Private Authentication with Alpha-Beta Privacy

Laouen Fernet[1] Sebastian Mödersheim[2]

Abstract: Alpha-beta privacy is a new approach for security protocols that aims to provide a logical and intuitive way of specifying privacy-type goals. Recently the tool *noname* was published that can automatically analyze specifications for a bounded number of sessions, but ships only with a few simple examples. This paper models two more complicated case studies, namely the ICAO 9303 BAC and the Privacy Authentication protocol by Abadi and Fournet, and applies the noname tool to analyze them, reproducing known vulnerabilities and verifying the corresponding fixes, as well as providing a better understanding of the privacy properties they provide.

Keywords: Privacy; Authentication; Unlinkability; Security Protocols

1 Introduction

The specification of privacy-type properties in security protocols is quite tricky and differs from the specification of standard secrecy goals. For a normal secrecy goal, e. g., secrecy of a key, one can simply specify which protocol parties are allowed to learn the key, and if the intruder learns the key (while not being one of the specified parties) it counts as an attack. This works for cryptographically strong secrets, but not for agent names, because they are at least in principle public values. The secret here is their relation, e. g., which agent has performed a particular transaction. The most common way to specify privacy-type properties for security protocols is based on a notion of observational equivalence between processes: one specifies two processes that represent two possible realities and the intruder should not be able to distinguish them. For instance, unlinkability in a protocol between cards and a card reader can be specified by the indistinguishability of the following two processes: one where a single card performs any number of sessions (protocol runs) with the reader, and one where any number of cards perform sessions with the reader. This expresses the goal that an intruder cannot tell whether it was all the same card or rather different cards.

Observational equivalence has several challenges: it can be rather unintuitive and technical to specify privacy goals as indistinguishabilities, and one can hardly be sure that one has not forgotten anything; moreover, automated verification seems to be only possible under major restrictions, either in what processes can do or in fixing the number of sessions. For an overview of models and methods in this area see [DH17].

[1] DTU Compute, Kgs. Lyngby, Danmark lpkf@dtu.dk ORCID: 0000-0001-9028-1480
[2] DTU Compute, Kgs. Lyngby, Danmark samo@dtu.dk ORCID: 0000-0002-6901-8319

doi:10.18420/OID2023_05

(α, β)-privacy [MV19] is an alternative approach to privacy for security protocols implemented in the novel *noname* tool for privacy analysis [FMV23]. Rather than reasoning about the distinction of two processes, a protocol is represented by a standard state transition system where every state stands for just "one reality". Every state contains two formulas α, expressing what information has been released to the public, and β, expressing what the intruder can observe. It is a violation of (α, β)-privacy, if there is a reachable state where the intruder can deduce anything from β that was not permitted by α. A modeler needs to specify what information is released in α, but does not specify β directly. Rather, the modeler specifies a protocol as a set of *transactions*, i. e., how honest agents send and receive messages and update their local state. The β of every reached state is then defined by the semantics of the transaction language and reflects all observations and deductions the intruder can make. Thus the modeler only needs to specify the protocol itself and what information is released in α and that is a positive specification, i. e., what the intruder is allowed to know. Thus, in the worst case the modeler errs on the safe side: if one forgets to specify something that the intruder can actually derive, then (α, β)-privacy is violated and the noname tool presents a violation. Then the modeler can decide whether the information that the intruder can derive is indeed acceptable, and specify that by adding an appropriate α release, or otherwise strengthen the protocol. In this way one can even *explore* what privacy guarantees a protocol gives by starting with α containing only what is obviously released and then incrementing the release until no more violations occur.

In this paper we describe exactly this process for two case studies. The first is the unlinkability in the ICAO 9303 BAC protocol for passport readers [IC], where we replicate known problems in some implementations [Ar10, CS10, Fi19]. The second is the private authentication in the Abadi-Fournet protocol [AF04] that hides as far as possible the identity of participants as well as the fact which participant is willing to talk to which other participants. To our knowledge this is the tightest characterization of the privacy goals that this protocol enjoys.

