

Formal Vulnerability Analysis of a Security System for Remote Fieldbus Access

Manuel Cheminod, Alfredo Pironti, and Riccardo Sisto

Abstract—As fieldbus networks are becoming accessible from the Internet, security mechanisms to grant access only to authorized users and to protect data are becoming essential. This paper proposes a formally-based approach to the analysis of such systems, both at the security protocols level, and at the system architecture level. This multi-level analysis allows the evaluation of the effects of an attack on the overall system, due to security problems that affect the underlying security protocols. A case study on a typical fieldbus security system validates the approach.

Index Terms—Computer security, cryptographic protocols, formal specifications, formal verification, industrial control, SCADA systems.

I. INTRODUCTION

IN the past, the security of industrial control systems (ICS) was mainly achieved thanks to their physical isolation. Nowadays, control systems are often interconnected to form complex distributed systems, such as for example SCADA (Supervisory Control and Data Acquisition) systems where a control centre monitors and controls geographically dispersed field sites. These systems tend to be no longer isolated, thus becoming prone to cyber attacks. Because of the serious effects that these attacks may have on the controlled assets, securing interconnected control networks against intruders has emerged as an important research topic [1]–[7] and has given rise to specific recommendations [8].

Security can be added to such systems by leveraging security methodologies already adopted in general-purpose computer networks, but with adaptations to address the special features of the industrial control networks. As an example, the limited computational power of network devices and the need for prompt real-time responses prevent the use of complex and time-consuming cryptographic primitives.

When developing a new security system, it is important to accurately verify that it actually achieves the desired protection, which is hard to do without proper methods and tools, because of the unconstrained behaviour of attackers and because of the inherent complexity of interconnected network systems. When an informal or semi-formal description of the system is the only available specification, some ambiguities about its interpretation may arise. These ambiguities can lead to different implementations being derived from the same protocol description, possibly creating flaws that can be exploited

by an attacker. For example, in [9] and [10] two attacks are reported against implementations of the SSH and SSL/TLS protocols respectively. Such attacks are possible because in both cases the standard is not precise enough at specifying how some checks on received data must be implemented. Some implementations of such protocols implement the checks in a way that is not susceptible to attacks, while other implementations of the same protocols implement the checks in a different way that is still acceptable with respect to the protocol specification but makes the protocol implementation vulnerable.

Formal methods can help in this respect, by providing rigorous ways for specifying such systems and for systematic reasoning about their security properties and vulnerabilities. Recently, formal methods for security assessment have made much progress, and automated tools that can be used even by non-experts are available [11], [12]. It is important to note that automated formal methods need to work on abstractions of the real systems in order to be successful. Consequently, they are useful for checking the logical correctness of an abstract model of a system and they may miss vulnerabilities not captured by the abstraction being used.

This paper shows how state-of-the-art automated formal methods can be used to analyse the correctness and effectiveness of a security system for distributed industrial control systems [13], and highlights the benefits that this kind of analysis can give. In [13], a security system for remote fieldbus access is only informally specified. One contribution of this paper is a formalization of the security-related parts of the system. This formalization resolves possible ambiguities, and makes explicit some security-relevant assumptions that were left implicit in the informal specification. It is also formally shown that, if not carefully resolved, such possible ambiguities could lead to the deployment of an insecure system.

Case studies and methodologies about using formal methods for analysing security protocols and networks, even specifically in the industrial networks area, have already appeared in literature (e.g. [12], [14], [15]). The novelty of the approach described in this paper is mainly in the exploitation of the positive interactions that arise from the use of two different modelling and verification techniques, at different abstraction levels. As a first step, the security protocols used in the system are modelled and analysed separately, obtaining a formal statement of the security properties they can guarantee. Based on such statements, the second step is to build and analyse a higher level model of the whole networked system, where protocol details such as exchanged messages are no longer represented, but the possible attacks on the protocols are represented abstractly. This model considers not only the

Manuscript received —; revised —.

M. Cheminod is with IEIIT, National Research Council, I-10129 Torino, Italy (e-mail: manuel.cheminod@polito.it).

A. Pironti and R. Sisto are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, I-10129 Torino, Italy (e-mail: {alfredo.pironti, riccardo.sisto}@polito.it).

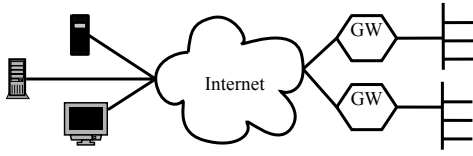


Fig. 1. Overall system architecture.

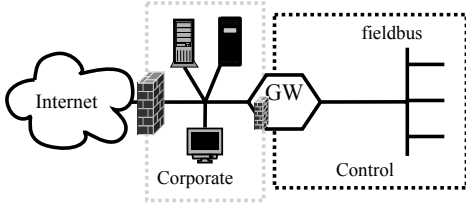


Fig. 2. Sample network architecture for single fieldbus.

security protocol vulnerabilities which emerged during the first verification step, but also the vulnerabilities of network components, thus being able to spot specific network configurations and actions that allow a malicious agent to successfully perform an attack on the system. The possibility to combine formal analyses at different levels in this way is important because it reduces the risk of incorrect modelling, which can arise if the security properties of a cryptographic protocol are not properly understood and modelled in the system model.

The rest of the paper is organized as follows. In section II, the security system to be analysed is presented. Then, section III presents the formal specification and analysis of the security protocol used in the system, while section IV shows how the results of the analysis of the protocol can be integrated into the analysis of the vulnerabilities of a specific networked system where the security system is deployed. Section V concludes this paper.

II. THE SECURITY SYSTEM

This paper considers a security system for protecting the access to interconnected fieldbuses [13], which is based on an architecture shown in figure 1. Each fieldbus can be accessed through the Internet via a gateway that performs protocol conversion. Each gateway is also used as an access control filter and as a means for enforcing data integrity and origin authentication for transmitted data. This kind of solution has the interesting feature of not requiring modifications to the fieldbus, and not burdening fieldbus nodes with cryptography. When the network is configured for each fieldbus as in figure 2, the system is also compatible with the recommended network architectures for industrial control systems [8], where firewalls are used for controlling the flow between a control network and a corporate network. The firewall in front of the control network can be implemented inside an application-proxy gateway, like the ones in the architecture proposed in [13].

Other solutions are also possible, such as for example the ones being proposed by the American Gas Association (AGA) [16], which has recently started the development of a suite of standards to protect the data transmitted by SCADA

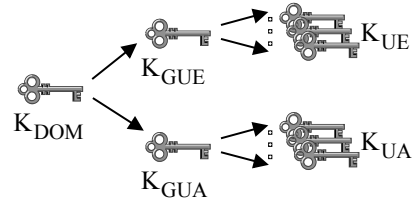


Fig. 3. Key derivation hierarchy.

systems and authenticate the originators of messages. At the moment, however, only the first part, on Background, Policy and Test Plan, has been released.

The security system proposed in [13] starts from the assumption that communication with the fieldbuses can be initiated by client users and is controlled by access control lists (ACL) located in the gateways. Special users with administrative rights can configure gateways and change their ACL. As in [13], here it is assumed that all system users, gateways and fieldbuses belong to the same domain. Interconnections between different domains are not considered.

