

# Automated Verification of Electrum Wallet

Mathieu Turuani<sup>1</sup>(✉), Thomas Voegtlin<sup>2</sup>, and Michael Rusinowitch<sup>1</sup>

<sup>1</sup> INRIA Nancy–Grand Est, Villers-lès-Nancy, France

`mathieu.turuani@inria.fr`, `rusi@loria.fr`

<sup>2</sup> Electrum Technologies GmbH, Berlin, Germany

`thomasv@electrum.org`

**Abstract.** We introduce a formal modeling in ASLan++ of the two-factor authentication protocol used by the Electrum Bitcoin wallet. This allows us to perform an automatic analysis of the wallet and show that it is secure for standard scenarios in Dolev Yao model [Dolev 1981]. The result could be derived thanks to some advanced features of the protocol analyzer such as the possibility to specify (i) new intruder deduction rules with clauses and (ii) non-deducibility constraints.

## 1 Context

Electrum is a popular Bitcoin Wallet. Thanks to a deterministic key derivation algorithm (BIP32), users can regenerate their wallet from a secret seed phrase, which protects them in case of loss or computer failure. It is a lightweight client, which means that it does not need to download the whole Bitcoin blockchain. Instead, the client communicates with a set of servers, and retrieves only needed information. The private keys used to sign Bitcoin transactions are never communicated to the servers, and servers do not store users accounts.

In order to protect users from Bitcoin theft, Electrum provides two-factor authentication, implemented using multi-signature addresses (P2SH) and an external co-signer (TrustedCoin). Our objective is to initiate the verification of Electrum’s two-factor authentication protocol in order to increase user confidence or detect potential weaknesses and rise warnings.

Several tools have been recently developed to perform fully automated analysis of cryptographic protocols (e.g. [Proverif]). Some have been able to discover new flaws and most of them rely on symbolic models where messages are considered as terms in some abstract algebra as opposed to sequences of bits in more concrete models. In these tools generally no properties are assumed (and exploited) about cryptographic primitives besides the fact that when a message has been encrypted by some key, it can be recovered by applying the inverse key to the ciphered text. Although symbolic analysis relies on high-level protocol abstractions it has been able to discover important flaws in real-world protocols [Armando 2008]. Moreover, under some suitable hypothesis these analyses are cryptographically sound, i.e. from the absence of flaws at the symbolic level we can derive the correctness of the protocol w.r.t. cryptographic models too.

---

This work has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 645865-SPOOC).

For this analysis we have employed Cl-Atse, a state-of-the-art protocol analyzer efficient and complete for bounded number of sessions [Turuani 2006]. It is compliant with ASLan++ specification language [von Oheim 2010], which allowed us to model relevant properties of the BIP32 key-derivation functions. Moreover, Cl-Atse’s unique ability to handle so-called non-deducibility constraints over the agent’s knowledge was fundamental in this analysis.

There exists a huge number of works on formal verification of security protocols. For instance, Bitcoin contracts have been modeled and verified using timed automata [Andrychowicz 2014]. However, to our knowledge previous verification works have not considered Bitcoin wallets.

## 2 Electrum Wallet

Electrum users may decide to enable two-factor authentication on their wallet. In that case, transactions will be signed by both the Electrum client and a cosigning server (TrustedCoin), if the user authenticates himself using a one-time password generated by Google Authenticator. The user (the human who owns the electronic wallet) keeps an offline copy of the wallet initialization data (seed phrase), in order to regenerate their wallet in case of data loss or disappearance of the cosigning service. A key feature of Electrum’s two-factor authentication protocol is that the wallet regeneration procedure only requires the seed phrase, and does not need the cosigning server.

The cosigning server, however, has no way to use the wallet without the client’s signature. Electrum’s two factor authentication uses Bitcoin Pay-to-Script-Hash addresses (P2SH), and the BIP32 standard for deterministic generations of keys. BIP32 allows the cosigning server to deterministically generate new private keys - following a ‘path’ from a root key - which is needed to sign Bitcoin transactions, while the client alone is able to generate the corresponding public keys, which are needed in order to create new P2SH addresses.

