Automatic Generation of Security Protocols Attacks Specifications and Implementations

Rémi Garcia and Paolo Modesti

Department of Computing and Games, Teesside University, United Kingdom
{r.garcia, p.modesti}@tees.ac.uk

Abstract. Confidence in a system's security is a key requirement for its deployment and long-term maintenance. Checking if a vulnerability exists and is exploitable requires extensive expertise. The research community has advocated for a systematic approach with formal methods to model and automatically test a system against a set of desired security properties. As verification tools reach conclusions, their applicability is limited unless expertly analysed. We propose a code generation approach to automatically build both a specification and an implementation of a Dolev-Yao intruder from an abstract attack trace, bridging the gap between theoretical attacks discovered by formal means and practical ones. Through our case studies, we focus on attack traces from the OFMC model checker, *Alice&Bob* specifications and Java implementations. We introduce a proof-of-concept workflow for concrete attack validation that allows to conveniently integrate user-friendly feedback from formal methods results into a Model-Driven Development process.

Keywords: Attack analysis · Code generation · Model-Driven Development · Formal methods for security [1]

1 Introduction

With the rise of large-scale automated vulnerability exploitation, security experts advocate for enhancing existing design practices [24,12], to avoid discovering and fixing security flaws in protocols embedded into deployed systems. To this end, Model-Driven Development and formal methods allow for a systematic and reliable vulnerability discovery process, while working on a simplified high-level model of a protocol. Once model checking is completed, an implementation phase translates the model into a concrete system. However, the implementation process can be particularly costly and error-prone, which is why an automatic code generation approach can be beneficial for quick prototyping.

In this paper, we accommodate the designer's vision, as the interpretation of an attack trace is particularly subtle, often requiring expertise in the attack trace format and verification tool's inner mechanisms. Our proof-of-concept reconstructs an attack trace into a protocol in the same format as the original

[1] Software available at `https://paolo.science/anbx/download/plas2022.zip`

specification. Next, we automatically generate an executable implementation of the exploit.

This can also be pedagogically beneficial to introduce learners to formal modelling and verification [20] of security protocols. They can then transfer the security notions of the abstract model to the concrete implementation in a standard programming language, like Java in this paper. In this context, students can work with known vulnerable protocols, understand the nature of the attacks, both in the abstract model and concrete system, test and fix them.

Moreover, we investigate the possible discrepancies between the concrete system's security and what can be modelled and verified with a high-level specification language. In reality, an attack trace on the model might exploit oversimplified protocol features and constitute a false positive. It should be noted that the investigation of false negatives is out of the scope of this work, as attacks that occurs only in the implementation but not in the abstract model are not something that the formal methods considered here could capture.

The proposed approach can also be of interest for the verification of protocols that have not been formally tested, giving a fast and intuitive prototyping and verification toolchain to developers with limited knowledge or resources. In this case, the main initial effort would be to write a model of the protocol in a simple and intuitive language for protocol narration. Our automated workflow can be used from there.

However, even security researchers who use formal methods as one of their investigation tools to model existing systems can benefit from the automated generation of proof-of-concept attacks in a concrete target language. In fact, end users will be more convinced that their systems are at risk if an attack can be demonstrated on a real (or realistic) system.

In essence, our approach injects an explicit Dolev-Yao intruder [13] into an *Alice&Bob* (*AnB* [17]) protocol narration. For the purpose of this paper, we use the OFMC model checker [5] (a component of the AVISPA toolkit), which performs both protocol falsification and bounded session verification. It leverages the lazy symbolic intruder representation and constraint differentiation as a general search-reduction technique. We choose OFMC because its verification is sound and complete, allows to specify the number of concurrent sessions, and accepts *AnB* as an input language. It also produces attack traces in a format that shares similarities with *AnB*, making its traces convenient to explain. The *AnBx* compiler [19] is used as a code generator to implement attacks in Java [8,18] and we extend its capabilities with this contribution.

As in this research we aim at providing a proof-of concept of the applicability of the proposed approach, our experiments target specific tools and languages that allow to clearly demonstrate the workflow in operation. However, the methodology is general and could be applied to other targets of interest for potential end users.

Outline of the paper Section 2 introduces previous works on the interpretation of verification results, the essentials of the *AnB* notation for protocol specification, and the formalism used to represent attack traces. Section 3 states our

methodology, presenting the main concepts of our approach for concrete attack prototyping and the workflow of the proposed solution. In Sections 4 and 5, we detail how the intruder's knowledge is modelled and the attack trace processed. The experimental results and correctness aspects are discussed in Section 6. Finally, we present our conclusions in Section 7.

2 Background

In this section, we introduce the main notions and formalisms used in the present paper and applied to the proposed approach. Moreover, we discuss the work undertaken by other researchers in this area.

2.1 Related Work

Several previous works investigated the trustworthiness of a formal attack trace. For instance, over-approximation may lead ProVerif [6] to report false attacks. Other modelling formalisms might introduce various kinds of false positives, depending on their internal assumptions. Despite being sound and complete, even OFMC can find attacks which are not concretely applicable in certain implementation contexts. For example, it does not distinguish ciphertexts from random numbers and allows them to be substituted by an attacker.

One major validation technique uses the *De Bruijn criterion* [4], where proof objects are independently checked by several tools, giving more certainty on a conclusion for the abstract model, but without application on a concrete system.

Armando *et al.* [1] have considered the applicability of abstract attack traces in protocols specified at the HTTP level. They bind a model to its concrete implementation and generate executable program fragments if an attack is found. Their prototype is tested, among others, against the Google Single Sign-On service.

A recent application to air traffic control is devised in [16], focusing on a much more concrete system description than what *AnB* expresses. It also translates security counterexamples to test cases, modelling aircraft landing behaviour with the *communicating sequential processes (CSP)* language.

Verification and test-case generation has been achieved from high-level HLPSL [15,11] or ASLan++ [7] models, as well as annotated sequence diagrams [9], but requiring pre-existing implementations to attack, while we can also generate the concrete system from the model. These approaches introduce faults in models through mutations, run model checking to obtain attack traces, and then simulate them on real communication channels.

Contributions on web applications also introduced similar concepts. With the AUTHSCAN framework [2], web authentication protocols specifications are automatically recovered from existing implementations and then verified. Similarly, Bansal *et al.* translate JavaScript and PHP to applied pi-calculus with semi-automatic ProVerif trace reconstruction [3]. The MobSTer framework [21]

introduces a user interface for the modelling and verification of web application, and attacks are automatically translated to HTTP requests. The SPaCIoS project [23] focuses on the Internet of Services, also automatically generating test cases for the system under validation, based on its formal description.

We start from a high-level *AnB* description of the system and use model checking for test case generation. We concretely validate attacks, while also enabling the user to see them described in *AnB*. The studied system's implementation is entirely generated, so we do not require a manual effort of implementing a system and tying it to a model.

