between secure channels, *(iv)* the authentication of user, client and server involved in a request and its response. This involves dealing with a complex state machine.

We *formally specify the security properties* (authentication and confidentiality) in Section 3.3. The properties are often tricky to formalize, and we need to condition them by the exact configurations in which a property is expected to hold, as the protocol also allows degraded configurations (that may be deployed simultaneously in other sessions). This also means that our model precisely describes in which configurations and threat models each property is actually expected to hold.

We *analyze OPC UA* using ProVerif. Given the size and the complexity of our model, and the large number of possible configurations, we provide tooling (Section 4.2) that allows to *(i)* generate ProVerif models for a particular set of configurations, and threat model; *(ii)* efficiently explore the lattice of configurations and threat models, and automatically find maximal configurations in which a property holds, as well as minimal configurations in which we find attacks (or reach the limits of ProVerif). Moreover, we used many advanced features of ProVerif [7] to fine tune the model and guide the proof search, to be able to conclude in complex configurations (Section 4.3). We believe that this is among the most complex analyses performed with ProVerif, both due to the size of the model as well as the complexity of the state machine, *e.g.,* the model requires a lot of information to be stored in a global state. To the best of our knowledge, it is the largest ProVerif model in terms of LoC and initial clauses (the internal protocol representation on which ProVerif reasons) ever analyzed.

Our analysis allowed to discover 8 *new vulnerabilities and other weaknesses* in OPC UA v1.05 (Section 5), 6 of these also affect v1.04. Each of these have been responsibly disclosed to and acknowledged by the OPC UA Foundation. We proposed provably secure fixes and other mitigations, most of them are now included in the specification. Finally, we draw more general lessons for the future of OPC UA (Section 5.7).

Artifacts. All models, results, and instructions to reproduce them are provided in the companion artifact [14].

2 OPC UA Protocol

2.1 Overview

We start by presenting the OPC UA protocol version 1.05.03 with a high-level overview of the sub-protocols and their interactions. In this work, we cover the security sub-protocols, namely the secure channel and the session sub-protocols. They are specified in the OPC UA standard, mostly in [22, Parts 4,6,7]. They involve three kinds of agents: *clients* (*e.g.,* user workstations), *servers* (*e.g.,* SCADA), and *users* (*e.g.,* humans operating user workstations). Each agent is assumed to be enrolled in a Public Key Infrastructure (PKI) and possesses a certificate that describes its identity and role (*e.g.,* a client certificate C_{cert}) with the associated private key (*e.g.,* C_{sk}); users store them on a smart card or may alternatively use a login password pair.

The main flow is depicted in Fig. 1: a client *opens* a *secure communication channel* with a server and *creates* a *session* inside this channel. A user on this client can then log in, *i.e., activate* a previously created session, and use this session to send/receive *requests* to/from the server. We now briefly present those sub-protocols.

Secure communication channels. Any communication between a client and a server in OPC UA starts by opening a secure channel. While there exists a specific configuration (OPC UA HTTPS) that leverages TLS for this, the more widely deployed profile (OPC UA Binary) has a dedicated channel sub-protocol called UASC. This sub-protocol is made of two parts: *(i)* a handshake layer, called *OpenChannel* (see Fig. 1), that establishes a new channel ID id_c and a set of symmetric keys sk, shared between client and server; *(ii)* a record layer that protects all subsequent messages with these symmetric keys (depicted by $\|.\|^{id_c,sk}$). The symmetric keys can be renewed by re-running OpenChannel on an existing channel.

Channels can be configured in three Modes:

- Enc: provides integrity and confidentiality.
- Sign: provides integrity only. This can be required for the system to be monitorable by third parties.
- None: insecure channels, which are reserved for scanning the network and debugging.

Finally, a channel SecurityPolicy defines the asymmetric and symmetric cryptographic algorithms to use. The protocol supports two families: RSA and ECC. The ECC family offers an Elliptic Curve Diffie–Hellman (ECDH) key exchange that can additionally provide PFS.

Session creation and activation. OPC UA sessions provide a user context for requests and their responses.[2] Sessions are created by a client in a channel (see CreateSession in Fig. 1), and yield a session identifier id_s and a session token tok_s (explained in Section 2.2.2). The options (SSec | SNoAA) define the security level provided by the session and will be explained later in Section 2.2.2. Created sessions can be activated (see ActivateSession) directly by the client with the anonymous user (Ano), or by a regular user. Regular users rely for this on login password pair (Pwd) or a certificate (Cert), which may be stored on a smart card for better security (see NIST [27] § 6.2.1.4.3).[3]

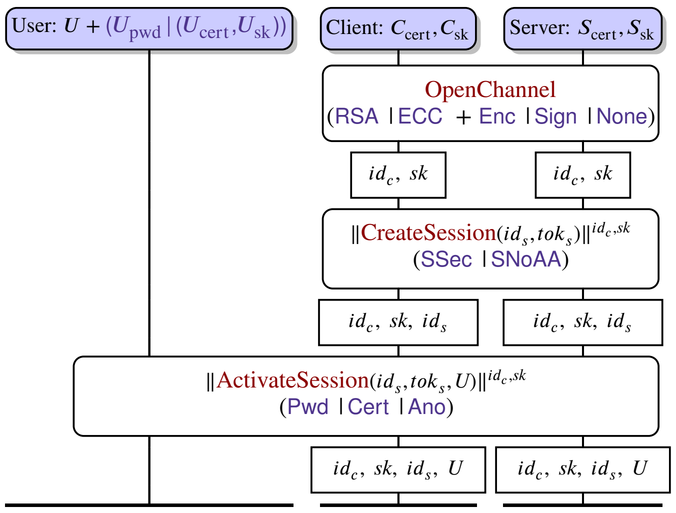
State machine. There exist various ways for channels and sessions to evolve and the resulting state machine, depicted in Fig. 2, is rather complex and further complicated by the various configurations that can be arbitrarily combined: (RSA | ECC + Enc | Sign | None + SSec | SNoAA + Pwd | Cert | Ano). Once opened, a channel can be *Reopened*, by running OpenChannel on the existing channel, resulting in a key renewal. Sessions can then be created and activated on any channel. By running ActivateSession on a previously activated session, users can:

- *ReActivate*: hand over sessions to other users (*e.g.,* at shift change), or
- *Switch*: transfer them to a new secure channel (*e.g.,* in case the previous channel was terminated).

Such complex operations are required for increased reliability, which is key in OT systems (see Appendix B.1).

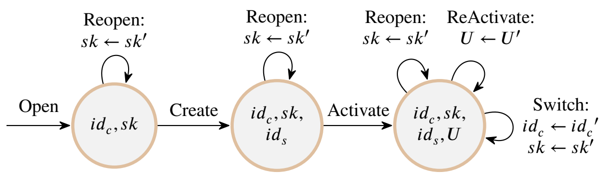
[2]There are also sessionless requests, but we choose not to cover them in this work because they rely on issued tokens (see next footnote).

[3]Another user authentication mode uses issued tokens (*e.g.,* WS-SecurityTokens (Kerberos), JSON Web Token (JWT) or OAuth 2 tokens). They rely on authorization services, that are protocols in their own rights. However, issued tokens details are neither specified nor included in the security profiles of version 1.05. Hence, we do not cover them here, and rather focus on the two core OPC UA user authentication methods.



User: $U + (U_{\text{pwd}} \mid (U_{\text{cert}}, U_{\text{sk}}))$
Client: $C_{\text{cert}}, C_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
OpenChannel
(RSA | ECC + Enc | Sign | None)
$id_c,\ sk$
$id_c,\ sk$
$\|$CreateSession$(id_s, tok_s)\|^{id_c, sk}$
(SSec | SNoAA)
$id_c,\ sk,\ id_s$
$id_c,\ sk,\ id_s$
$\|$ActivateSession$(id_s, tok_s, U)\|^{id_c, sk}$
(Pwd | Cert | Ano)
$id_c,\ sk,\ id_s,\ U$
$id_c,\ sk,\ id_s,\ U$

Figure 1: Overview of the OPC UA protocol: a client opening a secure channel (id_c with keys sk), creating a session id_s, and a user U activating it. Sub-protocols, which will be detailed later, are indicated in red, and protocol configurations in blue. $\|\cdot\|^{id_c, sk}$ denotes channel protection, that depends on the channel configuration (Encryption&Signature | Signature | None).



Open
Reopen: $sk \leftarrow sk'$
id_c, sk
Create
Reopen: $sk \leftarrow sk'$
id_c, sk, id_s
Activate
Reopen: $sk \leftarrow sk'$
ReActivate: $U \leftarrow U'$
id_c, sk, id_s, U
Switch: $id_c \leftarrow id_c'$ $sk \leftarrow sk'$

Figure 2: Overview of the OPC UA state machine

2.2 Protocol Description

For the sake of clarity, we first present the different sub-protocols when SecurityPolicy is set to ECC and Mode to Enc, and then explain the main differences with other configurations. We refer the reader to Appendix B for their detailed description.

Cryptographic Schemes. We use the following notations for cryptographic operations. $[\![m]\!]_{sk}$ denotes the signature of m with the private key sk, $\{m\}_k$ the symmetric encryption of m with the key k, and $[m]_{mk}$ the addition of a Keyed-Hash Message Authentication Code (HMAC) with key mk. The choice of the precise cryptographic suite implementing those depend on server configuration. *E.g.,* for SecurityPolicy `[ECC-B] ECC-nistP256` [23], they are respectively instantiated with ECDSA-SHA2-256, AES128-CBC and HMAC-SHA2-256. More importantly for this work, the combination of HMAC and symmetric encryption roughly follows the MAC-then-Encrypt construct with some data in plaintext. Formally:

$$[m, \{p\}_{ek}]_{mk} := m, \mathsf{AES}_{ek}(p, \mathsf{HMAC}_{mk}(m, p)).$$

The case of profiles with authenticated encryption schemes (a minority of profiles: 3 over 11) is a bit different (see Appendix B.2) and is left as future work.

2.2.1 UA Secure Conversation

We now present the UASC sub-protocol, including OpenChannel, see Fig. 3. The first two messages are for channel opening or reopening, and the last two are for requests and responses sent over the created channel. The requests that can be sent over the channels are described in Section 2.2.2.

Channel (re)opening. To open a secure channel, the client sends an *OpenSecureChannel* request. This request notably includes the client's certificate C_{cert} and is signed with the associated private key C_{sk}. The channel ID id_c^0 is initially set to 0 to indicate that a new channel should be opened. There is also a counter Rq, incremented at each request. Finally, the request includes the Diffie-Hellman (DH) half-key $C_{\text{hk}} = g^c$, from which the shared secret will be derived.

The server checks the client certificate and the signature of the client request. Its response is similar, but includes its own certificate S_{cert} and half-key $S_{\text{hk}} = g^s$. The server also chooses a new, fresh channel ID id_c (a counter) and tok_c (a nonce).

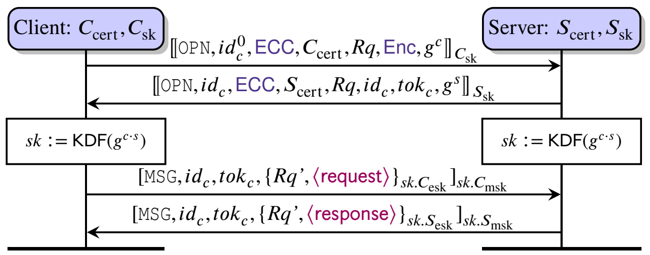

Client: $C_{\text{cert}}, C_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
$[\![\texttt{OPN}, id_c^0, \mathsf{ECC}, C_{\text{cert}}, Rq, \mathsf{Enc}, g^c]\!]_{C_{\text{sk}}}$
$[\![\texttt{OPN}, id_c, \mathsf{ECC}, S_{\text{cert}}, Rq, id_c, tok_c, g^s]\!]_{S_{\text{sk}}}$
$sk := \mathsf{KDF}(g^{c \cdot s})$
$sk := \mathsf{KDF}(g^{c \cdot s})$
$[\texttt{MSG}, id_c, tok_c, \{Rq', \langle\mathsf{request}\rangle\}_{sk.C_{\text{esk}}}]_{sk.C_{\text{msk}}}$
$[\texttt{MSG}, id_c, tok_c, \{Rq', \langle\mathsf{response}\rangle\}_{sk.S_{\text{esk}}}]_{sk.S_{\text{msk}}}$

Figure 3: (Re)Open Secure Channel (first two messages) and Channel Communications (last two) in $(\mathsf{ECC} + \mathsf{Enc})$. `OPN` is a header string. After the first two messages, client and server can derive a set of symmetric keys *sk* containing encryption keys $C_{\text{esk}}, S_{\text{esk}}$ and HMAC keys $C_{\text{msk}}, S_{\text{msk}}$. The last two messages describe the channel protection of subsequent requests/responses payloads $\langle\mathsf{request}\rangle / \langle\mathsf{response}\rangle$ (cf. Section 2.2.2).

The former identifies the channel, while the latter identifies the shared secret *sk* that will be derived. *sk* is computed by a Key Derivation Function (KDF) of the shared DH secret $g^{c \cdot s}$. More precisely, two keys for encryption ($sk.C_{\text{esk}}$ and $sk.S_{\text{esk}}$) and HMAC ($sk.C_{\text{msk}}$ and $sk.S_{\text{msk}}$) are derived. The client checks the server certificate and the signature of the response and derives *sk* as well. (id_c is included twice.)

An existing channel with ID id_c^{old} can be reopened by a client by choosing $id_c^0 := id_c^{\text{old}}$ (instead of zero). The server will then choose a new security token tok_c' but keep $id_c = id_c^{\text{old}}$. The client must reopen before the channel lifetime (depending on server configuration) expires.

Message security. Once a channel is open, client and server can send requests and responses, which are wrapped and cryptographically protected as shown in the last two messages of Fig. 3. In particular, reusing notation from Fig. 1, $\|\mathsf{request}\|^{id_c, sk} := [\texttt{MSG}, id_c, tok_c, \{Rq', \mathsf{request}\}_{sk.C_{\text{esk}}}]_{sk.C_{\text{msk}}}$.

Other configurations. When $\mathsf{SecurityPolicy} = \mathsf{RSA}$, the client and server exchange nonces C^c_{nonce} and S^c_{nonce} instead of DH half-keys. Those nonces are asymmetrically encrypted with the receiver's public key and are sent along with the hash of the public key used for encryption. When $\mathsf{Mode} = \mathsf{Sign}$, the encrypted parts are in plaintext in `MSG` messages. When $\mathsf{Mode} = \mathsf{None}$, signatures, HMAC, encryption, certificates, and half-keys (or nonces) are omitted in all messages. See a detailed description of those configurations in Appendix B.2.

2.2.2 Sessions

We now describe the CreateSession and ActivateSession sub-protocols (both depicted in Fig. 4). The first two messages allow the creation of a session, the third and fourth its activation, and the last two describe how *user requests*, at the application layer, are wrapped in a session. We stress that all those messages are sent over and protected by some channel, and act as $\langle\mathsf{request}\rangle$ and $\langle\mathsf{response}\rangle$ of that channel, cf. Fig. 3.

Session creation. The client sends a `Create` request with a fresh nonce C_{nonce} and C_{cert}. The server checks that the client certificate is the same as the one associated to the channel through which this message is sent (this client certificate was sent in the OpenChannel request). The server response provides a new, fresh session ID id_s (a counter) that will identify the session. It also includes a fresh *Session Authentication Token* tok_s (a nonce) that serves the same purpose as id_s but is supposed to be kept secret. A fresh server nonce S_{nonce} is provided and acts as a challenge to the client for the next session activation. The server also proves possession of his certificate, by signing C_{cert} and C_{nonce} with S_{sk}. Finally, a signed fresh DH half-key $S_{\text{ek}} = [\![g^s]\!]_{S_{\text{sk}}}$ is added in an additional field for later use, should the session be activated in configuration Pwd (described next). If all checks pass, clients and servers notably store the session information: id_s, tok_s, id_c (id_c refers to the channel on which the messages were sent).

First activation. Once created, a user can *activate* a session (id_s, tok_s, id_c) by sending an `Activ` request. This request includes tok_s and a signature of S_{cert} and S_{nonce}, sent in the previous session creation response, to prove possession of C_{sk}. In mode (Cert) (as in Fig. 4), the request must also contain the user's signature (with U_{sk} associated to U_{cert}) of S_{cert} and S_{nonce}. In practice, the signature can be computed on the user's Smart Card. The server verifies the signatures and responds with fresh S'_{nonce} and S'_{ek} for future reactivations.

User requests and responses. User requests $\langle\mathsf{Urequest}\rangle$, produced by the application layer, are wrapped in a session by appending the session token tok_s. The user response $\langle\mathsf{Uresponse}\rangle$ can then be linked to that session as well, since the response will use the

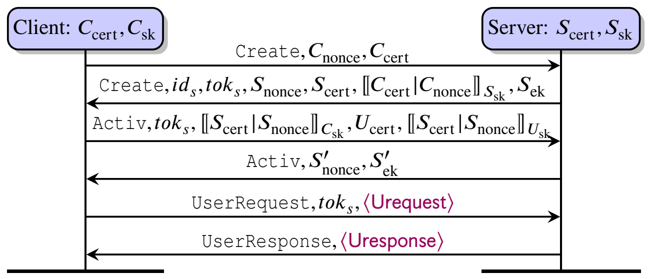

Client: $C_{\text{cert}}, C_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
Create, $C_{\text{nonce}}, C_{\text{cert}}$
Create, $id_s, tok_s, S_{\text{nonce}}, S_{\text{cert}}, [\![C_{\text{cert}} | C_{\text{nonce}}]\!]_{S_{\text{sk}}}, S_{\text{ek}}$
Activ, $tok_s, [\![S_{\text{cert}} | S_{\text{nonce}}]\!]_{C_{\text{sk}}}, U_{\text{cert}}, [\![S_{\text{cert}} | S_{\text{nonce}}]\!]_{U_{\text{sk}}}$
Activ, $S'_{\text{nonce}}, S'_{\text{ek}}$
UserRequest, tok_s, ⟨Urequest⟩
UserResponse, ⟨Uresponse⟩

Figure 4: Session creation (first two messages) and activation (third and fourth) and user requests/responses (last two) in (ECC + Enc + SSec + Cert). The session activation involves the user to produce the signature with U_{sk}. `Create` and `Activ` are header strings. The last two messages describe how user requests ⟨Urequest⟩ and responses ⟨Uresponse⟩ payloads, from the application layer, are wrapped in a session.

same Rq at channel level. Indeed, recall that such session requests are additionally sent through, and cryptographically protected by, some secure channel.

Reactivation and channel switching. Session handover to another user is achieved by sending an activation request referring to a previously activated session, on the same client and channel, but for a different user. Similarly, the switching of secure channel is an activation of the same session, on the same client by the same user, but carried out on a different secure channel, with the same security Mode and the same client and server certificates.

Other activations. In configuration Ano, U_{cert} and the signature with U_{sk} are omitted and the session is not bound to a user. In configuration Pwd, U_{cert} is replaced by a login and the user's signature is replaced by the password encrypted with a symmetric key derived from the latest S_{ek} and some fresh client DH half-key, both sent together with this ciphertext. When Mode = RSA, the only difference is that S_{ek} is omitted and, when necessary, the password is instead asymmetrically encrypted with the server's public key. Note that in configuration Enc , the password *may* be sent without symmetric encryption (*i.e.,* sent as plaintext at the session layer), as encryption is provided at the channel level.

Session security. We now explain the differences between SSec and SNoAA. By default, in SSec, server and client certificates are exchanged and trust checked against the PKI, at session creation and activation. However, the specification also defines a *No Application Authentication* mode [22, Part 7, § 5.3], where the server is configured to accept all certificates and only use them for message security. We call this mode SNoAA. We provide further details about SNoAA and the other aforementioned configurations in Appendix B.3.

2.3 Security Goals

In this section, we describe the security goals of OPC UA, and the threat model under which these goals should hold as described in [22, Part 2 *"Security Model"*]. In Appendix B.4, we provide more evidence, with references to the specification, supporting our interpretations.