The contributions of this paper are to provide the first models with (α, β)-privacy of protocols that go beyond the one-step examples and unlinkability goals that the *noname* tool ships with. It also gives a characterization of Abadi-Fournet that is quite different from previous indistinguishability approaches and where it is unclear to us how this could be captured as an observational equivalence of processes. All specifications are available at `https://www2.compute.dtu.dk/~samo/abp/bac-et-al.tar.gz`

2 ICAO 9303 BAC

The BAC protocol [IC] is an RFID protocol used for passports with RFID card readers. We simplify matters for two reasons: it allows us to focus on the actual issue, avoiding complicated and irrelevant modeling challenges, and moreover, the noname tool quickly runs into state space explosions otherwise. The full specification is found in `bac.nn`.

The first step is that an RFID tag `x` starts a new session, with a new random nonce `N` that it sends to the Tag Reader. There is a unique key `sk(x)` for each tag `x` that is derived from the passport data (that the reader obtains using OCR). The reader uses `sk(x)` to encrypt `N` as a response to the challenge, as well as a challenge `N'` of its own. We omit `N'` here for simplicity. Also for simplicity, not to have to model the exchange of `sk(x)`, we put these two steps into one atomic transaction:

```
1  Transaction Challenge:
2  * x in {t1,t2}.
3  new N. send session(x,N). send N.
4  send scrypt(sk(x),N). nil
```

Here, Line 2 means that `x` is non-deterministically chosen from a set of two tags `{t1,t2}` (we have to bound the number of tags tightly for the tool performance). The `*` symbol indicates an α-release: at this point the intruder is allowed to know that `x in {t1,t2}`, but nothing more. If this transaction is executed repeatedly, then the variables will all be freshly renamed each time, the intruder obtains an α formula like `x1 in {t1,t2},..., xn in {t1,t2}`, saying that there have been `n` transactions performed by some tags `x1,...,xn` and the intruder is not allowed to learn anything more than that they are tags. In particular the intruder is not allowed to know whether or not `x1=x2` holds. This is indeed all that needs to be specified to formulate unlinkability as a goal in (α,β)-privacy.

Line 3 creates a new random number, sends it out on the network (so the intruder can see it), and we send also a special *pseudo-message* `session(x,N)`, i. e., a message that only exists in our model: it formalizes the session state of the tag `x` were `session` is a private function that the intruder cannot apply. Line 4 represents the answer that the server sends: a symmetric encryption (`scrypt`) with key `sk(x)` and content `N`. The **`nil`** command finishes the transaction.

The next step is that the tag receives the messages from the reader, tries to decrypt it, check that it contains the number `N` challenged to the server, and sends an error message otherwise:

```
Transaction Response:
receive Session.
try X = sfst(Session) in
try N = ssnd(Session) in
receive M.
try NN = dscrypt(sk(X),M)
in State := noncestate[N].
   if N=NN and State = fresh
   then noncestate[N]:=spent.
     send ok. nil
   else send nonceErr  . nil # nonce check failed
catch   send formatErr . nil # decryption failed
catch   nil
```

```
14  catch  nil
```

In Line 2, the tag actually tries to retrieve its session state (that we had sent as a pseudo-message in the other transaction). This is supposed to be of the form `session(X,N)` consisting of the tag name and challenge of that session. To check and extract this information, we have two private functions (i. e., functions not accessible to the intruder) called `sfst` and `ssnd` with the property that they the return `X` and `N`, respectively if the given message has the form `session(X,N)`, and give an error otherwise. The **`try`-`in`-`catch`** construct accordingly continues with the **`in`** part if there is no error, and to the **`catch`** part otherwise; here that would be the last two lines where the transaction does nothing.

In Lines 5 and 6, the tag receives the actual message that (supposedly) the card reader has sent and tries to decrypt it with its key `sk(X)` (where `X` is the value just retrieved from its session state). The function `dscrypt` is public, i. e., also the intruder can apply it with known keys, but the intruder in this example does not know any `sk(X)`. (One could model that the intruder has its own passport with tag `tI` and knows the key `sk(tI)`.) Again the decryption function either returns the content if this is a symmetric encryption with the given key, or an error otherwise. In the error case (Line 12) the tag responds with a `formatErr` code.