2.2 Security properties and types of attacks

When looking at a vulnerability, it is important to precisely define its nature in order to assess the possible consequences. We consider two main classes of security violations: *authentication* and *secrecy*.

On authentication, we refer to Lowe's injective and non-injective agreements [14]. For a non-injective agreement to succeed between two agents A and B on some term M, A must complete a protocol run believing to have been communicating with B. They agree that M was exchanged, with B believing to have been talking to A. The injective agreement adds a freshness requirement, where every run of A corresponds to a unique run of B. With an authentication violation, the intruder would deceive an agent into believing that it talks to someone honest.

Secrecy for a term M is usually specified with respect to a set of agents meant to know the secret, of which the intruder is absent. In terms of verification technique, secrecy is expressed as a reachability problem, where a reachable state of the system in which the intruder knows M represents an attack. It is worth noting that the secrecy of M between two agents A and B stipulates that A and B must share the same value M.

2.3 Protocol specification notation

The *AnB* notation is an abstract, intuitive, and compact specification language in which a designer can declare communicating agents, functions, and literals. It defines a sequence of actions performed by each honest participant in the protocol. A protocol step is denoted as a channel exchange, like $A \to B : Msg$, specifying that an agent playing the role A sends a message Msg to another agent playing the role B. For the sake of brevity, we refer to the agents directly by their roles in the rest of the paper.

In these actions, an agent can send messages composed of built-in and custom functions applications, or fresh or constant values. This knowledge is formed by what the agent has sent or received so far, as well as its initial knowledge, the set of terms known before performing any action.

We use the *AnBx* language [8], an extension of *AnB* that notably provides type signatures and forwarding channels, enforcing security guarantees from the message originator to the final recipient along a chain of intermediate agents.

In AnB, desired security properties are specified as predicates in the *Goals* section. A goal is violated if there exists a corresponding reachable attack state for a Dolev-Yao intruder. While in our experiments we also considered large protocols, for the sake of clarity we use Woo-Lam as a running example, as it is simple and highlights interesting behaviours.

Using $AnBx$, we can model the Woo-Lam protocol, which is a one-way authentication of the initiator A to a responder B. It involves a trusted third-party server s with whom A and B share long-term symmetric keys $shk(A, s)$ and $shk(B, s)$. The server extracts a random number NB (generated by B) from the ciphertext $\{|A, B, NB|\}shk(A, s)$ and sends it back to B. B compares NB from s with the one it sent to A. If they match, then B knows that the initiator of the protocol is indeed A since it knows that A and s communicate securely. In essence, the goal of this protocol is that B *authenticates* A *on* NB.

```
Protocol: Woo-Lam
Types:
    Agent A,B,s;
    Number NB;
    Function [Agent,Agent ->* SymmetricKey] shk
Knowledge:
        A: A,B,s,shk(A,s);
        B: A,B,s,shk(B,s);
        s: A,B,s,shk(A,s),shk(B,s)
Actions:
        A -> B: A
        B -> A: NB
        A -> B: {|A,B,NB|}shk(A,s)
        B -> s: {|A,B,{|A,B,NB|}shk(A,s)|}shk(B,s)
        s -> B: {|A,B,NB|}shk(B,s)
Goals:
        B authenticates A on NB
```

2.4 Attack trace and reached state information

The verification tool OFMC reasons with states that represent what every agent knows at each step of a protocol execution. The final knowledge each honest agent possesses is summarised in the *Reached State* section of an OFMC attack trace. Let A be an honest agent's role in the trace. Its reached state maps roles to subjective concrete identities of the form xY, $Y \in \mathbb{N}$. After executing n steps of the trace related to the agent, its local knowledge is listed. This final knowledge K_n is a set of terms composed of what the agent initially knows, and what it received or freshly generated during its protocol run. This happens in a numbered session σ. With a $[A \mapsto x23]$ mapping, $x23$ being A's identity from its own point of view, we obtain:

$$\text{state\_r}A(x23, n, K_n, \sigma)$$

The above Woo-Lam specification is vulnerable to an attack that OFMC outputs as follows, where i is the intruder:

```
ATTACK TRACE:
i -> (x501,1): x502
(x501,1) -> i: NB(1)
i -> (x501,1): NB(1)
(x501,1) -> i: {|x502,x501,NB(1)|}_(shk(x501,s))
i -> (x501,1): {|x502,x501,NB(1)|}_(shk(x501,s))
% Reached State:
% request(x501,x502,pBANB,NB(1),1)
% state_rB(x501,3,shk(x501,s),s,x502,NB(1),NB(1),{|x502,x501,
    NB(1)|}_(shk(x501,s)),{|x502,x501,NB(1)|}_(shk(x501,s))
    ,1)
% state_rA(x20,0,shk(x20,s),s,x31,1)
% state_rs(s,0,shk(x34,s),shk(x35,s),x34,x35,1)
```

In this trace, A knows its own identity $x20$, reached step 0 as not being involved in the attack, knows the shared key with the server $shk(x20, s)$, followed by s and $x31$, then believed to be B. Finally, this knowledge is constituted in session number 1, the same session as the other agents.

To break authentication, the intruder gets around encryption by deceiving B with a potentially fraudulent A identity and some message formatting tricks:

1. To start the attack, the intruder sends an identity $x502$ to B, which is not necessarily A's true identity, as A defines itself as $x20$.
2. B creates and sends NB, as specified originally.
3. The intruder sends back NB, when B expects $\{|A, B, NB|\}shk(A, s)$.
4. B creates a ciphertext with the key $shk(B, s)$, in which the last received message NB is the third variable.
5. The intruder then obtains $\{|A, B, NB|\}shk(B, s)$, and sends it back to B.

The intruder does not need to know what is encrypted, only to forward it. B expects $\{|A, B, NB|\}shk(B, s)$ as the final message, and assumes it is coming from A and decrypted by s, without an actual guarantee.

Formally, events annotations are used to guide the goal verification. The event $request(x501, x502, pBANB, NB(1), 1)$ with $B = x501$, states that B believes to have been talking to A with identity $x502$, and received NB in this session. As per the authentication goal, this should have been accompanied by a corresponding *witness* event where A intends to run the protocol with B and agree on NB. A is never active in the protocol, so no agreement is achieved between A and B.

2.5 Intermediate format for protocol implementation

Before generating concrete code from an *AnBx* specification, we need to consider that this formalism describes the ideal execution of a protocol from the point of view of an external observer. Therefore, for a concrete implementation, we

need to use an intermediate format allowing to explicitly represent actions and consistency checks on reception from the viewpoint of each participating agent.

We use existing functionalities in the *AnBx* compiler to convert the protocol specification into such an intermediate format, called Executable Narration (*ExecNarr*) [19]. This notation can capture a large number of formalisms, as it is used by the *AnBx* compiler to generate code in ProVerif, Java, VDM-SL, etc.