The interactions between agents are described by four sequences of actions: initialisation (user/client), registration and confirmation (client/server), and (recurring) transaction phase. Authentication is carried out via the Google OTP function using a secret and the current time. BIP32 extended keys are denoted by  $k$ ,  $K$ ,  $C$  for private/public keys and the associated chain parameter. Personal data are denoted by: *email* for the client’s e-mail (public); *time* for the current time (public); *database* for the server’s database (private); and  $a$ ,  $b$ ,  $c$ ,  $d$ , ... for some paths (public) used to build child keys in the BIP32 structure.

### *User and Client’s Initialization.*

<b>Initial knowledge</b>	: Both knows $K_s$ , $C_s$ ;	
User generates two fresh seeds	: Seed1 and Seed2;	-secret-
User extracts the priv./public	: $k_1, K_1, C_1$ from Seed1;	-secret-
keys and chain parameters	: $k_2, K_2, C_2$ from Seed2;	-secret-
User $* \rightarrow *$ Client	: email.k1.C1.K1.C2.K2	
Client computes the user IDs	: LongUserID from $K_1, C_1, K_2, C_2$ ;	-secret-
	UserID prefix of LongUserID;	



Now, the client must confirm his registration before signing transactions. The Electrum client does not store the OTP secret, but displays it to the user who stores it on an external device with Google Authenticator. However, this cannot be modeled directly since the D.Y. intruder rather concentrates on internet communications between the client and the server. Therefore, this step is abstracted by a client who generates the OTP himself. The confirmation phase starts with the client opening a secure channel as before to send his request and OTP. Depending on server's agreement, *true* or *false* is returned and the client data is either confirmed or deleted from the database. The *exit* keyword indicates procedure termination for both parties when the server disagrees.

*Signing a Transaction with the Server.*

<b>Initial knowledge</b>	: Client was confirmed; Server knows ks, Ks, Cs;	
Client opens a channel	: Session key Sk shared with Server;	-secret-
Client creates a list	: SigList = empty list of signature requests;	
<hr/>		
<b>Client iterates</b>		
choose a path	: Path = sequence of a, b, c, d, ...	
create a script	: Script using key from K1,C1 following Path;	
add to the list	: Add Path.Script to SigList;	
Client builds the OTP	: OTP = googleOTP(Sec,time);	-secret-
Client → Server	: { sign.UserID.tx_data.SigList.OTP }_Sk;	
Server reads database	: UserID.email.K1.C1.K2.C2.true	
Server builds the Secret	: Sec from UserID, Ks and Cs;	-secret-
Server tests the OTP	: OTP ?= googleOTP(Sec,time);	
<hr/>		
<b>if data not found or OTP differs</b>		
Server → Client	: { false }_Sk; <i>exit</i> ;	
<hr/>		
<b>otherwise Server iterates</b>		
select element	: pick-up Path.Script from SigList;	
derive new key	: k4 from ks,Cs following LongUserID.Path;	
sign the transaction	: Sign tx_data and Script with k4 in signed_tx;	
Server → Client	: { signed_tx }_Sk;	

Once the client is registered, it is allowed to request transaction signatures. As before, this is done by opening a secure channel with the server, and sending a transaction request containing the one-time password generated with the current time. The request also includes the transaction data *tx\_data* to be signed, along with a list *SigList* of signature requests. Each element in this lists consists in a Bitcoin script using a key derived from the client's root key *K1* (plus *C1*) using the BIP32's CKD\_pub (and CKD\_priv) method. The path used to derive this key is also included in the list, so that the server can use it to derive his own key too. This method allows the client and the server to forget all the keys locally used in each transaction, as long as the root keys are safely stored along with the paths used to derive new keys. Moreover, the two keys used to sign transactions - client side and server side - are generated using the same, unique, path. On the server side, this path is extended with the long user ID so that all the keys used for all the clients are generated from one, single, root key pair

$ks, Ks$ . Moreover, the server has no need to store key paths: the responsibility to keep these is left to the client. In practice, these paths are of length two and sequentially generated, so that a fast sieve allows the user to recover his keys in case of client loss. Once successful, the server returns the signed transaction to the client. While not modeled here, the Electrum server can also push the signed transaction to the Bitcoin network for the client.