2.3.1 Threat Model

We first summarize the threat model in terms of attacker capabilities and compromise scenarii detailed in [22, Part 2, § 4.3 *"Security threats to OPC UA systems"*]. OPC UA is claimed to resist the following threats: eavesdropping (§ 4.3.3), message spoofing (§ 4.3.4), message alteration (§ 4.3.5) and replay (§ 4.3.6), emitting malformed messages (§ 4.3.7). The attacker is also able to set up a rogue server (§ 4.3.10), and to act as a MiM (§ 4.3.9). Therefore, we shall consider an *active adversary on the network* who can eavesdrop, intercept, manipulate, and inject messages. Such an attacker is often referred to as a *Dolev-Yao attacker* [15].

Moreover, user credentials (passwords, or certificate private keys) may be compromised (§ 4.3.12). Client and server certificates and associated private keys are also considered as potentially compromised [22, Part 4, § 6.1.4]. In short, the attacker can compromise agents' long-term secrets. Channel symmetric keys sk are more ambivalent: their lifetime depends on a parameter (*i.e., revisedLifetime*) spanning from a few minutes to 40+ days and is not enforced by security profiles. We conservatively consider those keys as semi-long-term. We do not consider compromise of such keys by default, but will allow such compromise in specific cases (password confidentiality), when mentioned explicitly. However, full attacker control of a machine, allowing ephemeral secret leaks (*e.g.,* DH client's secret c), remains out of scope [22, Part 2, § 4.4].

2.3.2 Security Properties

We now discuss the security properties that OPC UA is supposed to guarantee as per [22, Part 2, § 4.2 *"Security objectives"*] (with additional supporting evidence in Appendix B.4).

Confidentiality. Secure channels with Mode = Enc must provide confidentiality of data exchanged on the network: user requests and responses. Passwords must remain confidential for all channel modes. Moreover, channels configured with ECC additionally provide PFS, *i.e.,* any message exchange prior to the compromise should remain secure. To sum up, OPC UA is expected to guarantee that: *(i) The attacker cannot learn the user requests and responses payloads when sent over channels in configuration Enc. (ii) PFS is guaranteed for the payloads sent over channels in configuration (ECC + Enc). (iii) The attacker cannot learn the user passwords.*

Integrity and Authentication. Integrity and authentication of user requests are key requirements for OPC UA [22, Part 2, § 4.2,5.1.1]. We express those more formally as *agreement properties* in Lowe's authentication taxonomy [20], where properties are expressed as relations between different agents' point of views of a protocol execution. Data and entity authentication can be expressed as a single, strong *agreement property*: *if a server S accepts a user request R supposedly from a user U on client C, then U has indeed initiated request R for server S on C*. Conversely, *if a user U receives on client C a user response R supposedly from server S, then S has indeed responded with R to client C for user U*. As the protocol is supposed to protect against replays, we moreover require that these agreements are *injective*, *i.e.,* each received request corresponds to a distinct request emission.

3 OPC UA Formal Model

3.1 Proverif Background

In ProVerif, protocols are formally modeled in a dialect of the applied pi-calculus. Messages are abstracted by terms, to focus on their functionality *e.g.,* a ciphertext perfectly hides its plaintext, unless the attacker knows the decryption key. Protocols are described by parallel processes, that model message exchanges and verification steps performed by agents. Finally, public communication channels are fully controlled by a Dolev-Yao attacker who can intercept, send, but also forge new messages. Given a security goal, ProVerif either returns true if security is proved, or false and automatically produces an attack trace, or may return cannot-be-proved in some cases where an attack cannot be reconstructed (due to internal abstractions). Moreover, ProVerif may not terminate, since the underlying problem is undecidable. Below, we briefly recall the main concepts of ProVerif's syntax and semantics and refer the reader to [7] for a comprehensive description.

Messages. In symbolic models, messages are abstracted by formal terms in a term algebra. For example, a digital signature scheme is modeled by the function symbol $\mathsf{sign}(\cdot,\cdot)$ and the term $\mathsf{sign}(m,k)$ represents the signature of message m using the secret key k. Similarly, signature verification is modeled by two additional function symbols: $\mathsf{checkSign}(s,\mathsf{pk}(k))$ is the application of the verification algorithm, on the signature s and the public-key $\mathsf{pk}(k)$ associated to k. Finally, the expected functionality of a digital signature, *i.e.,* the fact that the verification of a valid signature succeeds, is expressed by an equation: $\mathsf{checkSign}(\mathsf{sign}(m,k),\mathsf{pk}(k)) = \mathsf{true}$. We additionally provide the means to retrieve the signed message with a function symbol $\mathsf{readSign}(\cdot)$ and equation $\mathsf{readSign}(\mathsf{sign}(m,k)) = m$. One can similarly model DH exponentiation, with function symbol $\mathsf{exp}(\cdot,\cdot)$ and constant g representing the group generator. The equation $\mathsf{exp}(\mathsf{exp}(\mathsf{g},x),y) = \mathsf{exp}(\mathsf{exp}(\mathsf{g},y),x)$ expresses that exponents commute.

Protocol roles. ProVerif relies on a process calculus to model the roles of a protocol. We illustrate the syntax on an example. Consider the Server role in the UASC sub-protocol, where the server receives an initial *OpenChannel* request from a client, and sends back a corresponding response message. A process modeling such a server running with SecurityPolicy ECC could be specified as follows (some parts are omitted with [...] for brevity):

```
let Server(S_cert: cert, S_sk: skey) =
in(c, OPN_req: bs);
let (=OPN, =zero, =ECC, C_cert: cert, Rq: nonce, mode: chmode, C_hk: G) = readSign(OPN_req) in
if checkSign(OPN_req, get_pk(C_cert)) then
new s: ex; new tok_c: chtoken; new id_c: chid;
let S_hk = exp(g, s) in let seed = exp(C_hk, s) in
let C_enc_k = enc_key_ECC(seed) in [...]
let symkeys = (tok_c, C_enc_k, [...]) in
event NewSharedKey(C_cert, S_cert, mode, seed);
insert Schan(C_cert, S_cert, ECC, mode, id_c, symkeys);
let OPN_re = (OPN, id_c, ECC, S_cert, Rq, id_c, tok_c, S_hk) in
out(c, asign(OPN_resp, S_sk)) [...]
```

The Server process, parametrized with a certificate S_cert and an associated private key S_sk (line 1), initially waits for a message OPN_req from a client (line 2) on a public channel c; cert, nonce, chmode, and G are the expected types[4]. The server extracts the different fields from the message and checks that they contain the header string OPN, a channel ID set to zero and that SecurityPolicy = ECC; other fields are bound to the variables C_cert, Rq, mode, and C_hk (line 3). Next, it checks the signature (line 4). If all those checks succeed, the server generates new *names* that are fresh values, namely a new ephemeral secret, channel identifier, and token (line 5), and computes the shared DH key and associated symmetric channel keys C_enc_k, etc. (lines 7, 8). The server then logs the fact that a new shared key has been computed with the *event* NewSharedKey (line 9). *Events* are used to model security properties (see next paragraph). The server also stores the channel information in a *table* Schan (line 10) and computes the *OpenChannel* response that is signed (line 11) and output to the client (line 12). *Tables* are used to store evolving states of the different agents. For instance, the table Schan(C_cert, S_cert, ECC, mode, id_c, symkeys) stores the channels, that have been opened by the server identified by the certificate S_cert. When the server receives the *CreateSession* request in the session sub-protocol, it will retrieve the channel information from this table, *e.g.,* a server with certificate S_cert can retrieve symkeys and C_cert of a channel identified by id_c with: get Schan(C_cert, =S_cert, sec_policy, mode, =id_c, symkeys).

Security Properties. Intuitively, events are merely annotations in execution traces that record some specific steps of the protocol. Security properties are then expressed as logical formulas over events. *E.g.,* the following formula expresses the impossibility for an attacker to learn some shared secret seed, that has been computed for securing a channel in mode Enc by a server, and hence is logged in a NewSharedKey event: event(NewSharedKey(S_pk,C_pk,Enc,seed))&& attacker(seed)==> false.

Note that attacker(seed) is true whenever the adversary can compute the value of seed. The attacker is able to compute new terms by applying function symbols on previously observed outputs. This property thus models confidentiality of Enc channel shared secret keys, from the point of view of a server.

3.2 Modeling in the Applied Pi Calculus

We now discuss our formal model of OPC UA [14], and explain some of our modeling choices. As a general guideline, we strive to make attack conservative choices. For example, when the specification allows several options we allow all of them, or choose the least secure one.

Overall architecture of processes. Our model is based on three main processes, one for each kind of agents (users, clients and servers), running in parallel. We consider an unbounded number of agents, and each agent may engage into an unbounded number of protocol sessions. *E.g.,* A client may start an arbitrary number of sessions with different servers.

Each of these main processes contains subprocesses for specific tasks. The client process allows to (re)open channels and create sessions. The user process allows to activate session (using user's credentials) and send/receive user requests/responses. The server process handles the different kinds of requests (channel (re)opening, session creation, (re)activation and channel switching, as well as user requests). As the scheduling is adversarial, the attacker decides which actions are triggered and which agents execute a given sub-protocol.

PKI, user passwords and compromise. When an agent is created, it is enrolled in a PKI. The PKI issues a certificate modeled by a private function, *i.e.,* a function that the attacker is not allowed to apply: fun certify (kind,crypto,pkey) :cert [private]. A certificate specifies the kind of agent (Client, Server, or User), the crypto family RSA or ECC as well as the public key pkey. For each new user, a fresh password is also created.

At any time, we allow the attacker to compromise an agent. Compromise of agent A triggers the event leak(A) and results in the output of A's secret key, and password if A is a user.

Local state and tables. As illustrated in Fig. 2, the state machine is rather complex, and yet fully captured by our model. We use tables to store the local states of the protocol agents. In particular, we use tables for recording the list of channels that were opened and sessions that were created/activated by a client or server. Note that we distinguish these tables for clients and servers as their view may not coincide.[5] We introduced 4 tables in total: channels and sessions for clients and servers. This way, were able to fully capture the OPC UA state machine, which will, however, make the reasoning and proof search extremely complex and will require a carefully designed proof methodology (see Section 4).

Modes and configurations. Our model is designed to be modular and allows to enable/disable the modes and features that are to be supported. We can set the SecurityPolicy (ECC or RSA), channel modes (None, Sign or Enc), the session security level (SSec or SNoAA), as well as the supported user credential modes (Pwd, Cert or Ano). For example, the configuration (RSA | ECC + Enc | Sign | None + SSec | SNoAA + Cert) supports all kind of crypto, channels, and session modes as well as user identification through certificates. However, anonymous and password-based login are excluded. Further parameters allow

[4]Note that although the language is typed, the attacker is allowed to send ill-typed messages and ProVerif does capture type flaw attacks.

[5]However, user processes have access to client tables (users can access local clients' states they use and control).

to enable or disable agent and channel keys compromise, reopening of channels, and channel switching. Regarding the two admissible behaviors for password encryption in Enc channels mentioned in Section 2.2.2 (double encryption or only channel encryption), we conservatively chose the least secure option (channel encryption only).

Being able to use such parameters has proven to be extremely useful to establish our proof methodology (Section 4). We stress that we aim to obtain results with respect to a rich model with almost all modes enabled (although we sometimes consider some option values in isolation).

Size of the model. The complexity of our model is also reflected by its size: while the ProVerif files were written in a modular way that amount to 2.4k LoC (including all declarations and property definitions), the unfolded process that ProVerif generates to reason corresponds to 8.6k LoC that are translated into 2.3k initial clauses (for the full configuration when verifying a confidentiality property). As a point of comparison the most comprehensive model of TLS 1.3 [5] has an unfolded process of 7.3k LoC translating to 1.4k clauses.

3.3 Formal Security Properties

In this section, we explain how we formally modeled in ProVerif, the security properties identified in Section 2.3.2.

***Conf*$_{\mathrm{C}}$ and *Conf*$_{\mathrm{S}}$: Confidentiality of user requests and responses.** When Mode = Enc, user requests and the corresponding responses are supposed to be confidential. In addition, when SecurityPolicy = ECC, the protocol is expected to provide PFS. These requirements are formalized as follows, where C_data is an event triggered by clients whenever they send/receive a user request/response R:

```
event(C_data(C_pk, S_pk, SecPo, Enc, SE, R))@t && attacker(R)
 ==> event(leak(S_pk))@ts && (SecPo = RSA || ts < t)
```

For any user request R sent at time t in a session SE with Mode = Enc, such that R is known to the attacker, it must be that the peer has been compromised (leak(S_pk)). Moreover, if the SecurityPolicy (SecPo) is not RSA, and thus is ECC, then PFS requires that the leak must have occurred *before* the request (ts < t). A similar property models the confidentiality from servers' point of view, using an event S_data triggered whenever a request/response is received/sent. We respectively call those two properties *Conf*$_{\mathrm{C}}$ and *Conf*$_{\mathrm{S}}$.

***Conf*$_{\mathrm{Pwd}}$: Confidentiality of passwords.** A second confidentiality requirement is that passwords remain secret:

```
event(new_user(U, pwd, U_sk, U_pk, U_cert)) && attacker(pwd)
 ==> event(leak(U_pk)) ||
     ( event(C_Activ_req(C_pk,S_pk,mode,check,SE,U,pwd)) &&
       event(leak(S_pk)) )
```

The property states that for any declared user, if the attacker knows her password, then one of the following must have happened: *(i)* the user has been compromised, or *(ii)* the password has been used by a client to activate a session (event C_Activ_req) with a compromised server S_pk. Indeed, in both cases the password is trivially leaked. Otherwise, we expect the password to remain confidential.

***Agr*$_{\mathrm{S}}$ and *Agr*$_{\mathrm{C}}$: Data and entity authentication of user requests and responses.** Finally, we model authentication of user requests. As discussed in Section 2.3.2, this property is an injective agreement property:

```
inj-event(S_Rcv_Usr_Req(C_pk, S_pk, SE, U, R))
 ==> inj-event(C_Snd_Usr_Req(C_pk, S_pk, SE, U, R))
   || (event(leak(C_pk)) || mode = None) &&
      ( U = anon || event(leak(U)) ||
        event(C_Activ_req(C'_pk,S'_pk,mode,check,SE',U,pwd)) &&
        event(leak(S'_pk)) )
```

The property states that, whenever the server S_pk receives a request R from a user U_pk, sent on a client C_pk in a session SE, then U_pk did indeed send R to S_pk in session SE or the attacker can impersonate the user and either (a) the client was compromised (leak(C_pk)) or (b) the channel mode is None. To impersonate the user we have to consider three cases: *(i)* either the session is anonymous, *i.e.,* there is no user, or *(ii)* the user credentials have been leaked, or *(iii)* the user did use his password to authenticate to a corrupted server (on an arbitrary client). By declaring the agreement to be injective, we ensure that any user request event on the server side corresponds to a *distinct* user request event on the client side. This is called *Agr*$_{\mathrm{S}}$. The opposite property, *Agr*$_{\mathrm{C}}$, is also modeled and starts with a C_Rcv_Usr_Resp, ends with a S_Snd_Usr_Resp event, and is simply conditioned by || event(leak(S_pk)).

Sanity checks. In addition, we deploy a number of sanity checks to ensure the validity of our model. We check that all expected protocol flows are indeed executable in each of the configurations (encoded as reachability of events). For each of the security properties we also verify that the property is indeed violated when not conditioned correctly, *e.g.,* removing event(leak(U_pk)) in

Configuration option	Admissible values	# configurations
SecurityPolicy	(ECC | RSA)	3
Mode	(Enc | Sign | None)	7
SessionSecurity	(SSec | SNoAA)	3
UserAuthentication	(Cert | Pwd | Ano)	7
Leak	(Ltk | Chk / Nok)	4
Reopen	(T | F)	2
Switch	(T | F)	2

Table 1: Model configurations are defined by setting (possibly multiple) admissible values to each configuration option. The first four lines correspond to protocol configurations, the fifth line allows to enable/disable key leakage, and the last two lines correspond to protocol simplifications.

$Conf_{\text{Pwd}}$. Finally for each correspondence property `H ==> C` we check that the hypothesis `H` can be satisfied as otherwise the property would trivially hold.

4 Proof Methodology

We now present our analysis methodology, and some of the more advanced proof techniques we developed to overcome the complexity of our model. Those were pivotal to produce any meaningful analysis, as ProVerif was initially unable to successfully conclude, even for simplified configurations.

4.1 Protocol Configurations and Modularity

In order to isolate the different model features that were challenging for ProVerif to analyze, we developed a modular protocol model that allows to choose: *(i)* a particular protocol configuration, *(ii)* the possibility for compromise of long-term keys (Ltk) and/or channel keys (Chk), or no key leakage at all (Nok), *(iii)* to to turn on (T) / off (F) channel reopening and session switching. We summarize them in Table 1.

Note that a protocol configuration may enable multiple admissible values at once, *e.g.,* a deployed server may be configured to accept either ECC, RSA, or both. The protocol simplifications obviously allow to simplify the automated analysis by disabling some features. They also allow to minimize attack traces and narrow down the exact protocol features that are required to trigger a vulnerability. We use a preprocessor, based on a Python Jinja template engine, to generate the model corresponding to the chosen configuration. As each of these model options are pairwise independent, we end up with 7 056 possible configurations and as many models.

4.2 Systematically Exploring Configurations

As we shall see, the set of model configurations (see Table 1) forms a lattice with a single maximal element. Formally, we first define an order relation over configuration options as the set inclusion for protocol configuration and key leakage options (*e.g.,* Sign < Enc | Sign) and F < T for the other options. The order over configurations is then defined point-wise. Given a configuration c, we denote by $c^{\uparrow}$, respectively $c^{\downarrow}$, its upward, respectively downward, closed set.

We developed a Python script to efficiently explore the verification of all configurations for a given property, and extract maximal configurations for which the property holds, minimal configurations for which the property does not hold, *i.e.,* an attack is found, as well as minimal configurations for which ProVerif does not terminate successfully, *i.e.,* either returns cannot-be-proved or exhausts a given resource (time or memory) budget. Indeed, whenever a property holds on a configuration c, the property holds on any configuration in $c^{\downarrow}$ (as all traces in these configurations are included in the traces of c). Conversely, when a property is falsified on c, it is falsified on $c^{\uparrow}$ as well. As a heuristic, we assume that when ProVerif does not successfully conclude on c it will neither conclude on configurations in $c^{\uparrow}$.

As ProVerif may not terminate, we allocate a maximal time and memory budget. We start with a small resource budget (a few seconds and MO of RAM) that allows to quickly explore the lattice and often prune large parts on which ProVerif efficiently concludes. Our script then iteratively increases the budget and re-explores the configurations that exhausted the previous budget (more details in Appendix D.1). Moreover, our script allows to explore the lattice in parallel on multiple CPU cores. This lattice exploration script is reminiscent to previous work, *e.g.,* [17, 18] that faced similar challenges, but we push these ideas further with variable resource allocation and exploration heuristics (iteratively increasing resources).

4.3 Advanced proof techniques

As mentioned above, due to the complexity of the model, ProVerif fails to successfully conclude on most properties and configurations except for the most basic ones, *e.g.,* when only Mode = None is enabled as no secure channel can be opened in such a trivial case. To obtain meaningful results, we thus had to use a number of existing, advanced features that allow to fine-tune the generation of Horn clauses and the resolution underlying ProVerif's proof search. The authentication properties were particularly challenging and required what we believe to be a novel proof methodology combining existing features in order to successfully prove them.

4.3.1 Advanced features

In order to finitely represent the potentially unbounded number of fresh values a *name* can have, ProVerif uses a sound abstraction: a name new n is internally represented as a function of all previous (in the syntax tree) inputs, table accesses and (internal) replication indices resulting in a term. It is possible to fine-tune such contexts without impacting the model semantics; *i.e.,* ProVerif is sound independently of the contexts. Larger contexts generally enlarge the verification complexity but yield more precise results, that is ProVerif will less likely return a cannot-be-proved result. Smaller contexts can drastically reduce the verification complexity, and even address non-termination issues, but can result in more cannot-be-proved. For our model, we have modified the context of 34 out of 39 new names, often to mitigate non-termination issues. In some cases, different properties required different name contexts; this is handled by our Python script generating the model for a given configuration and property.

Similarly, internal abstractions on inputs may hamper ProVerif's ability to conclude. Adding a [precise] statement on selected inputs was often required for the attack reconstruction to succeed. Such statements instruct ProVerif to generate clauses with additional information. However, the additional precision significantly degrades performance and required to fine-tune when to use it (depending on the property and configuration being proved).