The next step is to compare the received nonce `NN` with the actual nonce `N` from the session state. Here we have however a slight challenge in modeling: since the session state is modeled here as a pseudo-message that was sent in the first transaction and received back in the second, an intruder can replay an old session state that was actually already processed by the tag, and we cannot prevent that in our model. However, (α, β)-privacy transactions have a notion of long-term state that we can use. Here we use an (infinite) array of memory cells `noncestate[.]` that is initialized with `fresh`. In Line 7 we retrieve the State of the nonce `N` in question, check that the State is still `fresh` in Line 8, and then set it to `spent` in Line 9, effectively blocking a nonce from being used twice. Finally the tag responds with an `ok` message or a `nonceErr`.

The Attack A sequence of three transactions is sufficient to get to a violation of the (α, β)-privacy. We start with two Challenge transactions giving `alpha = x1,x2 in {t1,t2}` and respective messages observed by the intruder:

```
l1 -> session(x1,N1)          l3 -> session(x2,N2)
l2 -> scrypt(sk(x1),N1)       l4 -> scrypt(sk(x2),N2)
```

Next, we execute the Response transaction where the intruder chooses for `Session` the message `l1` and for `M` the message `l4`. Now there are two possibilities for what can happen: either `x1=x2`, then the decryption works (Line 6 of the Response transaction), but the nonce check fails (Line 8), or `x1/=x2` and then already the decryption fails. The error message by the tag is `nonceErr` in the first case, and `formatErr` in the second case, so the intruder now knows whether or not `x1=x2`. Since this does not follow from α, the observations β of the intruder violate (α, β)-privacy, and we can find this attack with the noname tool.

This attack was first reported in [Ar10] and it is interesting that the French implementation of the protocol was vulnerable to this attack, while the British implementation was not. The ICAO 9303 standard [IC] requires the tag to send error messages when receiving an ill-formed or incorrect message from the reader, however this standard does not prescribe what the error message should be. In the French implementation, two different error messages were used (represented here with `nonceErr` and `formatErr`), while the British implementation uses the same error message in both cases, and in fact, doing so we can verify the specification with the noname tool (for up to four transactions).

Another Problem [Fi19] actually pointed out that the protocol has another problem that is here (and in several other models) buried by the fact that the first exchange between tag and reader, namely the nonce `N` from the tag and the response `scrypt(sk(x),N)` from the reader is merged into one transaction. This does not allow for a possible confusion that can arise when multiple tags in parallel are shown to the same or different readers.

We thus split the Challenge transaction into two transactions (see `bac-parallel.nn`):

```
Transaction InitSession:
* x in {t1,t2}.
new N.
send session(x,N).send N.nil
Transaction Challenge:
receive Session.
try X=sfst(Session) in
  receive N.
  send scrypt(sk(X),N).nil
catch nil
```

Here the `InitSession` is the part of the tag creating a new nonce and session state, and this is where the only α-release occurs, thus one may not learn more than that `x` is a tag.

The `Challenge` transaction now receives the pseudo-message `Session`, which simply models that the reader receives the (claimed) identity `X` of the card and can compute the key `sk(X)`. Note that we would be "cheating" if the server also received the nonce `N` from the session state, because that is actually transmitted over a public channel that the intruder can access and manipulate. This `Challenge` transaction now allows for the confusion that the reader gets the claimed identity and shared key from one passport, and the nonce from another.

Now suppose the following transactions: two tags `x1` and `x2` (possibly the same) perform the transaction `InitSession` and the intruder sends the session message from `x1` and the nonce `N2` from `x2` to the server in the `Challenge` transaction, who thus produces `scrypt(sk(x1),N2)`.

The intruder feeds this message to the second tag, i. e., executing the `Response` transaction with the session state of `x2`. This will give the `ok` message if and only if `x1=x2`.