Privacy, authentication and integrity protection are achieved by shared key cryptography, using a hierarchical key system, with keys stored in smartcards (one smartcard for each user and one for each gateway). According to the nomenclature given in [13], the K_{DOM} key is the domain root key, from which all other shared keys are derived. From K_{DOM} , the Gateway User Authentication key (K_{GUA}) and the Gateway User Encryption key (K_{GUE}) are derived. From K_{GUA} and K_{GUE} , it is possible to respectively derive for each user a User Authentication key (K_{UA}) and a User Encryption key (K_{UE}). The key derivation tree is depicted in figure 3. Each user needs to store a local copy of its own K_{UA} and K_{UE} ; each gateway stores a local copy of K_{GUA} and K_{GUE} , so that the specific user keys can be generated on the fly, when a user connects. Note that there exists a different K_{UA} and K_{UE} for each user of the system, and one user always uses the same key-pair to contact any gateway.

A similar architecture is adopted for the administrative keys, each gateway storing its specific administration authentication and encryption keys, and each administrator storing the administration master keys, from which gateway specific keys are derived.

Note that since all gateways in the same domain share the same K_{GUA} and K_{GUE} keys, if just one gateway is compromised, the keys must be changed in the whole system (including the users, since they have derived keys). This issue could be avoided by using asymmetric keys, so that each gateway and user has their own key. However, symmetric keys are chosen for this system architecture in order to cope with the limited computational power available at the gateways, and possibly at the user (mobile) devices.

A custom request/response security protocol controls communication between a user and a gateway. The request/response scenario is depicted in figure 4. The user initiates a session by sending a request to a gateway. While the message structure will be detailed in section III-B, it is worth pointing out now that the content of the request is protected

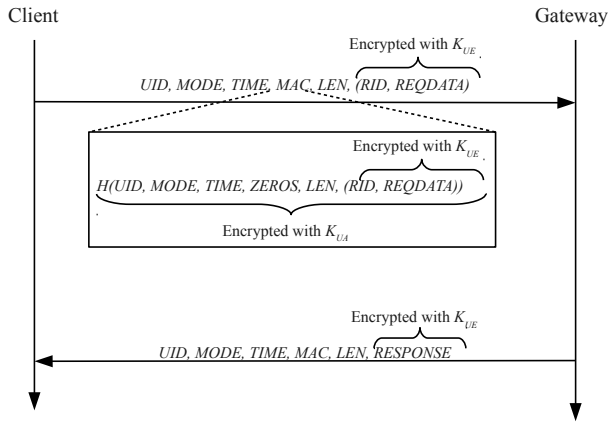


Fig. 4. Request/response security protocol scenario.

by a Message Authentication Code (MAC), whose purpose is to ensure integrity, and the application data are encrypted, in order to provide privacy. Authentication is implicitly achieved, because only authorized users and gateways can access the shared authentication and encryption keys. In particular, the K_{UA} and K_{UE} (derived from K_{GUA} and K_{GUE}) are respectively used to compute the MAC and encrypt the application data (details are given in section III-B).

The gateway replies by sending a response to the user, again protected by a MAC, with encrypted application data.

If the application data contain administrative commands, then the message payload is further authenticated and encrypted by using the administrative keys, so that only the intended gateway can understand the content (and further check its integrity).

III. SECURITY PROTOCOL ANALYSIS

A. Formalism and Background

The formal models of protocol agents are specified in this paper by using the applied pi calculus (*a-pi* for short) modelling language [17]. Essentially, a security protocol is composed of several agents, each implementing one of the protocol roles. A protocol role can be modelled in a-pi by creating its corresponding a-pi process. Protocol roles communicate with each other by communication channels. The statement $\text{out}(c, M)$ means output of message M over channel c ; while $\text{in}(c, x)$ represents input from channel c with storing of input data in variable x . The shorthand $\text{in}(c, = \text{data})$ means that input from channel c is accepted only if it matches the content of data. That is equivalent to the more verbose

$$\text{in}(c, \text{tmp}); \text{ if tmp} = \text{data then}$$

Comments are enclosed between $(*$ and $*)$.

In a-pi, cryptographic primitives are symbolically represented by pairs of constructors and destructors. For example, symmetric encryption of message M with key k is represented by the constructor $\text{enc}(k, M)$ and a corresponding destructor is defined to represent decryption by the rewriting rule

$$\text{dec}(k, \text{enc}(k, M)) = M$$

Hashing of message M is represented by the $H(M)$ constructor. The absence of a hash-related destructor represents the fact that a hash cannot be inverted. This symbolic idealized view of cryptography (perfect cryptography) is effective in efficiently spotting logical errors in the use of cryptography within security protocols. However, by abstracting away all computational-hardness theories of cryptography, low-level errors such as bad interactions between specific cryptographic algorithms cannot be discovered.

The attacker is assumed to have complete control over public channels: it can drop, forge, alter and eavesdrop messages. This abstract way of modelling a security protocol and an attacker comes from Dolev and Yao [18]. Even if it cannot represent all the possible weaknesses, it is still considered a valid means to reason about security protocol logic.

The a-pi language has been chosen over other modelling languages (e.g. spi calculus, or CSP) because it is easily extensible, enables precise models to be defined by explicitly including the checks on received data, and can be analysed by ProVerif [11], one of the most powerful tools currently available for security protocol analysis. ProVerif is a security protocol specific theorem prover that can be used by non-experts, too: differently from other general purpose theorem provers, which require manual interaction, it is fully automatic. Moreover, being a theorem prover, it can prove security properties for an unbounded number of parallel protocol sessions, a feature not usually allowed by state exploration tools.

Essentially, in order to verify a security protocol with ProVerif, an a-pi model of the protocol is enriched by adding a formal description of the intended security properties (see section III-C for details). Then, this enriched model is given to the ProVerif tool, which transforms the a-pi representation of the protocol into a set of Horn clauses, and then uses a custom resolution algorithm¹ in order to prove the requested security properties, expressed as logical formulas. When (if) ProVerif terminates execution, it either gives a proof of correctness of the protocol with respect to the requested security properties, or shows an attack breaking them. Since verification of typical security properties on protocols is undecidable [19], the tool may also not terminate, or terminate without an answer. This may happen especially when the extension features of a-pi are exploited by adding special equational theories. Nevertheless, ProVerif has been extensively used, even with large protocols (e.g. [20], [21]), and it has been shown to return an answer in all the most significant cases and to scale well.

B. Formalizing the Protocol

In the analysed request/response protocol, two roles can be identified, namely the user (initiator) role and the gateway (responder) role.