### 3 Modeling BIP32

The BIP32 standard describes a hierarchy of keys derived from a single seed and by following paths, i.e. a sequences of integers. The master key generation function is a hash and does not require any specific construction. The Child Key Derivation (CKD) functions admit two variants: CKD<sub>pub</sub> for deriving public keys and CKD<sub>priv</sub> for private keys. Each one takes a public (resp. private) key, a chain parameter, and a path to follow, and produces the child public (resp. private) key. Consequently the algebraic relation linking CKD<sub>pub</sub> and CKD<sub>priv</sub> is:  $CKD_{priv}(inv(K), C, S) = inv(CKD_{pub}(K, C, S))$  where  $inv$  is the key inversion function, used here to specify the private key associated to the public key  $K$  or to  $CKD_{pub}(..)$ . For simplicity we have omitted the chain parameter in our description: it will be modeled by a function called cCKD in our ASLan++ specification. In our model, each public key is either the root public key  $pk(A)$  of some agent  $A$ , or a child public key derived with BIP32. Note that  $pk$  already exists in ASLan++ syntax and thus will not be redefined. The two CKD functions can be modeled by declaring a single  $pCKD$  function which takes a private key (as would CKD<sub>priv</sub>) but computes the corresponding public child key (as would CKD<sub>pub</sub>):

```
noninvertible pCKD(private_key,message,message): public_key;
```

where *message* is the generic type (used for a chain parameter and a path here), and *noninvertible* ensures that this function can be freely used but the intruder cannot retrieve the arguments from the result.

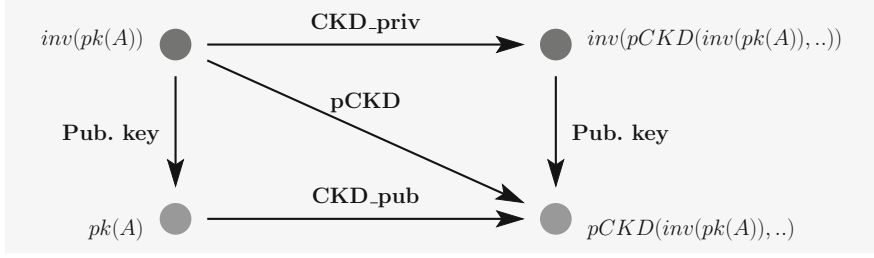
In our protocol model every occurrence of  $CKD_{pub}(K, C, S)$  from the initial specification is represented by  $pCKD(inv(K), C, S)$  and every occurrence of  $CKD_{priv}(K, C, S)$  is represented by  $inv(pCKD(K, C, S))$ . This modeling ensures the validity of the above algebraic relation since for all  $K, C, S$ :

$$CKD_{priv}(inv(K), C, S) = inv(pCKD(inv(K), C, S)) = inv(CKD_{pub}(K, C, S))$$

The intruder deduction ability should be extended to take into account the CKD functions from BIP32. Therefore we define these two new intruder deduction rules expressed by Horn Clauses in ASLan++:

```
iknows(pCKD(inv(K),C,S)) :- iknows(K.C.S);   \% for CKD\_pub
iknows(inv(pCKD(K,C,S))) :- iknows(K.C.S);   \% for CKD\_priv
```

The ability for the intruder to deduce a public key from its inverse (private) key is expressed using two Horn Clauses:



**Fig. 1.** Example of key derivations with fixed chain and path

```

iknows(pk(A))          :- knows(inv(pk(A)));
iknows(pCKD(K,C,S))    :- knows(inv(pCKD(K,C,S)));

```

These functions are summarized in Fig. 1, where each arrow represents an operator over keys, where both the chain parameter and the path are fixed.

## 4 ASLan++ Wallet Model

The ASLan++ model for the Electrum wallet is detailed here. It specifies the user agent and his client software (separately), plus the three server agents described previously. It also specifies the security properties to be preserved in any run, plus the intruder capabilities. Conforming to BIP32's modeling from Sect. 3, the CKD\_priv and CKD\_pub function are replaced by combining *pCKD* and *inv*. Note that *pCKD* is universally known and thus can be used by the intruder. The list containing the elements that the client want to be signed, and the list of these elements once they are signed by the server, are represented with concatenation of pairings. While the model contains variants for choosing *Input\_info* (the list of signature requests), only the most general one where the intruder has full control over it is used for the analysis. Moreover, the ASLan++ syntax uses names starting with a non-capital letter to denote constants, e.g. *atoms/nonces/...*, and names starting with a capital letter for variables. Therefore, a key pair denoted by *Ks, ks* (public/private) in the description of the protocol will now be written *Ks/inv(Ks)* instead, with *Ks* a variable of type *public\_key* and *inv(Ks)* the inverse of its value, thus the associated private key.