This attack can be found by the noname tool, however due to complexity, we introduced a "guardrail", guiding the tool to first execute two `InitSession`, followed by a `Challenge` and a `Response`. With this guidance we prune other interleavings from the search tree, and it is then small enough to find the attack in a reasonable amount of time. We also verified with the tool under this guardrail that the attack does not exist when encrypting the responses from the tag.

We see here a clear advantage of (α, β)-privacy: the attack description in [Fi19] requires one page of explanation and an understanding of their particular notion of bi-similarity between processes. It may be impossible to make that intuitive to non-mathematical readers because it refers to a world in which only one tag exists, so that the above strategy of the intruder cannot lead to an error. In contrast our specification of the privacy goal is rather simple (the intruder may not find out more about the involved tags other than that they are tags) and also the violation is: confusing the steps of two parallel sessions leads to an observable error message unless the two sessions are with the same tag.

Finally, one may wonder if this is really an issue, because RFID tags do not actually really perform multiple sessions at the same time. If every tag works strictly sequentially, i. e., a new session can only be started once the current session is finished or timed out, then the attack is also prevented. However, this opens another can of worms: since the intruder also knows that tags cannot participate in two sessions at the same time, the successful completion of two interleaved sessions means that distinct tags are involved. The encryption of all response messages also prevents this attack, see `bac-sequential.nn`.

3 Private Authentication

We model the private authentication protocol by Abadi and Fournet, AF for short [AF04]. The protocol contains two roles, initiator and responder. The initiator sends an encrypted message containing a nonce to the responder, who either replies with a nonce for authentication or with a decoy message. The purpose of the decoy is to hide the fact that the responder does not want to talk to the initiator, or that the incoming message does not have the right format. The intruder should not be able to tell the difference between a decoy message and a properly encrypted reply. AF is parameterized over a specification of which agent is willing to talk which other agents. We first look at simple variant AF0 where everybody is willing to talk to everybody.

3.1 AF0: Initial attempt

This first version is found in `af0-initial.nn`. We consider three agents `a,b,i` in this specification where `a` and `b` are honest, and `i` is the intruder, all of which can play as participants here. Each participant `x` has a public key `pk(x)` and the corresponding private key `inv(pk(x))`. The intruder knows all public keys (because `pk` is a public function, and agent names are public constants) and their own private key `inv(pk(i))`.

The first transaction describes how an honest agent `xA` initiates communication with a (possibly dishonest) agent `xB` (the case of a dishonest `xA` is already covered by the intruder model):

```
1  Transaction Initiator:
2  * xA in {a,b}.
3  * xB in {a,b,i}.
4  if xB=i then
5    new NA,R. send crypt(pk(xB),pair(xA,NA),R).
6    * xA=gamma(xA) and xB=gamma(xB). nil
7  else
8    new NA,R. send crypt(pk(xB),pair(xA,NA),R).
9    * xB in {a,b}. nil
```

Like in the previous case study, Lines 2 and 3 specify the non-deterministic choices of agent names from the respective domains, and thus specify that the intruder so far is only allowed to know that `xA` and `xB` are chosen from these domains. The initiator sends an encrypted message to the recipient, containing a pair of their name and a fresh nonce `NA`. (`R` is also a nonce for randomized encryption.) If the recipient is dishonest, then the intruder will learn the values of `xA` and `xB` from this message (knowing the private key to decrypt it). Thus we get in this case a violation of (α, β)-privacy unless we release this information, which we do in Line 6. Here the formula `x=gamma(x)` means that we release the true value `gamma(x)` of `x`: `gamma(x)` will be replaced by the true value of `x` when reaching this point. Also in the case that the recipient is honest, the intruder can learn something from the fact that they cannot decrypt the message: that `xB` is honest, which we release in Line 9. Releasing means that we *allow* this information to be known by the intruder, so that it does not count as an attack if the intruder finds this out. If we had forgotten any of these releases, the noname tool would have notified us with an attack. Note that, except for the α release, the **`then`** and **`else`** branches are identical: they reflect the steps that `xA` indeed performs, while the **`if`**-**`then`**-**`else`** and α-releases are only for specifying the privacy goal.