Let UID be the identifier of a user. As specified in [13], the function that derives each User Authentication key (K_{UA}) from K_{GUA} can be simply modelled as

$$\begin{aligned} K_{UA} &= f_{GUA}(UID) \\ &= \text{enc}(K_{GUA}, H(UID)) \end{aligned}$$

¹The standard Prolog resolution algorithm would not terminate.

```

1U: let user =
2U:  (* Prepare and send request *)
3U:  let request = (RID,REQDATA) in
4U:  let encReq = enc(Kue,request) in
5U:  new cTime;
6U:  let reqMac = enc(Kua,H((UID, RKES_AUTH_ENC_3_DES,
7U:    cTime, ZEROS, len(encReq), encReq))) in
8U:  event sentReq(RID,REQDATA);
9U:  out(c, (UID, RKES_AUTH_ENC_3_DES, cTime, reqMac,
10U:    len(encReq), encReq));
11U:  (* Receive and parse response *)
12U:  in(c, (=UID, =RKES_AUTH_ENC_3_DES, sTime, resMac,
13U:    encResLen, encRes));
14U:  (* Check MAC *)
15U:  if resMac = enc(Kua,H((UID, RKES_AUTH_ENC_3_DES,
16U:    sTime, ZEROS, encResLen, encRes))) then
17U:  let response = dec(Kue,encRes) in
18U:  event receivedRes(response).

```

Fig. 5. User role model.

The same reasoning applies for the User Encryption key (K_{UE}).

In the formal model developed here, only communication that happens over the Internet or the corporate network is considered, while communication happening over the fieldbus is left out. Indeed, fieldbus protocols are susceptible to attacks [22], and in many cases they even lack security protection measures, thus being evidently insecure. However, here the choice has been to focus on the vulnerabilities of the security protocol between user and gateway, rather than those of the security protocols in the fieldbus. It should be noted that neglecting attacks on the fieldbus protocol is safe if the attacker has no physical access to the fieldbus, e.g. because it is placed in a safe building, and the only means for an attacker to access the fieldbus is from the Internet, or the corporate network, through the gateway.

Referring to figure 4, a user initiates a session by sending a message with the following format:

$$UID, MODE, TIME, MAC, LEN, REQUEST$$

where UID is the identifier of the user, $MODE$ is a field indicating the authentication and encryption algorithms used in the message, $TIME$ is a timestamp indicating the user's notion of the current time at the gateway, MAC is the Message Authentication Code, LEN is the length of the request data, and finally $REQUEST$ is the user request data.

As prescribed in [13], the MAC is computed by encrypting the hash of the whole message with key K_{UA} , where the bytes reserved for the MAC itself are all set to zero.

The request is composed of two fields, RID containing the receiver ID, and $REQDATA$, containing the payload of the request. In order to ensure confidentiality, the request is encrypted with key K_{UE} .

The user and gateway roles can be modelled by the a-pi processes in figures 5 and 6 respectively. The full model that can be analysed by ProVerif can be found in [23].

In the user model, the request data are prepared at line 3U and they are encrypted at line 4U. Then the timestamp is generated at line 5U, and the MAC is computed at lines 6U–7U. Note that ProVerif does not provide full support for time, however, the timestamp is modelled as fresh data, since it is

```

1G: let gateway =
2G:  (* Receive and parse request *)
3G:  in(c, (rUID, =RKES_AUTH_ENC_3_DES, cTime,
4G:    reqMac, encReqLen, encReq));
5G:  (* Generate the User Keys on the fly *)
6G:  let sKua = enc(Kgua,H(rUID)) in
7G:  let sKue = enc(Kgue,H(rUID)) in
8G:  if reqMac = enc(sKua,H((rUID, RKES_AUTH_ENC_3_DES,
9G:    cTime, ZEROS, encReqLen, encReq))) then
10G:  (* No actual check on cTime *)
11G:  let (rid,reqData) = dec(sKue,encReq) in
12G:  event receivedReq(rid,reqData);
13G:  (* Prepare and send response *)
14G:  let encRes = enc(sKue,res(rid,reqData)) in
15G:  new sTime;
16G:  let resMac = enc(sKua,H((rUID, RKES_AUTH_ENC_3_DES,
17G:    sTime, ZEROS, len(encRes), encRes))) in
18G:  out(c, (rUID, RKES_AUTH_ENC_3_DES, sTime, resMac,
19G:    len(encRes), encRes)).

```

Fig. 6. Gateway role model.

likely to be different at each protocol session execution.²

Line 8U introduces a fictitious event, signalling that the request is going to be sent by the user. This event is only needed for security properties verification, and it will be discussed later. Finally, at lines 9U–10U the request message is sent over channel c .

At the gateway side, the message is received and parsed; in the gateway model, this is done at lines 3G–4G. After a message has been received, even before the MAC can be verified, the gateway needs to use the received UID ($rUID$ in the model) in order to derive the appropriate user keys. The keys are generated at lines 6G–7G.

At line 8G the MAC is checked. If the MAC is correct, the gateway shall check that the received user time is within the allowed Time Window (TW), which is $\pm 150s$ as defined in [13]. As ProVerif has no notion of quantitative time, this check is conservatively modelled here by the gateway always accepting any received timestamp (this corresponds to having an infinite TW). This behaviour over-approximates the real one because it includes some protocol executions that are impossible in reality because the check fails. Over-approximations preserve soundness because the security of the over-approximated protocol implies the security of the real protocol for the most relevant security properties. In practice, this means that if no attack is found on the over-approximated protocol, none exists on the real protocol. Instead, if a freshness attack is found on the model, the attack also exists on the real protocol but only provided the attacker can conduct it within the $150s$ time window.

At line 11G the encrypted request data are decrypted. At line 12G a fictitious security-related event is emitted (discussed later in detail); it signals that the gateway has received a request, and this request is considered valid by the gateway, because it passed MAC validation and decryption.

From line 14G to line 19G the gateway prepares and sends the response. A response message shares the same format of a request message, except that the timestamp is newly generated by the gateway, and the response data do not contain the receiver ID, because the user is the implicit receiver of

²It is different provided an adequately fine time resolution is chosen.

the response message. The response data are modelled as a function $\text{res}()$ of the request data, which enables reasoning about the relation between each request and its response.

Finally, on the user side, the response is received and parsed at lines 12U–13U. The MAC is checked at lines 15U–16U, and the encrypted response data are decrypted at line 17U. Finally, a security-related event is emitted, signalling that the user has received a response that is considered to be valid.

The informal protocol description does not specify whether a new request can or cannot be issued before the previous request has been received. In order to cope with the most general case, it is assumed (and thus made explicit) here that new requests can be emitted before the previous requests have been received.

As cited above, the protocol also allows administrative messages to be exchanged, by further applying authentication and encryption to the request and response data by using the Gateway Authentication and Gateway Encryption keys. For brevity, the administrative messages are not considered in the model shown.

C. Security Considerations

As stated in [13], the protocol should provide the following security properties: *privacy*, meaning that no-one except the sender and the intended recipient should be able to understand the relevant content of the message; *authentication*, meaning that only the enabled users should be able to interact with the system; *integrity*, meaning that if sensitive data are altered, alteration can be recognized and the altered message discarded.

Closely related to authentication, *freshness* should be achieved by this protocol too, meaning that an authentic message is considered authentic by the receiver only once. If an attacker can re-play a message more than once, so that the receiver considers it as valid more than once, we consider this a violation of authentication, because the (non-authorized) attacker was able to successfully interact with the system.

Another property of interest when analysing this kind of protocols is *non-repudiation*, meaning that when a protocol agent sends (receives) a message, evidence is produced at the same time, so that the agent cannot deny the message was sent (received). The custom protocol analysed here is not designed to fulfil this property: by using symmetric key encryption, both the initiator and the responder may forge messages intended to be created by the other agent.