### 4.1 Attacker Model and Assumptions

Our analysis relies on the perfect cryptography hypothesis: no information about the content of a ciphertext can be derived by agents that do not possess the decryption key. Fresh random numbers used in a ciphering algorithm cannot be derived either. Random numbers cannot be predicted by the intruder. A ciphertext is in general not malleable and therefore cannot be used to generate a different ciphertext without being decrypted first. This can be slightly relaxed

when the protocol uses specific operators (Xor, exponentiation, ..): the algebraic properties of these operators that can be exploited are made explicit. Horn Clauses are also used to express agent capabilities to modify or learn messages that goes beyond the perfect cryptography hypothesis.

A honest agent follows the protocol specification and is assumed to be *prudent*, that is, the agent decrypts as soon as possible every part of a received message, checks the content by relating it to its known data. In the same way signatures are verified by agents whenever possible.

## 4.2 Security Properties

While the built-in ASLan++ security properties are fine for protecting objects with scope larger than just one session, they were not adequate for protecting local objects inside sessions where one agent (e.g., the server) has no idea, and do not truly care, if he is speaking to a honest client or not. The server has to serve any client, honest or not, and he has no idea which one will contact him. Therefore, instead of tweaking the built-in properties, we have designed new specific ones for this kind of model:

```
uSecret(message, message) : fact;
uChecks(agent, message, message, message, message) : fact;
```

which associated security properties are:

```
uSecret: [](forall M K. (!uSecret(M,K) |
    !iknows(M) | iknows(K) ));
uChecks: [](forall A M K U V. (!uChecks(A,M,K,U,V) | A=i
    | K=M | !U=V));
```

where  $[]$  means ‘in every trace at any moment’ and  $!$  denotes the negation. The first one,  $uSecret(M, K)$ , ensures that the message  $M$  remains unknown for the intruder, unless he can build or deduce  $K$ . This is widely used to preserve the security of some data inside a session, where  $K$  will be the session key. This way, an attack is raised on a local data, or a data transmitted during a session, only if the intruder has no access to the main key protecting it. Naturally, the session key itself has its own security property, to check whether it can be captured by the intruder. This method builds an interesting hierarchical structure among security properties reflecting the dependencies between data and the means used to protect them. This was made possible only thanks to the recent extension of Cl-Atse (the analysis backend used for this study) that allows it to check for non-deducibility constraints over the intruder knowledge. Indeed, this does not simply require that the key  $K$  was received by the intruder, but that among the infinite set of possible intruder knowledge solutions of some set of constraints, only those where he is unable to deduce that key, or to rebuild it from pieces, are kept.

The second property,  $uChecks(A, M, K, U, V)$ , ensures that if the agent  $A$  is not played by the intruder, then either the message  $M$  and  $K$  are equal, or the messages  $U$  and  $V$  differ. This is widely used to check the server’s responses to

the client's requests. Therefore, in this context an attack is conservatively raised iff: (i) the agent  $A$  was an honest agent; (ii) the response  $M$  from the server differs from the expected result  $K$ ; (iii) the request (list of) parameter  $U$  was correct, i.e. identical to its expected value  $V$ .

### 4.3 User Role

The ASLan++ *User* role models human owner(s) of an Electrum wallet to be distinguished from a role played by software client(s), that owns only a subset of User's knowledge. The User role appears only during the initialization phase as described in Sect. 2. Thus, he will first generate two seeds using the *fresh()* nonce generator, and declares them as secrets:

```
secret_Seed1:(Seed1) := fresh();
secret_Seed2:(Seed2) := fresh();
```

Then, he derives two key pairs  $K1/inv(K1)$  and  $K2/inv(K2)$  from the seeds, plus their associated chain parameters  $C1$  and  $C2$ , and declares the private keys as secrets. The chain parameters (or the public keys) being sent to the server, they can also be declared as secrets (for testing, as they are not critical) but under the assumption that the server was not compromised (actively or passively), i.e. that its root key remains private:

```
K1 := pk(Seed1); uSecret(inv(K1),Seed1);
K2 := pk(Seed2); uSecret(inv(K2),Seed2);
C1 := cp(Seed1); C2 := cp(Seed2);
uSecret(C2,inv(pk(server)));
```

Finally, he gives these initialization data to his client, necessarily through a secure channel:

```
Actor *->* Client : email.inv(K1).C1.K1.C2.K2;
```

The secure channel is specified in ASLan++ syntax by arrow decorations: left star for origin authenticity, and right-star for read-protection and guaranty of delivery to the correct recipient.