The response is now described from the perspective of an honest `xB`:

```
1  Transaction Responder:
2  * xB in {a,b}.
3  receive M.
4  try DEC = dcrypt(inv(pk(xB)),M) in
5    try A = proj1(DEC) in
6      if A=i then
7        new NB,R. send crypt(pk(A),NB,R).
8        * xB=gamma(xB). nil
```

```
 9       else
10         new NB,R. send crypt(pk(A),NB,R). nil
11     catch new NB. send NB. nil
12   catch new NB. send NB. nil
```

It may be surprising that `xB` is here also non-deterministically chosen. The example protocol actually leaves the communication model abstract and just models that a message may arrive at *any* participant, since this may be caused by the intruder. `xB` receives a message `M` that they try to decrypt with their private key. The operator `dcrypt` satisfies the equation `dcrypt(inv(K),crypt(K,M,R))=M`. If the decryption succeeds, the result `DEC` must be a pair of a sender name `A` and a nonce. To obtain `A`, the responder uses `proj1` which satisfies the equation `proj1(pair(X,Y))=X`. If this succeeds, `xB` sends an answer encrypted for `A` containing a fresh nonce `NB` and randomization `R` (Lines 7 and 10). As before, we must take into account what the intruder can learn if `A=i`: since then the answer is decipherable for them, they learn the name of `xB` (in case they did not know already). If anything goes wrong (if decryption fails or the content is not a pair) then the recipient sends a decoy message, a random nonce `NB` that is not distinguishable from an encrypted message that the intruder does not have the decryption key for.

The Attack The noname tool reports an attack on this specification. In this attack, only the `Responder` transaction was executed where the intruder provided as input for message `M` the following message: `crypt(pk(a),pair(i,R48),R38)` where `R48` and `R38` are recipe variables that stand for arbitrary messages. The intruder has thus sent a message to recipient `a` under their true name `i`. We have thus two cases. First, `xB=a` and thus the message is correctly decrypted by `xB` and the intruder receives as an answer `crypt(pk(i),NB,R)` for some fresh values `NB` and `R`. Second, `xB/=a` and the decryption fails and `xB` sends a decoy message `NB`. This is of course observable for the intruder, since they can decrypt in the first case, but not in the second. The concrete state that the noname tool presents is the latter case, and the intruder thus learns `xB/=a` without that being released.

This illustrates how we may forget something that the intruder might find out and we may then decide that this is completely benign: the intruder here acts under their real name and just finds out that `xB` who did answer the message was not the intended recipient. Without a basic change of communication model, this information release is unavoidable and the solution is thus to release just this information in this case.

3.2 AF0: Corrected release

We update the responder transaction to add the information being released in the case that the decryption fails, i. e., the **catch** branch following Line 12, see `af0-corrected.nn`:

```
1 Transaction Responder:
 2  * xB in {a,b}.
 3  receive M.
 4  try DEC = dcrypt(inv(pk(xB)),M) in
 5    try A = proj1(DEC) in
 6      if A=i then
 7        new NB,R. send crypt(pk(A),NB,R).
 8        * xB=gamma(xB). nil
 9      else
10        new NB,R. send crypt(pk(A),NB,R). nil
11    catch new NB. send NB. nil
12  catch
13    try C = recipient(M) in
14      try DEC = dcrypt(inv(pk(C)),M) in
15        try A = proj1(DEC) in
16          if A in {a,b,i} and C in {a,b} then
17            new NB. send NB.
18            * not (C=xB and A=i). nil
19          else new NB. send NB. nil
20        catch new NB. send NB. nil
21      catch new NB. send NB. nil
22    catch new NB. send NB. nil
```