We also had to design *lemmas*. Lemmas use the same syntax as security properties (queries) and are mainly invariants on the protocol, whose validity is first checked by ProVerif. Lemmas are then applied during the proof search when proving queries by providing additional information. The most basic lemmas are of the form not attacker(t), which specifies that the attacker is unable to learn t. We have introduced such lemmas notably to express the secrecy of the DH exponents in the protocol which significantly speed up the verification.

4.3.2 A novel proof methodology

Despite the addition of proof helpers as described above, the verification did not terminate for authentication properties on most configurations. Introspection of the ProVerif's proof search procedure revealed that it was looping. The loop resulted from the handling of sessions in tables and correspond to actual loops in the protocol state machine. For instance, in order to (re)activate a session, a table entry of a previous session is necessary and a new entry of the updated session is created. Hence, insertion of a table entry requires the existence of a table entry. We therefore applied the following principled methodology that consists in two steps and that is quite effective at thwarting such loops, which is a recurring issue in complex protocol verification. (A more detailed presentation with a running example is given in Appendix D.2).

Breaking loops. ProVerif's resolution procedure is guided by a *selection function* which can be modified using a noselect statement. Manually inspecting the detailed output, *i.e.,* created clauses displayed with the option `verboseRules`, we are able to identify the facts that are almost always selected and that cause the loop, and *break* this loop by adding noselect statements on those facts. In our case, as explained above, the loop was caused by table accesses. We therefore added statements of the type noselect table(S_sessions ([...])) for session and channel tables to deprioritize the selection of a given table fact and avoid entering the loop.

Adding invariants. However, as tables are not selected anymore, this results in cannot-be-proved. Indeed, without the ability to trace back from where a table entry originates, it conservatively assumes tables could initially contain arbitrary entries, leading to "false attacks". Fortunately, ProVerif explicitly describes those "false assumptions" it made in the *goal reachable*, *i.e.,* the clause that falsifies the property. One can then contradict these with invariants, confirm they are correct by proving them with ProVerif, and then retrying to prove the initial property assuming the invariants. In short, such invariants "compensate" for the unexplored part (tracing back the origin of a table entry). This is an iterative and generic process, where invariants are added when needed by inspecting the false assumptions that ProVerif generates.

Properties	OPC UA v1.05.03		v1.05.04 RC
$Conf_{\mathrm{C}}$	✗	5.6 (C.4.2)	✓
$Conf_{\mathrm{S}}$	✓		✓
$Conf_{\mathrm{Pwd}}$	✗	5.6 (C.4.1, C.4.2)	✓
Agr_{S}	✗	5.1, 5.2, 5.3, 5.4, 5.5, (C.3)	✗/✓$^-$
Agr_{C}	✗	5.1, 5.2, 5.6 (C.4.2)	✗/✓$^-$

Table 2: Summary of our automated security analysis with respect to the properties from Section 3.3. ✗: an or several attack(s) are automatically found by ProVerif, with references to the corresponding subsections (those in appendices are shown in brackets). ✓: ProVerif proved the property. ✗/✓$^-$: ProVerif shows that some residual risks remain in some configurations (attacks against Agr_*) and proves instead a weaker property that does not consider the residual risks (Agr_*^-). When we claim a proof: not all configurations have necessarily been proven, but all configuration options are covered (see "Scope of the proofs" in Section 5).

For instance for OPC UA, when the client receives a message signed with the server's signing key $sk.S_{\mathrm{msk}}$, this signing key is the only known link between the session entries on the client and server side that ProVerif infers (as the noselect avoids resolution on the table facts). In this case we can provide an invariant which states that if the client and server session table entries use channel keys sk_1 and sk_2 that coincide on S_{msk} ($sk_1.S_{\mathrm{msk}} = sk_2.S_{\mathrm{msk}}$), then these session entries also coincide on the client and server identities and all the channel keys ($sk_1 = sk_2$), unless the client long-term key was leaked.[6]

These invariants are first proved without the noselect statement and then declared as axioms, *i.e.,* lemmas that are admitted, when proving the security property. Finally, we manually check that the dependency graph between invariants is not cyclic (*i.e.,* we never assume inv1 to prove inv2 and assume inv2 to prove inv1). In our case, we provide this graph in [14].

We heavily relied on this technique to thwart proof search loops caused by session re-activations, channel switchings, etc. The proof of the authentication properties required addition of 10 noselect statements and of more than 100 lemmas.

Soundness. We emphasize that our methodology is sound and results in a fully checked proof. Soundness relies on the fact that all axioms are first proved as queries before being assumed, and that ProVerif guarantees soundness independently of name contexts and other proof options (see Appendix D.2).

5 Analysis Results

We have leveraged ProVerif to analyze the properties stated in Section 3.3. Our analysis uncovered several vulnerabilities affecting OPC UA version 1.05.03, some of them also affecting version 1.04. Each of these vulnerabilities, and the fixes we have designed to address them, have been responsibly disclosed to the OPC Foundation. Then, during the embargo period, we discussed with the Foundation (through their ticketing system, emails, and attending their working group meetings) the different fixes and their impact on existing implementations. The Foundation decided to keep most of them, which are now part of the updated specification v1.05.04 RC (Release Candidate). As we shall explain in detail, some of the vulnerabilities have not been entirely fixed (mostly for backward compatibility reason) and some residual risks remain. Therefore, we had to weaken the properties Agr_{S} and Agr_{C} we could prove on the updated specification, to reflect these residual risks. Excluding those residual risks, we show that our fixes do indeed avoid these attacks in the configurations for which we initially found them.

We detail the found attacks, their impact, root causes and remediation in Table 3. We summarize our formal findings in Table 2 and detail below the scope of our formal proofs. For each of the vulnerabilities, we add links to tickets that describe follow-up actions and modifications to the standard resulting from our disclosure and the discussions with the OPC Foundation. Since the vulnerabilities and weaknesses we describe target a specification, rather than an implementation, we did not request CVEs.

Scope of the proofs. Our detailed results can be found in [14] and are summarized next. Proofs for $Conf_{\mathrm{C}}$ and $Conf_{\mathrm{S}}$ (obtained in maximum 10 hours) are with respect to almost maximal configurations. Namely, we achieve the maximal configuration for RSA. Proofs are more challenging in ECC because they require PFS. We can nevertheless prove the maximal configuration (excluding None and SNoAA): (i) without Leak, as well as (ii) with Leak and either without Reopen, or without Switch. We proved $Conf_{\mathrm{Pwd}}$ for the maximal configuration without Leak. With Leak, proofs were harder to obtain as we must choose between Reopen and Switch. We otherwise capture all configurations when considering in isolation SecurityPolicy and Mode.

[6] ProVerif does actually handle tables and events differently. We therefore have to add an event of the form `event insert_t (...)` before each `insert t (...)` statement together with axioms `t (...) ==> insert_t (...)`. We can then directly reason about those events in the invariants. Note that even though these axioms hold syntactically, we prove them as queries with Proverif to ensure every insertion is indeed preceded by the corresponding event.

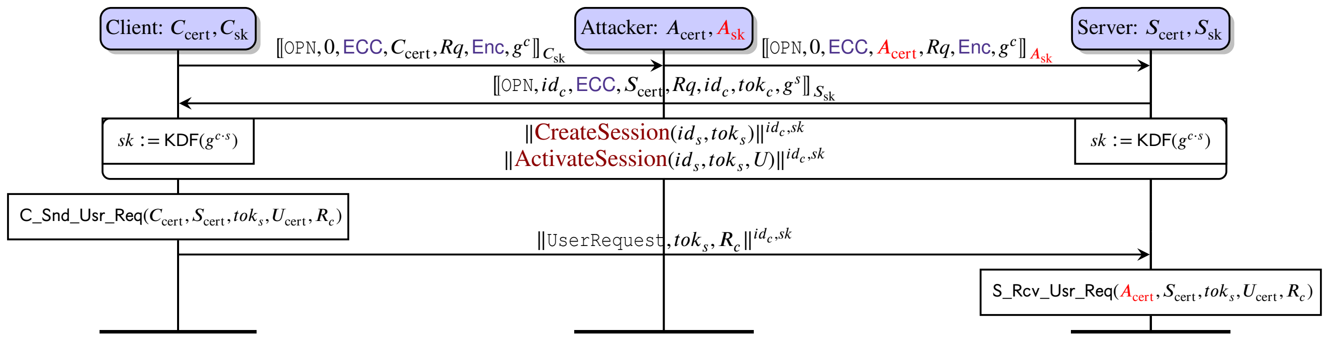

Client: $C_{\text{cert}}, C_{\text{sk}}$
Attacker: $A_{\text{cert}}, A_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
$[\![\mathtt{OPN}, 0, \mathsf{ECC}, C_{\text{cert}}, Rq, \mathsf{Enc}, g^c]\!]_{C_{\text{sk}}}$
$[\![\mathtt{OPN}, 0, \mathsf{ECC}, A_{\text{cert}}, Rq, \mathsf{Enc}, g^c]\!]_{A_{\text{sk}}}$
$[\![\mathtt{OPN}, id_c, \mathsf{ECC}, S_{\text{cert}}, Rq, id_c, tok_c, g^s]\!]_{S_{\text{sk}}}$
$sk := \mathsf{KDF}(g^{c \cdot s})$
$\|\mathsf{CreateSession}(id_s, tok_s)\|^{id_c, sk}$
$\|\mathsf{ActivateSession}(id_s, tok_s, U)\|^{id_c, sk}$
$sk := \mathsf{KDF}(g^{c \cdot s})$
$\mathsf{C\_Snd\_Usr\_Req}(C_{\text{cert}}, S_{\text{cert}}, tok_s, U_{\text{cert}}, R_c)$
$\|\mathtt{UserRequest}, tok_s, R_c\|^{id_c, sk}$
$\mathsf{S\_Rcv\_Usr\_Req}(A_{\text{cert}}, S_{\text{cert}}, tok_s, U_{\text{cert}}, R_c)$

Figure 5: Client impersonation attack when SecurityPolicy = ECC and SessionSecurity = SNoAA. Lack of agreement on the client during user Authentication is made possible by the same weakness that allows the KCI attack, *i.e.,* the lack of link to the client.

The weakened agreement properties, Agr^-_S and Agr^-_C (formally defined in Appendix A), are even more complex to prove, due to the conditioning of all residual risks. We were thus not able to prove them for the maximal configurations, and we instead explored some configuration options separately. Regarding Agr^-_S, we were able to obtain proofs without Leak for Reopen + Switch. With Leak, additional residual attacks remain in presence of Reopen (5.4, 5.5) or Switch (5.4). We nevertheless obtained proofs without Reopen nor Switch. Regarding Agr^-_C, we additionally prove it for RSA + Leak + Reopen + Switch (without Enc).

We report all of our results in [14]. Finally, except for the signature oracle attack (C.4.2), all vulnerabilities we report were automatically discovered using ProVerif on our model.

5.1 Race Condition for User Contexts

We discovered a *race condition* after the handover of a session to a new user, that breaks the authentication of user requests. This violation occurs on all versions of OPC UA and does not require any key compromise.

Description. Suppose that an honest client creates a session, that is activated by user U^1 who sends a user request R. Before the request is received (either intercepted by an attacker, or delayed) the session is transferred to user U^2 by a new activation request. After the session transfer, the request R is received by the server and interpreted as a request from U^2, hence violating Agr_S. The detailed message exchange is given in Appendix C.1. A similar attack can be mounted against Agr_C. The attack relies on the fact that a user handover neither modifies the session identifier nor the session token, none of them being bound to the user identity.

Impact. Since a user U^2 may have higher security clearance or privileges than U^1, this attack may lead to an abuse of access rights and privilege escalation. It also leads to wrong security logs and user requests attributions.

Mitigations and fix. The confusion of user context could be avoided by using a fresh *Session Authentication Token* tok_s for each new user, *i.e.,* in each session activation response ([22, Part 4, § 5.6.3.2 table 17]). This would avoid the above attack, but would not be backward compatible.

The OPC Foundation has acknowledged the problem, and plans to include the following mitigations in v.1.05.04 RC (#9351): *(i)* clients should *"create a new session for the new credentials, do the higher privilege operation and close the sessions"*; *(ii) "do not process new requests until activate session completes; any existing requests finish with the current credentials"*. As a result, the privilege escalation exploit discussed above is no longer possible. However, Agr_S and Agr_C are still formally violated due to the lack of agreement on the user's identity (and not only his rights). Therefore, in order to further analyze Agr_S and Agr_C, we had to weaken the properties to allow the user to differ, if there has been a previous activation with another user. The weakened properties are formally defined in Appendix A.

5.2 Client Impersonation in ECC

Our analysis uncovered that the supposedly more secure ECC policy suffers from a client impersonation attack when SessionSecurity = SNoAA that violates Agr_S and Agr_C. The attacker can deceive a server into believing that the user issuing the request is on a different client.

Description. We illustrate the attack against Agr_S in Fig. 5 (detailed flow in Appendix C.2) and suppose an honest client C, a compromised client A (presented as the attacker) and an honest server S.

Name (Section)	Violated Properties	Violation	Threat Assumptions	Configuration Assumptions	Practical Impact	Root Causes	Remediations: Fix, Mitigation (Mit), Enhancement (Enh) and their Tickets
Race Condition for User Contexts (5.1)	$\mathsf{Agr}_{\mathrm{CS}}[U]$	Server S receives a user request allegedly from C,U but U_2 sent it	$\emptyset$	$\emptyset$	Abuse of user rights, wrong logging (user)	• RC1: missing binding btw user and activated session (SAToken tok_s)	• Fix1: bind tok_s to user (change at each activation) • **Mit1: limit new activation to a user with same or lower rights (9351)**
Client Impersonation in ECC (5.2)	$\mathsf{Agr}_{\mathrm{CS}}[C]$	Server S receives a user request allegedly from C^a,U but it comes from C,U	Att(C^a_{sk})	ECC + SNoAA	Confusion (client,user) at server, potential confusion test vs prod. client, wrong client logging	• RC2: lack of receiver identity	• **Fix2: enforce receiver identity in ECC (9349)** • **Enh2: clarify stronger checks in SNoAA (9350, 9427)**
KCI: User Impersonation (5.3)	$\mathsf{Agr}_{\mathrm{S}}^{-0}[U]$	Server S receives a user request allegedly from C^a,U but user U is on client C and did not send it	Att(C^a_{sk}), Att(S_{sk})	Cert	User impersonation	• RC3: lack of binding of user's authentication (user's signature) to full context (client identity)	• Fix3: add client's identity (public key or certificate) in user's authentication signature • **Enh3: plan to release some enhanced security versions of the user tokens (9809, 9810)**
Downgrade of Password Secrecy (5.6)	$\mathsf{Conf}_{\mathrm{Pwd}}$	Compromission of user password	Att(ch_{sk})	Enc + Pwd	Stealing of user password, full user impersonation	• RC4: relaxed security configuration to avoid what may appear to be redundant encryption but is not	• Fix4: use a challenge response mechanism for user password authentication or use proper PAKE protocol • **Enh4: recommend dedicated password encryption, even in mode Enc (9432)**
Risk of Signature Oracle (5.6)	$\mathsf{Conf}_{\mathrm{C}}$, $\mathsf{Conf}_{\mathrm{Pwd}}$, $\mathsf{Agr}_{\mathrm{C}}$	Server impersonation towards honest clients and users in ECC, a less severe UKS attack for RSA	$\emptyset$	ECC + SNoAA	Stealing of user password, full server impersonation	• RC5: lack of context in signatures	• Fix5: add context and tags to the signature • **Enh5: checked well-formedness of certificates (9594, 9596, 9597 and 9598) and nonce length (9595, 9599)**
Session Hijack—Reopen (5.4)	$\mathsf{Agr}_{\mathrm{S}}^{-}[U]$	Server S receives a user request allegedly from C^a,U but user U is on client C and did not send it	Att(C_{sk})	Sign (+ Reopen)	User impersonation	• RC6: lack of binding btw channel symmetric keys before and after renewal (Reopen) • RC7: no session ownership proof (tok_s not secret)	• **Fix6: forbid reopen takeover by linking channel secret keys through renewal (10056)** • Fix7: keep tok_s secret even is Sign mode, and use it as a MAC computed on each request • **Mit2: disable Sign mode by default (9875)**
Session Hijack—Switch (C.3)	$\mathsf{Agr}_{\mathrm{S}}^{-}[U]$	Server S receives a user request allegedly from C^a,U but user U is on client C and did not send it	Att(C_{sk})	Sign (+ Switch)	User impersonation	• RC7: no session ownership proof (tok_s is not secret)	• Fix7: keep tok_s secret even is Sign mode, and use it as a MAC computed on each request • **Mit2: disable Sign mode by default (9875)**
KCI: Session and User Confusion (5.5)	$\mathsf{Agr}_{\mathrm{S}}^{-}[U]$	Server S receives a user request allegedly from C,U but U^d sent it and U^d can be a dummy user without any valid credential	Att(S_{sk})	(Reopen)	User impersonation	• RC6: lack of binding btw channel symmetric keys before and after renewal (Reopen) • RC7: no session ownership proof (tok_s not secret)	• **Fix6: forbid reopen takeover by linking channel secret keys through renewal (10056)** • Fix7: (keep tok_s secret even is Sign mode, and) use it as a MAC computed on each request

Table 3: Summary of the attacks found that affect OPC UA v1.05.03. Additionally, all attacks except those that require ECC also affect OPC UA v1.04. Att(d) denotes the attacker should know or compromise d. We indicate with $\mathsf{Agr}_{\mathrm{CS}}$ when both $\mathsf{Agr}_{\mathrm{C}}$ and $\mathsf{Agr}_{\mathrm{C}}$ are violated (similarly for the weakened variants). For agreement properties, we indicate in square bracket the piece of data on which the disagreement occurs. For instance, $\mathsf{Agr}_{\mathrm{CS}}[U]$ indicates that clients and servers both end up in disagreement on the user U who sent/received a request. Finally, $\mathsf{Agr}_{\mathrm{S}}^{-0}$ denotes the first weakening of $\mathsf{Agr}_{\mathrm{S}}$ (see Appendix A). We indicate in bold font the remediations chosen by the OPC UA Foundation to be included in the specification and in parentheses the tickets that can be found at `https://mantis.opcfoundation.org/view.php?id=<ticket>`.