Let us consider a very simple protocol with a single action: $A \rightarrow B : A$. We suppose that B already knows A. It can be compiled to an *ExecNarr* as follows:

```
A: send(B;A)           # Insecure
B: R0 := receive()     # <- A  Insecure
B: eq(A,R0)
```

Here, $R0$ would be what B receives from an insecure channel supposedly from A, but under Dolev-Yao assumptions, B would have no guarantee that variable $R0$ contains the value A. B does an equality check against $R0$, but it checks that it is equal to its subjective expectation of A, based on its initial knowledge. However, as the intruder controls the network, such a check is not sufficient to determine if its vision of A matches who the sender really is.

3 Methodology

In our approach, we provide a viable automated solution to the onerous and error-prone manual process of expressing an attack in a high-level notation or in concrete code. A fast prototyping workflow also allows to catch false positives resulting in a failure of the attack in the concrete generated code.

From a developer's standpoint, one of the main objectives is to be able to conveniently reason on an abstract representation of the protocol without losing control of the concrete implementation. A high-level of abstraction is convenient, but iterative development with immediate feedback involving the concrete code can provide advantages in terms of security testing and low-level insights. With this goal in mind, we propose the following workflow (Fig. 1):

1. Specify the protocol in a high-level specification language, like *AnBx*.
2. Use formal verification tools to exhibit potential vulnerabilities.
3. If one is found, rewrite the attack trace as a protocol in the same format used for the specification of the original protocol.
4. Generate the attack implementation in a concrete programming language (e.g. Java) and run it.

More in detail, we consider a security protocol P specified in a formal language $\mathcal{L}$, vulnerable to an attack described by an attack trace *Atk*. *Atk* specifies the state reached by the system with a violated goal, including the agents' knowledge and a list of actions needed to achieve it. In every action, the intruder acts either as the sender or recipient of a message. Such a list describes the steps that the intruder has to perform in order to successfully complete the attack.

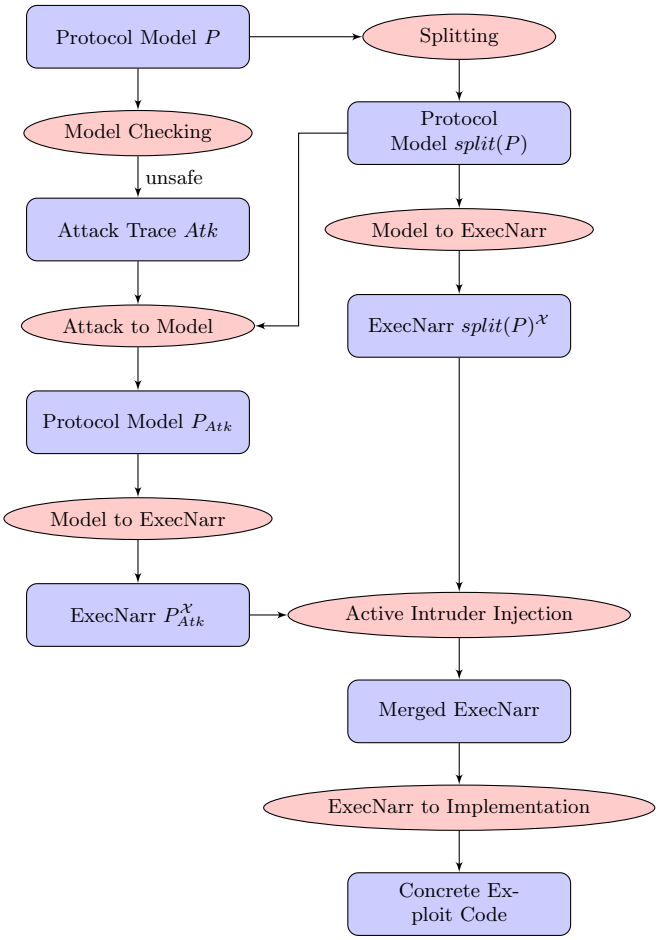

Fig. 1: Workflow of attack reconstruction and implementation

We first need to apply a transformation on P, called *split*, to explicitly add the intruder in a Dolev-Yao style. Therefore, we split every action of P in two, denoting the presence of a man-in-the-middle agent called $Intr$. In *AnBx*, this consists, for example, in transforming the action $A \to B : Msg$ into two consecutive actions $A \to Intr : Msg$ and $Intr \to B : Msg$. $Intr$ here has its own knowledge according to Dolev-Yao rules as discussed in Section 4, and corresponds to a passive intruder. This new protocol is called $split(P)$.

Then, we reconstruct the attack trace Atk into a protocol P_{Atk} encoding Atk in the same language $\mathcal{L}$ as P (procedure *Attack to Model* in Fig. 1). This enables the user to observe the attack in a side-by-side comparison with P. This particular reconstruction requires two inputs (the protocol $split(P)$ and the attack trace Atk) and is described in Section 5.

As discussed in the previous section, the construction of an implementation requires to compute the $ExecNarr$ of a protocol. We denote with $P^{\mathcal{X}}$ the $ExecNarr$ of the protocol P.

However, we cannot directly generate an attack implementation from P_{Atk}, as $P^{\mathcal{X}}_{Atk}$ does not include the information about the checks on reception performed by the honest agents in the original protocol P. In fact, in the attack implementation, honest agents will run their standard code, while the intruder will follow the actions specified in P_{Atk}. Therefore, we need to merge $P^{\mathcal{X}}_{Atk}$ with $P^{\mathcal{X}}$ (*Active intruder injection* procedure in Fig. 1).

In Atk, the intruder can be active. In $split(P)^{\mathcal{X}}$, $Intr$'s actions would then be replaced by their counterparts in $P^{\mathcal{X}}_{Atk}$. If there are inactive agents in P_{Atk}, their actions can be omitted in the merged $ExecNarr$, for the final code generation.

A simple example of the merging procedure *Active intruder injection* is given in Fig. 2: by explicitly adding the man-in-the-middle for the protocol P with one action $A \to B : A$ discussed in Section 2.5, we obtain case (a).

```
A: send(Intr;A)
Intr: R0 := receive()
Intr: send(B;R0)
B: R1 := receive()
B: eq(A,R1)
```

(a) $split(P)$ $ExecNarr$

```
Intr : send(B; Intr)
B : R1 := receive()
B : eq(Intr, R1)
```

(b) P_{Atk} $ExecNarr$

```
Intr : send(B; Intr)
B: R1 := receive()
B: eq(A,R1)
```

(c) merged $ExecNarr$

Fig. 2: *Active intruder injection*: merging process of Executable Narrations for a protocol with a (a) passive and (b) active intruder; (c) is the merged $ExecNarr$

The other case (b) is $P^{\mathcal{X}}_{Atk}$, where the intruder sends its own identity to B. This is the explicit impersonation of A.