The case where the actual recipient `xB` could not decrypt the given message is complicated. To even talk about who is the intended recipient `C` (if the message is even an encryption) we need to model a function that agents in reality cannot perform, namely extracting the name of the recipient from the message, if it exists (Line 13). That is the purpose of the private function `recipient` which satisfies the equation `recipient(crypt(pk(B),M,R))=B`. These steps do not correspond to actions an agent would do and which we only need in order to determine whether the intruder is allowed to learn something. This is the case if the message is indeed encrypted for some agent `C` and contains a pair of an agent name `A` (the claimed sender) and a nonce. We can even restrict this to an honest `C`, as the intruder can otherwise already decrypt the given message and learn `A` and `C`. If `A` is an agent and `C` is honest, then the intruder learns that at least one of two things must be true: `C` is not the actual recipient `xB` or `A` is honest, for if `C=xB` and `A` dishonest, then the intruder could decipher the answer.

In the case where the first specification had the attack, the intruder knew that `A=i` and `C=a` by construction. The released formula α in the updated specification is thus `not(a=xB and i=i)` or simply `a/=xB`. We can indeed verify with the noname tool that there are no more violations of (α, β)-privacy under a bound of three transactions.

3.3 AF

We now lift the simplification of AF0 that everybody is willing to talk to everybody, see af.nn. We define a binary relation talk, where talk(x,y) means that x is willing to talk to y. The noname tool requires to give a fixed interpretation of such a relation. We choose for our experiments the interpretation: talk: (a,b),(a,i),(b,a) which specifies that talk is true for the listed tuples and false otherwise. The aim of the protocol is that privacy holds for every interpretation of talk, but we cannot encode this in the noname tool and in fact the definition of (α,β)-privacy requires a fixed interpretation of all relation symbols [GMV22].

The initiator now checks that the given xA really wants to talk to the given xB in Line 4:

```
 1  Transaction Initiator:
 2  * xA in {a,b}.
 3  * xB in {a,b,i}.
 4  if talk(xA,xB) then
 5    if xB=i then
 6      new NA,R. send crypt(pk(xB),pair(xA,NA),R).
 7      * talk(xA,xB) and xA=gamma(xA) and xB=gamma(xB). nil
 8    else
 9      new NA,R. send crypt(pk(xB),pair(xA,NA),R).
10      * talk(xA,xB) and xB in {a,b}. nil
11  else * not talk(xA,xB). nil
```

In the positive cases the intruder learns that talk(xA,xB) (in case xB=i the intruder learns also the concrete agent names, in case xB/=i the intruder learns that xB is honest), in the negative case, the intruder learns not talk(xA,xB). This case is a bit artificial, because if xA does not want to talk to xB, they would not even start this transaction in reality, but here we need to non-deterministically choose the agent names first and then abort if not talk(xA,xB), and then the intruder learns that because no message goes out.

This has an interesting consequence: suppose we are in a state where the intruder, as part of α, has learned that agent a talks to every other agent, and now observes not talk(xA,xB). From xA in {a,b} follows that xA=b. This is not a violation of (α,β)-privacy, since this xA=b holds in every model of α. In general, it is completely acceptable that the intruder learns the value of privacy variables like xA, as long as this is a consequence of α.

```
Transaction Responder:
* xB in {a,b}.
receive M.
try DEC = dcrypt(inv(pk(xB)),M) in
  try A = proj1(DEC) in
    if A=i then
      if talk(xB,A) then
```

```
 8          new NB,R. send crypt(pk(A),NB,R).
 9          * xB=gamma(xB) and talk(xB,A). nil
10        else
11          new NB. send NB.
12          * not talk(xB,A). nil
13      else if A in {a,b} then
14        if talk(xB,A) then
15          new NB,R. send crypt(pk(A),NB,R). nil
16        else new NB. send NB. nil
17      else new NB. send NB. nil
18    catch new NB. send NB. nil
19  catch
20    try C = recipient(M) in
21      try DEC = dcrypt(inv(pk(C)),M) in
22        try A = proj1(DEC) in
23          if A in {a,b,i} and C in {a,b} then
24            new NB. send NB.
25            * not (C=xB and A=i and talk(xB,A)). nil
26          else new NB. send NB. nil
27        catch new NB. send NB. nil
28      catch new NB. send NB. nil
29    catch new NB. send NB. nil
```

New is here the case split on whether `talk(xB,A)` in Lines 7 and 14: if not, we get into the decoy cases. Observe that only in the dishonest cases the intruder learns whether `talk(xB,A)` or not in Lines 9 and 12. In case the message is a valid message to some different agent `C`, the intruder learns a bit less as compared to the AF0 version in Line 25: it basically says that now the reason for not being able to decipher the reply could be that `not talk(xB,A)`.