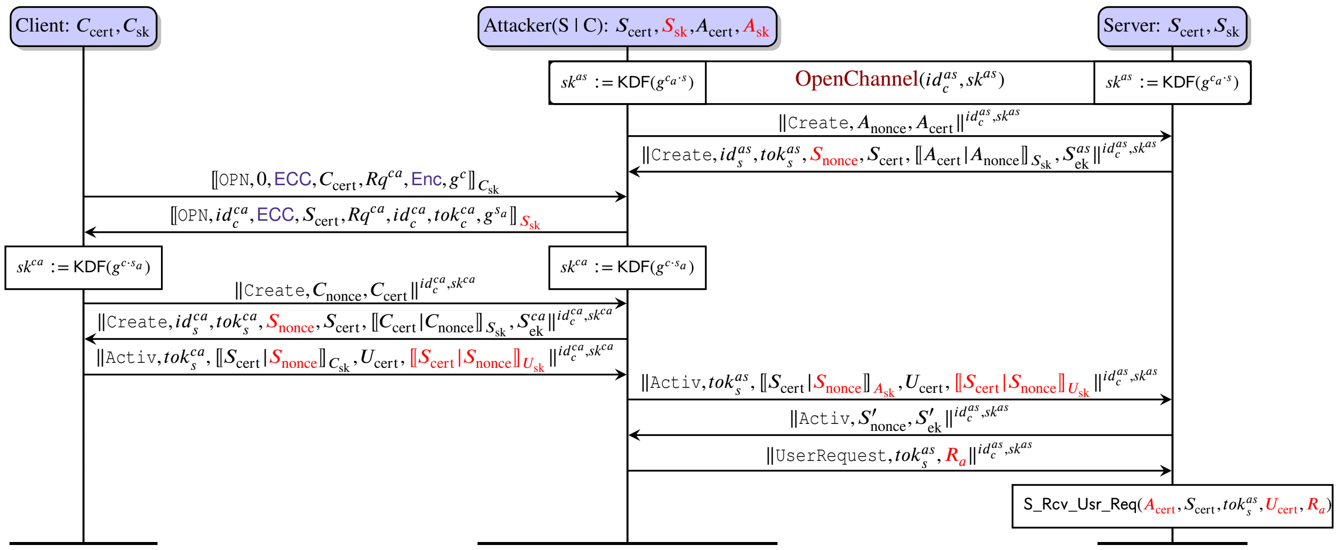

Client: $C_{\mathrm{cert}}, C_{\mathrm{sk}}$
Attacker(S | C): $S_{\mathrm{cert}}, S_{\mathrm{sk}}, A_{\mathrm{cert}}, A_{\mathrm{sk}}$
Server: $S_{\mathrm{cert}}, S_{\mathrm{sk}}$
$sk^{as} := \mathsf{KDF}(g^{c_a \cdot s})$
OpenChannel(id_c^{as}, sk^{as})
$sk^{as} := \mathsf{KDF}(g^{c_a \cdot s})$
$\|\mathtt{Create}, A_{\mathrm{nonce}}, A_{\mathrm{cert}}\|^{id_c^{as}, sk^{as}}$
$\|\mathtt{Create}, id_s^{as}, tok_s^{as}, S_{\mathrm{nonce}}, S_{\mathrm{cert}}, [\![A_{\mathrm{cert}} | A_{\mathrm{nonce}}]\!]_{S_{\mathrm{sk}}}, S_{\mathrm{ek}}^{as}\|^{id_c^{as}, sk^{as}}$
$[\![\mathtt{OPN}, 0, \mathsf{ECC}, C_{\mathrm{cert}}, Rq^{ca}, \mathsf{Enc}, g^c]\!]_{C_{\mathrm{sk}}}$
$[\![\mathtt{OPN}, id_c^{ca}, \mathsf{ECC}, S_{\mathrm{cert}}, Rq^{ca}, id_c^{ca}, tok_c^{ca}, g^{s_a}]\!]_{S_{\mathrm{sk}}}$
$sk^{ca} := \mathsf{KDF}(g^{c \cdot s_a})$
$sk^{ca} := \mathsf{KDF}(g^{c \cdot s_a})$
$\|\mathtt{Create}, C_{\mathrm{nonce}}, C_{\mathrm{cert}}\|^{id_c^{ca}, sk^{ca}}$
$\|\mathtt{Create}, id_s^{ca}, tok_s^{ca}, S_{\mathrm{nonce}}, S_{\mathrm{cert}}, [\![C_{\mathrm{cert}} | C_{\mathrm{nonce}}]\!]_{S_{\mathrm{sk}}}, S_{\mathrm{ek}}^{ca}\|^{id_c^{ca}, sk^{ca}}$
$\|\mathtt{Activ}, tok_s^{ca}, [\![S_{\mathrm{cert}} | S_{\mathrm{nonce}}]\!]_{C_{\mathrm{sk}}}, U_{\mathrm{cert}}, [\![S_{\mathrm{cert}} | S_{\mathrm{nonce}}]\!]_{U_{\mathrm{sk}}}\|^{id_c^{ca}, sk^{ca}}$
$\|\mathtt{Activ}, tok_s^{as}, [\![S_{\mathrm{cert}} | S_{\mathrm{nonce}}]\!]_{A_{\mathrm{sk}}}, U_{\mathrm{cert}}, [\![S_{\mathrm{cert}} | S_{\mathrm{nonce}}]\!]_{U_{\mathrm{sk}}}\|^{id_c^{as}, sk^{as}}$
$\|\mathtt{Activ}, S'_{\mathrm{nonce}}, S'_{\mathrm{ek}}\|^{id_c^{as}, sk^{as}}$
$\|\mathtt{UserRequest}, tok_s^{as}, R_a\|^{id_c^{as}, sk^{as}}$
S_Rcv_Usr_Req($A_{\mathrm{cert}}, S_{\mathrm{cert}}, tok_s^{as}, U_{\mathrm{cert}}, R_a$)

Figure 6: KCI user impersonation attack when SessionSecurity = SSec (illustrated here when SecurityPolicy = ECC and Mode = Enc). Values in a session between the attacker and the server (respectively client) are annotated with ^{as} (respectively ^{ca}). We write OpenChannel(id_c^{as}, sk^{as}) for the channel open request and response messages with id_c^{as} identifier and keys sk^{as}.

C sends an *OpenSecureChannel* request to S. The attacker intercepts this request and modifies the client's identity and resigns the request as coming from A. The server sends its response that the attacker forwards to C, who accepts and opens the channel. Indeed, when SecurityPolicy = ECC, the server's response does *not* include the client's identity. According to [22, Part 6, § 6.7.2.3 table 51] the *ReceiverCertificateThumbprint*, that indicates what public key was used to encrypt the remaining of the message, *shall be null if the message is not encrypted*, which is the case in ECC (but not in RSA, for which the attack is impossible). Therefore, client C is unable to detect that the server's response was intended for A.

At this stage, this already violates authentication of the *OpenSecureChannel* request, which one could state as an intermediary property, as there is disagreement on the client identity between C and S. The attack is also reminiscent of Lowe's attack on the NSPK protocol [19].

We now show the attacker can continue and violate Agr_S. Once the channel is established, the client C will use it to create a session. A user on client C will activate this session and send a user request. These requests will be accepted when SessionSecurity = SNoAA as the server does *not* check that the client certificate corresponds to the one of the channel.

Hence, the server believes that the request received stems form user U on client A while it actually originates from user U on client C. As illustrated by the events at the end of the client and server's execution in Fig. 5, this disagreement violates Agr_S (note that user U is neither compromised nor using passwords). The same attack and disagreement on the client identity does actually also violate Agr_C.

Interestingly, this attack affects the newly introduced ECC but not the "old" and supposedly less secure RSA security policy since the *ReceiverCertificateThumbprint* is not null. It witnesses the fact that cryptographic suites cannot be simply replaced without a thorough security analysis.

Impact. Such an attack could for instance be exploited to create wrong security logs or circumvent user authorization based on the client machine (*e.g.,* Role Based Access Control). Indeed, the server will associate the wrong client-user pair to a user request. A more dramatic example of a potential exploit scenario of the lack of agreement allows an attacker to redirect legitimate (say highly privileged) user U's requests at a training client C intended to a training server, to an in-production SCADA server S, who will accept and log them as coming from a control room client A, that we assume compromised (say decommissioned). This kind of confusion is made possible by the lack of agreement on the pair user-client and a "Non-Transparent Redundancy" server mode. We further explain this and report an additional, less serious Denial-of-service attack (DOS) attack in our full vulnerability report [14].

Mitigation and fix. We proposed to fix this vulnerability by including *ReceiverCertificateThumbprint* (*i.e.,* hash of the receiver's public key) in ECC as well. This fix was accepted by the OPC Foundation and changes in the specification are documented in tickets #9349, #9350 and #9427 including the fix and several clarifications (notably the Conformance Unit 3781). In particular, they updated [22, Part 6] v1.05.04 RC to *"require the ReceiverCertificateThumbprint to be set for all ECC policies"*. Since the

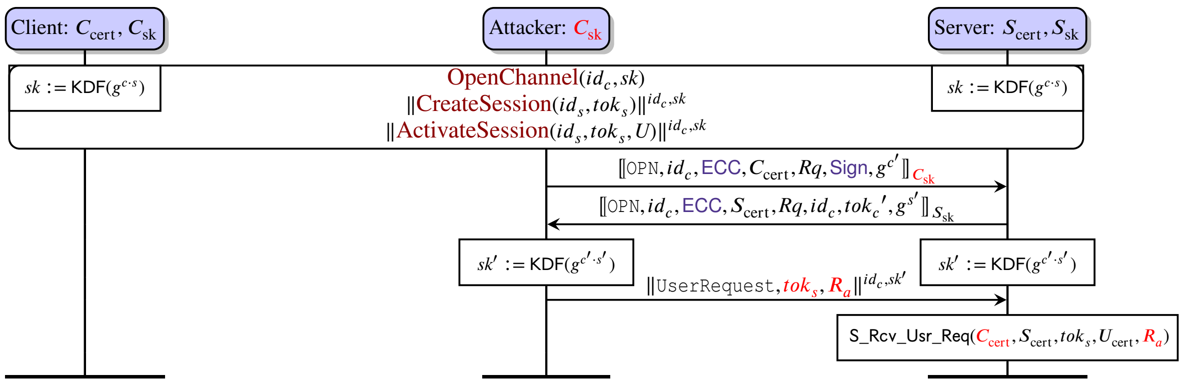

Client: $C_{\text{cert}}, C_{\text{sk}}$
Attacker: C_{sk}
Server: $S_{\text{cert}}, S_{\text{sk}}$
$sk := \mathsf{KDF}(g^{c \cdot s})$
OpenChannel(id_c, sk)
$\|$CreateSession$(id_s, tok_s)\|^{id_c, sk}$
$\|$ActivateSession$(id_s, tok_s, U)\|^{id_c, sk}$
$sk := \mathsf{KDF}(g^{c \cdot s})$
$[\![\texttt{OPN}, id_c, \mathsf{ECC}, C_{\text{cert}}, Rq, \mathsf{Sign}, g^{c'}]\!]_{C_{\text{sk}}}$
$[\![\texttt{OPN}, id_c, \mathsf{ECC}, S_{\text{cert}}, Rq, id_c, tok_c', g^{s'}]\!]_{S_{\text{sk}}}$
$sk' := \mathsf{KDF}(g^{c' \cdot s'})$
$\|\texttt{UserRequest}, tok_s, R_a\|^{id_c, sk'}$
$sk' := \mathsf{KDF}(g^{c' \cdot s'})$
$\mathsf{S\_Rcv\_Usr\_Req}(C_{\text{cert}}, S_{\text{cert}}, tok_s, U_{\text{cert}}, R_a)$

Figure 7: Session hijacking by reopening a channel when Mode = Sign, illustrated here with SecurityPolicy = ECC and SessionSecurity = SSec.

issue is corrected in the specification, we included the fix in our model to show its effectiveness. In particular, while ProVerif finds an attack against Agr^-_{C} on v1.05.03 for configuration (ECC + Enc + SNoAA + Cert + Leak), it proves v1.05.04 RC secure for the same property and configuration.

5.3 KCI: User Impersonation

While key compromise allows to trivially impersonate the compromised party, a Key Compromise Impersonation (KCI) attack allows to impersonate an uncompromised party to the compromised party. Modern protocols are designed to resist such attacks. For instance, the TLS 1.3 specification explicitly lists resistance to KCI attacks as one of its goals [25, App. E] for certificate-based authentication. Yet, we have discovered a KCI attack that allows to impersonate an honest user on an honest client to a server whose keys have been leaked.

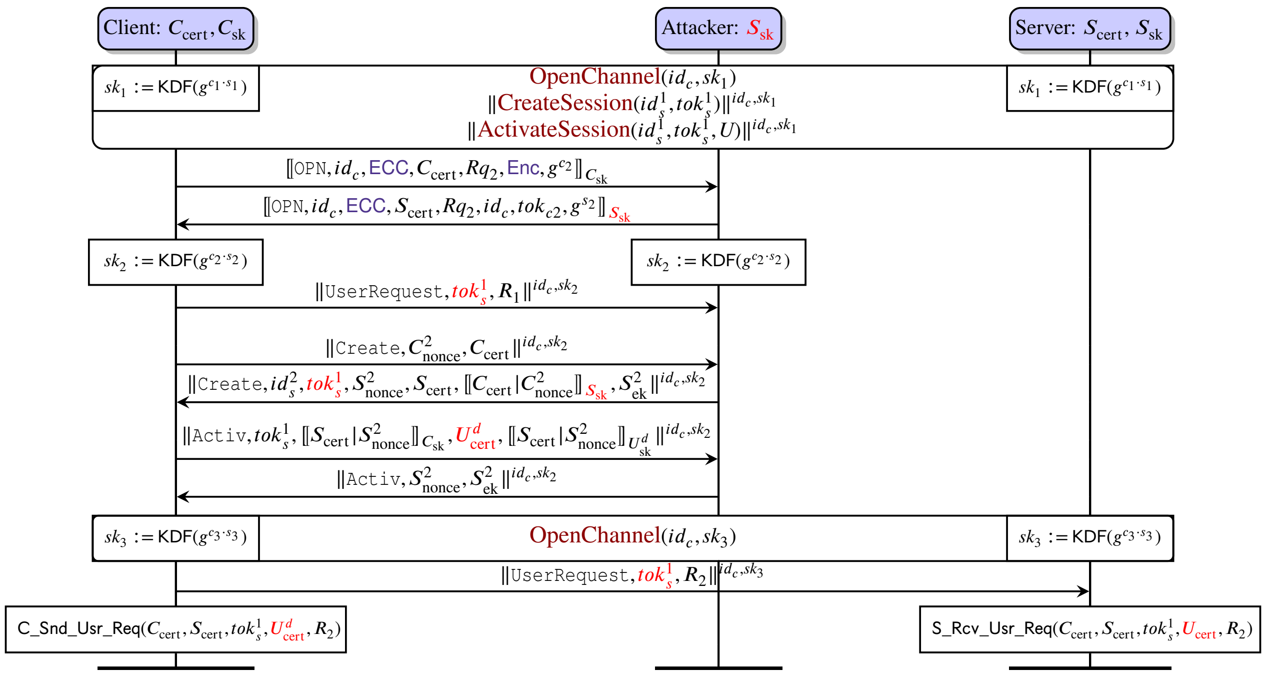
We note that password-based authentication is broken in OPC UA as soon as a user connects to a server whose keys have been compromised since clients send user's passwords to the server. Certificate based authentication is supposed to resist this since it involves a signature by the user's private key (*e.g.,* computed on a smart card) of a nonce generated by the server. The attack we found shows the contrary: a user can be impersonated even when using a certificate to authenticate.

Description. The attack involves an uncompromised client C, an uncompromised user U, and a compromised server S. We also suppose that the keys of another client C^A have been compromised. We explain the attack when SecurityPolicy = ECC and Mode = Enc. The detailed message flow is provided in Fig. 6.

1. The attacker opens a channel between C^A and S and creates a session se_1 between C^A and the honest server S, in which S generates the nonce S_{nonce} (see Fig. 4).
2. C sends an Open Channel Request to S that is intercepted by the attacker. Knowing S's long-term keys, the attacker forges and sends back to C a valid response. Hence, the attacker also knows the channel keys.
3. C sends a Create Session Request to S. The attacker intercepts it. Knowing S's long-term keys and the channel keys, the attacker can impersonate the server and forges a valid Response. Moreover, the attacker uses the same server nonce S_{nonce} as in sessions se_1. C records this session se_2 in its local state.
4. User U on client C sends an activation request for se_2 to S. The attacker intercepts it and extracts the signature $[\![S_{\text{cert}}|S_{\text{nonce}}]\!]_{U_{\text{sk}}}$ created by U to authenticate, where U_{sk} is the private key associated to U_{cert}.
5. The attacker activates session se_1 using U's signature $[\![S_{\text{cert}}|S_{\text{nonce}}]\!]_{U_{\text{sk}}}$. As the server S has created S_{nonce} for session se_1, this session creation is accepted by S.
6. The attacker can now send arbitrary user requests in se_1 from client C^A without any further intervention from U. Note that the user U had no interaction with client C^A.

This attack violates Agr_{S} as a server accepts a user U's request without the user having sent any request.

Impact. Assuming long-term keys S_{sk} of server S were compromised, an attacker can send arbitrary user requests to the legitimate server S on behalf of a user U, even if U authenticates with an uncompromised certificate on an uncompromised client (*e.g.,* using state-of-the-art authentication on a smart card), and even if U did not send any user request. We stress that we solely assume that S_{sk} are compromised (*e.g.,* through a flawed certificate management/renewal or flawed certificate storage/generation)



Client: $C_{\text{cert}}, C_{\text{sk}}$
Attacker: S_{sk}
Server: $S_{\text{cert}}, S_{\text{sk}}$
$sk_1 := \mathsf{KDF}(g^{c_1 \cdot s_1})$
OpenChannel(id_c, sk_1)
$\|$CreateSession$(id_s^1, tok_s^1)\|^{id_c, sk_1}$
$\|$ActivateSession$(id_s^1, tok_s^1, U)\|^{id_c, sk_1}$
$sk_1 := \mathsf{KDF}(g^{c_1 \cdot s_1})$
$[\![\mathtt{OPN}, id_c, \mathsf{ECC}, C_{\text{cert}}, Rq_2, \mathsf{Enc}, g^{c_2}]\!]_{C_{\text{sk}}}$
$[\![\mathtt{OPN}, id_c, \mathsf{ECC}, S_{\text{cert}}, Rq_2, id_c, tok_{c2}, g^{s_2}]\!]_{S_{\text{sk}}}$
$sk_2 := \mathsf{KDF}(g^{c_2 \cdot s_2})$
$sk_2 := \mathsf{KDF}(g^{c_2 \cdot s_2})$
$\|\mathtt{UserRequest}, tok_s^1, R_1\|^{id_c, sk_2}$
$\|\mathtt{Create}, C_{\text{nonce}}^2, C_{\text{cert}}\|^{id_c, sk_2}$
$\|\mathtt{Create}, id_s^2, tok_s^1, S_{\text{nonce}}^2, S_{\text{cert}}, [\![C_{\text{cert}} | C_{\text{nonce}}^2]\!]_{S_{\text{sk}}}, S_{\text{ek}}^2\|^{id_c, sk_2}$
$\|\mathtt{Activ}, tok_s^1, [\![S_{\text{cert}} | S_{\text{nonce}}^2]\!]_{C_{\text{sk}}}, U_{\text{cert}}^d, [\![S_{\text{cert}} | S_{\text{nonce}}^2]\!]_{U_{\text{sk}}^d}\|^{id_c, sk_2}$
$\|\mathtt{Activ}, S_{\text{nonce}}^2, S_{\text{ek}}^2\|^{id_c, sk_2}$
$sk_3 := \mathsf{KDF}(g^{c_3 \cdot s_3})$
OpenChannel(id_c, sk_3)
$sk_3 := \mathsf{KDF}(g^{c_3 \cdot s_3})$
$\|\mathtt{UserRequest}, tok_s^1, R_2\|^{id_c, sk_3}$
C_Snd_Usr_Req$(C_{\text{cert}}, S_{\text{cert}}, tok_s^1, U_{\text{cert}}^d, R_2)$
S_Rcv_Usr_Req$(C_{\text{cert}}, S_{\text{cert}}, tok_s^1, U_{\text{cert}}, R_2)$

Figure 8: Session confusion illustrated here with Mode = Enc, SecurityPolicy = ECC and SessionSecurity = SSec.

but do not assume that the server is fully compromised. In particular, the server S could be protected with additional layers of security and can have access to high-privileged data and actions that the attacker does not have access to.

Mitigation and fix. We proposed to fix this vulnerability by adding the client's identity to the user's signature, *i.e.,* $[\![C_{\text{pk}} | S_{\text{cert}} | S_{\text{nonce}}]\!]_{U_{\text{sk}}}$, where C_{pk} is the public key of the client's certificate. Such a fix would however break backwards compatibility of systems in production. The OPC Foundation acknowledged the attack, but considered the compromise of server long-term keys to be a very strong threat model that should be accepted for now, given the cost of a non-backward compatible fix. In the mid-term, they want to consider to *"release some enhanced security versions of the user tokens"*, notably using ideas from `RFC 8705` (*e.g.,* CertificateBound JWTs); see the associated tickets #9809 and #9810.

To further analyze OPC UA, we had to weaken $\mathsf{Agr}_{\mathsf{S}}^{-}$ to accept this residual risk, and to not deem as attacks the executions where a client is compromised and an honest user activates a session towards a compromised server (see Appendix A).

5.4 Session Hijack by Reopening or Switching

Even though the weakened property $\mathsf{Agr}_{\mathsf{S}}^{-}$ purposely avoids the previous attacks, we were able to uncover yet another attack when channels only provide integrity, *i.e.,* Mode = Sign.

Description. The attack involves a compromised client C, an uncompromised user U and an uncompromised server S. The detailed message flow is provided in Fig. 7.