In order to reason about the model, and to get a rigorous proof of correctness (or in-correctness) of the protocol, the security properties must be formally defined. In ProVerif, privacy claims about this protocol can be specified by

$$\text{query attacker : RID, attacker : REQDATA, attacker : res(RID, REQDATA).}$$

meaning that neither the receiver ID, nor the request data, nor the response shall ever be known by the attacker. ProVerif provides a proof stating that this property is true, thus effectively stating that the protocol is correct with respect to privacy.

When it comes to authentication, integrity and freshness (collectively called agreement here, for brevity) it turns out

that their formalization is not trivial: at least some assumptions that were left implicit in the informal protocol description must be made explicit.

As a first step, let us concentrate on agreement upon the request message. Formally, it is required that each time the gateway believes it has received a valid message from the user, then the user previously and intentionally sent that message. This property is also known as injective agreement [24]. In order to prove agreement properties like this, fictitious security-related events are used. In ProVerif, this agreement property can be expressed by the query

$$\text{query evinj : receivedReq}(x, y) ==> \text{evinj : sentReq}(x, y). \quad (1)$$

referencing the fictitious events receivedReq and sentReq . Unfortunately, ProVerif can prove that this property is false, meaning that the modelled protocol does not fulfil the injective agreement property. Indeed, once a first valid session has been executed, the attacker can replay the request of the first session to the gateway, which accepts the message as genuine, without the user getting involved at all. By taking into account the real TW, which is smaller than the modelled infinite one, it follows that the attacker has about 150s after the first valid session, in which the message can be replayed as many times as possible and accepted as genuine by the gateway.

However, even if it is true that the attacker can replay already sent messages, it cannot forge new messages from scratch. This can be captured by a weaker form of agreement, called non-injective agreement. Non-injective agreement means in this case that if the gateway believes it has received a valid message from the user once or more, then the user previously sent that message intentionally at least once. In ProVerif, this property is expressed by the query

$$\text{query ev : receivedReq}(x, y) ==> \text{ev : sentReq}(x, y).$$

which is in fact proved to be true in this protocol model.

Now, let us focus on the response messages. Besides an agreement property like the one expressed for the request messages, an additional property of this protocol can be expressed: the response must be considered valid only if it is a response to the original request. For example, it should not be possible that the responses to two different requests with the same user ID can be swapped, or that a single response can be used as response to two different requests with the same user ID. This property can be expressed in ProVerif as

$$\text{query evinj : receivedRes}(z) ==> (\text{evinj : receivedReq}(x, y) \ \& \ z = \text{res}(x, y)). \quad (2)$$

meaning that each time the user accepts a message as a valid response, the gateway must have received a request, and the message accepted by the user must be a response to the original request. Once again ProVerif proves this property is false, due to a replay attack similar to the one that affects the request messages. However, since the protocol prescribes that the user accepts any timestamp sent by the gateway as valid, it follows that the TW is in fact infinite, meaning that any response can be replayed by the attacker at any time.

Surprisingly, even the non-injective version of the above property is false, meaning that the attacker can somehow even forge new responses. Suppose the user sends out a request, and the attacker immediately replays the request back to the user. Since both request and response messages have the same format, it is possible that the user mis-interprets a request as a valid response. The informal protocol specification does not prescribe any specific way to distinguish request/response data payloads, so implementations of this protocol may be affected by this issue.

The above problem can be solved by explicitly tagging requests and responses with different identifiers, so that they cannot be confused. Under this assumption, the non-injective agreement becomes true, while the injective agreement is still false, because replay attacks are still possible on the user, as well as on the gateway.

Another security-relevant detail that is made explicit here is the assumption that the receiver IDs are unique in the whole domain, so that the same message replayed on any gateway has the same effect on the fieldbus network. If this assumption is dropped, then the attacker, by replaying to gateway G a (non-administrative) message intended for gateway H, can maliciously alter the state of the system. This is possible for non-administrative messages, because the same key is used by the user with all gateways, while administrative messages get encrypted with gateway specific keys.

Response messages can be replayed to the user they are intended to, but not to other users, as they already contain the *UID*, which is required to be unique in the system.

As ProVerif has no notion of quantitative time, effects due to jitter or time spent in computation are not taken into account. Yet, it is considered that the attacker may drop or reorder messages.

Another class of attacks that would be possible on the gateway side is the class of denial of service (DoS) attacks. Since K_{UA} and K_{UE} are computed on the fly, at each session at least two encryptions and two hashing operations must be performed by the gateway in order to check the MAC, that is before the received data are authenticated (K_{UE} can be in fact computed after MAC checking, but it does not alter the effectiveness of the attack). Conversely, the attacker can forge a malicious request with minimal effort, enabling the DoS attack. There are no trivial ways to mitigate this issue. User keys are computed on the fly, due to the constraints on memory, so it is not reasonable to assume that they are all stored at the gateway. Also, checking the timestamp before checking the MAC has no positive effect, since the attacker could easily forge it to make it appear as valid.

In general, DoS attacks can be identified by tracking the amount of resources used by an attacker and by the attacked agent before the malicious session gets dropped. A DoS attack is possible if the attacker uses less resources than the attacked agent. It is worth pointing out that ProVerif has no support for such kind of properties. So, the manual reasoning above shows the DoS attack, but ProVerif did not give any proof or hint about it.

Even if the protocol taken in this paper as a case study is not intended to provide non-repudiation, it may be interesting

```

1Uf: let user =
2Uf:   (* Prepare and send request *)
3Uf:   let request = (RID,REQDATA) in
4Uf:   let encReq = enc(Kue,request) in
5Uf:   new cTime;
6Uf:   new cNonce;
7Uf:   let reqMac = enc(Kua,H((UID, RKES_AUTH_ENC_3_DES,
8Uf:     cTime, cNonce, ZEROS, len(encReq), encReq))) in
9Uf:   event sentReq(RID,REQDATA);
10Uf:  out(c, (UID, RKES_AUTH_ENC_3_DES, cTime, cNonce,
11Uf:    reqMac, len(encReq), encReq));
12Uf:  (* Receive and parse response *)
13Uf:  in(c, (=UID, =RKES_AUTH_ENC_3_DES, sTime, rCNonce,
14Uf:    resMac, encResLen, encRes));
15Uf:  (* Check MAC *)
16Uf:  if resMac = enc(Kua,H((UID, RKES_AUTH_ENC_3_DES,
17Uf:    sTime, rCNonce, ZEROS, encResLen, encRes))) then
18Uf:  if rCNonce = cNonce then
19Uf:  let response = dec(Kue,encRes) in
20Uf:  event receivedRes(response).

```

Fig. 7. Fixed user role model.

to mention how in general ProVerif can be used to verify this kind of properties. ProVerif does not support verification of non-repudiation properties natively; however, the methodology proposed in [25], which can be applied in this context too, uses ProVerif to automate significant parts of the required proofs, while completing the remaining proof steps manually.

D. Fixing the Protocol

Even if it is assumed that the receiver IDs are unique in the whole system, and that request and response messages cannot be confused, the replay issue, both at the user and at the gateway sides, still remains to be addressed. (The replay attack on responses also includes matching each response with its request.)

In order to fix this issue, a possibility is to add a “nonce” field (a number used only once) after the time field of both the request and response messages.