### 4.4 Client Role

The *Client* role is played by Electrum software client on behalf of the user. After a first initialization phase:

```
User *->* Actor : ?Email.inv(?K1).?C1.?K1.?C2.?K2;
```

where it receives its parameters from the User role, it computes the long user ID (*Long-ID*) and declares it as secret, truncates it to *UID*, and deduces the main server's public key  $K3$  and chain parameter  $C3$  devoted to this user. Also non essential or obviously public data are leaked to the intruder, in case some real user would do so involuntarily:



```

K3  := pCKD(inv(pk(server)),cp(server),Long_ID);
C3  := cCKD(      pk(server) ,cp(server),Long_ID);
Actor -> ? : Email.K1.C1.K2.K3.C3;

```

Then, the clients starts a registration process to the server. To do so, it first generates a fresh session key  $Sk$  and declares it as secret, then transmits it to the server through a read-protected channel followed by the registration request protected by that key:

```

secret_Sk:(Sk) := fresh(); Actor ->* Server : Sk;
Actor -> Server :
  {|register.{Email.K1.C1.K2.C2}_pk(server)|}_Sk;

```

After processing the request, the servers gives its answer protected by the session key too:

```

Server -> Actor : {|{?Secret}_K1|}_Sk; uSecret(Secret,Sk);

```

This answer contains the *Secret* that will be used for further authentication using the *google\_OTP* function. Naturally *Secret* is declared as a secret w.r.t. the session key  $Sk$ . This goal is made inactive if the intruder can deduce or compute  $Sk$ , namely if the session officially involves an intruder (but this is not an attack) or if the session was compromised (but the goal associated to  $Sk$  must trigger instead). Then, following the same method and using an OTP declared as secret and generated through *google\_otp(Secret,Time)* with *Time* being the current time of the client (correct or not), it casts the confirmation request (with a new session key) and gets the server *Result*, that could be *true* or *false*. This response is checked for validity:

```

uChecks(Server,true,Result,ctime,Time);

```

i.e. the *Result* must be *true* unless the client did not use the current time properly. This may generate an attack, see Sect. 5.2. Finally, the client casts a signature request. This request uses a *Raw\_transaction* which is some fresh data leaked to the intruder, plus a list of signature requests called *Input\_info* which must be as general as possible, and thus, directly provided by the intruder at its own discretion. This way, the intruder can choose the most dangerous values for the client. At the end, the client checks the server's response similarly as before.

## 4.5 Server Registration Role

The registration role is the first of the three server roles. Its goal is to record new user's registration data. The model does not even need to check if a previous user was already registered with the same ID, since the probability to find hash collisions by chance is considered to be negligible. Therefore, this role first opens a new connection to a client by receiving a session key, though a read-only channel, followed by the request itself:

```

Client ->* Actor : ?Sk;
Client -> Actor :
  {|register.{?Email.?K1.?C1.?K2.?C2}_pk(server)|}_Sk;

```

This allows him to rebuild the long and short user ID in the same way as the client did on his side, along with the OTP's secret which is an hash over the user's short ID, the private server's key, and the server's chain parameter:

```
Secret := hash(UID.inv(pk(server)).cp(server));
uSecret(Long_ID, Sk); uSecret(Secret, Sk); uSecret(Long_ID, C2);
```

Naturally these data are also declared as secrets, w.r.t. the session key  $Sk$  since the data allowing to rebuild  $Long\_ID$ , and  $Secret$ , are both transmitted in a channel protected by  $Sk$ . However, we can notice that losing the session key  $Sk$  here does not necessarily implies losing the long user ID, since the data building it are also protected by the server's public key. Therefore, it is declared as secret a second time, but w.r.t. one of its element that was already declared as secret by the User (c.f. Sect. 4.3), i.e.  $C2$ . We note here that this study does not consider cryptographic attacks on the primitives. In particular, the server and the hashing method must be resistant by other means to an intruder performing e.g. huge series of (fake) client/server registrations to retrieve many pairs of  $UID$  and  $Secret$  with the objective of rebuilding the secret associated to a known, honest, user ID. Finally, the server adds the client's data to its database, and sends the  $Secret$  back for further authentication:

```
database->add(UID.Email.K1.C1.K2.C2.false);
Actor -> Client : {|{Secret}_K1|}_Sk;
```

where *false* refers to the unconfirmed status, and the secret is protected by both the session key and an encryption with the public key  $K1$ .