1. Client C (whose key C_{sk} has been compromised) opens a channel in Mode = Sign and creates a session with S. The attacker learns the session token tok_s as only integrity is provided. (Note that although the attacker knows C_{sk}, it does not know the channel keys sk.)
2. Honest user U on client C activates the session using its user credentials (which are not leaked to the attacker).
3. Using C_{sk} the attacker re-opens the channel with S and learns the updated channel keys sk'.
4. Knowing sk' and tok_s, the attacker can send arbitrary user requests impersonating U.

Session Hijack by Switching variant. Note that a variant of this attack relies on switching the channel of a session, rather than reopening the channel. In the above attack, a channel reopening request was issued by the attacker to gain access to the channel underlying the session. Alternatively, an attacker can intercept a session re-activation, open a new channel, and replay the re-activation attack on the new (attacker controlled) channel (see Appendix C.3 for details).

Impact. Assuming that a client's long-term is compromised, an attacker can completely hijack the session of a user who activated a session on that client. The attacker can therefore send arbitrary requests on behalf of U.

Mitigation. This attack has also been acknowledged by the OPC Foundation. As a counter-measure the foundation considers deactivating the mode Mode = Sign by default (tickets #9874, #9875). For a future version, the foundation also considers chaining channel keys through reopening (ticket #10056), which requires knowledge of previous channel keys to initiate a Reopen.

5.5 KCI: Session and User Confusion

As a response to our Session Hijack attacks, the OPC Foundation decided to disable Mode = Sign by default. However, even when Mode = Enc, we found another KCI attack that violates the weakened property $\mathsf{Agr}^{-}_{\mathsf{S}}$, which excluded the previous KCI attack (Section 5.3). We shall also assume a compromise of a server's long-term key. Recall that an impersonation of this server is obviously possible. What we additionally show is an impersonation of honest users towards that honest server.

Description. The attack involves an honest client C and user U, a compromised server S. We additionally assume the adversary has normal access to C but is not necessarily registered in the user database. The detailed message flow is provided in Fig. 8.

1. User U on client C opens a channel, creates and activate a session s_1 with server S.
2. When C renews the channel keys (Reopen), the attacker impersonates S and learns the new channel keys sk_2. He will then learn tok^1_s for session s_1.
3. The adversary initiates a new session s_2 creation and activation on C with a *dummy user* U^d (U^d does not have to be a valid, registered user). The adversary continues impersonating S: it proceeds and accepts those requests and uses the exact same tok^1_s from s_1. C has now registered two sessions s_1 (with U) and s_2 (with U^d), both with the same tok^1_s. Note that S is neither aware of session s_2 nor of user U^d.
4. C will eventually reopen the channel with the honest server S: the attacker can pretend to lose the connection, or can wait for the channel to be reopened or switched.
5. From now on, when C sends requests from U^d to S in session s_2, it uses tok^1_s, which causes S to believe this request is coming from the honest user U in session s_1.

Impact. The impact is similar to the previous KCI attack (Section 5.3). Assuming long-term keys of server S were compromised, an attacker having unprivileged access to an honest client C can send arbitrary user requests to the legitimate server S on behalf of an honest, possibly high-privileged, user U, who previously activated a session on that client. Without assuming having access to C, a variant of this attack where U^d is a valid, honest user is still possible and has a similar impact as the Race Condition attack (Section 5.1): confusion of user context for user requests. Note that the attack is also possible when Mode = Sign.

Mitigation. This attack has also been acknowledged by the OPC Foundation. As a counter-measure the foundation considers chaining channel keys through reopening (ticket #10056).

5.6 Other Findings

We report on additional attacks and weaknesses in Appendix C.4 that we briefly summarize here.

Downgrade of Password Secrecy. Our analysis uncovered that, when channel keys are leaked, counter-intuitively, Mode = Enc is less secure than Mode = Sign since it leaks user passwords. This is due to a relaxed security requirement in the specification that allows to send passwords in plaintext when Mode = Enc since they are under channel encryption anyway. Channel keys can be considered less secure than long-term keys in some scenarios (e.g., when stored in memory while long-terms keys are stored in Hardware Security Module (HSM)s).

The OPC Foundation acknowledged this issue and decided to add a warning and a recommendation to always encrypt passwords (see ticket #9432).

Risk of Signature Oracle. A server configured to allow SNoAA may not verify the certificate sent by a client application at session creation, but simply append the client nonce and sign the resulting bitstring, to prove possession of its private key. This induces a "signature oracle" to produce valid server signatures on adversarially-chosen data. In particular, this signature oracle can be exploited to forge valid *OpenChannel* response and *CreateSession* responses, allowing to fully impersonate an honest server towards an honest client in ECC. Hence, all security goals are violated and the attacker can learn users' passwords. For RSA, the attack is less severe as the attacker does not learn the channel keys and can only perform an Unknown Key Share (UKS) attack.

The OPC Foundation acknowledged the weakness, and to mitigate the vulnerability, *(i)* adds explicit requirements for v.1.05.04 RC, and conformance tests to ensure that certificates are validated before being used to produce a signature (tickets #9594, #9596,

#9597 and #9598); *(ii)* states that all Nonces shall have a length specified by the SecurityPolicy (tickets #9595, #9599). We reflect those fixes in our model and prove that they thwart the attack we found on the same configuration.

5.7 Lessons Learned

First, our results confirm that, without key compromise, OPC UA satisfies all security goals except Agr_{S},Agr_{C} due to the Race Condition attack (5.1) (Agr^{-}_{S},Agr^{-}_{C} hold nevertheless). This mostly aligns with prior analyses by the BSI [13,30]. Our work extends beyond this baseline by considering strong attacker models including key leakage that are *also* in scope of the specification's threat model (Section 2.3).

Within that scope, our analysis revealed several attacks and design weaknesses. Our analysis highlights that many attacks exploit *(i)* the multi-layer nature of the protocol: channels, created and activated sessions, requests, *(ii)* the agility of the protocol due to OT constraints: switch/reopen channels, re-activate sessions, etc., *(iii)* the complexity of the protocol: combination of multiple options and configurations. We believe that this protocol complexity is not inherently insecure, but requires a careful analysis. Several protocol design flaws are the root causes of our attacks and are often related to missing bindings between critical components, such as (see Table 3 for a detailed list): activated sessions are not bound to their user (5.1), signed requests are not bound to the recipient (5.2), signed user tokens are not bound to the client (5.3), new channel keys obtained by reopening are not bound to previous keys (5.4, Appendix C.3, 5.5), and the absence of a context binding in signatures (C.4.2). This should be remembered when designing future protocol evolutions.

Our analysis also highlights the pitfalls of adding new features on top of an existing protocol. For instance, the addition of ECC by simply replacing a cryptographic primitive suite by another one has led to a quite subtle attack (5.2).

Finally, our interaction with the OPC Foundation demonstrates the importance of proactive reporting and collaboration between researchers and standardization bodies to enhance the resilience of critical infrastructures. We were able to adapt our models and analyses to the proposed fixes that were validated by the OPC Foundation.

6 Related Work

Formal symbolic verification, using tools such as ProVerif and Tamarin, has been used on many deployed and standardized protocols, including TLS [4,5,11], EMV [3], 5G [2,10], Bluetooth [31], LAKE-EDHOC [18] or WiFi [26]. We continue this line of work, and exploit several of the latest features of ProVerif [7] to enable what we believe to be among the most complex formal analyses. While several of these works also generate different protocol configurations, there are key differences. For instance, Wu *et al.* [31] use a modular encoding that allows to easily select a given subprotocol of each kind out of many: this results in a large number of rather small models (200 LoC for the largest unfolded process and 100 initial clauses vs 8.6 kLoC and 2.3 k clauses in our model); moreover, as these models do not consider key compromise (as those of Shen *et al.* [26]), their verification was fully automatic, and did not face the challenges we needed to address (such as providing lemmas, or fine-tuning of the selection function; see Section 4.3).

Puys *et al.* [24] previously used ProVerif to analyze OPC UA v1.03, that does not include ECC. Moreover, their models are minimalistic and only consider channel opening and user sessions without analyzing their interaction, omitting many details and configurations studied here. This is reflected by <100 LoC and <100 initial clauses for each of the (separate) models of channel and session protocols.

A quite exhaustive security analysis of OPC UA v1.04 was performed by the German BSI [30]. It includes a risk analysis, a dynamic security analysis through fuzzing and static code analysis of the open62541 implementation. Deployed OPC UA products have also been studied through large-scale Internet measurements [12] and an analysis of products and libraries for OPC UA [16]: these works demonstrate a large number of misconfigured OPC UA artifacts (*e.g.,* certificate checks disabled for testing purpose) enabling attacks that OPC UA was designed to resist (*e.g.,* rogue server); often these misconfigurations were due to unclear instructions or implementations that rely on incomplete libraries. In contrast, our attacks affect even a perfectly well-configured OPC UA protocol deployment. Being performed on version 1.04, these analyses do not include the ECC policy. Moreover, these analyses do not include any security proofs, and we consider them as complementary to our formal verification effort.

Finally, note that six, out of the height vulnerabilities we found, already affected versions 1.03 and 1.04, and yet had not been discovered by these previous efforts.

7 Conclusion

We proposed a verification framework for OPC UA based on a comprehensive protocol description, a ProVerif model using a number of advanced ProVerif features and a new proof methodology, and some tooling to mitigate the complexity of the proof effort. Our framework automatically discovered several vulnerabilities that have all been acknowledged by the OPC Foundation. In response, the specification has been updated with fixes and mitigations that we had analyzed.

We believe that our framework can serve as a starting point for other researchers and practitioners working with OPC UA (notably Section 5.7). Our proof methodology may also serve for other ambitious case studies whose automated verification do not work "out-of-the-box". As discussed, the protocol residual risks induced additional analysis challenges, especially for proving the agreement properties. Should those risks be accepted by the Foundation in the longer term, we plan to further improve our proof methodology and develop dedicated invariants to obtain complete proofs for maximal configurations.

8 Ethics Considerations and Compliance with the Open Science Policy

8.1 Ethics Considerations

Our paper describes vulnerabilities and weaknesses in a deployed security protocol, OPC UA. Some findings only apply to the latest version of this protocol, that is not yet deployed (version 1.05). As discussed in the paper, all findings have been *responsively disclosed* to the OPC Foundation. To do so, we used the Foundation's reporting tool: `https://mantis.opcfoundation.org`. For each vulnerability, a private ticket has been issued, and the vulnerabilities have been discussed during the embargo period. We also attended several meetings of the working group for further discussion. The resulting changes to the specification have been documented in public tickets that are cited in the paper.

The vulnerability report that we transmitted to the OPC Foundation for *responsible disclosure* is in the file `vulnerabilities.pdf` [14]. It provides details on the attacks that we present in Section 5. Moreover, Table 3 gives further details on root causes and potential fixes.

No other ethics considerations have been identified.

8.2 Open Science

As mentioned in the paper, our models are available in a companion artifact [14], that includes the vulnerability report, all ProVerif models, the dependency graph between properties, and the necessary scripts to reproduce the results from Section 5.

The `README.md` **file** provides detailed instructions on how to *(i)* install ProVerif, *(ii)* configure the Python environment for the scripts, *(iii)* compute attack traces as PDF files, *(iv)* prove the security properties in maximal configurations, *(v)* launch lattice exploration campaigns.

The Python script `opcua.py` (configured by `config.py`) takes Jinja2 template files `opcua-jinja.pv` and `config-jinja.pvl` to *(i)* produce the `tmp_opcua.pv` and `tmp_config.pvl` ProVerif input files according to the selected query and configuration, and *(ii)* launch ProVerif on these files.

The former template file `opcua-jinja.pv` contains the complete ProVerif model of the protocol, all queries corresponding to the security properties and the invariants. Advanced features described in Section 4.3 such as *Name* contexts, noselect statements and *invariants* may vary depending on the selected query. The latter template file `config-jinja.pvl` is used for configuring the model, as explained in Section 3.2, Section 4.1 and Table 1. It may also modify the behavior of ProVerif to prevent attack reconstruction or make the output more verbose.

As an example of attack reconstruction, the attack 5.4 session hijack by reopening, that breaks Agr_S, is found with the command: `python3 opcua.py -q "3.1.reopen" -c "ECC, Sign, reopen, SSec, cert, no_switch, lt_leaks" -html` and the detailed trace can be observed in the file `output/trace1.pdf`.

As an example of a security property proof, the confidentiality of user requests Conf_C is proved for SecurityPolicy = ECC without Leak by: `python3 opcua.py -q "Conf[C]" -c "ECC, None|Sign|Encrypt, reopen, SNoAA|SSec, anon|pwd|cert, switch, no_leaks"`.

The novel proof methodology presented in Section 4.3.2 requires, for the proof of a formal property to be sound, all its dependencies to be also proven. The dependency graph of our formal model is provided in the file `dependencies.txt`.

As an example of an advanced proof, the weakened authentication of user responses (property Agr_C^-), stated as query "3.2", relies on invariants "3.2.axioms", "3.2.A", "3.1.A" and "3.1.C". Each of these needs to be proven separately for the same configuration: `python3 opcua.py -q "3.2" -c "ECC, Encrypt, no_reopen, SSec, cert, no_switch, lt_leaks"` and then `python3 opcua.py -q "3.2.axioms" -c "ECC, Encrypt, no_reopen, SSec, cert, no_switch, lt_leaks"` etc. The script `prove.sh` (see below) allows to prove such a conjunction of properties.

The systematic exploration of configurations mentioned in Section 4.2, requires the script `prove.py` (that imports `configurations.py`). It provides the following lists: *(i)* the maximal true configurations, for which the property holds, *(ii)* the minimal false configurations, *(iii)* the minimal configurations for which ProVerif did not finished.

The bash script `prove.sh` starts a new campaign that iteratively increases the time budget for computations. A complete exploration of $\mathit{Conf}_{\mathrm{C}}$ is launched (preferably on a server for which long runs can be launched) by: `./prove.sh "Conf[C]" "RSA, None|Sign|Encrypt, reopen, SNoAA|SSec, anon|pwd|cert, switch, lt_leaks"`. A restart from a previous campaign (Appendix D.1), for example from the log file `query_Conf[C]_2560.txt` (2560 is the timeout in seconds used at the last step) is launched with a doubled timeout by: `./prove.sh "Conf[C]" "RSA, None|Sign|Encrypt, reopen, SNoAA|SSec, anon|pwd|cert, switch, lt_leaks" 5120 "query_Conf[C]_2560.txt"`. Timeouts for future steps will then be automatically doubled, and handled by the `prove.sh` script. Note that for agreement properties, it is possible to explore proving the conjunction of $\mathit{Agr}^{-}_{\mathrm{S}}$ and all its axioms (as queries) with the query argument "3.1.all" (and "3.2.all" for $\mathit{Agr}^{-}_{\mathrm{C}}$).

The file `results.md` presents the maximal true configurations for each security property.

Acknowledgments

This work benefited from funding managed by the French National Research Agency under the France 2030 program (ANR-22-PECY-0006) and the ANR chair in AI ASAP (ANR-20-CHIA-0024) with support from the region Grand Est. We thank Alexandre Debant and Vincent Cheval, for in-depth discussions about ProVerif proofs. We also thank Randy Armstrong and the OPC UA Foundation for the discussions about potential fixes.

References

[1] M. Baezner and P. Robin. Stuxnet. http://hdl.handle.net/20.500.11850/200661, 2017.

[2] D. A. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler. A formal analysis of 5G authentication. In *Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[3] D. A. Basin, R. Sasse, and J. Toro-Pozo. The EMV standard: Break, fix, verify. In *Symposium on Security and Privacy (S&P)*. IEEE, 2021.

[4] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[5] K. Bhargavan, V. Cheval, and C. A. Wood. A symbolic analysis of privacy for TLS 1.3 with encrypted client hello. In *Conference on Computer and Communications Security (CCS)*. ACM, 2022.

[6] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop (CSFW)*. IEEE, 2001.

[7] B. Blanchet, V. Cheval, and V. Cortier. Proverif with lemmas, induction, fast subsumption, and much more. In *Symposium on Security and Privacy (S&P)*. IEEE, 2022.

[8] S. Brizinov and N. Moshe. Exploiting OPC-UA in Every Possible Way: Practical Attacks Against Modern OPC-UA Architectures. https://www.blackhat.com/us-23/briefings/schedule/}exploiting-opc-ua-in-every-possible-way-practical-attacks-against-modern-opc-ua-architectures-31535, 2023.

[9] P. Cheremushkin and S. Temnikov. OPC UA security analysis. https://ics-cert.kaspersky.com/publications/reports/2018/05/10/opc-ua-security-analysis/, 2018.

[10] C. Cremers and M. Dehnel-Wild. Component-based formal analysis of 5g-aka: Channel assumptions and session confusion. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[11] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[12] M. Dahlmanns, J. Lohmöller, I. B. Fink, J. Pennekamp, K. Wehrle, and M. Henze. Easing the conscience with OPC UA: an internet-wide study on insecure deployments. In *Internet Measurement Conference (IMC)*. ACM, 2020.

[13] Damm, Gappmeier, Zugfil, Plöb, Fiat, and Störtkuhl. OPC-UA security analysis. Technical report, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2017.

[14] V. Diemunsch, L. Hirschi, and S. Kremer. Submission artifacts. `https://archive.softwareheritage.org/swh:1:rev:1528b4a1dbd05e4e509dabdfd8d7f4a0cd3a6dac;origin=https://github.com/vdh-anssi/opc-ua_security`, 2025.

[15] D. Dolev and A. Yao. On the security of public key protocols. *IEEE transactions on information theory*, IT-29(2), 1983.

[16] A. Erba, A. Müller, and N. O. Tippenhauer. Security analysis of vendor implementations of the OPC UA protocol for industrial control systems. In *Workshop on CPS & IoT Security and Privacy (CPSIOTSEC@CCS)*. ACM, 2022.

[17] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. Basin. A spectral analysis of noise: A comprehensive, automated, formal analysis of Diffie-Hellman protocols. In *USENIX Security Symposium*, 2020.

[18] C. Jacomme, E. Klein, S. Kremer, and M. Racouchot. A comprehensive, formal and automated analysis of the EDHOC protocol. In *USENIX Security Symposium*, 2023.

[19] G. Lowe. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1996.

[20] G. Lowe. A hierarchy of authentication specifications. In *Computer Security Foundations Workshop (CSFW)*. IEEE Comput. Soc. Press, 1997.

[21] S. Obermeier, R. Schlegel, and J. Schneider. A security evaluation of IEC 62351. In *Journal of Information Security and Applications*, 2016.

[22] OPC Foundation. OPC Unified Architecture Specification. `https://reference.opcfoundation.org/`, 2023. Release 1.05.03.

[23] OPC Foundation. OPC Unified Architecture Specification. `https://profiles.opcfoundation.org/profile/`, 2024. Release 1.05.

[24] M. Puys, M.-L. Potet, and P. Lafourcade. Formal Analysis of Security Properties on the OPC-UA SCADA Protocol. In *Computer Safety, Reliability, and Security (SAFECOMP)*, volume 9922 of *LNCS*. Springer, 2016.

[25] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018.

[26] Z. Shen, I. Karim, and E. Bertino. Segment-based formal verification of wifi fragmentation and power save mode. In *Asia Conference on Computer and Communications Security, (ASIA CCS)*. ACM, 2024.

[27] K. Stouffer, M. Pease, C. Tang, T. Zimmmerman, V. Pillitteri, S. Lightman, A. Hahn, S. Saravia, A. Sherule, and M. Thompson. Guide to Operational Technology (OT) Security. `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r3.pdf`, 2023.

[28] Systerel. A safe and secure OPC UA implementation. `https://www.s2opc.com/`, 2023. Version 1.3.1.

[29] Team82 Research. OPC UA Deep Dive: A Complete Guide to the OPC UA Attack Surface. `https://claroty.com/team82/research/opc-ua-deep-dive-a-complete-guide-to-the-opc-ua-attack-surface`, 2023.

[30] J. vom Dorp, S. Merschjohann, D. Meier, F. Patzer, M. Karch, and C. Haas. OPC-UA security analysis. Technical report, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2022.

[31] J. Wu, R. Wu, D. Xu, D. J. Tian, and A. Bianchi. Formal model-driven discovery of bluetooth protocol design vulnerabilities. In *Symposium on Security and Privacy (S&P)*. IEEE, 2022.