Injecting $Intr$'s action from $P^{\mathcal{X}}_{Atk}$ into $split(P)^{\mathcal{X}}$ – case (c), Fig. 2 – displays the same check on reception as in case (a).

Let us consider that, for example, the goal of the protocol is for the recipient to authenticate the message received from the sender. In this case, the goal fails as the intruder can send any value, and decide whether the check performed by B fails or succeeds, preventing B to detect the attack in the latter case. This leaves the entirety of the honest code untouched, following the philosophy that honest agents correctly execute the protocol, but are betrayed by their environment.

With this merged $ExecNarr$, we can generate concrete code to investigate the applicability of Atk in a target environment, bridging the gap between Atk as an abstract model and its concrete execution. We should consider here two scenarios where attacks can be detected or not by honest agents. We assume that an honest agent who detects an anomaly would abort the protocol run or at least give notice of the attack. A designer can look at what part of the code mitigated the attack, identifying how careful an implementation has to be. Some of those insights are given in Section 6.2 and Appendix B.

In this workflow, protocols can be specified, verified and run. The validity of the theoretical attack can be empirically tested and the entire workflow is automated. The developer only needs to adjust the first specification. This approach allows a non-expert to swiftly check the presence of the attack in an automatically generated implementation, and compare two protocol models, both specified in an easy-to-understand notation.

Along with the design and automation of the entire workflow, the specific contribution of this work includes the development of the *Attack to Model* and *Active intruder injection* procedures shown in Fig. 1.

4 Intruder knowledge and subjective identities

A Dolev-Yao intruder can deceive honest agents. As such, the intruder must know what is publicly obtainable by the honest agents, or known by some agent about another one. To mimic an honest behaviour, the intruder knows its own derivations of what honest agents know about themselves. For example, if A knows its public and private key pair $pk(A)$, $inv(pk(A))$, as well as a shared key with s $shk(A, s)$, the intruder knows $pk(Intr)$, $inv(pk(Intr))$ and $shk(Intr, s)$.

Some ground rules exist to limit the intruder's capabilities:

- Constant agents, like trusted servers, cannot be impersonated, and their knowledge cannot be extracted.
- For variable agents, the knowledge they have about themselves can only be known by the intruder if another variable agent knows it as well.

Formally, the initial knowledge of the honest agents specified in the $Knowledge$ section is a mapping of n agent roles to n knowledge sets $[a_1 \mapsto k_1, a_1 \mapsto k_2, \ldots, a_n \mapsto k_n]$ that we denote as $know(Ag)$ given a protocol role Ag. Given the predicate $var(Ag)$, true if Ag is a variable agent, and our declared honest agents Ags, the intruder's knowledge $know(Intr)$ is defined as the union of the knowledge of variable agents, where each agent's own name is replaced by $Intr$:

$$\bigcup_{Ag \in \{a \mid a \in Ags \wedge var(a)\}} know(Ag)[Ag \mapsto Intr]$$

In the Woo-Lam example, the intruder can know B because B is in the initial knowledge of another variable agent A and the same logic applies to A and s. The last pieces of the initial knowledge of A and B are the shared symmetric keys between A, B and s. As A and B can be impersonated, the term $shk(Intr, s)$ is available to $Intr$. This models $Intr$ sharing a key with the server s as any other agent. However, $shk(A, s)$ and $shk(B, s)$ cannot be known by the intruder because A and B only know their own keys, and honest agent s has a protected knowledge. The rationale is that $Intr$ should have the combined capabilities of all variable agents that can be impersonated/compromised by an attacker.

Thus, given the original honest initial knowledge presented in Section 2.3, the intruder's initial knowledge is:

```
Intr: Intr,A,B,s,shk(Intr,s);
```

This initial knowledge is consistent with the *AVISPA Intermediate Format IF* [17] initialisation *iknows* facts. *IF* is internally used by OFMC for the verification process. The *iknows* facts for Woo-Lam include:

```
iknows(i).iknows(shk(i,s)).iknows(s).iknows(B11).iknows(A21).
```

$B11$ is a subjective B identity, so B can be impersonated. The initial knowledge of an agent can be compromised, and every identity known throughout the protocol is purely subjective, except for constant agents. To understand it, consider the *IF* code produced by OFMC from the Woo-Lam *AnB* specification:

```
initial_state init1 :=
state_rA(A11,0,shk(A11,s),B11).
state_rB(B21,0,shk(B21,s),A21).
& A11/=i & B21/=i
```

In this initial knowledge, the agents know themselves, with two additional constraints that enforce that no honest agent is the intruder. $A21$ is B's vision of B while $A11$ is A's vision of itself, but they do not have to be equal. $A21$ could be the intruder, since no identity is confirmed yet at the start of the protocol.

5 Attack trace processing and protocol reconstruction

We detail here how to reconstruct an OFMC attack trace into an *AnBx* protocol. The actions and reached state parts of *Atk* are both analysed, removing session and step numbers, and extra parenthesis and underscores from the trace. This corresponds to the *Attack to Model* procedure in Fig. 1, divided into three steps:

1. Replace the concrete identities with the agents' roles in the trace.
2. Replace impersonated agents' roles in the initial knowledge.
3. Identify forgeries and generate the appropriate declarations and messages.

5.1 Actions coupling and impersonation

Discrepancies between the intended protocol and the attack trace can only be highlighted if we compare the correct corresponding actions. We then need to have a proper coupling between the actions in $split(P)$ and in Atk.

It should be noted that the first and last actions in Atk do not necessarily correspond to the first and last actions of $split(P)$. To locate exactly where the attack happens, we must find an anchor point in the specification. In the OFMC reached state introduced in Section 2.4, each agent reached a certain step number corresponding to the last action it is a sender in or, when the number goes higher, the last it is a receiver in. In this Woo-Lam example, B reached step 3, and we have two actions with B as sender. The last action has B as receiver, so this is the anchor point that we will use.

With numbered actions, we show in Fig. 3 which trace actions correspond to which original actions. Starting from the last action of the last honest agent (here $x501$ with $B = x501$), simple comparisons of senders and receivers allow us

to precisely determine what is the correct coupling for actions between $split(P)$ and Atk. We removed session and step numbers annotations for clarity. We remind that here the ciphertext $\{|A, B, NB|\}shk(A, s)$ is replaced by NB in the attack, as explained in Section 2.4. This is possible as in OFMC a ciphertext and a random number are indistinguishable, unless the receiving agent knows the decryption key.

```
   A->Intr:A
   Intr->B:A
1  B->Intr:NB
   Intr->A:NB
   A->Intr:{|A,B,NB|}shk(A,s)
   Intr->B:{|A,B,NB|}shk(A,s)
2  B->Intr:{|A,B,{|A,B,NB|}
                  shk(A,s)|}
                     shk(B,s)
   Intr->s:{|A,B,{|A,B,NB|}
                  shk(A,s)|}
                     shk(B,s)
   s->Intr:{|A,B,NB|}shk(B,s)
3  Intr->B:{|A,B,NB|}shk(B,s)
```

(a) actions in $split(P)$

```
   i->x501:x502
1  x501->i:NB

   i->x501:NB
2  x501->i:{|x502,x501,NB|}
                  _(shk(x501,s))

3  i->x501:{|x502,x501,NB|}
                  _(shk(x501,s))
```

(b) actions in Atk

Fig. 3: Matching result between actions from $split(P)$ and Atk

The semantic differences that exist between those two versions denote a deviation from the intended specification. Determining what data has changed in Atk is crucial to precisely define how we will model it in AnB.