On the user side, the nonce is used to match a response with its request: a response is valid only if the received nonce value matches the nonce value that was sent in the request. In order to implement this feature, it is just required that the nonce value is stored on the user side between the output of the request and the input of the valid response, which is acceptable even on a resource constrained device. When this check is in place, replay attacks on the user side are no longer possible, because each request uses a different nonce.

The a-pi model of the user can be updated according to the proposed fix, as shown in figure 7. With respect to the original model of figure 5, the fixed model differs by the introduction of the nonce. In particular, the nonce is generated after the timestamp, at line 6Uf, and it is included in the MAC (line 8Uf) and in the request (line 10Uf). The response, received and parsed by the user at lines 13Uf–14Uf, contains the *rCNonce* field, that is the nonce sent back by the gateway. After the MAC of the response is checked (lines 16Uf–17Uf), at line 18Uf it is checked whether the received nonce *rCNonce* matches the original nonce of this session, *cNonce*. If this is the case, this is the matching response for the request, and the protocol continues; otherwise, this is a mis-matched response, and the protocol execution terminates.

Under the assumption that requests and responses are tagged and therefore cannot be confused, ProVerif can finally prove that query (2) is true for the model of figure 7, meaning that replay attacks on the user side are no longer possible.

On the gateway side, in order to check that nonces are not used twice by the same user, all received nonces and their associated user IDs should be stored, but this would not be acceptable due to the device memory constraints. However, the timestamp mechanism can be exploited in order to limit the number of nonces to be stored.

In the proposed fix, the timestamp is checked for validity within the TW as usual. If this first check is successful, a second check is performed to see whether a message with the same nonce has already been received from the same user within the same TW. Older messages with the same nonce would not pass the timestamp check. This allows the gateway to store only used nonces associated with their user and with timestamps in the current TW, while older nonces can be deleted.

If the check on the nonce is successful, i.e. the user has not already used the same nonce in the TW, the message is valid, and the nonce of the current message is added to the list of “used” nonces for that user along with the corresponding timestamp. When the timestamp associated with one of the “used” nonces expires, that is when the message is not in the TW any more, the entry is deleted.

Unfortunately, there is no straightforward way to describe such a nonce freshness check mechanism in ProVerif, for an arbitrary and possibly infinite number of sessions, and in any case the resulting model would probably be too complex to be automatically verified by ProVerif.

Nevertheless, a correctness proof about the fixed gateway version can be obtained in a scenario with a fixed number of valid sessions and a still unbounded number of sessions started by the attacker. As an example, here the simplest scenario is described, where a single valid session is modelled. In this case, there is a single user instance (described by the fixed user model of figure 7). On the gateway side, the model is shown in figure 8. A single instance of the `gatewayFirst` process models the first protocol session. At the end of this session (line 33Gf), the replication of the `gatewayOther` process models all possibly infinite subsequent sessions (initiated by the attacker). Since the `gatewayFirst` model handles the first session, any value for the nonce is accepted as valid, and it is stored in `firstNonce`. In the `gatewayOther` model, the nonce received in the request is stored in `curNonce` (line 3Gf), and after checking the MAC of the request (lines 8Gf–9Gf), at line 11Gf `curNonce` is checked against `firstNonce`, the nonce used in the first session. If they match, then this session is a replay attack, and it is correctly recognized and terminated. If instead `curNonce` is different from `firstNonce`, the gateway accepts the request as valid. However, in this single valid session scenario, this is something that should never happen, because it would mean the attacker had succeeded in forging a valid new request. This is why in the model a `failure` event is emitted in this case, representing an unsafe protocol state.

In fact, ProVerif successfully proves that the `failure` event

```

1Gf: let gatewayOther =
2Gf: (* Receive and parse request *)
3Gf: in(c, (rUID2, =RKES_AUTH_ENC_3_DES, cTime2, curNonce,
4Gf:   reqMac2, encReqLen2, encReq2));
5Gf: (* Generate the User Keys on the fly *)
6Gf: let sKua2 = enc(Kgua, H(rUID2)) in
7Gf: let sKue2 = enc(Kgue, H(rUID2)) in
8Gf: if reqMac2 = enc(sKua2, H((rUID2, RKES_AUTH_ENC_3_DES,
9Gf:   cTime2, curNonce, ZEROS, encReqLen2, encReq2))) then
10Gf: (* No actual check on cTime2 *)
11Gf: if curNonce = firstNonce then 0
12Gf: else event failure(). (* should never be emitted *)

13Gf: let gatewayFirst =
14Gf: (* Receive and parse request *)
15Gf: in(c, (rUID, =RKES_AUTH_ENC_3_DES, cTime, firstNonce,
16Gf:   reqMac, encReqLen, encReq));
17Gf: (* Generate the User Keys on the fly *)
18Gf: let sKua = enc(Kgua, H(rUID)) in
19Gf: let sKue = enc(Kgue, H(rUID)) in
20Gf: if reqMac = enc(sKua, H((rUID, RKES_AUTH_ENC_3_DES,
21Gf:   cTime, firstNonce, ZEROS, encReqLen, encReq))) then
22Gf: (* No actual check on cTime *)
23Gf: (* firstNonce: accepted and stored as `used` *)
24Gf: let (rid, reqData) = dec(sKue, encReq) in
25Gf: event receivedReq(rid, reqData);
26Gf: (* Prepare and send response *)
27Gf: let encRes = enc(sKue, res(rid, reqData)) in
28Gf: new sTime;
29Gf: let resMac = enc(sKua, H((rUID, RKES_AUTH_ENC_3_DES,
30Gf:   sTime, firstNonce, ZEROS, len(encRes), encRes))) in
31Gf: out(c, (rUID, RKES_AUTH_ENC_3_DES, sTime, firstNonce,
32Gf:   resMac, len(encRes), encRes));
33Gf: !gatewayOther.

```

Fig. 8. Fixed gateway role model for a single valid session scenario.

can never be emitted (because the attacker cannot forge new requests from scratch). With this model, ProVerif can also prove that query (1) is true (in fact it proves the non-injective version of the property because the injective one is trivially implied in this particular scenario with a single user instance). This means that, in the modelled one-session scenario, replay attacks are not possible at the gateway, while replay attacks on the user side have already been ruled out for the most general case with possibly infinite sessions.

This proposed fix requires that the gateway stores, for each user, the pairs of used nonces-timestamps for the current TW (and constantly updates this list as new messages arrive and the TW scrolls). If this overhead is not acceptable due to the constrained resources of the gateway, sequence numbers instead of nonces can be used. In this case, the gateway need only store, for each user, the last seen sequence number. However, by re-ordering messages, the attacker may invalidate some valid requests. Suppose user U sends three requests with sequence numbers 1, 2 and 3; if the attacker delays requests 1 and 2, and immediately forwards request 3, then requests 1 and 2 will not be accepted. They would have been accepted if nonces were used. Also, sequence numbers can be suitable when a connected transport layer, such as TCP, is used.

Finally, using the timestamp field of the user directly as a nonce is not recommended, because the user’s notion of the gateway time is updated from time to time, making it possible to have two different valid messages with the same timestamp.

The full formal model of the fixed protocol, where the security properties can be rigorously proved to be true, can be found in [23].