We have verified privacy with the noname tool up to four transactions, both for the above interpretation of talk, and when everybody talks to everybody.

4 Conclusion

Using (α, β)-privacy, we can write specifications of protocols and express privacy properties in a more intuitive way by writing explicitly what information the intruder is allowed to learn. This declarative way of specifying privacy goals allows for a better understanding of what information a protocol is revealing. For instance, in our study of the Abadi-Fournet protocol, we adapted the information released until there were no more violations. This gives a novel characterization of the privacy guarantees provided by the protocol, which is not based on formulating indistinguishabilities between different scenarios but rather on what deductions the intruder can make from observing a concrete execution.

Similarly, our model of the BAC protocol not only demonstrates how the unlinkability analysis with noname is declarative and simple, but also allows for an easier understanding of the known vulnerabilities: while [Fi19] needs to refer to a non-trivial notion of bisimilarity to demonstrate that BAC violates their model of unlinkability, we can simply give an attack trace. It is also noteworthy that an early analysis of BAC [Ar10] failed to notice the problem that [Fi19] pointed out: this may be due to the fact that the intricacy of specifying unlinkability with indistinguishability notions forced the authors to make simplifications that simply bury the attack. Thus more declarative models can be very helpful for avoiding false negatives.

One limitation of the *noname* tool is the state space explosion, since all traces with different sequences of atomic transactions are explored (up to a fixed number of transactions). This can be partially addressed by "guiding" the tool to consider relevant traces, where some transactions need to be executed before others. Indeed this study could thus also provide ideas for the future development of the tool, such as search heuristics based on partial-order reductions (exclusion of redundant interleavings) to quickly find attacks.

Bibliography

[AF04] Abadi, M.; Fournet, C.: Private Authentication. Theor. Comput. Sci., 322(3):427–476, 2004.

[Ar10] Arapinis, M.; Chothia, T.; Ritter, E.; Ryan, M.: Analysing Unlinkability and Anonymity Using the Applied Pi Calculus. In: CSF 2010. IEEE, pp. 107–121, 2010.

[CS10] Chothia, T.; Smirnov, V.: A Traceability Attack against e-Passports. In: FC 2010. volume 6052 of LNCS. Springer, pp. 20–34, 2010.

[DH17] Delaune, Stéphanie; Hirschi, Lucca: A Survey of Symbolic Methods for Establishing Equivalence-based Properties in Cryptographic Protocols. J. Log. Algebraic Methods Program., 87:127–144, 2017.

[Fi19] Filimonov, I.; Horne, R.; Mauw, S.; Smith, Z.: Breaking Unlinkability of the ICAO 9303 Standard for e-Passports Using Bisimilarity. In: ESORICS 2019. volume 11735 of LNCS. Springer, pp. 577–594, 2019.

[FMV23] Fernet, L.; Mödersheim, S.; Viganò, L.: A Decision Procedure for Alpha-Beta-Privacy for a Bounded Number of Transitions. Technical report, DTU Compute, 2023. Together with noname tool available at `https://www2.compute.dtu.dk/~samo`.

[GMV22] Gondron, S.; Mödersheim, S.; Viganò, L.: Privacy as Reachability. In: CSF 2022. IEEE, 2022.

[IC] ICAO: , Machine Readable Travel Documents. Doc Series, Doc 9303. `https://www.icao.int/publications/pages/publication.aspx?docnum=9303`.

[MV19] Mödersheim, S.; Viganò, L.: Alpha-Beta Privacy. ACM Trans. Priv. Secur., 22(1):1–35, 2019.