## 4.6 Server Confirmation Role

The confirmation role is mandatory to allow the client to send further requests. It ensures that Google OTP can be used for authentication and if so, switch its status from unconfirmed to confirmed. The server starts this role as before, by receiving a new session key though a read-only channel, followed by the request itself containing the user ID from which the secret can be rebuilt:

```
Client ->* Actor : ?Sk;
Client -> Actor : {|confirm.?UID.?OTP|}_Sk;
Secret := hash(UID.inv(pk(server)).cp(server));
```

This way, the server does not need to store each user's secret in the database. Then, the server has three choices, depending on the validity of the one-time password and the content of the database for this user ID:

- Either the database contains a non-confirmed user under that ID, and the provided password was correct. Then the server validates the request by switching the user status to confirmed in the database, and sending back the *true* response:

```
database->remove(UID.Email.K1.C1.K2.C2.false);
database->add( UID.Email.K1.C1.K2.C2.true );
Actor -> Client : {|response.true|}_Sk;
```

- Or the database contains a non-confirmed user under that ID, but the provided password was not correct. Then the server deletes (yes, it does!) the unregistered user account from the database, and sends back a *false* response. Whatever the reason was, a user mistake or an intruder trying to interfere with a wrong password, in both cases the client has to register again:

```
database->remove(UID.Email.K1.C1.K2.C2.false);
Actor -> Client : {|response.false|}_Sk;
```

- Or the database does not contains any non-confirmed user under that ID. If there exists some user registered and already confirmed under that ID, it is not seen here and thus it remains unchanged. Then the server simply sends back a *false* answer:

```
Actor -> Client : {|response.false|}_Sk;
```

Moreover, in all three cases above, the server also declares the local data as secrets within the session, i.e.:

```
uSecret(Secret,Sk); uSecret(OTP,Sk);
```

Other cases than these three could be introduced in future protocol variants. For example, the server could send an adequate response when the user is already registered. Depending of these future improvements on the server's tests and actions on data, a new analysis would be necessary.

## 4.7 Server Signature Role

The signature role is the master piece of the Electrum two factor authentication protocol. It allows the client to cast Bitcoin transactions signed both by him and the server. This is the most complex part of the protocol since the number of signatures to perform for one single request is not fixed, and thus requires an iterative process. The server starts this role as usual by receiving the session key followed by the request and uses it to rebuild the user secret and query the database:

```
Client ->* Actor : ?Sk;
Client -> Actor :
    {|sign.?UID.?Raw_transaction.?Input_info.?OTP|}_Sk;
Secret := hash(UID.inv(pk(server)).cp(server));
if (database->contains(UID.?Email.?K1.?C1.?K2.?C2.true)
    & OTP = google_otp(Secret,ctime)) { ...
```

Similarly to the confirmation role, the server tests the database content and one-time password for user authentication. If this test is positive then the server processes *Input\_info*, i.e. the list of objects that must be signed, by the server for some of them, and not for some others. This list is modeled by right-parenthesised pairings, and the server iterates the process until no more pairing is found (the last element in the list is supposed to be *end*):

```

while (Input_info = ?Elem.?Next) {
  if (Elem = mustSign(?A.?B.?Script)) { ...

```

The list elements of the form *mustSign(..)* are objects that the server is asked to sign. Others are objects that can be simply ignored. For each object of this kind found in the list, thus for each elementary signature request *A.B.Script* found inside, the server computes the key pair  $K3/inv(K3)$  derived from his private root key using BIP32 and following the long user ID plus *A* and *B* as path. For readability, this is decomposed in two steps here:

```

K3 := pCKD(inv(pk(server)),cp(server),Long_ID);
C3 := cCKD(      pk(server) ,cp(server),Long_ID);
K4 := pCKD(inv(pCKD(inv(K3),C3,A)),cCKD(K3,C3,A),B);
C4 := cCKD(      pCKD(inv(K3),C3,A) ,cCKD(K3,C3,A),B);