IV. SYSTEM VULNERABILITY ANALYSIS

In this section the whole system infrastructure is taken into account, moving the analysis and the model of the system to a different level of abstraction. The focus here is on analysing the vulnerabilities of a particular network configuration. According to network topology, firewall position and configuration, weaknesses of protocols or of other software may or may not be exploitable. This can be formally analysed by the Prolog-based modelling approach and tool described in [12]. The core of the analysis mainly involves the potential actions that an attacker can perform in a system leveraging the particular configuration of the system itself. From an initial *state* of the system, the tool concatenates all the possible actions that the attacker can perform generating a set of *attack paths* which are then combined in an *attack graph*.

In this paper the analysis presented in [12] is extended and improved with the adoption of a more rigorous formalization of *states* and *transitions*, which enables the description of parts of the model as *state transition systems*.

Indeed, the whole system is modelled as a state transition system, i.e. as a potentially infinite set of different states connected through a set of different transitions. The transitions, in fact, define the possible *evolutions* of the system from a set of initial states. In this model a set of properties of interest related to each state is defined. The objective of the analysis is then to assess the reachability of particular states that satisfy or violate some property given the set of initial states and the set of possible transitions.

This reachability analysis is performed by means of an exhaustive state generation process. From the initial set of states the tool generates all possible states into which the system can evolve by triggering the proper transitions. The procedure continues by recursively generating all the possible evolutions starting from the newly generated states. The tool keeps track of the states and the transitions involved in the analysis by means of a directed graph whose nodes represent states and whose edges represent transitions. This structure allows the tool to look for all the possible paths that lead from an initial state to a state that violates a property.

Since the model is expressed in the logic programming language Prolog, the main elements of the language used are presented here. A formula like L_1 . represents a *fact* in Prolog, that is a predicate asserted to be valid. Instead, a formula like $L_0 : -L_1, \dots, L_n$. defines a *rule*. The semantics of this formula is that L_0 is true (satisfied) if all the L_i predicates with $i > 0$ are true (satisfied).

This state transition system analysis approach is applied to the system described in section II, assuming a network infrastructure organized according to the schema of figure 2.

A. System Model

The system model is composed of a set of *nodes* which are defined by some *attributes*. For instance, `field_line` is defined as a node that models a field line and that has an

attribute named `status`:

```
node(field_line).
attribute(field_line,status).
status(field_line,normal). (3)
```

Fact (3) states that the value of the attribute `status` for the node `field_line` is `normal`.

The collection of all the values of all the attributes for all the nodes defines a system state. Two states are thus equal if and only if all the nodes are the same and the values of each attribute are equal for corresponding nodes.

Some attributes are *static* while other ones are defined as *dynamic* since their values can change over time (e.g. the `status` attribute).

The interactions among elements of the system define the *transitions* of the system model. For instance, a supervisor in the field network can perform a *stop* operation on the `field_line` node in order to enable maintenance or diagnostic operations on the machines. This kind of interaction is modelled in the Prolog model by a rule

```
trans(S,S',state_change) :-
  select(status(field_line,Old),S,T), (4)
  append([status(field_line,stopped)],T,S'). (5)
```

Predicate `trans`, in this case, defines *how* and *when* a system state can change into another system state. In particular, in this case, premises (4) and (5) specify that the new state represented by variable³ S' has one single difference with respect to state S : the value of attribute `status` for node `field_line`. Predicate `select` removes a predicate from the complete definition of state S obtaining a partial definition of the state (represented by variable T). Premise (5) injects the new predicate in the partial definition obtaining a full definition in S' . In this example the value is changed from a generic `Old` value to the `stopped` value; actual transitions in the model are defined by means of more complex rules. For instance the *domain* of the values of the attributes is checked. In fact, it is assumed that each variable can be bound to values belonging to a finite set of possible values. This limits the complexity of the overall model.

B. User/Gateway Security Protocol Model

The security protocol analysed in section III is now modelled at a more abstract level. In fact, the main focus here is on the objectives of the protocol, not on the details of its exchanged messages. For example, the fact that communication can start only if the client has the proper K_{UA} and K_{UE} keys, is modelled by the following rule:

```
send(U,DP,M) :-
  regulated(DP,GW), (6)
  hasProperKeys(U,GW). (7)
```

Predicate `regulated` (6) represents the relationship between a gateway and the datapoints that it exposes. For

³In Prolog, capital letters denote variables.

instance, the following fact models that the datapoint `field_line` can be accessed only through the gateway `gw`:

```
regulated(field_line, gw).
```

For administrative messages (responsible for modification in the ACL of gateways) the scenario is similar:

```
send(U, GW, modify_access(U, DP, Permission)) :-
    regulated(DP, GW),
    hasProperGatewayKeys(U, GW). (8)
```

Premise (8) checks whether the client `U` issuing the request has the proper gateway keys needed to send such kind of messages. At the head of the rule the source of the message is `U`, the destination is `GW` and the content is a request to modify the access limitation for the pair `(U, DP)`, enabling a set of `Permission` (read, write or create).

The logic introduced by the ACL on the gateways is directly introduced in the definition of the receive rule. The send rule by which a client can send a message is defined so that a message is delivered only if the client has sufficient rights to access the destination datapoint. For instance, client user sends a write request to datapoint `field_line`. This message is delivered by the gateway and received by the datapoint only if in the ACL matrix of the proper gateway there is a pair `(user, field_line)` with write operations allowed. In the Prolog model such a scenario is represented by the rule

```
receive(DP, write(U, DP, M)) :-
    regulated(DP, GW),
    allow(GW, U, DP, write). (9)
```

Predicate `allow` (9) heavily depends on the ACL definition and it is worth noting that the ACL matrix can change by means of particular messages.

In the actual implementation of these rules `status` has been added as a key element in the validation of these rules. For instance, `regulated(DP, GW)` can depend on the current state of the system, thus `regulated(DP, GW, S)` depends on `S`, where `S` is the state of the system.

C. Case Study

A small scenario was defined involving a user and three different elements in the field network: a field line (`field_line`) representing a production line where several pieces are processed by a robotic arm (`field_roboarm`). Defective pieces on the line cause the line itself to become stuck and this triggers the intervention of a supervisor that: sets to stopped the status of `field_line` and `field_roboarm` (by writing values into the proper datapoints exposed by the gateway), modifies the configuration of the gateway in order to be able to trigger a discard operation (by writing a value into the proper datapoint) which discards the defective piece and restarts production. The following steps are involved:

- The status of `field_line` suddenly changes from normal to stuck because of a defective piece;
- The supervisor sets both `field_line` and `field_roboarm` to stopped;

```
1T: trans(S, S', change_acl) :-
2T:   send(supervisor,
3T:     modify(supervisor, gw, set(supervisor, discard, w)), S),
4T:   select( acl(gw, List), S, T1),
5T:   select( allow(supervisor, discard, _Perm), List, TList),
6T:   sort( [allow(supervisor, discard, [w])|TList], List1),
7T:   append( [acl(gw, List1)], T1, T2),
8T:   sort( [sent(supervisor, modify(supervisor, gw,
9T:     set(supervisor, discard, w))] | T2], S').
```

Fig. 9. Change ACL transition.

```
10T: trans(S, S', perform_discard) :-
11T:   send(supervisor, wr(supervisor, discard, 1), S),
12T:   receive(discard, wr(supervisor, discard, 1), S),
13T:   select(status(field_line, LS), S, T1),
14T:   select(status(field_roboarm, AS), T1, T2),
15T:   append([status(field_line, normal),
16T:     status(field_roboarm, normal)], T2, T3),
17T:   append([sent(supervisor,
18T:     wr(supervisor, discard, 1))] | T3, T4),
19T:   append([performed(discard, [LS, AS])] | T4, T5),
20T:   sort(T5, S').
```

Fig. 10. Perform discard operation.