As we are considering a fully compromised environment, any unconfirmed identity in the reached state such as $x502$ for A in Section 2.4 is considered as the intruder. Thus, P_{Atk}'s actions are specified as follows:

```
Intr -> B: Intr
B -> Intr: NB
Intr -> B: NB
B -> Intr: {|Intr,B,NB|}shk(B,s)
Intr -> B: {|Intr,B,NB|}shk(B,s)
```

5.2 Forgery

When the intruder tampers with the protocol run, it can replace messages with new forged ones. In the OFMC attack trace, forged terms are represented by variables of the same xY format as the agents' identities.

We map those terms with their counterparts in P. Given a non-agent term xY substituting a term of type T in P, we generate a fresh variable XY of type T in P_{Atk}'s $Types$ section. We detail special cases as follows.

Exponentiation forgery

With the exponentiation function exp, the base can be well-known constant. In this case, we consider that the intruder only forges the exponent. One example of this situation is a violation of Diffie-Hellman secrecy in a specification safe for weak authentication only. In the obtained attack trace, the intruder replaces the symmetric key $exp(exp(g, Y), X)$ by $exp(x304, X)$.

Since g is publicly known, the intruder can compose it. Consequently, we can rewrite the expression as $exp(exp(g, X304), X)$ and preserve the original key structure. Using another base would lead to detection, as honest agents use g.

Mimicking encrypted messages

An attack can exploit the fact that an agent uses encrypted data that it is not able to decrypt. An intruder sending random data could deceive such an agent, given that the encryption method provides ciphertext indistinguishability. Noise sending or bit padding cannot be expressed in a classical AnB notation, so we use a clone of the honest message with respect to its format, types and encryption scheme, but with entirely forged contents.

We could have a message $\{|NB, A|\}shk(A, s)$ where NB and A are respectively of type $Number$ and $Agent$. They are encrypted with a symmetric key shared between A and s, so another agent B could not decrypt it. Using arbitrary data, the intruder can replace the message with $\{|Xxx, Intr|\}Key$ where Xxx and Key are freshly generated and respectively of type $Number$ and $SymmetricKey$. Thus, type-checking would not enable B to detect the forgery.

An advantage of this approach is its genericity. Assuming that honest agents can distinguish between a ciphertext and a random number, and even identify what encryption scheme is used, the attack would still go undetected. This reconstruction addresses the worst-case scenario corresponding to a Dolev-Yao intruder having perfect knowledge of the executing protocol behaviour.

6 Evaluation

In our evaluation of the proposed technique, we reconstruct 56 attack traces in total from 27 vulnerable protocols shown in Table 1. The test suite includes complex protocols as the e-commerce iKP and SET payment systems along with smaller components, like Diffie-Hellman, used in larger protocols to perform tasks as key establishment.

We limit the scope of this paper to attacks occurring in the same session, without reconstructing replay attacks that work in two or more parallel sessions. Our protocols are taken from AnB and $AnBx$ case studies. Most are available through the OFMC distribution and tutorials [22], and some like SET and iKP are studied in [8].

6.1 Reconstruction correctness

As we reconstruct an attack trace Atk on a protocol P to an $AnBx$ protocol P_{Atk}, the correctness of the procedure needs to be assessed. For that purpose, we devise a *cross-validation* strategy in which we use OFMC to verify P_{Atk}.

We obtain a second attack trace Atk_2, and if the actions of Atk_2 are a subsequence of those of Atk, bar variables α-renaming, then the sequence of actions in P_{Atk} that lead to a compromised goal in Atk_2 has an isomorphism in Atk. We express such a relationship between Atk and Atk_2 as $Atk_2 = IsoSubSeq(Atk)$.

To attain it often requires to amend violated security goals, to reflect how the verification should target the new network exchanges. The goals are rewritten as follows.

With an authentication or secrecy goal on a term $Payload$, we replace it with a term sent by $Intr$, i.e. what agents in the goal believe to be $Payload$ in Atk. We denote the result as $Payload_{Atk}$. As for the agents in the goal:

- In secrecy goals as "$Payload_{Atk}$ *secret between* $AgentSet$", we remove any agent who does not know $Payload_{Atk}$ at the end of P_{Atk}.
- In authentication goals as "$Agent_1$ *authenticates* $Agent_2$ *on* $Payload_{Atk}$" , if $Agent_2$ is inactive or does not know $Payload_{Atk}$, we apply $[Agent_2 \mapsto Intr]$.

For example, the Woo-Lam protocol exhibits the same attack trace discussed in Section 2.4 with a modified goal "B *authenticates* $Intr$ *on* NB".

```
i -> (x501,1): x502
(x501,1) -> i: NB(1)
i -> (x501,1): NB(1)
(x501,1) -> i: {|x502,x501,NB(1)|}_(shk(x501,s))
i -> (x501,1): {|x502,x501,NB(1)|}_(shk(x501,s))
% Reached State:
% request(x501,x502,pBIntrNB,NB(1),1)
% state_rB(x501,3,shk(x501,s),s,x502,NB(1),{|x502,x501,NB(1)
    |}_(shk(x501,s)),1)
% state_rA(x33,0,shk(x33,s),s,x34,1)
% state_rIntr(x20,0,shk(x20,s),s,x31,x32,1)
% state_rs(s,0,shk(x37,s),x37,1)
```

The reached state includes the new agent $Intr$ in the *request* part, reflecting the new goal. As expected, our new intruder replaces $Intr$ in the trace.

As it can be observed in the experimental results summarised in Table 1, there are cases where the second trace is semantically different from the first one, or even shows no attack at all. In our case studies, these differences are justified by the impossibility of having the same attack behaviour, or the existence of multiple ways to perform an attack that were available from the start.

We detail the AndrewSecureRPC case below, and give more examples in Appendix A. At the end of the day, we found no case of flawed reconstruction from an OFMC attack trace to $AnBx$ in our cross-validation process.