A Weakened Properties

Weakening to avoid race condition. As a result of the changes in the specification detailed in Section 5.1, the privilege escalation due to the race condition is mitigated, but the agreement properties Agr_S, and Agr_C are still violated. We therefore weaken Agr_S as follows:

```
inj-event(S_Rcv_Usr_Req(C_pk, S_pk, SE, U, R))
 ==> ( inj-event(C_Snd_Usr_Req(C_pk, S_pk, SE, U', R))
       && event(S_Activation(C_pk, S_pk, SE', U')) )
  || (event(leak(C_pk)) || mode = None) &&
       ( U = anon || event(leak(U)) ||
           event(C_Activ_req(C'_pk,S'_pk,mode,check,SE',U,pwd))
        && event(leak(S'_pk)) )
```

where SE' is the same as SE, except for the secure channel. Hence, we explicitly tolerate a mismatch on the user if this user has sent an activation request to the server (as in the race condition). Allowing the channel to differ in the sessions is needed as channels may be reopened. Agr_C can be weakened in an analogous way (see [14]).

Weakening to avoid the user impersonation KCI attack. As the first KCI attack (Section 5.3) has currently not been fixed, we weaken Agr_S to exclude this attack and further analyze the protocol. Namely, we add a disjunct that excludes the combination of server's and client's long-term keys compromise. We additionally need to take into account other possible exploits of the KCI attacks that open up new ways to compromise user's authentication tokens. As a result, the condition

```
|| (event(leak(C_pk)) || mode = None) &&
     ( U = anon || event(leak(U)) ||
         event(C_Activ_req(C'_pk,S'_pk,mode,check,SE',U,pwd))
      && event(leak(S'_pk)) )
```

is replaced by

```
|| (event(leak(C_pk)) || mode = None) &&
     ( U = anon || event(leak(U)) ||
      event(leaked_server(rogue_server)) ||
      event(C_Activ_req(C'_pk,S'_pk,mode3,check,SE',U,u_tk))
        && (mode3 = None || mode3 = Sign)
      )
```

Indeed, if the client is compromised and there is a rogue server (*i.e.,* a compromised server), the attacker could use it to inject a S_nonce, honestly generated by an honest server, to obtain and steal an authentication token from an uncompromised user. Note that this now replaces the previous condition for the case u_tk=pwd. Similarly, if a user authenticates using a client in a channel in mode Sign or None, an attacker may act as a MiM and eavesdrops on the signed user's authentication token.

We define Agr_S^- to be the property obtained by weakening Agr_S to both avoid the race condition and the first KCI attack. Similarly, Agr_C^- corresponds to the weakening of Agr_C to avoid the race condition. (The KCI attacks do not violate Agr_C).

We do not claim Agr_S^- is "optimal", in the sense that it would perfectly capture the residual risks; it could be an over-approximation since the residual risks are hard to define. However, Agr_S^- nonetheless remains a sound approximation of a model that would consider that "all is lost" as soon as there is a rogue server. Supporting this is the fact that we later found another KCI attack (*i.e.,* KCI: Session and User Confusion, see Section 5.5) that violates Agr_S^- despite being a KCI attack too. The crux of this new attack is that it does not assume any client compromise, but only a server compromise.

B OPC UA Protocol Description

B.1 On Reliability Constraints

We mentioned in Section 2.1 that some of the complexity of the OPC UA state machine was due to OT inherent constraints. We now illustrate such constraints and their rationale with some use cases.

At shift change in a control room, an operator may want to hand its session over to his colleague that will take its place at the same workstation, so that all his work done to set up the monitoring of an ongoing operation be not lost, but could be directly used by its replacement. Therefore, OPC UA allows a new activation of an already activated session, by a new user on the same secure channel.

In the event of a network outage, the connection, and hence the secure channel, between a client and a server can be lost. When network connection is restored, the client can open a new secure channel and let the user reuse its current session on the server

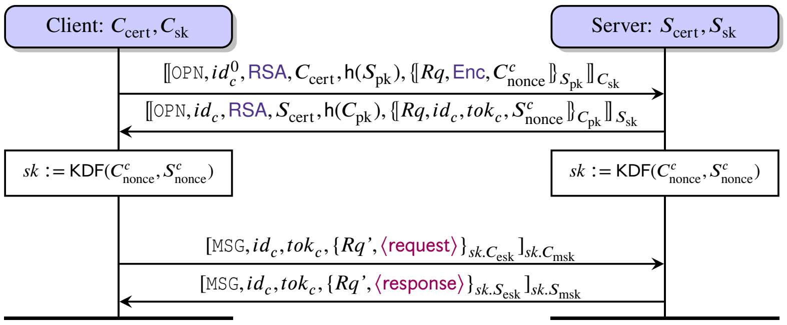

Client: $C_{\text{cert}}, C_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
$\llbracket \mathtt{OPN}, id_c^0, \mathsf{RSA}, C_{\text{cert}}, \mathsf{h}(S_{\text{pk}}), \{\!| Rq, \mathsf{Enc}, C^c_{\text{nonce}} |\!\}_{S_{\text{pk}}} \rrbracket_{C_{\text{sk}}}$
$\llbracket \mathtt{OPN}, id_c, \mathsf{RSA}, S_{\text{cert}}, \mathsf{h}(C_{\text{pk}}), \{\!| Rq, id_c, tok_c, S^c_{\text{nonce}} |\!\}_{C_{\text{pk}}} \rrbracket_{S_{\text{sk}}}$
$sk := \mathsf{KDF}(C^c_{\text{nonce}}, S^c_{\text{nonce}})$
$sk := \mathsf{KDF}(C^c_{\text{nonce}}, S^c_{\text{nonce}})$
$[\mathtt{MSG}, id_c, tok_c, \{Rq', \langle\mathsf{request}\rangle\}_{sk.C_{\text{esk}}}]_{sk.C_{\text{msk}}}$
$[\mathtt{MSG}, id_c, tok_c, \{Rq', \langle\mathsf{response}\rangle\}_{sk.S_{\text{esk}}}]_{sk.S_{\text{msk}}}$

Figure 9: (Re)Open Secure Channel in (RSA + Enc). Except for the first two messages, the protocol is the same as in ECC Fig. 3. We let S_{pk} (respectively C_{pk}) be $\mathsf{pk}(S_{\text{cert}})$ (respectively $\mathsf{pk}(C_{\text{cert}})$) where $\mathsf{pk}(c)$ returns the public key associated with the certificate c. $\mathsf{h}(\cdot)$ is a hash function depending on the security policy (*e.g.,* SHA-256). Other notations are defined in Section 2.2.

through a new authentication, on the new channel. Therefore, OPC UA relies on such a reliability mechanism we called Switch (see Fig. 2 and Section 2.2.2).

B.2 Secure Communication

UASC in mode RSA. Following Section 2.2.1, we now describe the UASC sub-protocol, and notably OpenChannel, when SecurityPolicy =RSA. We shall use the notation $\{\!| m |\!\}_{pk}$ to denotes the asymmetric encryption of m with the public key pk.

The sub-protocol is depicted in Fig. 9. As explained in Section 2.2.1, the main difference lies in how the symmetric shared keys sk are computed based on a KDF of nonces exchanged in the first two messages. We already explained how it can be adapted to the configurations (Sign | None).

Unauthenticated vs Authenticated Encryption Algorithms. We presented the protocol for configurations that define an encryption scheme corresponding to an *unauthenticated encryption algorithm*, such as SecurityPolicy `[B] - Basic256Sha256` which defines RSA-OAEP-SHA1 as the asymmetric encryption algorithm and RSA-PKCS15-SHA2-256 for signature, and AES256-CBC as the symmetric encryption algorithm and relies on HMAC-SHA2-256 for (symmetric) message authentication. Most of *SecurityPolicies* rely on *unauthenticated encryption algorithms*: all RSA policies are such, and the majority of non-deprecated ECC policies are such as well. However, 3 non-deprecated ECC policies rely on an *authenticated encryption algorithms*: ChaCha20Poly1305. They are not covered by our analysis and left as future work.

On Unauthenticated Encryption We now give more details about the use of unauthenticated encryption in OPC UA. We used the notation $\llbracket m, \{\!| p |\!\}_{k_1} \rrbracket_{k_2}$ in Fig. 9 to denote the combination of signature and asymmetric encryption. OPC UA specifies that it is computed as follows:

$$\llbracket m, \{\!| p |\!\}_{k_1} \rrbracket_{k_2} := m, \mathsf{RSA}^{\mathsf{enc}}_{k_1}(p, \mathsf{RSA}^{\mathsf{sign}}_{k_2}(m, p)).$$

Similarly, the symmetric encryption $[m, \{p\}_{ek}]_{mk}$ is computed:

$$[m, \{p\}_{ek}]_{mk} := m, \mathsf{AES}_{ek}(p, \mathsf{HMAC}_{mk}(m, p)).$$

On Authenticated Encryption Note that for profiles with authenticated encryption, *i.e.,* those based on ChaCha20Poly1305, the message $[m, \{p\}_{ek}]_{mk}$ is computed differently:

$$[m, \{p\}_{ek}]_{mk} := m, \mathsf{CHACHA}_{ek,mk}(m, p)$$

where $\mathsf{CHACHA}_{ek,mk}(m, p)$ denotes the encryption of p with associated data m, with key (ek, mk) (we abstract away the ChaCha20Poly1305 encryption nonce here). We left this authenticated mode as future work.

On Channel Reopening. Servers are configured with a *RevisedLifetime* value, typically 1 hour, that is the maximum time a channel can be open. At 75% of the *RevisedLifetime*, the client is required to reopen the channel to renew the set of symmetric keys and get a fresh channel token tok_c' as explained in Section 2.2.1. The client will use the new set of keys first, and the server will definitively switch to the new set at the first message received with the new token tok_c'.

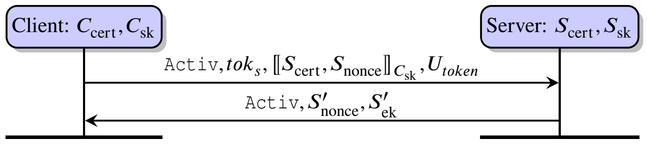

Client: $C_{\text{cert}}, C_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
$\texttt{Activ}, tok_s, [\![S_{\text{cert}}, S_{\text{nonce}}]\!]_{C_{\text{sk}}}, U_{token}$
$\texttt{Activ}, S'_{\text{nonce}}, S'_{\text{ek}}$

Figure 10: Session activation in configurations (RSA | ECC + Sign | Enc + SSec | SNoAA + Pwd). U_{token} is defined in Appendix B.3 depending on whether Mode is RSA or ECC. Other notations are explained in Fig. 4.

B.3 Sessions

We described in Section 2.2.2 the session sub-protocol for the configurations (ECC + Enc | Sign | None + SSec | SNoAA + Cert) and how to adapt it when SecurityPolicy = RSA and/or in configuration Ano. We now describe in details how the sub-protocol differ for Pwd, depending on whether SecurityPolicy = RSA or SecurityPolicy = ECC. In both cases, the password will be encrypted in a *EncrypteSecret* structure (cf. [22, Part 4, § 7.41.2.3 *"EncrypteSecret Format"*]), that depends on the SecurityPolicy.

Configuration Pwd + ECC. At session activation, users can authenticate themselves with a login and password (configuration Pwd). In that case, the exchanged messages are a bit different since the client needs to send the encrypted password to the server. This is done as shown in Fig. 10 with:

$$U_{token} = U, [\![C_{\text{cert}}, g^s, g^c, \{S_{\text{nonce}}, U_{\text{pwd}}\}_{k_e}]\!]_{C_{\text{sk}}}$$

where:

- Username U acts a login.
- g^s is the server's half-key (from $S_{\text{ek}} = [\![g^s]\!]_{S_{\text{sk}}}$) and S_{nonce} the server nonce that were both sent in the last session activation or creation response (for that session).
- g^s is a new, fresh client's DH half-key.
- $k_e = \mathsf{KDF}(g^{c \cdot s})$ is a fresh symmetric encryption key.

This mechanism is supposed to guarantee PFS to the passwords. Moreover, it allows the server to pass the container to a secondary authentication server. Note that, if the DH half-key $S_{\text{ek}} = [\![g^{er}]\!]_{S_{\text{sk}}}$ provided by the server were not signed, an attack would be possible as soon as the channel is compromised, so this signature must be properly checked.

Configuration Pwd + RSA. In that case, passwords are asymmetrically encrypted with the server's public key. The flow of message is then exactly as in Fig. 10 with:

$$U_{token} = U, [\mathsf{h}(S_{\text{cert}}), \{\![k_e, k_m, iv]\!\}_{S_{\text{pk}}}, \{iv, S_{\text{nonce}}, U_{\text{pwd}}\}_{k_e}]_{k_m}$$

where k_e and k_m are fresh symmetric keys (for symmetric encryption and HMAC), and iv is a fresh initialization vector.

When are passwords encrypted? We described how passwords are encrypted at the session layer. For the configuration Enc, this can be considered redundant with the encryption at the channel layer. For this reason, the specification allows to skip the encryption of the password at the session layer when the channel is encrypted *i.e.,* in that case the user token contains the password in plaintext, but the whole messages will be wrapped in and encrypted by the communication channel:

$$U_{token} = U, U_{\text{pwd}}$$

As shown in the quote below, the specification allows both behaviors in Enc (configured with *UserTokenPolicy SecurityPolicy*): the password (*UserIdentityToken*) can be encrypted at the session layer or not. However, in Sign the password must always be encrypted at the session layer:

> **[22, Part 4, § 7.41.4 *"UserNameIdentityToken, Table 193"*]** *[In Enc]*
> • *UserTokenPolicy SecurityPolicy:* ***Other***
> • *UserIdentityToken:* ***Asymmetric algorithm for "Other"***
> *[In Enc]*
> • *UserTokenPolicy SecurityPolicy:* ***None***
> • *UserIdentityToken:* ***No encryption but encrypted SecureChannel***

[In Sign]
• *UserTokenPolicy SecurityPolicy:* ***Other***
• *UserIdentityToken:* ***Asymmetric algorithm for "Other"***
[In Sign]
• *UserTokenPolicy SecurityPolicy:* ***None***
• *UserIdentityToken:* ***Invalid configuration shall be rejected***

SSec vs SNoAA. We already explained the differences between those two modes in Section 2.2.2 and we now complement this with the rationale behind them.

Certificate trust checking (against the PKI) at session creation and activation is actually redundant with UASC, since those checks were already performed at channel opening and certificates are checked to be the same at channel and session layers. However, performing the trust checks at the session layer is necessary for HTTPS profiles since, then, channel-level certificates are TLS certificates, which are different from the OPC UA application certificates used at the session layer. In short, certificates at the channel level (TLS certificates) can be different from those at the session level (OPC UA certificates).

For the sake of unity, those session-layer trust checks are prescribed in the default configuration that we call Session Security (SSec).

However, the specification allows a relaxed configuration, where all certificates are accepted by the server (SNoAA):

[22, Part 7, § 5.3 table 11 *"No Application Authentication"*] *The Server supports being able to be configured for no application authentication, just User authentication and normal encryption/signing:*
- *Configure Server to accept all certificates*
- *Certificates are just used for message security (signing and encryption)*
- *Users level is used for authentication*

B.4 OPC UA Model and Security Goals

We provide more details to support our interpretation of the security goals and threat models of OPC UA, that we presented in Section 2.3.

B.4.1 Tables in ProVerif

Continuing Section 3.1, we now provide an example of ProVerif code for handling tables. Consider the table

```
table Csess(pkey,pkey,sid,stok,checkMode,chan,user,cred).
```

and its (simplified) usage in an initial session activation by user U of a session created between client C_pk and server S_pk:

```
get Csess(=C_pk,=S_pk,id_s:sid,tok_s:stok,m,ch:chan,=none,_) in
[...] (* send activation request and receive response *)
event C_First_Activation(ch, id_s, U);
insert Csess(C_pk,S_pk,id_s,tok_s,m,ch,U,U_cert); [...]
```

The above code illustrates that we retrieve an existing channel between C_pk and server S_pk. The fact that the user is set to the constant none, models that the session has not yet been activated. After execution of the activation sub-protocol, we insert an entry recording that the session has been activated by U identified by her certificate Ucert. The event C_First_Activation(ch, id_s, U) records the first activation by user U of session id_s on channel ch.

B.4.2 Threat Model

We explained our interpretation of the OPC UA threat model in terms of attacker capabilities and compromise scenarii including relevant references from [22, Part 2, § 4.3 *"Security threats to OPC UA systems"*] in Section 2.3.1. We now give further quotes of the specification to support these choices.

We consider the presence of *compromised* agents. For users, their credentials (passwords, or certificate private keys) may be compromised (§ 4.3.12). For clients and servers, we consider the possibility that their long term private keys (associated to their certificate) have been compromised. Such leaks may happen, for instance, when a device is decommissioned without appropriate erasure of sensitive data. This is a common threat in the context of PKI, and is addressed in the specification:

[22, Part 4, § 6.1.4 *"Creating a SecureChannel"*] *Certificates can be compromised, which means they should no longer be trusted. Administrators can revoke a Certificate by removing it from the trust list for all applications or the CA can add the Certificate to the Certificate Revocation List (CRL)*

We however exclude full attacker control of a machine which would give access to all ephemeral secrets and any credential entered by users, as acknowledged by the specification:

[22, Part 2, § 4.4 *"OPC UA relationship to site security"*] *Threats to infrastructure components that might result in the compromise of operating systems, where OPC UA Applications are running, are not addressed by OPC UA.*

B.4.3 Security Properties

We now discuss the security properties that OPC UA is supposed to guarantee as described in [22, Part 2, § 4.2 *"Security objectives"*].

Confidentiality. Secure channels with a Mode set to Enc must provide confidentiality of data exchanged on the network. Even though [22, Part 2, § 4.2.4] mentions "data protection from passive attacks such as eavesdropping", the specification explicitly considers that active adversary capabilities can impact confidentiality as well. In particular, the following threats are considered to "impact confidentiality": compromising user credentials (§ 4.3.12), session hijacking (§ 4.3.9), and rogue server (§ 4.3.10). We thus conservatively interpret this as confidentiality in presence of an active attacker.

The encrypted messages should protect the data of user requests and responses, but also credentials such as passwords. Moreover, channels relying on the ECC family, should provide PFS to user requests/responses, *i.e.,* any message exchange prior to the compromise should remain secure.

To sum up, OPC UA is expected to guarantee that: *(i) The attacker cannot learn the user requests and responses payloads when sent over channels in configuration Enc. (ii)* PFS *is guaranteed for the payloads sent over channels in configuration (ECC + Enc). (iii) The attacker cannot learn the user passwords sent during create sessions (independently of the configuration since they are always encrypted).*

Integrity and Authentication. Integrity and authentication are key requirements for OT protocols. OPC UA is claimed to provide those properties in presence of an active attacker [22, Part 2, § 5.1.1]. Below we express these security goals more formally as *agreement properties* in Lowe's authentication taxonomy [20], where properties are expressed as relations between different agents' point of views of a protocol execution.

Secure channels with a Mode set to Enc or to Sign should provide message integrity, also known as data authentication, in presence of an active attacker.

[22, Part 2, § 4.2.5 *"Integrity"*] *Receivers receive the same information that the original sender sent, without the data being changed during transmission.*

Similarly, endpoints must be identified, *i.e.,* the protocol must provide entity authentication

[22, Part 2, § 4.2.2 *"Authentication"*] *Entities such as clients, servers, and users should prove their identities. Authentication can be based on something the entity is, has, or knows.*

Identifying endpoints correctly is also essential to grant access rights corresponding to the authorization policy, often on a role-based access control (RBAC) basis.

[22, Part 2, § 4.2.3 *"Authorization"*] *The access to read, write, or execute resources should be authorized for only those entities that have a need for that access within the requirements of the system.*

Data and entity authentication can be expressed as a single, strong *agreement property*: *if a server S accepts a user request R from a user U on client C, then U has indeed initiated request R for server S on C*. Conversely, *if a user U receives on client C a user response R from server S, then S has indeed responded with R to client C for user U*. As the protocol is supposed to protect against replays , we moreover require that these agreements are *injective*, *i.e.,* each received request corresponds to a distinct request emission.

Note that authentication is essential for logging lifetime and security events for auditability.