- The supervisor sends a request to modify the ACL of gw;
- The gateway grants the supervisor's request;
- The supervisor writes the value 1 on the discard element;
- After discard, the elements involved are reset to the normal status.

It is worth noting that (in this example) the supervisor does not have complete control over the datapoints exposed by the gateway. This means that the supervisor cannot write values in some datapoints (like the datapoint for the discard operation) unless the gateway explicitly allows such a write operation. This limitation was introduced as, in some cases, there are operations that should not be always possible and that should be enabled only in particular circumstances.

The steps in the modification of the ACL are modelled by the transition described in figure 9. Lines 2T-3T involve the rule defined for the send operation. Lines 4T-7T actually modify the proper ACL list elements building the new state. Lines 8T-9T store in the new state the fact that a message has been sent.

The steps in the execution of the discard operation and in the reset of the status of `field_line` and `field_roboarm` are modelled by the transition described in figure 10. The reset of the ACL is omitted for brevity. Lines 11T-12T involve both the send rule and the receive rule. Lines 13T-16T reset the status of the involved nodes. Finally, lines 17T-19T store in the new state the message sent and the information about the discard operation performed. Lines 13T-14T get the status of the nodes involved while the discard operation is performed and store this information in the new state (line 19T).

The scenario defined here involves a change in the ACL of a gateway. This modification, however, is triggered only if the status of other two nodes is stopped. In order to model this "correct" behaviour the rule for the send operation can

be modified as follows:

```
send(U, gw, modify_access(U, discard, w), S) : –
    regulated(discard, gw),
    hasProperGatewayKeys(U, gw),
    attr_is(field_line, [status, stopped], S),
    attr_is(field_roboarm, [status, stopped], S).
```

The difference with respect to the previous definition is the presence of the last two lines, which ensures that the involved nodes have the `status` attribute set to `normal` before actually sending the request.

Once the “correct” behaviour of the agents is defined, the property that shall be verified is formulated: “a `discard` operation has to be performed if and only if the status of `field_line` and `field_roboarm` is `stopped`”.

D. Analysis

The analysis does not report any property violation on the previously described system. This means that in the abstract model the protocol and the gateway behaviours ensure the correct evolution of the system.

It is worth noting that this analysis is not meant to verify the protocol itself, rather it assumes the protocol as correct and flawless. The model of the gateway and protocol, in fact, is oversimplified, having assumed as valid the properties of the protocol described in [13]. In Section III, however, it has been shown that the protocol may suffer from the so-called “replay attack”. An attacker with enough control on the network media can resend messages that have been sent in the recent past. The attacker cannot forge new messages, though.

To take this into account the model is updated accordingly by modifying predicate `send` by adding the following rule:

```
send(U, D, M, S) : –
    member(sent(U, M), S).      (10)
```

meaning that in a given state `S` it is possible to send a message `M` if this has already been sent (a replay). It is worth noting that the destination `D` is not relevant in this rule.

The analysis is run again on the updated model in order to assess the effects of a replay attack (a flaw in the protocol) on the overall system. Assessing the effects of lower level details (the protocol) in the higher level system is a benefit of using this approach.

The analysis reports the violation of the property previously defined and produces one of the possible counter-examples:

- `trans(S, S1, state_change)`, the status of `field_line` changes from `normal` to `stuck`;
- `trans(S1, S2, state_change)`, the status of `field_line` is set to `stopped` by the supervisor;
- `trans(S2, S3, state_change)`, the status of `field_roboarm` is set to `stopped` by the supervisor;
- `trans(S3, S4, change_acl)`, the supervisor modifies the ACL of gateway `gw`;
- `trans(S4, S5, perform_discard)`, the supervisor triggers the `discard` operation;

- `trans(S5, S6, change_acl)`, the supervisor resets the ACL of gateway `gw` to the old value;
- `trans(S6, S7, state_change)`, the status of `field_roboarm` and `field_line` is set to `normal`;
- `trans(S7, S8, change_acl)`, the attacker replays the message that modifies the ACL of gateway `gw`;
- `trans(S8, S9, perform_discard)`, the attacker replays the message that performs the `discard` operation;
- *Property violation*, in fact, a `discard` operation has been performed while both `field_line` and `field_roboarm` have their status set to `normal` (and not to `stopped`).

Looking into further details of the trace it is obvious that the replay attack has caused the predicates `send` in figure 9 and in figure 10 to become valid even if the `field_line` is not in the `stopped` status. Considering the specification of the protocol and assuming that a `discard` operation is not a slow operation, this attack is feasible even within the TW of 150s.

Another possible consideration is related to the real possibility for an attacker to be able to actually store and replay messages directed to the field network. An attacker who is internal (in the Corporate Network of figure 2) can gain access to the communication medium very easily. An external attacker, for instance in the Internet Zone of figure 2, cannot directly access the Control Network but can nonetheless try to penetrate the system exploiting flaws in the configuration. As showed in [12], vulnerabilities and flaws in the configuration of network elements can become attack vectors that allow the external attacker to gain access to internal nodes.

E. Scalability considerations

The analysis performed by the tool described in this section is an exhaustive state exploration which can be affected in principle by the *state explosion problem* to the limit of having an infinite state space. However, the analyser tool has been designed carefully in order to keep the state space finite (thus ensuring termination) and to restrict the number of states generated during the analysis.

The state space is kept finite by having a finite number of possible transitions and of possible values describing the attributes of states. This implies that the states to be generated are finite and the analysis always terminates with a definite answer telling whether or not there are states that violate some property. The tool also avoids multiple generations of the same states by means of a state equivalence check that collapses equal states. In order to limit the growth of the state space when the complexity of models increases, several techniques are used. In typical scenarios a dramatic simplification in the model can be achieved by collapsing system elements that share the same configuration, and are therefore equivalent, into a single model node. Moreover, the notion of “state equivalence check” itself can be finely tuned in order to ignore irrelevant details, thus reducing the complexity of the analysis. Finally, analysis time can be saved by stopping the algorithm as soon as the first state violating a property is found. Some preliminary tests show that, in the average case, this greatly reduces execution time.

Based on these considerations, and on the fact that the tool took less than 1 second (on a Linux PC with a 3.5 GHz AMD

Athlon CPU and 2 GByte of RAM) to analyse the system presented in the paper, it can be expected that systems which are more complex in terms of number of different elements and interactions can be tackled as well.

V. CONCLUSION

This paper proposes a novel, formally-based approach to multi-level analysis of interconnected fieldbus systems, considering both low level communication protocols, and the overall system architecture. Interactions between the security services offered by the communication protocols and system behaviour are formally analysed, so that it is possible to observe the effects of potential inadequacies of the underlying security protocols on the whole system.