Protocol	*Authentication attacks*	*Secrecy attacks*	*Cross-validation*	*Java attack success*
AnBx_Xor_NSL	1	2	1|2|0/3	✓
AndrewSecureRPC	1	0	0|0|1/1	✓*
Diffie-Hellman_auth	0	1	✓	✓
Ebook	0	2	✓	✓
EMV_Visa_k3	1	1	✓	✓
EPMO	1	0	✓	✓
Goss	0	2	✓	✓
GSM	1	1	✓	✓
ISO4Pass	1	0	✓	✓
ISO5Pass	2	0	✓	✓
ISOCCFT2PassMutualAuth	1	0	✓	✓
Kerberos_PKINIT	1	0	✓	✓
KeyEx1	2	1	✓	✓
KeyEx2	2	1	✓	✓
KeyEx3	2	0	✓	✓
KeyEx5	2	0	✓	✓
MS-CHAPv2	0	1	✓	✓
NSPK	1	2	1|2|0/3	✓
Orig_1KP	2	1	✓	✓*
Orig_1KP_Fixed	0	1	✓	✓*
Orig_2KP	2	1	✓	✓*
Orig_2KP_Fixed	0	1	✓	✓*
Orig_3KP	2	1	✓	✓*
Orig_3KP_Fixed	0	1	✓	✓*
SET_Orig_Signed	2	3	2|1|2/5	✓*
SET_Orig_Unsigned	2	3	2|2|1/5	✓*
TMN	0	1	✓	✓
Woo-Lam	1	0	✓	×

Authentication/Secrecy - x: the studied protocol exhibits x attacks of this type.
Cross-validation - ✓: all verifications exhibit $Atk_2 = IsoSubSeq(Atk)$.
$x|y|z/N$: out of N trace reconstructions:
x are successfully cross-validated - as with ✓.
y are partially cross-validated, i.e. with Atk_2 denoting a behaviour that would have been possible in Atk.
z fail to cause a violation of their goal.
Java attack success - ✓: all processes pass the sanity checks and terminate.
✓*: attempted checks succeed, but some agents' runs do not terminate, because the OFMC attack trace does not contain sufficient information to terminate the protocol run.
×: the attack fails in Java.

Table 1: Comparison of reconstructed protocols attacks

It is worth to remind that our approach is not strictly bound to the languages and tools presented in this paper. In particular, we have verified our original and reconstructed protocols generating ProVerif specifications from *AnBx* (see foot-

note 1). We reproduced the relevant steps of the workflow and observed that the results are consistent with the *AnBx*/OFMC configuration. With AndrewSecureRPC, ProVerif could not terminate the verification because of internal loops caused by message symmetries. This kind of behaviour, discussed in the ProVerif manual, is not unexpected. In summary, the verification results in ProVerif did not contradict the ones presented in Table 1. The only exception is with Woo-Lam which has no attack in ProVerif because numbers and ciphertexts can be modelled with distinguishable data types unlike with OFMC.

AndrewSecureRPC cross-validation

AndrewSecureRPC as specified below is vulnerable to a reflection attack.

```
A -> B: A,{|NA|}shk(A,B)
B -> A: {|succ(NA),NB|}shk(A,B)
A -> B: {|succ(NB)|}shk(A,B)
B -> A: {|NB2|}shk(A,B)
```

The function *succ* increments a number. Here, A tries to authenticate B on $NB2$. In the trace, the last action is decisive, with $A = x601, B = x602$:

```
i -> (x601,1): {|NA(1)|}_(shk(x601,x602))
```

This attack assumes the lack of a freshness check on A's side, who could have otherwise noticed that it received the same message as it sent. As the intruder never knows NA, the original goal "*A authenticates B on NB2*" cannot be rewritten in a way that preserves its intended meaning.

6.2 Generated code execution

Concretely testing an attack found on the model would enhance the confidence and understanding in our results. This investigation is conducted in Java, using the *AnBx* compiler as a code generator and the *AnBxJ* library [18] as a common cryptographic and communication API.

Checks on message reception are automatically generated for every agent. As faulty or absent checks can lead to critical vulnerabilities like the OpenSSL *Heartbleed* bug [10], a model-driven approach helps to ensure consistency between high and low-level specifications of security properties.

Impersonation in the initial knowledge is handled through a configuration file. In practice, an agent would look up a table for known identities. We permit it, without any modification to the honest identity checks. Every agent records the other agents' identities and roles. Replacing an honest identity with another one achieves transparent impersonation while being faithful to the philosophy that honest agents are betrayed by their environment.

With the results compiled in Table 1, we conclude that the attacks given by OFMC are mostly applicable in a concrete environment. There are a few exceptions, where some honest agents' runs do not terminate as detailed in Appendix B, or where type-checking fails. Both might lead to attack detection, but the implementations could be hardened with freshness checks on received values, as demonstrated by the AndrewSecureRPC and Woo-Lam examples. The attack on Woo-Lam fails in the generated Java, as explained below:

Woo-Lam Java attack mitigation

The attack trace for the Woo-Lam protocol is a clear example of how a reasonable implementation can be more defensive than what a verification tool assumes. The message $\{|A, B, NB|\}shk(A, s)$ is swapped with NB, which has a different type. As a result, an error occurs during type-checking on B's side. This attack could be successful if the received data was only ever treated as type *Object*, but mitigated by minimal measures. Another fix could be to have freshness checks on NB, and B would notice that it received what it just sent.

Thus, the low-level type checks automatically generated by the *AnBx* compiler do not permit an undetected round-trip of NB, as B expected a ciphertext. This confirms the need of checking if attacks on the model occur in the concrete implementation, which is one of the main motivations of this work.

6.3 Performance

The execution time of the trace reconstruction and Java generation is by far inferior to the time required to verify a protocol with OFMC as the state-space grows. This shows that the overhead introduced in the workflow is minimal, if not negligible. In fact, it takes less than 10 seconds to reconstruct every attack and generate its Java code for the entire test suite.

This is conducted with a single-threaded process on a laptop i5-5200U CPU, and leads to a real-time workflow. The peak RAM usage is under 50MB and we compile the results in Fig. 4. We use a logarithmic scale to highlight the reconstruction component, and note that the Java generation performance is tied to code independent of the specific contribution presented in this paper.

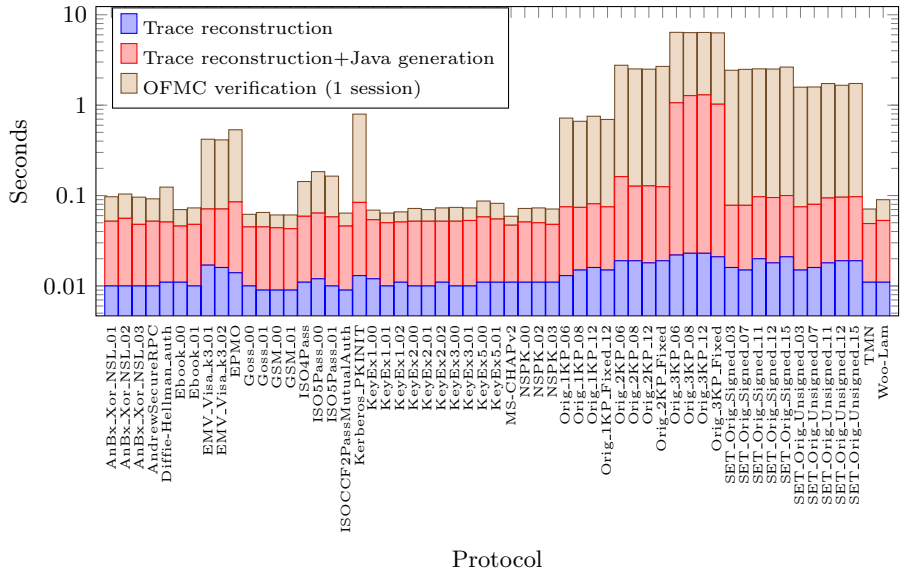

Fig. 4: Execution times needed for our workflow compared to OFMC verification

7 Conclusion and future work

In this paper, we explore a new angle of a Model-Driven Development approach with both abstract and concrete feedback to analyse verification results. Given an insecure protocol, we automatically reconstruct its abstract attack trace to the same notation used to specify the protocol (e.g. the user-friendly *AnBx*). We also generate an executable Java implementation of the entire attacked protocol, including the intruder's code.

Our solution provides a fully automated workflow suitable for real-time trace reconstruction and assessment of the applicability of an abstract attack trace in a concrete system. We showed examples where common mechanisms in low-level systems would prevent an abstract OFMC attack from being undetected, or even executable.

We plan to reconstruct a broader class of attacks in future work, notably with replay and session manipulation. Besides, we could cover other target languages, as allowed by the Executable Narration intermediate representation.

References

1. Armando, A., Pellegrino, G., Carbone, R., Merlo, A., Balzarotti, D.: From model-checking to automated testing of security protocols: Bridging the gap. In: Brucker, A.D., Julliand, J. (eds.) Tests and Proofs - 6th International Conference, TAP@TOOLS 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7305, pp. 3–18. Springer (2012), https://doi.org/10.1007/978-3-642-30473-6_3
2. Bai, G., Lei, J., Meng, G., Venkatraman, S.S., Saxena, P., Sun, J., Liu, Y., Dong, J.S.: AUTHSCAN: automatic extraction of web authentication protocols from implementations. In: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013. The Internet Society (2013), https://www.ndss-symposium.org/ndss2013/authscan-automatic-extraction-web-authentication-protocols-implementations
3. Bansal, C., Bhargavan, K., Delignat-Lavaud, A., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. J. Comput. Secur. **22**(4), 601–657 (2014), https://doi.org/10.3233/JCS-140503
4. Barendregt, H., Geuvers, H.: Proof-assistants using dependent type systems. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 1149–1238. Elsevier and MIT Press (2001), https://doi.org/10.1016/b978-044450813-3/50020-5
5. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. International Journal of Information Security **4**(3), 181–208 (2005). https://doi.org/10.1007/s10207-004-0055-7
6. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Computer Security Foundations Workshop, IEEE. pp. 0082–0082. IEEE Computer Society (2001)
7. Büchler, M., Oudinet, J., Pretschner, A.: Semi-automatic security testing of web applications from a secure model. In: Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012. pp. 253–262. IEEE (2012), https://doi.org/10.1109/SERE.2012.38

8. Bugliesi, M., Calzavara, S., Mödersheim, S., Modesti, P.: Security protocol specification and verification with AnBx. Journal of Information Security and Applications **30**, 46–63 (2016). https://doi.org/10.1016/j.jisa.2016.05.004
9. Carbone, R., Compagna, L., Panichella, A., Ponta, S.E.: Security threat identification and testing. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015. pp. 1–8. IEEE Computer Society (2015), https://doi.org/10.1109/ICST.2015.7102630
10. Cassidy, S.: Existential type crisis : Diagnosis of the OpenSSL Heartbleed Bug (2014), http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html
11. Dadeau, F., Héam, P., Kheddam, R., Maatoug, G., Rusinowitch, M.: Model-based mutation testing from security protocols in HLPSL. Softw. Test. Verification Reliab. **25**(5-7), 684–711 (2015), https://doi.org/10.1002/stvr.1531
12. Dark, M., Belcher, S., Bishop, M., Ngambeki, I.: Practice, practice, practice... secure programmer! In: Proceeding of the 19th Colloquium for Inf. System Security Education (2015)
13. Dolev, D., Yao, A.: On the security of public-key protocols. IEEE Transactions on information Theory **2**(29) (1983). https://doi.org/10.1109/tit.1983.1056650
14. Lowe, G.: A hierarchy of authentication specifications. In: CSFW'97, pp. 31–43. IEEE Computer Society Press (1997)
15. Maatoug, G., Dadeau, F., Rusinowitch, M.: Model-Based Vulnerability Testing of Payment Protocol Implementations. In: HotSpot'14 - 2nd Workshop on Hot Issues in Security Principles and Trust, affiliated with ETAPS 2014. Grenoble, France (Apr 2014), https://hal.inria.fr/hal-01089682
16. Mihret, Z., Liu, L.: Attack-driven test case generation approach using model-checking technique for collaborating systems. In: 2nd IEEE/ACM International Workshop on Engineering and Cybersecurity of Critical Systems, EnCyCriS@ICSE 2021, Madrid, Spain, June 3-4, 2021. pp. 1–8. IEEE (2021), https://doi.org/10.1109/EnCyCriS52570.2021.00008
17. Mödersheim, S.: Algebraic properties in alice and bob notation. In: Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan. pp. 433–440. IEEE Computer Society (2009), https://doi.org/10.1109/ARES.2009.95
18. Modesti, P.: Efficient Java code generation of security protocols specified in AnB/AnBx. In: Security and Trust Management - 10th International Workshop, STM 2014, Proceedings. pp. 204–208 (2014). https://doi.org/10.1007/978-3-319-11851-2_17
19. Modesti, P.: AnBx: Automatic generation and verification of security protocols implementations. In: 8th International Symposium on Foundations & Practice of Security. LNCS, vol. 9482. Springer (2015). https://doi.org/10.1007/978-3-319-30303-1_10
20. Modesti, P.: Integrating formal methods for security in software security education. Informatics in Education **19**(3), 425–454 (2020). https://doi.org/10.15388/infedu.2020.19
21. Peroli, M., Meo, F.D., Viganò, L., Guardini, D.: Mobster: A model-based security testing framework for web applications. Softw. Test. Verification Reliab. **28**(8) (2018), https://doi.org/10.1002/stvr.1685
22. Sebastian Mödersheim: OFMC distribution and tutorials (2022), https://www2.compute.dtu.dk/~samo/

23. Viganò, L.: The spacios project: Secure provision and consumption in the internet of services. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013. pp. 497–498. IEEE Computer Society (2013), https://doi.org/10.1109/ICST.2013.75
24. Walden, J., Frank, C.E.: Secure software engineering teaching modules. In: Proceedings of the 3rd Annual Conference on Information Security Curriculum Development. pp. 19–23. InfoSecCD '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1231047.1231052

A Cross-validation of OFMC traces

This appendix details the reasons why some attack trace reconstructions cannot be cross-validated according to our criteria in Section 6.1.

AnBx_Xor_NSL

This protocol has two secrecy violations on the terms M and N, where the second traces denote different intruder behaviours than the first, that would though have been possible in the first ones. Both second traces display the same behaviour, and we detail the M case here. With "M *secret between* A, B" being violated, the original actions are:

```
A -> B: {N,A}pk(B)
B -> A: {M,xor(N,ag2xor(B))}pk(A)
A -> B: {M}pk(B)
```

The first trace, where $A = x20, B = x30$ ends with:

```
(x20,1) -> i: {M(2)}_(pk(i))
i -> (x30,1): {M(2)}_(pk(x30))
```

The intruder impersonated B, and A sent M encrypted with the wrong key, allowing M to be known by the intruder. The *AnBx* reconstruction ends with:

```
A -> Intr: {M}pk(Intr)
Intr -> B: {M}pk(B)
```

The second trace for "M *secret between* A, B", where $A = x32$, is:

```
(x32,1) -> i: {N(1),x32}_(pk(i))
i -> (x32,1): {x305,N(1) XOR x306}_(pk(x32))
(x32,1) -> i: {x305}_(pk(i))
```

Here, $x305$ replaces M. The intruder chose to swap the message instead of forwarding the original one. While this behaviour is different, it could have been done as well in the first trace, and B is still impersonated here. Note that N is also known by the intruder here, violating its associated secrecy goal.

NSPK

NSPK is specified as below, and $NxNB$ should be secret between A and B:

```
A->B: {NxNA,A}pk(B)
B->A: {NxNA,NxNB}pk(A)
A->B: {NxNB}pk(B)
```

The first trace with $A = x20, B = x30$ ends with:

```
(x20,1) -> i: {NxNB(2)}_(pk(i))
i -> (x30,1): {NxNB(2)}_(pk(x30))
```

The reconstructed actions for "$NxNB$ *secret between* A, B" end with:

```
A -> Intr: {NxNB}pk(Intr)
Intr -> B: {NxNB}pk(B)
```

And the second trace has $A = x31$, without B appearing:

```
(x31,1) -> i: {NxNA(1),x31}_(pk(i))
i -> (x31,1): {NxNA(1),x305}_(pk(x31))
(x31,1) -> i: {x305}_(pk(i))
```

In both traces, B is impersonated. Like with AnBx_Xor_NSL, data is swapped and not forwarded, but the two behaviours are possible in both cases. The other secrecy goal on $NxNA$ is also violated with the exact same trace.

SET_Orig_Signed/Unsigned

Those two versions of SET are vulnerable to the same attack on the goal "C *authenticates* a *on* $NxAuthCode$". The last two original actions are:

```
a -> M: {{C,...,NxAuthCode}inv(sk(a))}pk(M)
M -> C: {C,...,NxAuthCode}inv(sk(M))
```

In the trace where $C = x601$, the intruder impersonates M and skips the action with a, to jump to the last action, with $x511$ replacing $NxAuthCode$:

```
i -> (x601,1): {x601,...,x511}_inv(sk(i))
```

The attack on the new goal "C *authenticates* $Intr$ *on* $X511$" disappears, because there is no more possible short-circuiting. Similarly, a first goal "C *authenticates* M *on* $NxAuthCode$" could not have been violated.

Both SET versions also fail to satisfy "C *authenticates* a *on* $Contract$", which would have been satisfied if M was to be authenticated like previously, so short-circuiting is also part of those attacks. For the signed version, the attack is not reproducible, for the same reasons as with $NxAuthCode$.

The unsigned one, on its third action, is of the form:

```
C -> M : hash(PIData),...
```

Here, $hash(PIData)$ is a component of $Contract$. This action is followed by exchanges between a and M before M talks to C. The first trace focuses on bypassing a and M, ending with this, where $C = x601$:

```
(x601,1) -> i: hash(PIData),...
i -> (x601,1): {x601,...}_inv(sk(i))
```

Components of $PIData$ are included in the last action but do not originate from a or M. Short-circuiting a is no longer possible, but on the second-to-last action, $hash(PIData)$ is not signed with C's private key, so it can be swapped in transit. In the second trace, this is what we observe for the goal "C *authenticates* $Intr$ *on* $Contract$", with $C = x901, Intr = x902$:

```
(x901,1) -> i: hash(PIData),...
i -> (x902,1): x712,...
```

$x712$ replaces $hash(PIData)$, and $Intr$ has a different version of $hash(PIData)$ than C. We encode it as $hash(X712)$ for type consistency in $AnBx$. $Intr$ does not know some components of $PIData$, so it cannot realise that it received an arbitrary hash. Furthermore, $hash(PIData)$ is not sent back to C for confirmation, so the agreement on the same value is not verified. This attack would have worked on M in the first trace, but a simple bypass of a and M was chosen.

The goal "$NxAuthCode$ *secret between* C, M, a" sees a last discrepancy between traces. $NxAuthCode$ is transmitted only encrypted with the private key of a, thus not ensuring its confidentiality. In the first trace, the intruder is purely passive. The second time, it bypasses C and produces its own intermediate protocol values, but this is not necessary to the attack.

B Attack detection in Java

This part explains why some attack traces' behaviours could be detected by the honest agents.

AndrewSecureRPC

Here, all checks are passing, but the trace stops involving B before the end of B's original actions. The trace ends as follows, with $A = x601, B = x602$:

```
(x601,1) -> i: {|succ(NB(2))|}_(shk(x601,x602))
i -> (x601,1): {|NA(1)|}_(shk(x601,x602))
```

B never receives $\{|succ(NB)|\}shk(A, B)$ and could notice something. The intruder ignored the actions for B after the third step. Yet, two actions could be introduced in the narration before the last one, allowing B to terminate its run:

```
Intr -> B: {|succ(NB)|}shk(A,B)
B -> Intr: {|NB2|}shk(A,B)
```

On top of it, A could easily notice that it received the same NA as it sent.

Orig_iKP

The iKP reconstructions do not terminate when: "$Auth$ *secret between* C, M, a" is violated. $Auth$ is transmitted in plaintext during the last two original actions:

```
a -> M: Auth,...
M -> C: Auth,...
```

However, the last trace action involves a sending the message to the intruder. The forwarding to M and then C is absent from the trace, which would have been possible because the attack does not require an active intruder.

SET_Orig_Signed/Unsigned

The three secrecy violations occur with a purely passive intruder. In those traces, C and sometimes M do not complete their last actions, because the intruder does not carry the messages to them. Having OFMC print the entire protocol narration would have been sufficient to ensure termination.