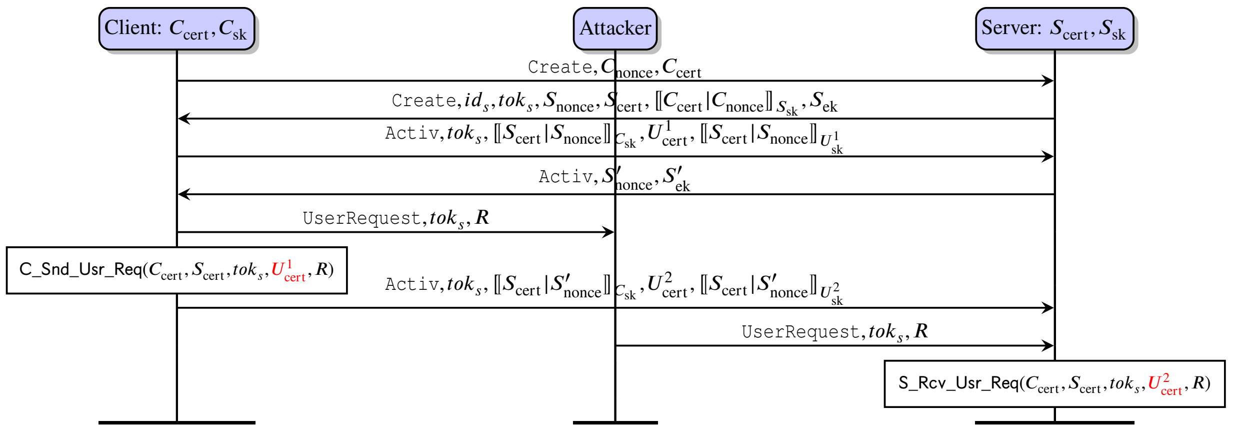

Client: $C_{\mathrm{cert}}, C_{\mathrm{sk}}$
Attacker
Server: $S_{\mathrm{cert}}, S_{\mathrm{sk}}$
Create, $C_{\mathrm{nonce}}, C_{\mathrm{cert}}$
Create, $id_s, tok_s, S_{\mathrm{nonce}}, S_{\mathrm{cert}}, [\![C_{\mathrm{cert}} | C_{\mathrm{nonce}}]\!]_{S_{\mathrm{sk}}}, S_{\mathrm{ek}}$
Activ, $tok_s, [\![S_{\mathrm{cert}} | S_{\mathrm{nonce}}]\!]_{C_{\mathrm{sk}}}, U^1_{\mathrm{cert}}, [\![S_{\mathrm{cert}} | S_{\mathrm{nonce}}]\!]_{U^1_{\mathrm{sk}}}$
Activ, $S'_{\mathrm{nonce}}, S'_{\mathrm{ek}}$
UserRequest, tok_s, R
C_Snd_Usr_Req($C_{\mathrm{cert}}, S_{\mathrm{cert}}, tok_s, U^1_{\mathrm{cert}}, R$)
Activ, $tok_s, [\![S_{\mathrm{cert}} | S'_{\mathrm{nonce}}]\!]_{C_{\mathrm{sk}}}, U^2_{\mathrm{cert}}, [\![S_{\mathrm{cert}} | S'_{\mathrm{nonce}}]\!]_{U^2_{\mathrm{sk}}}$
UserRequest, tok_s, R
S_Rcv_Usr_Req($C_{\mathrm{cert}}, S_{\mathrm{cert}}, tok_s, U^2_{\mathrm{cert}}, R$)

Figure 11: Race condition on user contexts. A server may interpret a request from user U^1_{cert} as a request from user U^2_{cert}. We neither depict the channel establishment nor the channel protection of the exchanged messages since the attack does not rely on these.

> **[22, Part 2, § 4.2.7 *"Auditability"*]** *Actions taken by a system must be recorded in order to provide evidence to stakeholders:*
> - *that this system works as intended (successful actions are tracked)*
> - *that identify the initiator of certain actions (user activity is tracked)*
> - *that attempts to compromise the system were denied (unsuccessful actions are tracked)*

Availability In OT systems, availability is also a major concern.

> **[22, Part 2, § 4.2.8 *"Availability"*]** *is impaired when the execution of software that needs to run is turned off or when the software or communication system is overwhelmed by processing input.*

While we do not directly model availability we consider the consequences on availability of the vulnerabilities we discover.

C Additional Details on Attacks

We summarize in Table 3 all the attacks found, their impact, root causes, and associated fixes/mitigations. We provide additional details on some individual attacks in the following subsections.

C.1 Race Condition for User Contexts

We provide the detailed message flow of the race condition which leads to a potential confusion of the user context in Fig. 11. The figure illustrates how Agr_S is violated in the configuration (ECC + Enc + SSec + Cert), but can be adapted to any other configuration.

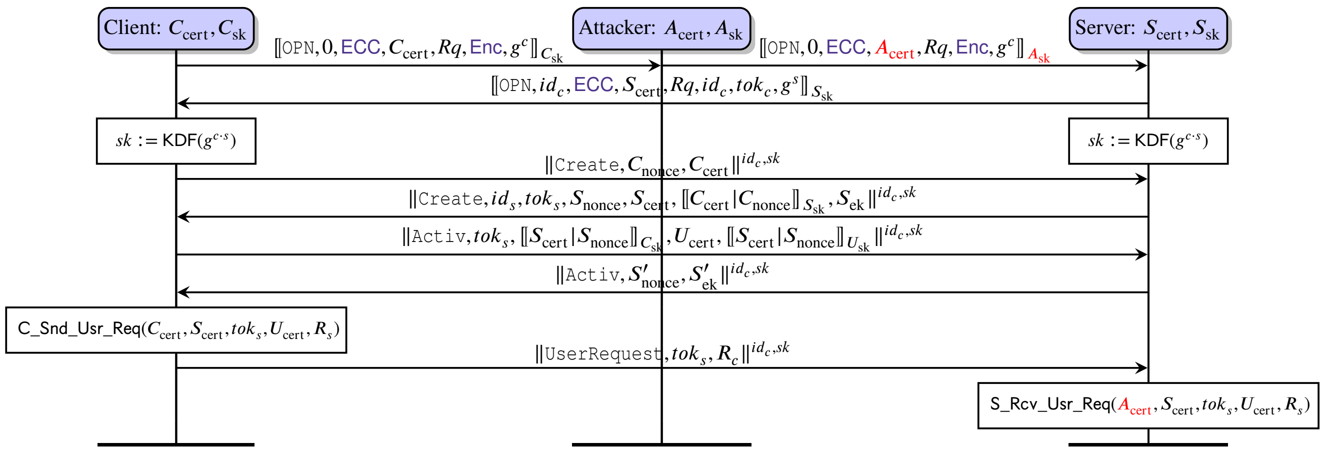
The attack can be easily adapted to violate Agr_C: the attacker does not intercept the user request, but the request response. If the response is only delivered after the re-activation, we end up with a similar mismatch on the user identity from the client's point of view.

C.2 Client Impersonation in ECC

The detailed message flow of the client impersonation attack is given in Fig. 12.

C.3 Session Hijack Attacks

Interestingly, both of the attacks presented below directly contradict claims in the specification. Indeed, a "session hijack" threat is identified in [22, Part 2, § 4.3.9 *"Security threats to OPC UA systems"*]. But the specification later argues why OPC UA is supposed to thwart such a threat:



Client: $C_{\text{cert}}, C_{\text{sk}}$
Attacker: $A_{\text{cert}}, A_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
$[\![\texttt{OPN}, 0, \text{ECC}, C_{\text{cert}}, Rq, \text{Enc}, g^c]\!]_{C_{\text{sk}}}$
$[\![\texttt{OPN}, 0, \text{ECC}, A_{\text{cert}}, Rq, \text{Enc}, g^c]\!]_{A_{\text{sk}}}$
$[\![\texttt{OPN}, id_c, \text{ECC}, S_{\text{cert}}, Rq, id_c, tok_c, g^s]\!]_{S_{\text{sk}}}$
$sk := \text{KDF}(g^{c \cdot s})$
$sk := \text{KDF}(g^{c \cdot s})$
$\|\texttt{Create}, C_{\text{nonce}}, C_{\text{cert}}\|^{id_c, sk}$
$\|\texttt{Create}, id_s, tok_s, S_{\text{nonce}}, S_{\text{cert}}, [\![C_{\text{cert}} | C_{\text{nonce}}]\!]_{S_{\text{sk}}}, S_{\text{ek}}\|^{id_c, sk}$
$\|\texttt{Activ}, tok_s, [\![S_{\text{cert}} | S_{\text{nonce}}]\!]_{C_{\text{sk}}}, U_{\text{cert}}, [\![S_{\text{cert}} | S_{\text{nonce}}]\!]_{U_{\text{sk}}}\|^{id_c, sk}$
$\|\texttt{Activ}, S'_{\text{nonce}}, S'_{\text{ek}}\|^{id_c, sk}$
$\text{C\_Snd\_Usr\_Req}(C_{\text{cert}}, S_{\text{cert}}, tok_s, U_{\text{cert}}, R_s)$
$\|\texttt{UserRequest}, tok_s, R_c\|^{id_c, sk}$
$\text{S\_Rcv\_Usr\_Req}(A_{\text{cert}}, S_{\text{cert}}, tok_s, U_{\text{cert}}, R_s)$

Figure 12: Client impersonation attack when SecurityPolicy = ECC and SessionSecurity =SNoAA. Detailed message flow of Fig. 5

> **[22, Part 2, § 5.1.9 *"Reconciliation of threats with OPC UA security mechanisms"*]** *Hijacking a Session would thus first require compromising the security context.*

The two following attacks show that this is not the case in that our attacks do not assume compromising channel keys.

Session Hijack by Reopening. We provide the detailed message flow of the session hijack attack in Fig. 7. The attack requires that Mode = Sign and is illustrated here when SecurityPolicy = ECC and SessionSecurity = SSec.

Session Hijack by Switching. Note that a variant of this attack relies on switching the channel of a session, rather than reopening the channel. In the above attack a channel reopening request was issued by the attacker to gain access to the channel underlying the session. Alternatively, an attacker can intercept a session re-activation, open a new channel, and replay the re-activation attack on the new (attacker controlled) channel. We provide the detailed message flow of this variant in Fig. 13. This variant shows that the attack is not due to the channel re-opening mechanism, but rather relies on the lack of session ownership proof (tok_s is not secret). Indeed, tok_s is described in [22, Part 4, § 7.36] as *secret*, but is sent unencrypted on the network when Mode = Sign.[7] Since no other mechanism is in place for the server to guarantee that his peer is the client who created the session, the attacker can simply replay the tok_s as shown by the two session hijack attacks we found.

C.4 Additional Findings

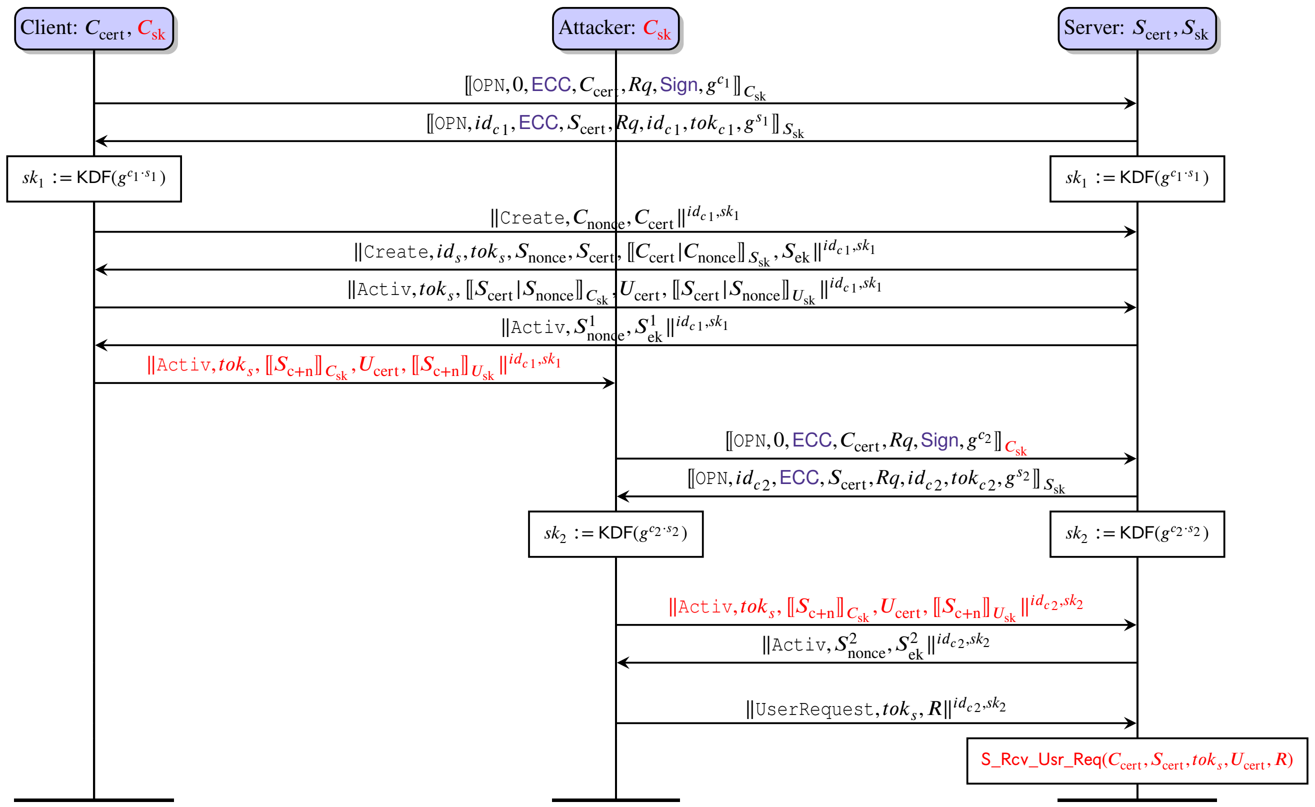
C.4.1 Downgrade of Password Secrecy

We show that when channel keys are leaked, counter-intuitively, Mode = Enc is less secure than Mode = Sign since it leaks user passwords.

Description. When a password is transmitted for user authentication at session activation, it is embedded in a dedicated *EncryptedSecret* data container (see Appendix B.3 for details). As mentioned in Section 2.2.2, the specification however tolerates this container to not be encrypted when encryption is already provided by the channel, *i.e.,* Mode = Enc. Hence, when channel keys are leaked, and Mode = Enc, the confidentiality of the password ($\mathit{Conf}_{\text{Pwd}}$) is violated (while preserved when Mode = Sign).

Impact. An attacker that records the traffic and is able to compromise channel keys, can acquire the plain text of the encrypted traffic, and in particular the user password, if not encrypted at session level. As mentioned in our threat model Section 2.3.1, we consider those keys as semi-long-term and allow the attacker to compromise them. Indeed, best practices recommend storing long term keys in a HSM, while symmetric channel keys are stored in memory during their lifetime, and hence are more exposed and subject to compromise. This means that the compromise of semi-long-term keys can entail the compromise of a password (which is a long term secret). Of course, once a user U's password leaks to the attacker, the latter can impersonate U to honest clients and servers.

[7] We reported this mismatch to the OPC Foundation which has updated the specification [22, Part 4, § 7.36 *"SessionAuthenticationToken"*] to clarify that the value is not secret.



Client: $C_{\text{cert}}, C_{\text{sk}}$
Attacker: C_{sk}
Server: $S_{\text{cert}}, S_{\text{sk}}$
$[\![\texttt{OPN}, 0, \text{ECC}, C_{\text{cert}}, Rq, \text{Sign}, g^{c_1}]\!]_{C_{\text{sk}}}$
$[\![\texttt{OPN}, id_{c1}, \text{ECC}, S_{\text{cert}}, Rq, id_{c1}, tok_{c1}, g^{s_1}]\!]_{S_{\text{sk}}}$
$sk_1 := \text{KDF}(g^{c_1 \cdot s_1})$
$sk_1 := \text{KDF}(g^{c_1 \cdot s_1})$
$\|\texttt{Create}, C_{\text{nonce}}, C_{\text{cert}}\|^{id_{c1}, sk_1}$
$\|\texttt{Create}, id_s, tok_s, S_{\text{nonce}}, S_{\text{cert}}, [\![C_{\text{cert}}|C_{\text{nonce}}]\!]_{S_{\text{sk}}}, S_{\text{ek}}\|^{id_{c1}, sk_1}$
$\|\texttt{Activ}, tok_s, [\![S_{\text{cert}}|S_{\text{nonce}}]\!]_{C_{\text{sk}}}, U_{\text{cert}}, [\![S_{\text{cert}}|S_{\text{nonce}}]\!]_{U_{\text{sk}}}\|^{id_{c1}, sk_1}$
$\|\texttt{Activ}, S^1_{\text{nonce}}, S^1_{\text{ek}}\|^{id_{c1}, sk_1}$
$\|\texttt{Activ}, tok_s, [\![S_{\text{c+n}}]\!]_{C_{\text{sk}}}, U_{\text{cert}}, [\![S_{\text{c+n}}]\!]_{U_{\text{sk}}}\|^{id_{c1}, sk_1}$
$[\![\texttt{OPN}, 0, \text{ECC}, C_{\text{cert}}, Rq, \text{Sign}, g^{c_2}]\!]_{C_{\text{sk}}}$
$[\![\texttt{OPN}, id_{c2}, \text{ECC}, S_{\text{cert}}, Rq, id_{c2}, tok_{c2}, g^{s_2}]\!]_{S_{\text{sk}}}$
$sk_2 := \text{KDF}(g^{c_2 \cdot s_2})$
$sk_2 := \text{KDF}(g^{c_2 \cdot s_2})$
$\|\texttt{Activ}, tok_s, [\![S_{\text{c+n}}]\!]_{C_{\text{sk}}}, U_{\text{cert}}, [\![S_{\text{c+n}}]\!]_{U_{\text{sk}}}\|^{id_{c2}, sk_2}$
$\|\texttt{Activ}, S^2_{\text{nonce}}, S^2_{\text{ek}}\|^{id_{c2}, sk_2}$
$\|\texttt{UserRequest}, tok_s, R\|^{id_{c2}, sk_2}$
$\text{S\_Rcv\_Usr\_Req}(C_{\text{cert}}, S_{\text{cert}}, tok_s, U_{\text{cert}}, R)$

Figure 13: Session hijacking by switching to a compromised channel when Mode = Sign, illustrated here with SecurityPolicy = ECC and SessionSecurity = SSec. We denote $S_{\text{cert}}|S^1_{\text{nonce}}$ with $S_{\text{c+n}}$.

Mitigation and fix. A simple fix is to always encrypt the password at session level. The OPC Foundation acknowledged the problem and will add a warning explaining the impact of not using an *EncryptedSecret* in an encrypted channel in v.1.05.04 RC of [22, Part 4, § 7.41.4 table 193] and a recommendation to always use it; see ticket #9432.

C.4.2 Risk of Signature Oracle

A server configured to allow SNoAA may not verify the certificate transmitted by a client application at session creation, but simply append the client nonce and sign the resulting bitstring, to prove possession of its private key. This induces a "signature oracle" to produce valid server signatures.

Description. In a *CreateSession* reply, servers prove possession of their private keys by the means of the signature $[\![C_{\text{cert}}|C_{\text{nonce}}]\!]_{S_{\text{sk}}}$ specified in [22, Part 4, § 5.6.2.2 table 15 *"by appending the client Nonce to the client Certificate and signing the resulting sequence of bytes"*]. We note that the payload that is signed neither contains any tag nor context. The conformance unit [22, Part 7, § 5.3 table 11 *"Security - No Application Authentication"*] (see Appendix B.3) authorizes a server to accept all certificates, when SessionSecurity = SNoAA. Moreover, the client nonce must have a minimal size, but no maximal size is prescribed in [22, Part 4, § 5.6.2.2 table 15 *"This number shall have a minimum length of 32 bytes. Profiles may increase the required length"*]. Therefore, if a client provides two bitstrings of minimum length in a create session request, the server replies with its signature of the concatenated bitstrings. Hence, a compromised client may open a channel with an honest server S, and exploit the signature oracle through session creation requests, in order to then impersonate the server towards any honest client during an *OpenChannel* response, for both SecurityPolicy RSA and ECC; see the first two messages of Fig. 14.