This proposed methodology was applied to a typical ICS using a security system for remote fieldbus access [13]. As this system was only informally specified in [13], a formalization of it has been provided. The formalization work helped to highlight possible weaknesses or ambiguities of the informal, descriptive specification, such as replay attacks possible under some assumptions. A formal model of a fixed version of the protocol was developed, on which secrecy and injective authentication can be proved.

Thanks to the multi-level analysis, the consequences of protocol attacks on the whole system can be inspected automatically. As an example, a production line where products can be discarded by an administrator only if such products are faulty, was considered. The analysis of this scenario automatically shows that if the underlying protocol is not fixed, and is therefore subject to the previously detected replay attack, then after a faulty product has been legitimately discarded by an administrator, the attacker has a time frame in which he can discard valid products. To our knowledge, this is the first work where the effects of attacks on a proprietary protocol for accessing a fieldbus gateway are formally stated and propagated to the control system that uses it by a rigorous and systematic approach.

Future work could apply the approach presented in this paper to larger systems, such as smart grids or power plants that, as they become distributed, need to be made secure. Moreover, the security protocols and the reference industrial control systems proposed by the IEC 62351 standard (not fully released yet) may be analysed too. This would also practically show the scalability of the proposed approach, because more complex security protocols and larger systems would be considered. Another extension to this work could be to go beyond the Dolev-Yao level, taking into account the computational complexity of the cryptographic operations. This would lead to a more precise security analysis, and the specific computational constraints of the network devices considered could also be taken into account.

REFERENCES

- [1] M. S. DePriest, "Network security considerations in TCP/IP-based manufacturing automation," *ISA Transactions*, vol. 36, no. 1, pp. 37–48, 1997.
- [2] R. Zurawski, Ed., *The Industrial Information Technology Handbook*. CRC Press, 2005.
- [3] V. M. Ijure, S. A. Laughter, and R. D. Williams, "Security issues in SCADA networks," *Computers & Security*, vol. 25, no. 7, pp. 498–506, 2006.
- [4] M. Brändle and M. Naedele, "Security for process control systems: An overview," *IEEE Security and Privacy*, vol. 6, no. 6, pp. 24–29, 2008.
- [5] A. Miller, "Trends in process control systems security," *IEEE Security and Privacy*, vol. 3, no. 5, pp. 57–60, 2005.
- [6] P. Ralston, J. Graham, and J. Hieb, "Cyber security risk assessment for SCADA and DCS networks," *ISA Transactions*, vol. 46, no. 4, pp. 583–594, 2007.
- [7] L. Piètre-Cambacédès and P. Sitbon, "Cryptographic key management for SCADA systems – issues and perspectives," in *International Conference on Information Security and Assurance*, 2008, pp. 156–161.
- [8] NIST, "SP 800-82: Guide to industrial control systems (ICS) security," September 2008, final Public Draft.
- [9] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *IEEE Symposium on Security and Privacy*, 2009, pp. 16–26.
- [10] V. Klíma, O. Pokorný, and T. Rosa, "Attacking RSA-based sessions in SSL/TLS," in *Cryptographic Hardware and Embedded Systems*, 2003, pp. 426–440.
- [11] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *Computer Security Foundations Workshop*, 2001, pp. 82–96.
- [12] M. Cheminod, I. Bertolotti, L. Durante, P. Maggi, D. Pozza, R. Sisto, and A. Valenzano, "Detecting chains of vulnerabilities in industrial networks," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 2, pp. 181–193, 2009.
- [13] T. Sauter and C. Schwaiger, "Achievement of secure Internet access to fieldbus systems," *Microprocessors and Microsystems*, vol. 26, no. 7, pp. 331–339, 2002.
- [14] J. Edmonds, M. Papa, and S. Sheno, "Security analysis of multilayer SCADA protocols," in *Critical Infrastructure Protection*, 2007, pp. 205–221.
- [15] B. Dutertre, "Formal modeling and analysis of the Modbus protocol," in *Critical Infrastructure Protection*, 2007, pp. 189–204.
- [16] AGA, "Report no.12, part 1: Cryptographic protection of SCADA communications: Background, policies and test plan," March 2006.
- [17] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," *ACM Special Interest Group on Programming Languages*, vol. 36, no. 3, pp. 104–115, 2001.
- [18] D. Dolev and A. C.-C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–207, 1983.
- [19] N. A. Durgin, P. Lincoln, and J. C. Mitchell, "Multiset rewriting and the complexity of bounded security protocols," *Journal of Computer Security*, vol. 12, no. 2, pp. 247–311, 2004.
- [20] K. Bhargavan, C. Fournet, and A. D. Gordon, "Verified reference implementations of WS-security protocols," in *Web Services and Formal Methods*, 2006, pp. 88–106.
- [21] M. Abadi, B. Blanchet, and C. Fournet, "Just Fast Keying in the Pi Calculus," in *European Symposium on Programming*, 2004, pp. 340–354.
- [22] A. Treytl, T. Sauter, and C. Schwaiger, "Security measures for industrial fieldbus systems - state of the art and solutions for IP-based approaches," in *International Workshop on Factory Communication Systems*, 2004, pp. 201–209.
- [23] M. Cheminod, A. Pironti, and R. Sisto, "Online resources about formal analysis of fieldbus systems," 2010, available at: <http://staff.polito.it/riccardo.sisto/fieldbus/>.
- [24] G. Lowe, "A hierarchy of authentication specifications," in *Computer Security Foundations Workshop*, 1997, pp. 31–43.
- [25] M. Abadi and B. Blanchet, "Computer-Assisted Verification of a Protocol for Certified Email," *Science of Computer Programming*, vol. 58, no. 1–2, pp. 3–27, 2005.



Manuel Cheminod received the M.S. and Ph.D. degrees in computer engineering from Politecnico di Torino, Torino, Italy, in 2005 and 2010 respectively.

He is now working with the Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni (IEIIT). His current research interests include formal verification of cryptographic protocols and formal methods applied to vulnerability and dependability analysis in distributed networks.



Alfredo Pironti received the M.S. and Ph.D. degrees in computer engineering from Politecnico di Torino, Torino, Italy, in 2006 and 2010 respectively.

He is currently a Post Doctoral Researcher at Politecnico di Torino, where he is also Teaching Assistant for undergraduate courses. His main research interests are on formal methods applied to security protocols and security-aware applications, as well as software engineering and model driven development. During winter of 2008, he was a visiting Ph.D. student at Open University and Microsoft Research in

Cambridge, UK.



Riccardo Sisto received the M.S. degree in electronic engineering in 1987, and the Ph.D. degree in computer engineering in 1992, both from Politecnico di Torino, Torino, Italy.

Since 1991 he has been working at Politecnico di Torino, in the Computer Engineering Department, first as a researcher, then as an associate professor and, since 2004, as a full professor of computer engineering. He teaches introductory courses on programming and undergraduate and graduate courses on network and distributed programming. Since the

beginning of his scientific activity, his main research interests have been in the area of formal methods, applied to software engineering, communication protocol engineering, distributed systems, and computer security. On this and related topics he has authored and co-authored more than 70 scientific papers.

Dr. Sisto has been a member of the Association for Computing Machinery (ACM) since 1999.