```

An implementation of the server would certainly build  $K3/inv(K3)$  before the loop, but this has no impact on the analysis. The server can now use  $inv(K4)$  to sign the requested element, and add it to the list of signatures that he is building, and proceed to the next element:

```

Signatures := Signatures.{Raw_transaction.Script}_inv(K4);
Input_info := Next; secret_K4p:(K4p) := inv(K4);
Empty := false;

```

The private key  $inv(K4)$  is also declared as secret, but this time, for the server: it is used for signing, so it must remain perfectly private to the server even if this one communicates with a compromised client. The *Empty* variable records the information that the signature list is not empty anymore. This is used right after the end of the iterative process, to declare the final signature list as a secret w.r.t. the session key (since it is transmitted during the session):

```

if (Empty=false) { uSecret(Signatures,Sk); }

```

Note that this secrecy goal would be trivially invalidated without the condition. Finally, the server can send back the signed list to the client:

```

Actor -> Client : {|response.Signatures|}_Sk;

```

If the password is incorrect or if the database does not contain a confirmed used of that ID, then the server simply replies *false* instead of the signature list. In both cases (correct or not), the secrecy of the local data (such as the user IDs or the OTP secret) is ensured w.r.t. the session key, in the same way as in the previous roles.

## 5 Results

We have performed several analyses using our ASLan++ model of Electrum's two factor authentication protocol. We first tested the executability of the protocol, i.e. to make sure that participants can truly run their parts as expected. Second we present a replay attack on the confirmation message that was unveiled by the tools. This attack does not threaten private data but makes the client

erroneously believe that it is in some state with the server and this may block him in future actions. Finally we evaluate the protocol security according to several scenarios and show that no attack is possible for these configurations (beside the previously mentioned one). The investigated scenarios cover standard uses of the protocol (with honest server and client) but also critical cases where the client or the server is dishonest. We have employed a computer cluster for increasing the number of protocol sessions considered during our experiments.

## 5.1 Executability Checking

To check the absence of blocking during protocol execution and therefore that the protocol can reach a successful final state, we have considered several scenarios too. We consider here the case of a single honest client and a single honest server. Hence the *body* section contains:

```
newSession(alice, client1, server, inv(pk(server)));
```

Moreover the fact that a client role receives a correct final answer from the server is encoded with an ASLan property added at the end of the client role.

```
uChecks(Actor, true, false, Signatures,
        start.{Raw_transaction.Script2}_inv(SerKey2).
        none.{Raw_transaction.Script1}_inv(SerKey1) );
```

This property requires that the signature *Signature* sent by the server is exactly *data*. For the server to provide this signature, the client request must be properly constructed. Cl-Atse generates this request. Signatures are built from paths *a, b* and *c, d* that have been used to compute the public keys *SerKey1* and *SerKey2* with the corresponding scripts, i.e.:

```
K3 := pCKD(inv(pk(server)), cp(server), Long_ID);
C3 := cCKD(pk(server), cp(server), Long_ID);
SerKey1 := pCKD(inv(pCKD(inv(K3), C3, a)), cCKD(K3, C3, a), b);
SerKey2 := pCKD(inv(pCKD(inv(K3), C3, c)), cCKD(K3, C3, c), d);
```

## 5.2 Attack by Confirmation Replay

In a scenario where the client and the intruder have each one an open session with the server, the intruder has the possibility to replay the registration confirmation sequence of the client towards the server. This is possible since the server authenticates the client solely by its transmitted information, basically *UserID* and *OTP*. The fact that the same session key is reused is neither detected by the server in our modeling, nor in real implementations. The fact that the *OTP* password used in a first request does not expire instantaneously and therefore remains valid for a second immediately following a session is not exploited here. However if an already confirmed client send a new confirmation request then according to the technical documentation the server will check whether the client is already recorded with *not confirmed* tag in the database. Hence the test will

fail since the client is recorded with another tag. By consequence the second session fails and the server returns *false*. Then the intruder needs only to replace the positive answer to the first request by the negative answer to the second one to deceive the client into thinking that its confirmation was denied. Moreover, by believing so the client will probably not emit signature requests. If he tries again to get a confirmation he will be rejected again without any action from the intruder. This forces the client to initiate a new registration. In order to correct this protocol behavior the server should check after a confirmation request if the client is already confirmed and reply positively in that case.