When SecurityPolicy = ECC, an attacker can further exploit this signature oracle to impersonate the server S towards any honest client C in session creations and activations (Fig. 14):

1. Client C sends an *OpenChannel* request to Server S. The attacker A intercepts this message. Using the signature oracle, the attacker computes a valid response with its own half-key g^a and derives the channel keys.
2. C sends a *CreateSession* request. Again the attacker exploits the signature oracle to obtain the server's signature on the

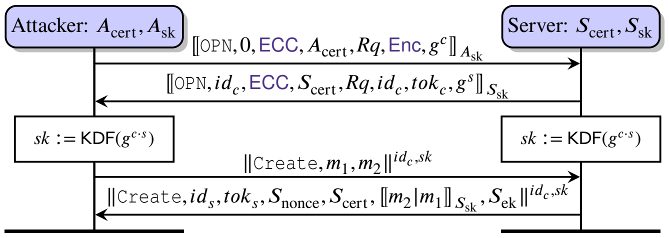

Attacker: $A_{\text{cert}}, A_{\text{sk}}$
Server: $S_{\text{cert}}, S_{\text{sk}}$
$[\![\texttt{OPN}, 0, \text{ECC}, A_{\text{cert}}, Rq, \text{Enc}, g^c]\!]_{A_{\text{sk}}}$
$[\![\texttt{OPN}, id_c, \text{ECC}, S_{\text{cert}}, Rq, id_c, tok_c, g^s]\!]_{S_{\text{sk}}}$
$sk := \text{KDF}(g^{c \cdot s})$
$sk := \text{KDF}(g^{c \cdot s})$
$\|\texttt{Create}, m_1, m_2\|^{id_c, sk}$
$\|\texttt{Create}, id_s, tok_s, S_{\text{nonce}}, S_{\text{cert}}, [\![m_2|m_1]\!]_{S_{\text{sk}}}, S_{\text{ek}}\|^{id_c, sk}$

Figure 14: Signature oracle for message $m = m_1|m_2$ with SecurityPolicy = ECC; when SessionSecurity = SNoAA, client certificates (here m_2) are not validated.

client's certificate and nonce. Therefore the attacker can forge a valid session creation response.

Once the session is created the attacker can impersonate the server in activation and user requests.

Note that when SecurityPolicy = RSA, the attacker can forge a valid *OpenChannel* response, but does not learn the channel keys, as the client nonce is encrypted with the server's public key (see Fig. 9). While this violates server authentication at channel establishment, such an UKS attack, cannot be further continued for session creation.

Impact. When SecurityPolicy = ECC, the signature oracle can be exploited to mount a full server impersonation, breaking all security properties on the client side: Conf_{C}, $\mathit{Conf}_{\text{Pwd}}$ and Agr_{C}. We stress that the signature oracle prerequisites should be considered plausible: *(i)* Allowing SNoAA in OPC UA Binary should not be rare in implementations favoring interoperability with OPC UA HTTPS. *(ii)* The signature oracle is at session creation, not at session activation, hence it does not require any user credential.

Mitigation and fix. The OPC Foundation acknowledged the weakness, and to mitigate the vulnerability, *(i)* adds explicit requirements for v.1.05.04 RC, and conformance tests to ensure that certificates are validated (*i.e.,* checked to be well-formed certificates, but not necessarily trust checked in SNoAA) before being used to produce a signature (tickets #9594, #9596, #9597 and #9598); *(ii)* states that all Nonces shall have a length specified by the SecurityPolicy (tickets #9595, #9599). We reflect those fixes and in our model and prove that they thwart the attack we found on the same configuration.

C.4.3 Other Weaknesses

There is a disagreement on the SessionSecurity mode. It is indeed possible that a client and a server do not have the same view regarding its value. The disagreement was found while trying to prove a lemma needed for proving $\mathit{Agr}^{-}_{\text{S}}$ when we allow both SSec and SNoAA. This disagreement does however not seem to lead to a vulnerability, but violated our formal agreement property.

D Security proofs

D.1 Starting New Proof Campaigns

As mentioned in Section 4.2 we use a proof script to deal with a large number of configurations, that gradually increases the computation timeout at each iteration.

However, in case of a change in the model, we need to start a new campaign, and recompute everything. But the results of the previous campaign give hints that revealed very useful to save time. Hence, we have adopted the following heuristic: we start with the minimal false and cannot-be-proved configurations, then the maximal true configurations with a timeout at least twice the duration of the previous computation, then the minimal out-of-memory and out-of-time configurations. As usual, we propagate each result into the configuration lattice, and stop only when all configurations are covered. This heuristic allowed us to recompute in a few days a complete campaign for a query, with final timeouts as long as 24 hours.

D.2 A novel proof methodology

We provide additional details and examples regarding our proof methodology discussed in Section 4.3.2.

D.2.1 Background on ProVerif

In ProVerif, protocols are specified in the applied pi calculus. The first step is to translate these processes into first-order Horn clauses, a logical representation that was shown to be a sound over-approximation [7]. ProVerif implements a *resolution procedure* that computes, out of an initial set of clauses, a fixed-point set of clauses that are reachable from the initial set. We briefly recall the key concepts of this procedure and refer the reader to [7] for a detailed and more formal account. The resolution procedure is based on Horn clauses that are first-order logic formulas of the form

$$c := H_1 \wedge \ldots \wedge H_n \rightarrow F$$

where $H_1, \ldots, H_n, F$ are *facts*, that are predicate symbols applied to terms. For instance, the clause

$$c_a^{\text{enc}} := \mathsf{attacker}(x) \wedge \mathsf{attacker}(y) \rightarrow \mathsf{attacker}(\mathsf{senc}(x, y))$$

describes the fact that, if the attacker knows some x and y, then he also knows $\mathsf{enc}(x, y)$ since he can apply the encryption function enc to x and y.

From an initial set of clauses, additional clauses are added by the means of *resolution*: two clauses

$$c_1 := H_1 \wedge \ldots \wedge H_n \rightarrow F$$

and

$$c_2 := H'_1 \wedge \ldots \wedge H'_k \rightarrow F'$$

such that F and H'_1 can be unified (say by σ) are *resolved* into a new clause

$$c' := H_1\sigma \wedge \ldots \wedge H_n\sigma \wedge H'_2\sigma \wedge \ldots \wedge H'_k\sigma \rightarrow F'\sigma$$

(note that H'_1 is removed from the hypotheses).

In ProVerif, the resolution is guided by a *selection function* that selects which clauses and facts to resolve, that is which c_1, c_2, and H'_1 to choose for the resolution. The built-in selection function is based on specific heuristics that aim at preventing the resolution to loop forever in many practical cases. Moreover, the selection function can be fine-tuned: ProVerif also provides a noselect statement that can be used to deprioritize the selection of a given fact.

Despite the built-in heuristics, the resolution procedure can loop indefinitely for complex models, especially when they involve tables and rich state machines. Intuitively, this is because ProVerif will end up resolving a clause c_1 that assumes a fact $H_i := \mathsf{table}T(y)$ (we assume an entry y is in table T) with a clause c_2 whose conclusion is $F := \mathsf{table}T(z)$ (we add an entry z to table T). The resolution will then produce a new clause that still assumes an entry in the table and that adds an entry to the table.

Example 1 (Running example: proving Agr_S^- for OPC UA)**.** *For instance, the proof for Agr_S^- in configuration (RSA + Sign + Reopen + SSec + Cert + Switch) does not terminate in reasonable time and shows an explosion of the number of generated clauses (clauses in the `Queue`).*

The following proof methodology aims at breaking such loops by manually guiding the resolution procedure.

D.2.2 Step 1: Identify the loops

When a proof does not terminate and seems to loop, the first step is to identify on which facts the resolution procedure seems to loop. For this, we re-run ProVerif with the option `verboseRules` that displays all the clauses created by the resolution procedure. In the produced output log, we manually inspect if a fact is selected in most resolution steps, past the first few resolution steps that are not looping yet. This way, we extract one or a few candidates for facts to *break the loop*.

Example 2. *The ProVerif output log with the option `verboseRules` shows that one quarter of the generated clause have the fact table(S_sessions(y)) as conclusion fact. We identify this fact as a candidate for breaking the loop.*

D.2.3 Step 2: Adding noselect statements to break the loops

We now add a noselect statement for the identified fact. We then re-run ProVerif and check if the proof terminates. If it does not, we repeat steps 1 and 2 until the proof terminates. If it does, it is very likely that ProVerif now returns a cannot-be-proved. We explain in the next steps how to deal with this.

Example 3 (Continuing the running example)**.** *We add*

```
noselect x1,x2:pkey, x3:seid, x4:stoken, x5:smode, x6:channel_, x7:login, x8:utoken, x9:exponent, x10:nonce;
    table(S_sessions( *x1,*x2,*x3,*x4,*x5,*x6,*x7,*x8,*x9,*x10))/9000.
```

We re-run the proof and still observe that the proof search seems to loop. Inspection of the output log reveals that many clauses are generated with the fact table(C_sessions(y)) *so we add:*

```
noselect x1,x2:pkey, x3:seid, x4:stoken, x5:smode, x6:channel_, x7:login, x8:utoken, x9:exponent, x10:nonce;
    table(C_sessions( *x1,*x2,*x3,*x4,*x5,*x6,*x7,*x8,*x9,*x10))/9000.
```

We re-run ProVerif: the proof now concludes in 4 seconds with a cannot-be-proved*.*

D.2.4 Step 3: Collecting false assumptions

Since the facts for which we added a noselect statement are not selected anymore, ProVerif has "no clue" about the terms that can be contained in those facts. For example, for table facts, ProVerif will not be able to infer which table entries are actually possible by the protocol. More generally, without the ability to trace back from where a fact (under noselect) originates, it conservatively assumes those facts could initially contain arbitrary terms, leading to "false attacks".

Fortunately, ProVerif explicitly describes the assumptions it made about the facts under noselect statements in the *goal reachable* and the *derivation* (produced in the HTML report, see the ProVerif manual). Those show the clause that falsifies the property and the derivation yielding this clause. In the hypotheses of this clause, one should manually inspect the facts under noselect statements and collect the terms that are assumed to be in those facts. For instance, when those facts are table facts, this allows to collect the entries that are assumed to be in the table for the "false" attack to happen. A manual inspection reveals what are the "false assumptions", namely the assumptions: (i) that ProVerif took because of the noselect statements, but (ii) that are impossible to obtain in normal executions of the protocol.

Example 4 (Continuing the running example)**.** *Inspecting the derivation produced by ProVerif for the* cannot-be-proved*, we observe two hypotheses over the facts under the* noselect *statements:*

```
event(insert_S_sessions(pk(sk_2),pk(sk_3),id_132,SAtk,SSec,
  chan(certify(client,RSA,pk(sk_2)),certify(server,RSA,pk(sk_3)),Sign,ch_n_1,
    keys(Token_1, enc_key_RSA(C_Nonce_1,S_Nonce_1), sign_key_RSA(C_Nonce_1,S_Nonce_1), enc_key_RSA(
         S_Nonce_1,C_Nonce_1), sign_key_RSA(S_Nonce_1,C_Nonce_1))
  ),usr_7,U_tk_8,S_er_2,S_nonce_22))
```

and

```
event(insert_C_sessions(C_pk_63,S_pk_61,id_131,SAtk,check_5,
  chan(certify(k_2,RSA,C_pk_63),certify(k_3,s_2,S_pk_61),Sign,ch_n_1,
    keys(Token_1, ek_62, sign_key_RSA(C_Nonce_1,S_Nonce_1), S_enc_k_68, S_sign_k_68)
  ),usr_8,U_tk_7,O_hk_7,O_nonce_9,C_sk_4))
```

where client signing key sign_key_RSA(C_Nonce_1,S_Nonce_1) *and* SAtk *match between* S_session *and* C_session*. However, the rest of the channel keys (e.g.,* S_enc_k_68*,* S_sign_k_68*) are not assumed to match.*

We know this is impossible since all channel keys are derived from the same seed in the protocol:

```
let sym_keys =
    let C_enc_k  =  enc_key_RSA(C_Nonce, S_nonce) in
    let C_sign_k = sign_key_RSA(C_Nonce, S_nonce) in
    let S_enc_k  =  enc_key_RSA(S_nonce, C_Nonce) in
    let S_sign_k = sign_key_RSA(S_nonce, C_Nonce) in
    keys(token, C_enc_k, C_sign_k, S_enc_k, S_sign_k)
```

Therefore, if sign_key_RSA(C_Nonce_1,S_Nonce_1) *matches between a client's session entry and a server's session entry, then all channel keys must match between those entries. Moreover, both entries should agree on client and server identities. We shall see how to add such an invariant in the next step.*

D.2.5 Step 4: Design invariants contradicting the false assumptions

Once a false assumption is identified, we write an invariant that states that these assumptions are actually false. Such invariants can be stated in ProVerif as an axiom that will be applied on the clauses during the resolution. Writing the invariant boils down to express the contradiction of the false assumption. We eventually use ProVerif to prove that these invariants indeed hold, see next step.

Example 5 (Continuing the running example)**.** *We had observed that client server session table entries agreed on the session token and the client signing key. We therefore add the following axiom*

```
let SK1 = keys(chtk1,C_enc_k1,C_sign_k,S_enc_k1,S_sign_k1) in
let SK2 = keys(chtk2,C_enc_k2,C_sign_k,S_enc_k2,S_sign_k2) in
let SC1 = chan(c_cert1,s_cert1,mode1,ch1,SK1) in
let SC2 = chan(c_cert2,s_cert2,mode2,ch2,SK2) in
event(insert_S_sessions(C_pk1, S_pk1, id1, SAtk, check1, SC1, usr1, U_tk1,  er, S_nonce1)) &&
event(insert_C_sessions(C_pk2, S_pk2, id2, SAtk, check2, SC2, usr2, U_tk2, ger, S_nonce2, C_sk2))
==> C_pk1 = C_pk2  && S_pk1 = S_pk2 && SC1 = SC2 ||
      event(leaked(C_pk1)) && event(leaked_server(S_pk2)).
```

that states that if client signing keys and SAtk *are equal then all channel keys are equal as well (*SC1 = SC2*) and the client and server's identities match (*pk1 = C_pk2 && S_pk1 = S_pk2*) or both of the client's and server's long-term keys leaked (*event(leaked(C_pk1)) && event(leaked_server(S_pk2))*).*

After adding this invariant we can verify again the main property (Agr_S^-) and observe that the false assumptions are indeed discarded by the invariant. Often, this requires several iterations to gradually remove false assumptions and direct ProVerif to a successful proof.

Example 6 (Continuing the running example)**.** *Rerunning ProVerif on the main property we obtain again* cannot-be-proved*. However, inspecting the derivation we observe that the session entries between client and server now coincide on channel keys and identities:*

```
event(insert_S_sessions(pk(sk_2),pk(sk_3),id_132,SAtk,SSec, chan(certify(client,RSA,pk(sk_2)),certify(
      server,RSA,pk(sk_3)),Sign,ch_n_1, keys(Token_1, enc_key_RSA(C_Nonce_1,S_Nonce_1), sign_key_RSA(
      C_Nonce_1,S_Nonce_1), enc_key_RSA(S_Nonce_1,C_Nonce_1), sign_key_RSA(S_Nonce_1,C_Nonce_1))),
      usr_7,U_tk_8,S_er_2,S_nonce_22))

event(insert_C_sessions(pk(sk_2),pk(sk_3),id_131,SAtk,check_5, chan(certify(client,RSA,pk(sk_2)),certify(
      server,RSA,pk(sk_3)),Sign,ch_n_1, keys(Token_1, enc_key_RSA(C_Nonce_1,S_Nonce_1), sign_key_RSA(
      C_Nonce_1,S_Nonce_1), enc_key_RSA(S_Nonce_1,C_Nonce_1), sign_key_RSA(S_Nonce_1,C_Nonce_1))),
      usr_8,U_tk_7,O_hk_7,O_nonce_9,C_sk_4))
```

However, they still differ on sessions ids and users. We identify additional false assumptions and add invariants to remove them, we list them all in [14].

D.2.6 Step 5: Proving the invariants

Once enough invariants are added and allow ProVerif to conclude the proof of the main property (without cannot-be-proved), we need to prove the invariants themselves. Indeed, since invariants are assumed to hold by ProVerif, adding an invariant that does not hold could therefore lead to wrong results.

Hence, we let ProVerif *prove* each invariant that we added to hold. This is done by removing the noselect statements on the facts identified at step 1 and declaring the axioms as queries. (These modifications to the files are handled by our script that declares the invariants as axioms when proving the main property and declares them as queries when checking their validity.)

Example 7 (Continuing the running example)**.** *The above invariant can actually easily be proved by ProVerif in about 1 second as a query.*

Remark 1. *ProVerif also has a notion of* lemma *that may be used as a proof helper. Unlike an* axiom*, a* lemma *is automatically proven. However, in our case the* noselect *statements would apply to both lemmas and queries and prevent ProVerif from proving the lemmas.*

D.2.7 Optional: Chaining invariants

Note that for proving some invariants, ProVerif may require other invariants/axioms (using the same methodology) to avoid the proof search to loop. In this case, we follow the same methodology where the main query becomes the invariant whose proof loops. Once this is completed, we can declare the new invariant as an axiom and prove the main property. We repeat this process until all invariants are proven and the main property is proven assuming those invariants. Resulting from this is a *dependency graph*: the main property is the root node (with no incoming edges), the invariants that need to be assumed to prove this main property are successor nodes, and the invariants that can be directly proven without assuming any other invariant are leaves, *i.e.,* they do not have any successor nodes. We make sure this dependency graph is indeed acyclic, *i.e.,* it is a Directed Acyclic Graph (DAG).

Example 8 (Continuing the running example)**.** *Part of the dependency graph of the invariants for the* $\mathsf{Agr}_{\mathsf{S}}^{-}$ *property is depicted below:*

```
agr_S^-
 |- 3.1.B
 |   |- 3.1.axioms.1, 3.1.axioms.2, [...]
 |
 |- 3.1.axioms.1, 3.1.axioms.2, [...]
 |
 [...]
```

In particular, we can see that:

1. *ProVerif can prove* $\mathsf{Agr}_{\mathsf{S}}^{-}$ *assuming the invariant* 3.1.B *and the series of invariants* 3.1.axioms.1*,* 3.1.axioms.2*, etc. (we omit some axioms for the sake of readability, see [14] for the complete DAG).*
2. *ProVerif can prove the invariant* 3.1.B *assuming the series of invariants* 3.1.axioms.1*,* 3.1.axioms.2*, etc.*
3. *ProVerif can prove the series of invariants* 3.1.axioms.1*,* 3.1.axioms.2*, etc.. without any assumption. Note that ProVerif proves this series of properties at once, i.e., in bulk.*

This dependency graph is useful to ensure that all invariants are eventually proven and that no cyclic dependencies are present. For our analysis of OPC UA, the largest dependency graph we used has a depth of 3, as the one depicted above.

D.2.8 Soundness

We emphasize that our methodology is sound and results in a fully checked proof. Indeed, by [7, Theorem 2], we know that given a protocol model, a query φ, and axioms $\mathcal{A}$, if

- each axiom in $\mathcal{A}$ holds for the protocol model in the applied pi calculus, and
- ProVerif outputs true for the query φ under axioms $\mathcal{A}$,

then φ holds on the protocol model in the applied-pi calculus.

As we first prove with ProVerif the invariants $\mathcal{A}_0$ that do not rely on any other axiom, we obtain by applying the theorem that those axioms $\mathcal{A}_0$ hold on the protocol model in the applied-pi calculus. Now, we can assume the invariants $\mathcal{A}_0$ as axioms to prove another property, say the main property φ, and guarantee that the first and second conditions of the theorem hold. We thus obtain that the target property φ holds on the protocol model in the applied-pi calculus.

If we used a dependency DAG of invariants, we can apply the theorem iteratively, starting from the leaves up to the root, to ensure that all invariants and the main property hold on the protocol model in the applied-pi calculus.

In a nutshell, soundness relies on the fact that all axioms are first proved as queries before being assumed, and that ProVerif guarantees soundness independently of name contexts and other proof options.

D.3 Full Results

In the companion artifact [14] we provide text files with the exhaustive list of results.

E List of Changes

- v1.0, January 31, 2025: Initial version, camera-ready version for USENIX Security 2025.