Cl-Atse has generated automatically the following attack trace that corresponds. Agents between brackets  $<$  and  $>$  are controlled by the intruder. We note that *client1* sends a request to the server that is duplicated, and he only receives the second answer.

```

client1    ->    <server> : _msg
<client1>  ->*    server   : n19(Sk)
<client1>  ->    server   : _msg
server     ->    <client1> : {|response.true|}_n19(Sk)
<i>        ->*    server   : n19(Sk)
<i>        ->    server   : _msg
server     ->    <i>      : {|response.false|}_n19(Sk)
<server>   ->    client1  : {|response.false|}_n19(Sk)

```

where for readability *\_msg* (and *\_uid*) are shortened forms of:

```

_msg: {|confirm._uid.googleOTP(hash(_uid.inv(pk(server))).
                                     cp(server)),time)|}_n19(Sk)
_uid: first10(hash(pk(n17(Seed1)).cp(n17(Seed1))).
               pk(n17(Seed2)).cp(n17(Seed2))))

```

*Suggestion:* the server's answer could be more explicit to let the client understand that it is under attack. Also, the client must not accept a server's response if it arrives after the end of validity for the OTP, to prevent replay. The non-confirmed data deletion is acceptable if the client understands that he must restart from registration because an intruder interfered.

### 5.3 Security Analysis

**Case of Dishonest Server:** With a compromised server only few security properties can be preserved for the client. In particular, secrecy of *C2*, *LongUserID*, *Secret*, *OTP* is lost. However, the client's master private keys and their children should remain secret even with a compromised server. This includes *K1\_priv* stored by the client and *K2\_priv* stored by *Human* role and the seeds that have been used for generating keys.

The model features a server that is compromised and thus, whose private root key is provided to the intruder. Therefore, its security properties are ignored and the intruder forces the server to follow or not the specification. The analysis of this specification by Cl-Atse shows that the client's security properties that are not related to data shared with the server are preserved.

**Case of Honest Server:** This is the main case. The number of honest clients running in parallel is a source of combinatorial explosion of the number of traces to analyze. To perform the analysis we have employed 50 nodes of a computation cluster. It appears that the limit was reached when analyzing this model for 2 to 3 concurrent sessions, each one being sequentially iterated 3 to 4 times. Each session is a block specifying one human owner and his software client, plus each of the three server's roles.

Finally, the scenarios containing two sessions featuring either two honest clients or one honest and one dishonest were found to be secure for up to four iterations (option ‘`--nb 4`’ of the tool). We have tried to speed-up analysis by ‘branching’ the client sessions to specific server sessions (these are perfectly interchangeable anyway) since this eliminates a large number of equivalent executions, but a scenario without this ‘branching’ was also tried and appeared to be truly difficult, when running on 30 nodes of the cluster for a week.

## 6 Conclusion

In this paper we have modeled Electrum's two factor authentication protocol using the ASLan++ language advanced features. Conditional security goals allow us to model several scenarios in one shot, without knowing beforehand which agent will be compromised. Horn Clauses allow us to model intruder capabilities (e.g. exploiting BIP32 related properties) in a flexible way and beyond standard Dolev Yao deduction rules. Our computer experiments have pointed a potential problem in the user registration process, and have shown that, assuming perfect cryptography, the protocol offers good security guarantees in standard scenarios. We have not studied privacy properties in this paper. In particular, a further analysis could check if an intruder is able to relate transactions that belong to the same user.

## References

- [Andrychowicz 2014] Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Modeling bitcoin contracts by timed automata. In: Legay, A., Bozga, M. (eds.) FORMATS 2014. LNCS, vol. 8711, pp. 7–22. Springer, Heidelberg (2014)
- [Armando 2008] Armando, A., et al.: Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps
- [Dolev 1981] Dolev, D., Yao, A.: On the Security of Public Key Protocols (Extended Abstract). In: FOCS, pp. 350–357 (1981)
- [von Oheim 2010] von Oheimb, D., Mödersheim, S.: ASLan++ — a formal security specification language for distributed systems. In: Aichernig, B.K., Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 1–22. Springer, Heidelberg (2011)

- [Proverif] Proverif. <http://prosecco.gforge.inria.fr/personal/bblanche/proverif/>
- [Turuani 2006] Turuani, M.: The CL-Atse protocol analyser. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 277–286. Springer, Heidelberg (2006)
- [Wuille 2012] Wuille, P.: Hierarchical Deterministic Wallets. Online specification for BIP32. <https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki>