The Journal of Systems and Software 89 (2014) 87–98

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A formal methodology for integral security design and verification of network protocols

Jesus Diaz*, David Arroyo, Francisco B. Rodriguez

Grupo de Neurocomputación Biológica, Departamento de Ingeniería Informática, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain

ARTICLE INFO

Article history:
Received 8 October 2012
Received in revised form 9 May 2013
Accepted 15 September 2013
Available online 8 October 2013

Keywords:
Secure development methodology
Formal model
Protocols security
Formal verification
WEP analysis

ABSTRACT

In this work we propose a methodology for incorporating the verification of the security properties of network protocols as a fundamental component of their design. This methodology can be separated in two main parts: context and requirements analysis along with its informal verification; and formal representation of protocols and the corresponding procedural verification. Although the procedural verification phase does not require any specific tool or approach, automated tools for model checking and/or theorem proving offer a good trade-off between effort and results. In general, any security protocol design methodology should be an iterative process addressing in each step critical contexts of increasing complexity as result of the considered protocol goals and the underlying threats. The effort required for detecting flaws is proportional to the complexity of the critical context under evaluation, and thus our methodology avoids wasting valuable system resources by analyzing simple flaws in the first stages of the design process. In this work we provide a methodology in coherence with the step-by-step goals definition and threat analysis using informal and formal procedures, being our main concern to highlight the adequacy of such a methodology for promoting trust in the accordingly implemented communication protocols. Our proposal is illustrated by its application to three communication protocols: MANA III, WEP's Shared Key Authentication and CHAT-SRP.

1. Introduction

The application of formal methods for the verification of security properties has received increased attention lately (see e.g. Weldemariam et al., 2011; Mohammad and Alagar, 2011). Additionally, there already exist plenty of automatic tools for the formal analysis of those security requirements (Blanchet, 2010; Andrew and Gordon, 2002). Several methodologies also endorse the application of these tools and theories in order to reach high security guarantees for critical systems and protocols (Hernan et al., 2006; Technical Report, 2009; Matsuo et al., 2010). However, a direct application of these tools may be very resource-consuming during the first stages of the design, since many flaws could be detected just by performing an informal analysis. We propose a methodology for the design of secure protocols that covers the whole design process and tackles the mentioned problems by following an iterative approach where the first steps are less resource consuming than the last ones. Consequently, the complexity of the detected flaws increases as we advance through the methodology steps. Also, special emphasis is made into creating a correct abstraction of the protocol environment and the attacker model. In this vein, the proposed methodology allows a finer control on the attacker modelization in order to fit it to the specific context in which the protocol will be executed. Like we will see, an active application of our methodology helps to reduce the impact of illegitimate actions over the resulting network protocols. In the examples shown here, we make use of ProVerif (Blanchet, 2010). In order to prove the queried security properties, ProVerif tries to automatically deduce them from the set of rules used to formalize the protocol. However, even though we have used ProVerif, any protocol verifier tool or method can be applied without affecting the other parts of our methodology.

The paper is structured as follows. We start in Section 2 with a brief introduction to the existing frameworks and procedures for the verification of security properties, along with a discussion on the advantages of applying the verification procedures at the different stages of the software life cycle. In Section 3 we introduce our methodology. In Section 4 we apply it to three different protocols (MANA III, WEP-SKA and CHAT-SRP). The three cases have been chosen because they are publicly available examples of protocols failing at some point of our methodology (for the first and second cases), allowing us to show its usefulness, while the third protocol has been explicitly created using our methodology, and successfully passes all its tests. Finally, we conclude this work in

* Corresponding author.
E-mail address: j.diaz@uam.es (J. Diaz).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.09.020

Section 5, with a global perspective of the benefits provided by our methodology.

2. Related work

According to the standard ISO27001, the deployment of any information system cannot be interpreted as a product. Certainly, the real implantation of an Information Security Management System (ISMS) is a process following a *Plan-Do-Check-Act* methodology: (1) design of an ISMS, (2) consequent system implementation, (3) resulting product monitoring, (4) maintenance and improvement of the ISMS and (5) eventual re-design of the system to overcome problems not included in the original solution but detected during production. This adaptive procedure should be applied for the definition of any component of an information security system. Thus, in this work we apply it to the design of cryptographic protocols.

On the grounds of the above introduced *Plan-Do-Check-Act* methodology, when creating secure cryptographic protocols we require a framework for assessing the fulfillment of the assumptions made at the design stage, a procedure for evaluating the goodness of the implemented product, and a model for identifying possible problems or security threats. Regarding the evaluation tasks, there are two main types of frameworks aimed to the task of creating secure systems and protocols: frameworks applied at the design phase (Hernan et al., 2006; Technical Report, 2009; Jurjens, 2003; Matsuo et al., 2010), and frameworks applied at the development stage (Swigart and Campbell, 2008; Bhargavan et al., 2010). In turn, the former sometimes rely on tools for automated reasoning specialized in the verification of security properties (Blanchet, 2010; Blanchet and Cadé, 2012; Barthe et al., 2009; Andrew and Gordon, 2002; di Genova et al., 2006; Paulson et al., 2012). Similarly, the latter are usually based on tools crafted for specific programming languages (Goubault-Larrecq and Parrennes, 2005; Chaki and Datta, 2009; Bhargavan et al., 2010; Aizatulin et al., 2011b,a; Bengtson et al., 2011). Although in some proposals only one type of framework is used, it should be noted that both of them should be applied if the goal is the *complete* identification and analysis of security requirements and assumptions. Imagine that we choose to apply a verification tool only to the obtained system implementation, and we indeed find security flaws. It may happen that, while trying to fix them, we realize that one (or several) of them is not just a coding flaw, but a design flaw. That means that we must go back to the design phase and fix the flaw at that stage. Moreover, if the flaw is serious enough, the existing implementation may need extensive changes, which will incur in unacceptable costs and delays. Yet another disadvantage is that code verification obviously relies on the existence of a tool for verifying the specific programming language that has been used. Although these tools are gaining popularity, the huge number of existing programming languages, along with the complexity of creating such a tool, suggests that in the real world we should not trust in having always a code verification tool suitable to our needs. On the other hand, applying a verification tool only to the design does not guarantee that the implementation of that design will also be secure even though the design seems to be so. However, it is technology independent. Thus, the verification of the security properties of both the design and implementation products should be applied when possible.

One major issue when creating security systems is the definition of the security requirements that are expected to be fulfilled (rather than what actions should and should not be executed Brown, 2013). In turn, the *attacker model* has direct influence when it comes to prove if the final system is compliant with those security requirements. This model defines the attacker capabilities that are necessary to conduct the *threat analysis* (Anderson, 2008, Chapter 11). Indeed this is of the utmost importance, since depending on what the attacker can do, a designer/developer may need to protect different resources or take one approach or another. There exist several classifications. For instance, an attacker can be said to be *internal* or *external*, depending on whether it is one of the entities which take part in the protocol/system, or a third party that is not included in it. An attacker can also be categorized as *passive* if the related attacks consist in observing the messages and a subsequent information inference, or as *active* if he/she actually inserts and/or modifies information on any communication link. On the other hand, there are *local adversaries* only threatening a subset of an information/communication system elements, versus *global adversaries* that can access every component of a system (Díaz et al., 2002). Major threats are determined by the so-called *Byzantine adversary*, most commonly used as reference when designing fault tolerant systems, and in the context of anonymous communication systems (Digital Privacy: Theory, Technologies, and Practices, 2007, p. 78). This kind of adversary is an internal attacker that behaves randomly in order to corrupt the system output (Lamport et al., 1982).

Additionally, the attacker model can be further refined by considering the attacker computing capabilities. One main approach is the Dolev-Yao attacker model (Dolev and Yao, 1983), which has proven to be a very powerful abstraction when used in conjunction with methodologies for the formal verification of protocols (Abadi and Rogaway, 2002; Kemmerer, 1987; Paulson, 1998; Houmani et al., 2009). It assumes an *omniscient* attacker who monitors and can modify the messages sent through all communications channels, but cannot break cryptographic primitives. In contrast with this attacker model, there is the computational attacker model (Bellare et al., 2000), which assumes that attackers are Probabilistic Polynomial Time Turing Machines (PPTTM) and hence their computing capabilities are consequently determined. This implies that the cryptographic primitives are not assumed to be perfect and thus any *breakable* primitive in a PPTTM scenario is vulnerable.

As main conclusion of this section we should recall the needs for an adaptive and feedback process when designing any information security protocol. Any possible methodology should be built upon evaluation tools for the goals and assumptions involved both in the design and the implementation stages, and any threat identified during implementation or production should be handled as feedback to further improve the system design. In fact, the more detailed attacker capabilities that are applied within the development level frameworks are not considered in their design level counterparts, and the more general models are ignored during development. This makes sense up to a certain point, since during the design phase the knowledge of the final system is not as deep as it is during its development. Hence, threats that are clear in the implementation stage can be quite obscure in the design process. However, there may be exceptions that could and should be taken into account during the design. For instance, the typical attacker models do not take into account pieces of information that can be easily accessible to an attacker by alternative means, like Facebook pages, and public databases. Although these are very specific facts, the effects that they could have on the ISMS performance can already be detected and considered at the design stage. Consequently, our proposal includes a finer control on the attacker capabilities during the design of the protocol, and sets explicit control points that provide feedback when security errors are found, easing their correction in subsequent iterations. Also, in order to reach the highest assurance levels of other methodologies, it includes the application of procedural (either formal or computational) verification methods.

3. The methodology

Our methodology consists of two main parts, like shown in Fig. 1. The first part includes the specification of the security aims/requirements of the protocol, and the informal evaluation of

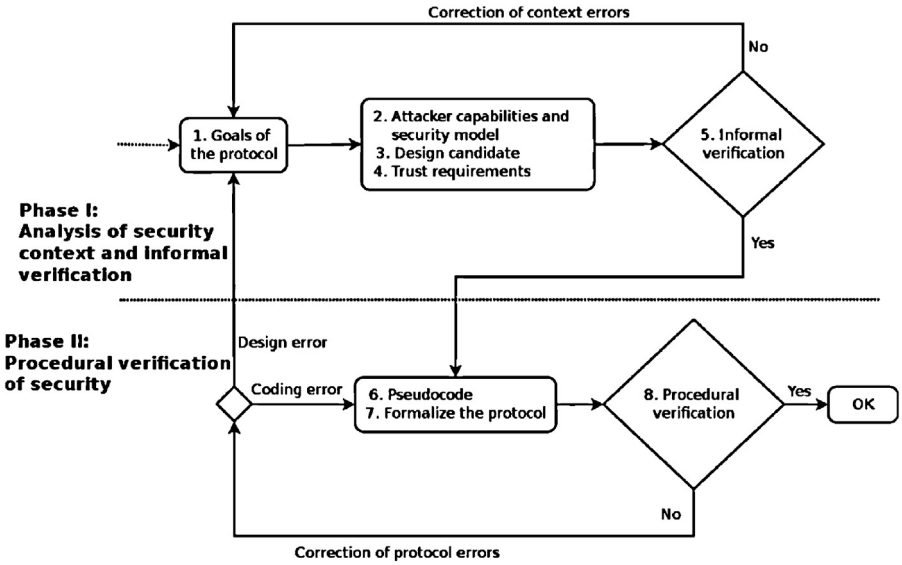

Fig. 1. Steps of the proposed methodology. The first phase performs an initial informal verification, establishing the protocol goals and requirements. The second phase performs a procedural verification of the security requirements.

the suitability of these goals regarding the protocol application context; the second part is centered in the formal definition of the protocol and its further verification.

3.1. Phase I: analysis of security context and informal verification

This first part is mainly intended to avoid incongruences related to security requirements that are unachievable or inappropriate for some reason. It also allows us to perform several informal checks to our first design candidates, and improves our understanding of the protocol. This part is depicted in the first phase of Fig. 1. Namely, here we face the following matters:

Step 1. Goals of the protocol. Within this step we have to create an informal specification of what we intend to achieve with our protocol, something like *"Complete certainty that the person being registered in our website is who he/she says he/she is"*. This step might seem too obvious to be needed, but specifying the security expectations for the protocol will help to improve its understanding and will be useful for the third and fourth steps.

Step 2. Attacker capabilities and security model. This step is essential, because depending on the capabilities the attacker has, some security requirements might or might not be attainable. Moreover, the type of attacker we choose will determine what tools or methods we are going to need in the next steps. First we have to take into account all the different *general capabilities* that our attackers will have. Following the summary in Section 2, we have to determine whether we are only worried about internal/external attackers, Dolev-Yao/computational models, etc. Once we have established these general capabilities, we have to add or remove any finer level capability that may influence in the system. As a result, with this step we will get a specification of the attacker model, e.g. the Dolev-Yao or computational model, along with a list of specific capabilities added or subtracted to it, like *"Knows the home address of everyone that may be involved in the protocol"* or *"Cannot eavesdrop wired communications"*. Henceforth, we will refer to this list as the *"+/−capabilities"* list.

Finally, note that in case we are not in the first iteration of the methodology, the output of both steps 5 and 8 may be used as input to this step. Indeed, verification failures might cause us to reconsider the attacker model, in case it was not realistic (either too optimistic or too pessimistic).

Step 3. Design candidate. Once we have pointed out the goals of our protocol, the attacker capabilities and security model we assume, we can cast a design candidate. For communications protocols, a sequence diagram is most commonly used, since it captures both the order of the messages composing the protocol and their contents. Therefore, as output of this step we will obtain a sequence diagram depicting the protocol.

Like in step 2, for iterations of the methodology other than the first, we should consider here the informal or formal requirements that failed in the previous iteration, in order to propose a solution to the encountered problems.

Step 4. Trust requirements. This step of the methodology is the first part of an informal verification procedure. Besides helping us to check the security requirements, it is also intended to prevent us from wasting our efforts needlessly. Before undertaking the time consuming formal verification task, it is worth making a simpler manual verification in order to detect obvious mistakes. For that purpose, we analyze the design candidate thoroughly by assigning *trust properties* to each component of the sequence diagram obtained in the previous step. Note that we use the term *trust property* rather than *security property* because at this point, these properties are something we expect the elements to keep and not something we state that they have. The procedure is as follows: for each step of the sequence diagram we note down the *trust properties* we expect from each of its elements to keep. If within a specific step we apply some function to merge/disassemble several elements, we may have to require additional *trust properties* (e.g., a message containing the a priori unauthenticated elements x_1, x_2 along with $MAC_K(x_1, x_2)$ can be considered authenticated by the holder of a key K, if the MAC verification succeeds provided that the key K is trusted).[1] Since the properties we require from a specific element may change

[1] Typically, this kind of rules are formally specified inside formal verification tools (e.g., Cryptyc Andrew and Gordon, 2002 is based on type deduction rules). However,

through the protocol, we will revise them for every step, even if the element has already been processed previously. This process is summarized in Algorithm 1. Some of the requirements that we may need to apply to the different elements are[2]:

None. When a field does not require any specific trust property.

Authenticity. When the authenticity of the field must be guaranteed.

Confidentiality. When the specific field must be kept secret.

Integrity. When the integrity of the field must be preserved by detecting any undesired modifications.

Uniqueness. When an element cannot be repeated among the same or different protocol runs. This property is commonly used to guarantee freshness and avoid replay attacks (Boyd and Mathuria, 2003, Chapter 1).

Algorithm 1. Algorithm for assigning requirements to protocol elements and messages. With elements we refer to any component calculated or sent within each specific step.

```
Data: The sequence diagram of the protocol.
Result: A set of trust requirements for each protocol step.
for s = first step; until s = last step do
  for e = first until e = last element of step s do
    Set reqs [s][e];
  end
  reqs[s] += Additional trust requirements for step s;
end
```

Therefore, as output of this step we get a list of *trust requirements*. Optionally, this list can be depicted jointly with the sequence diagram to create a requirement-tagged sequence diagram of our protocol design candidate.

Besides, we also have to decide in this step which verification tool or process we will apply in the second phase. Now that we know our specific *trust requirements*, we can choose an appropriate tool (depending on the security properties we want to verify).

Again, if this is not the first iteration of the methodology, we would be required to reconsider the points in the design where our (informal or formal) model of the protocol failed.

Step 5. Informal verification. Now we have to process the *+/−capabilities list* and the output of step 4. If any of the *trust requirements* enters in conflict with any of the capabilities we granted to the attacker, then we have a security failure and we need to redesign our protocol. Otherwise, we have informally verified the design candidate and we can continue. However, this does not guarantee that the design will also pass the next phase of the methodology. In case we find a failure, we will feed back a list of the *trust requirements* that have failed, in order to reconsider them in the next iteration of the methodology.

After applying our methodology up to this point, and assuming we passed the step 5, we would have reached the assurance level PAL1 defined in (Matsuo et al., 2010).

since this step is intended to be a preliminary informal approach, common sense and experience are enough.

[2] Although we propose these five requirements, more could be defined if necessary.

3.2. Phase II: procedural verification of security

This part of the methodology is devoted to the procedural verification of the security requirements established before. It is depicted in the second phase of Fig. 1. With procedural verification we refer to the fact that widely approved theories, methods or procedures should be applied here. Again, we can apply either the formal or the computational model. The main advantage of the formal methods is that there are plenty of tools that automatically analyze a protocol formalization. On the other hand, the computational model verification takes into account that the cryptographic primitives may not be perfect. However, in the last years there has been a lot of work to prove that formal security implies computational security (Abadi and Rogaway, 2002; Janvier et al., 2004) when the cryptographic primitives meet some properties. We do not enter in this matter here, and just assume that a "procedural" verification model has been adopted and is going to be applied. Nevertheless, we have found that the application of automatic tools (like ProVerif Blanchet, 2010 and Cryptyc Andrew and Gordon, 2002) allows to detect many flaws with little effort, so it may be a good choice to first apply formal methods, and then use the computational model for a more concrete evaluation.

Therefore, the steps we have included in this phase are as follows:

Step 6. Protocol pseudocode. In the same way that writing pseudocode is useful before coding a program, writing an informal narration of a protocol in the shape of pseudocode helps to reach a higher concretion level before properly formalizing it. As output of this step we get a written representation of the sequence diagram, with one process for each principal, which depicts the internal computations performed by each of them in order to generate the messages components along with the messages sent to the other principals.

Note that coding errors detected in step 8 might cause to come back to this step and modify the protocol's pseudocode (which will in turn induce changes in the code produced in the next steps).

Step 7. Formalization of the protocol. From the protocol pseudocode it is typically easy to produce a formalization in the language or definition model required by the chosen tools for verifying the protocol (the ones we decided to use in step 4). The result of this step must be a meticulous representation of the protocol after implementation.

Step 8. Procedural verification. Using the formalization obtained in the previous step, we use it as input for the chosen procedural verification model or tool. Additionally, depending on the tool that we have chosen, we will need to determine which verifications we want to perform. For this purpose, we use the *trust requirements* produced at step 4. Specifically, in Algorithm 1 for each *e* subindex within the variable *req* we look for the last sub-step in which it appears (the one with greater *s* subindex). The value in *req*[*s*][*e*] contains the formal security properties that we need to verify for the protocol element *e* (note that there may be elements with no requirements). Additionally, if the verification tool supports it, we can convert the *trust properties* that are required to each element when it first appears in the *req* variable as a condition that needs to be ensured when that element is created in our formalization of the protocol. Note that, even though the specific tool we use may not support this type of check, the system designers can always forward these requirements to the development team. That is, if the development team receives a requirement for an element *t* stating that it needs to be random and authenticated, they

will easily deduce that they need to fetch it from a randomness source and digitally sign it. The process for obtaining the formal security requirements to be verified in this last step is shown in Algorithm 2. The properties required to each element when it is created can be obtained using a similar algorithm, but going through the *trust requirements* in increasing order.

Algorithm 2. Algorithm for assigning the final security requirements to be verified during the formal verification of the protocol.

```
Data: The trust requirements of Step 4.
Result: Formal security requirements.
forall the e in reqs do
  sreqs[e] = NULL;
  s = last step;
  While s > = first step AND sreqs[e] == NULL do
    If sreqs[s][e] not empty then
      sreqs [e] = reqs [s][e];
    else
      s = s - 1;
    end
  end
end
```

Finally, if the tools or procedures followed in this step "output" that all requirements are fulfilled, we can conclude. Otherwise, we have to go back to the first step and correct errors in our design, checking also for errors in the formalization. Like with the informal verification, if we find a security failure, we will feed back a list of the security properties that have failed, in order to reconsider them in the next iteration of the methodology.

Note that for iterations of the methodology that are not the first one, all the information that has been fed back as a result of detecting a failure has already been processed during the first phase of the methodology. However, there is an exception, namely, when the failure was due to a coding error in the protocol formalization. In that case, the protocol designer would need to go back to the 6th step and revise the code, not affecting the previous steps of the methodology.

As final comments on the methodology, we must point out that, by itself, the methodology does not give as output an explicit measure of security of the analyzed protocol other than checking whether the specified security requirements are held. That will depend on the tools or the models used to verify the protocol. In any case, we will obtain an answer to whether or not the protocol meets the requirements specified in step 4 of our methodology. According to (Matsuo et al., 2010), any protocol successfully verified using our methodology would reach an assurance level equivalent to PAL2 or PAL3, depending on whether the used tool provides bounded or unbounded verification. Moreover, even the PAL4 level, still under consideration in Matsuo et al. (2010), can be achieved if we use a computational model verification tool or method. Also, it is important to emphasize that, for protocols where several different sequences are possible the methodology should be applied separately to each one of them. This includes error sequences, where the security properties of each element should be kept up to the point where the protocol has been executed. Note however that, if two different sequences share the same *sub-sequence* up to a certain point, and we have already verified that *sub-sequence*, it can be safely skipped. This may indeed be time-saving for the informal verification performed manually during the first phase of the methodology. Possibly, a good option (if the alternative sequences do not include additional factors, like new communication channels or special tokens) could be to only verify manually and formally the longest sequence. Once available its formalization, it would probably be easy to derive from it the alternative flows and formally verify them.

Table 1
Definition of the different outputs of the methodology.

O1:	Informal list of goals
O2:	Security model and *+/−capabilities list*
O3:	Sequence diagram
O4:	List of trust requirements/requirement-tagged sequence diagram
O5:	List of failed trust requirements or *Success*
O6:	Pseudocode
O7:	Formalization
O8:	List of failed security properties or *Success*

To summarize, the inputs and outputs of each of the different steps of our methodology are shown in Tables 2 and 3, corresponding to the first and second phases of the methodology. The acronyms used in the tables for the different outputs are explained in Table 1.

4. Case studies of real security protocols

In this section we provide three case studies by applying our methodology to real security protocols. For the first case, a problem is located in the first phase of the methodology, while in the second case a problem is detected with a formal analysis in the second phase.[3] On the other hand, in the third case we go through the whole methodology to verify one more protocol (Diaz et al., 2011, 2012) which successfully passes our tests. For the sake of brevity, we do not analyze different sequences of each protocol, since the process would be similar to the ones shown in the examples.

4.1. Case study I: context verification failure

Here we apply the first part of the methodology to the MANA III (MANual Authentication) authentication protocol (Gehrmann et al., 2004). MANA III is intended for wireless networks and bases its security in the requirement of manually introducing a short bitstring (R in Fig. 2) via keypads, previous to the execution, which is kept secret. Since this short secret bitstring is used in the calculation of MACs, an attacker will not be able to create fake MACs for authentication in real time. The MANA III protocol is informally depicted in Fig. 2.

In Wong and Stajano (2005) the authors suggest that this authentication method is no longer secure due to the proliferation of CCTV cameras. Indeed, such cameras can be used to observe the short bitstring introduced via the keypad. If the short secret random bitstring becomes known to the attacker, she could use it to mount a Man In The Middle attack (Wong and Stajano, 2005). Let us now see how this fact could have been detected with the first part of our methodology.

Step 1. Goals of the protocol. We want to achieve a final state in which Device A and Device B are certain that they are communicating with each other. That is directly translated to authenticity. Thus, our specific goals are:

1. Device A is successfully authenticated against Device B.
2. Device B is successfully authenticated against Device A.

Step 2. Attacker capabilities and security model. Let us assume that we adopt the Dolev-Yao attacker model (nevertheless, this decision does not affect what follows). As we said in

[3] It is not our purpose to present new flaws here. We rather intend to show how to avoid security design flaws using our methodology.

Table 2
Inputs and outputs of each step of the first phase of the methodology.

	Goals	Attacker model and capabilities	Design candidate	Trust requirements	Informal verification
Input	None	[O5 or O8][a]	[O5 or O8][a] and O1	[O5 or O8][a] and O3	O2 and O4
Output	O1	O2	O3	O4	O5

[a] Only when we are not in the first iteration.

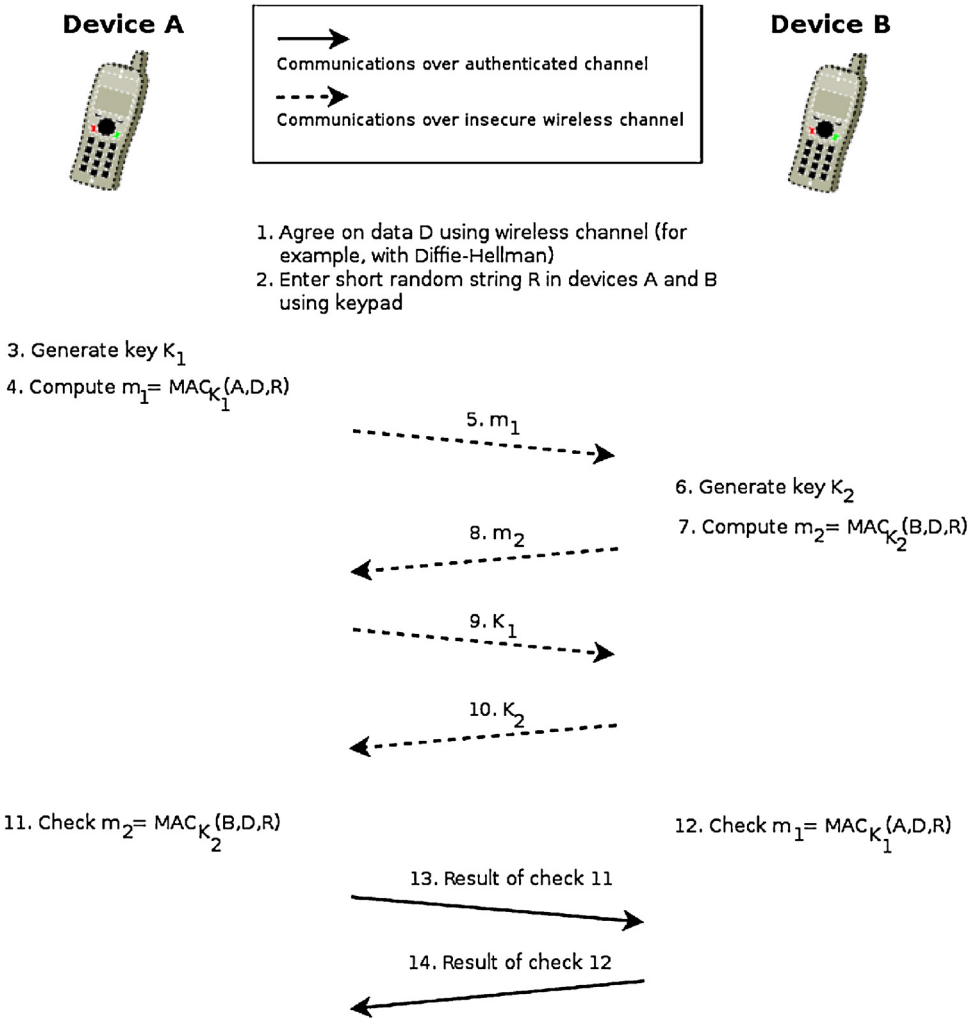

Fig. 2. Message sequence of the MANA III protocol (Gehrmann et al., 2004). (1) Devices A and B agree on some data D. (2) Enter shared random string via keypad. (3)–(8) A and B generate MAC keys and exchange keyed MACs. (9) and (10) Both devices exchange the keys used in the previous MACs. (11)–(14) Both devices check if the received MACs are correct, and inform the other device of the obtained result. Messages 5,8,9 and 10 are sent over insecure channels. Messages 13 and 14 are sent over an authenticated channel.

```
+ May observe keystrokes.
− Cannot find MAC collisions in real-time without R.
```

Listing 1. *+/−capabilities list* for the Dolev-Yao attacker for the MANA III protocol.

Section 3.1, we have to analyze our specific context to see if our attacker has some extra capabilities (or if we have to take out some capability from him). Now, according to Wong and Stajano (2005) given the current state of development of CCTV cameras, it is not safe to assume in MANA III that, in every situation, a password introduced via a keypad will always remain secret. Therefore, the more realistic decision of giving the attacker the capability to observe keyed passwords seems to be justified.[4] Besides, we will trust that the assumption made by the authors of the protocol stating that the short random string R prevents attackers

[4] However, this may be a too restrictive scenario for many situations, but for illustration purposes we will assume that we intend to use MANA III in a security critical context.

Initialization. A and B are publicly known values.
Step 1. D : Authenticity
Step 2. R : Confidentiality
Step 3. K_1 : Confidentiality
Step 4. $m_1 = MAC_{K_1}(A, D, R)$: Authenticity
(K_1 is trusted by A since she has created it.)
Step 5. m_1 : None
(At this point, no one knows K_1 and m_1 is sent over the public channel, so we cannot place any trust in this element.)
Step 6. K_2 : Confidentiality
Step 7. $m_2 = MAC_{K_2}(B, D, R)$: Authenticity
(K_2 is trusted by B since she has created it.)
Step 8. m_1 : None
(At this point, no one knows K_2 and m_2 is sent over the public channel, so we cannot place any trust in this element.)
Step 9. K_1 : None
(The used channel is insecure, anyone can send this.)
Step 10. K_2 : None
(The used channel is insecure, anyone can send this.)
Step 11. Nothing to do here.
Step 12. Nothing to do here.
Step 13. $check_{m_2}$: Authenticity.
Step 14. $check_{m_1}$: Authenticity.

Listing 2. List of trust requirements for MANA III.

Table 3
Inputs and outputs of each step of the second phase of the methodology.

	Pseudocode	Formalization	Procedural verification
Input:	O8[a] and O4	O2 and O5	O4 and O6
Output:	O6	O7	O8

[a] Only when we are not in the first iteration.

from finding, in real time, collisions for the calculated MACs is true. Summarizing, we use the Dolev-Yao model, and our *+/−capability list* will be as shown in Listing 1.

Step 3. Design candidate. From the goals we defined in the first step, we come up with a design candidate in the shape of a sequence diagram. Suppose that we produce the diagram in Fig. 2.[5]

Step 4. Trust requirements. With the sequence diagram in front of us, we apply Algorithm 1. As a result, we obtain the list of *trust requirements* shown in Listing 2 (we omit the complete process for brevity).

Step 5. Informal verification. Going through the list of *trust requirements* shown in Listing 2 we soon find an incompatibility with the attacker capabilities: in the second step, we see that we require R to be secret (confidentiality). Still, we gave the attacker the capability to observe keystrokes, and since R is entered in Devices A and B via a keypad, we cannot consider it as secret.

We should see now the importance of this part of the methodology, and why it has to be applied with care. Although the standard attacker models provide a powerful framework to start with, there are cases where these models do not cover all the possibilities that the attacker may have available. Therefore, in order to detect possible flaws on the context of the protocol we have to carefully establish the protocol goals, analyze the attacker capabilities given the current technologies, give a formal statement of the required security properties, and check if they hold given the previous facts. Once we successfully complete this phase, we can proceed with the second phase of the methodology, with a higher degree of certainty that we have correctly established the context of our protocol.

4.2. Case study II: procedural security verification failure

We dedicate this subsection to the procedural analysis of the Shared Key Authentication of the WEP standard (WEP-SKA from now on), as defined in IEEE (1999). Let us assume that the protocol successfully passes the first phase of our methodology. In this case, we apply the automatic formal verifier ProVerif (Blanchet, 2010).

The WEP-SKA protocol is one of the two authentication methods supported by the WEP standard. A normal execution consists of four messages between two stations: the Wireless Device (WD), and the Access Point (AP). In a typical protocol run the WD is authenticated by the AP as it is depicted in Fig. 3.

Nevertheless, as pointed out in the WEP standard (IEEE, 1999) sending both the challenge and its encrypted version may produce a security problem. This is due to the fact that WEP uses an encryption algorithm (the RC4 cryptosystem) that is a stream cipher which generates a pseudorandom sequence and XORs it to the plaintext in order to create the ciphertext. Therefore, if we know the ciphertext and the plaintext, we can obtain the keystream straightaway. However, in the standard it was only advised (but not required) to change the key and/or IV (Initialization Vector) frequently. As observed in Borisov et al. (2001), the fact of not being required to change the key/IV indirectly forces every receiver to accept repeated key/IV's, or risk otherwise not being compatible with some WEP compliant devices. That allows an attacker to successfully impersonate any station after having observed one single

[5] Obviously, in a real scenario we will produce it at this step and not before. We just showed the diagram in advance for introducing the case study.

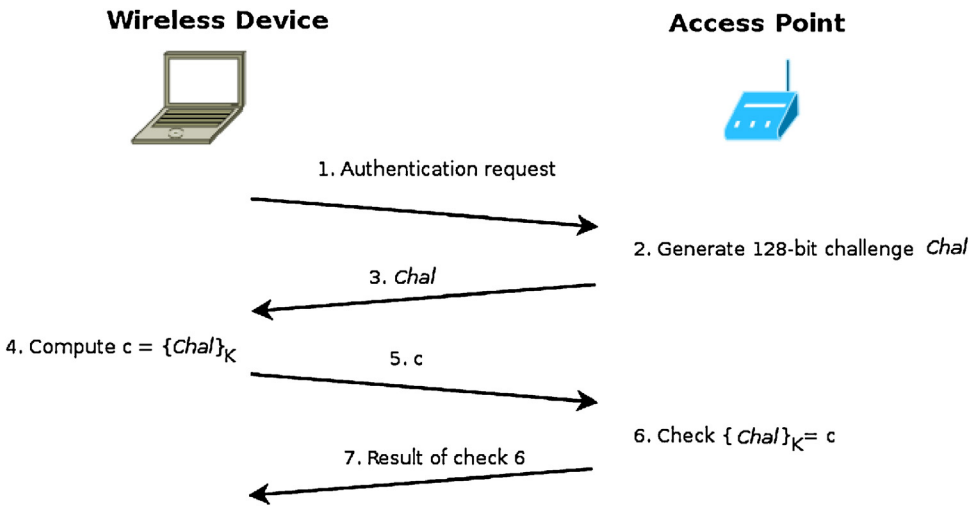

Fig. 3. Message sequence of the WEP-SKA protocol (IEEE, 1999). (1) The WD requests to be authenticated using WEP-SKA. (2) and (3) The AP generates a challenge and sends it, in plaintext, to WD. (4) and (5) The WD encrypts the challenge with a preshared key, and sends the result to the AP. (6) and (7) The AP checks the received encrypted challenge and informs WD of the result.

authentication. In Borisov et al. (2001) they call this attack *Authentication spoofing*.

Let us now forget for a moment that this flaw is known, suppose that we are asked to verify the security of WEP-SKA, and that it successfully passes the first part of our methodology. Since we will need it in order to include it as a query for the formal verification, assume also that we have reached the conclusion that the *trust requirement* (produced as output of step 4) for the element *c* shown in Fig. 3 is *authenticity* (which is inherited from the fact that the key *K* used to obtain *c* is a shared secret). Then, we would have to face the procedural verification of the protocol. The steps defined in Section 3.2 are as follows:

Step 6. Protocol pseudocode. The pseudocode for the WEP-SKA protocol is given in Listing 3.

Step 7. Formalization of the protocol. ProVerif has a problem here, because it does not support the XOR operation[6]. Nevertheless, since we do not need to apply complex derivation rules, we can get around this problem with a simple reduction rule simulating the XOR. This workaround can be seen in the code of the program available in Diaz et al. (2013). It is a rule stating that if the attacker knows both the challenge *c* and its encryption (through an XOR) with the key *k*, then she can apply the XOR function to recover the key *k*.

Step 8. Procedural verification. We have now formalized the protocol. Moreover, given that one of the *trust requirements* is for *c* to be authentic, we add an authenticity query within our ProVerif model. It is represented as a correspondence assertion stating that, each time the AP confirms having received a valid challenge response, a WD must have previously sent it. When we run ProVerif (the code for this example is available in Diaz et al. (2013)), we observe a trace like the one shown in Listing 4.

Since we have found an attack during the procedural verification with ProVerif, the requirements are not fulfilled. Therefore, we must go back to the first stage of the methodology after checking that no *coding* errors have been made, and re-design the protocol to avoid this attack. Moreover, since the specific security requirement that has failed is the authenticity of the challenge *c*, this is the component that we need to revise (along with everything interacting with it). For instance, we could proceed like it is suggested in Borisov et al. (2001), namely, we could disallow the reuse of IVs.

4.3. Case study III: complete verification

Now we shortly show how the full methodology is applied to the analysis of the protocol that was informally presented in Diaz et al. (2011), and whose formal security verification is performed in Diaz et al. (2012). Here we will just summarize the procedure followed for its verification (for further details, see Diaz et al. (2012)). While applying our methodology, we found flaws in the design of the protocol that could lead to replay attacks or impersonation attacks (Boyd and Mathuria, 2003). For instance, in order to solve the latter, we were forced to start a new iteration of our methodology and partially redesign the protocol, including the tickets explained below. The protocol is called CHAT-SRP (CHAos based Tickets – Secure Registration Protocol), and it is intended to be used in registration protocols for interactive platforms. The typical registration method used in these platforms requires the new users to provide an email address, verifying their identity when they access a link included in an email sent to the provided address.[7] This approach, although very user-friendly, is quite insecure, because emails are almost always sent unprotected (Farrell, 2009). Therefore, an attacker may impersonate a user just by eavesdropping the activation email. CHAT-SRP generates a ticket (basically, a pseudo-random number), links it to the requesting user email, and sends it encrypted via HTTPS. The user, who is initially identified as being the owner of a mobile number (which we assume to be previously known and verified), needs to (1) access the activation link sent by email, and (2) provide the right ticket in order to validate his new account. This way we confirm that who requested the registration is the one completing it. Once confirmed his identity, the protocol sends the new user a digital identity, which he could later use for performing robust cryptographic operations. Let us use our methodology to analyze it.

[6] Although there are approaches for XOR-aware modifications of ProVerif. See Küsters and Truderung (2011).

[7] This is known as EBIA (Garfinkel, 2003).

```
Process WD:
send(MA,AUTH,(id,'shared key',1));
receive(MA,AUTH,('shared key',2,info,result));
if result == successful then
chal = info;
send(MA,AUTH,('shared key',3,{chal}_k));
fi
receive(MA,AUTH,('shared key',4,result));

Process AP:
receive(MA,AUTH,(id,'shared key',1));
if successful then
result = successful;
chal = new pseudorandom;
send(MA,AUTH,('shared key',2,chal,result));
receive(MA,AUTH,('shared key',3,{chal2}_k));
if {chal2}_k == {chal}_k then
send(MA,AUTH,('shared key',4,successful))
fi
fi
```

Listing 3. Pseudocode for the WEP-SKA procedures. *WD* stands from the Wireless Device to be authenticated, *AP* stands from Access Point and *k* is the shared key. The field *info* conveys information dependent on the authentication algorithm, and *result* stores the result of the requested authentication. *MA, AUTH, 'shared key'* and the numbers used in the messages are fields specified in the standard, see IEEE, 1999.

Step 1. Goals of the protocol. We want to register new users, providing them with new virtual identities to be used in the platform. Therefore, our more formally stated goals are:

1. Authenticate a user requesting registration.
2. Send him/her a digital identity, keeping the identity's confidentiality and guaranteeing its authenticity (e.g., that it has been generated by a suitable Certification Authority).

Step 2. Attacker capabilities and security model. We assume the typical of the Dolev-Yao attacker. Note that this includes the capability of eavesdropping and blocking emails, which, as we said before, is our main concern. Besides, we also assume that our attacker can search in the Internet for the necessary information to start a registration process (in this case, a username and an email). Thus, our *+/−capabilities list* is shown in Listing 5.

Step 3. Design candidate. A sequence diagram for the protocol informally described at the beginning of this subsection is shown in Fig. 4, and we will use it as our design candidate. Keep in mind that we only show the result of several iterations of the methodology. Also, in this diagram and in the subsequent analysis, WS stands for Web Server, RA for Registration Authority and CA for Certification Authority. The three of them are assumed to be trusted authorities.

Step 4. Trust requirements. Applying Algorithm 1 to the elements of the diagram shown in Fig. 4, we obtain the *trust requirements* in Listing 6.

Basically, at steps 1, 3 and 4, we are considering the extra capability added to our attacker in our *+/−capabilities list*. Also, when the activation link is created in step 9, we require it to be authentic, confidential and unique (the confidentiality requirement is just a consequence of having just been created). However, in step 10.2 it is sent via email (an insecure channel), thus we cannot assume any trust property, since it could be eavesdropped or even originated by the attacker. The ticket and ID are expected to be kept authenticated, secret and unique, since they are always conveyed through SSL-secured channels. Since we have required the protocol to provide secrecy, authenticity and uniqueness, we decided to use ProVerif, which allows us to check authenticity and confidentiality, while uniqueness can be modeled by using the ProVerif `new` instruction.

Step 5. Informal verification. Going through all the steps in Listing 6 we do not find any inconsistency with the attacker capabilities nor the *+/−capabilities list* specified before. Moreover, we have taken into account the attacker's capability stating that she knows all the personal information required to initiate a registration request (since we have place no trust requirement in the username and email

WEP-SKA between legitimate WD_1 and AP:
$WD_1 \rightarrow AP$: WD_1
$AP \rightarrow WD_1$: $Chall_1$
$WD_1 \rightarrow AP$: $\{Chall_1\}_k$
Attacker : $k = Chall_1 \oplus \{Chall_1\}_k \implies$ Attacker gains k
$AP \rightarrow WD_1$: OK
WEP-SKA between illegitimate $fake_{WD_2}$ and AP:
$fake_{WD_2} \rightarrow AP$: $fake_{WD_2}$
$AP \rightarrow fake_{WD_2}$: $Chall_2$
$fake_{WD_2} \rightarrow AP$: $\{Chall_2\}_k$
$AP \rightarrow fake_{WD_2}$: OK

Listing 4. An example attack trace found by ProVerif for the *Authentication spoofing attack* over WEP-SKA introduced in Borisov et al. (2001). In the first block, the attacker eavesdrops on a WEP-SKA run between a legitimate WD and the AP. As a result, the attacker obtains the preshared key *k*. In the second block, the attacker uses the key *k* in order to successfully complete a WEP-SKA execution with the AP.

```
+ Knows all the information required for registration
for every user subject to be registered (username and email).
```

Listing 5. +/−capabilities list for the Dolev-Yao attacker for the CHAT-SRP protocol.

elements in steps 1, 3 and 4). For the remaining steps, the required *trust properties* are consistent with the properties of the channels used to conveyed the associated elements (we must remark here that the WS, RA and CA are trusted authorities). Thus, we can continue, considering that our design successfully passes our informal verification step.

Step 6. Protocol pseudocode. For brevity, we do not include the pseudocode here. Instead, it is accessible from Diaz et al. (2013). Note that, again, it was obtained after several iterations of our methodology, in which we detected and corrected security flaws.

Step 7. Formalization of the protocol. In step 4 we decided to use ProVerif for verification. From the pseudocode, it is not hard to obtain a formalization for ProVerif. For further details on the formalization of the protocol see Diaz et al. (2012) or the pseudocode and formalization for ProVerif in Diaz et al. (2013).

Step 8. Procedural verification. From the output of step 4, we can get what queries do we have to make to ProVerif. Namely, for the generated IDs, tickets, and codes, we have to check authenticity, confidentiality and uniqueness. In this case, uniqueness cannot be checked by ProVerif straight away. However, the three of them are created using the `new` instruction within each run of the protocol. Thus, we can assume that they are unique (nevertheless, this requirement could be passed to the development team). On the other hand, the link, username and email do not pose any security requirement. The confidentiality (secrecy) query about the three elements is easily done by adding queries of the type `query var:Type; attacker(var).`. In this case,

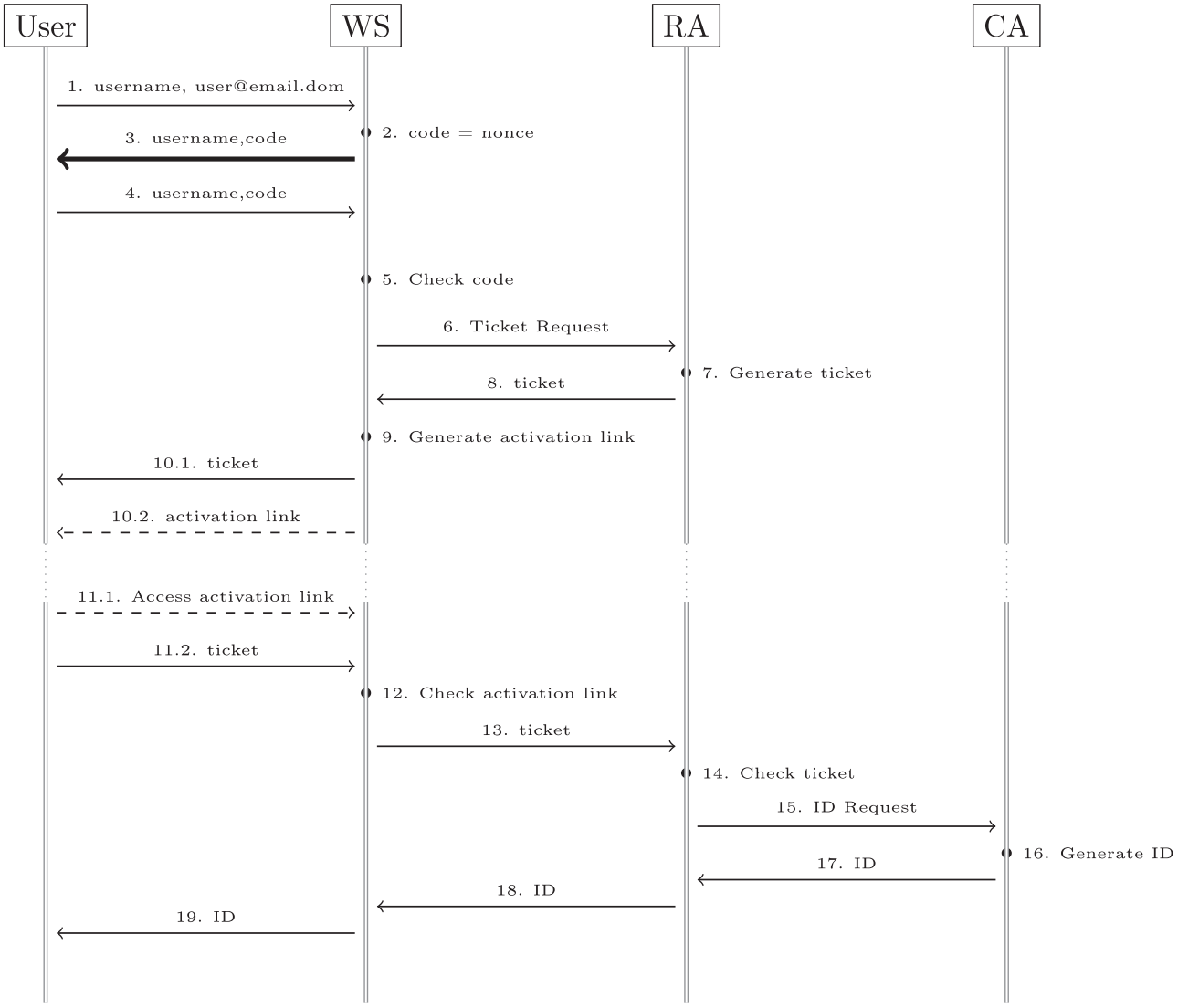

Fig. 4. Sequence diagram of the protocol messages. The dashed lines represent unprotected communications; the thin continuous ones represent SSL protected communications, and the thick continuous line represents the message sent using the extra authenticated channel (SMS). Although not shown in the diagram for readability, the ticket is generated as the hash of a nonce and the user's email.

```
Step 1. username : None
user@email.dom : None
Step 2. code : Uniqueness, authenticity, confidentiality
Step 3. username : None
code : Uniqueness, authenticity, confidentiality
Step 4. username : None
code : Uniqueness, authenticity, confidentiality
Step 5. Nothing to do here
Step 6. Ticket Request : Authenticity, confidentiality
Step 7. ticket : Authenticity, confidentiality, uniqueness
Step 8. ticket : Authenticity, confidentiality, uniqueness
Step 9. link : Authenticity, confidentiality, uniqueness
Step 10.1. ticket : Authenticity, confidentiality,
uniqueness
Step 10.2. link : None
Step 11.1. link : None
Step 11.2. ticket : Authenticity, confidentiality,
uniqueness
Step 12. Nothing to do here
Step 13. ticket : Authenticity, confidentiality, uniqueness
Step 14. Nothing to do here
Step 15. ID Request : Authenticity, confidentiality
Step 16. ID : Authenticity, confidentiality, uniqueness
Step 17. ID : Authenticity, confidentiality, uniqueness
Step 18. ID : Authenticity, confidentiality, uniqueness
Step 19. ID : Authenticity, confidentiality, uniqueness
```

Listing 6. List of trust requirements for CHAT-SRP.

the three secrecy properties are successfully verified. As for authenticity, we check it with several correspondence assertions:

1. For the code, we check that each time a user processes an SMS, he/she has previously requested registration and it has been sent by the WS.
2. For the ticket, we verify that each time a user receives it (along with a registration link), the RA has created the ticket, and the WS has created the registration link and sent it jointly with the ticket.
3. For the ID, we check that each time a user receives an ID, it has been generated by the CA, and that same user has requested an ID at least once (this includes the general case in which a user makes several requests, e.g., because some of them fail).

Running ProVerif on the code in Diaz et al. (2013) shows no violation on the stated security requirements. Thus, we can assume that they have been accomplished.

Therefore, following our methodology, as we have briefly shown, CHAT-SRP fulfills our secrecy, authenticity and uniqueness requirements.

As observed at the end of Section 3, in this case we have a total of 4 alternative flows, corresponding to failures in each of the checks performed by the servers (steps 2, 5, 12 and 14 in the diagram in Fig. 4). Therefore, it could be necessary to verify all of them. Note however that, in this case we do not need to verify them, since the "main" flow includes all of them. However, the correctness of this shortcut depends on our specific security requirements. For instance, in case we wanted to ensure anonymity, the fact of rejecting a registration request might leak knowledge to an attacker, hence breaking the protocol's security requirements.

5. Conclusion

In this work we have designed a methodology for verifying security properties of communication protocols. It is mainly divided in two parts. The first one is devoted to a preliminary study of the properties of the context where the protocol will be applied, its security requirements and an informal verification. The result of this first phase is an informally verified design candidate along with a set of informal *trust requirements*. The second part is dedicated to a procedural verification of these security requirements, applying well-known and widely accepted tools and/or procedures. Moreover, this second phase makes use of the design candidate produced at the first stage, and applies the therein obtained informal *trust requirements* to produce a final set of formal security requirements. By adopting an iterative methodology in which the early processes are less time consuming, we avoid wasting resources by increasing the probability of detecting basic flaws at the beginning of the analysis. Additionally, the fact of being constituted as an iterative methodology eases the task of producing useful feedback when security flaws are detected at the final steps, and take advantage of this feedback for solving design errors at the initial steps. We also provide three examples in order to help to understand which are the benefits of applying this methodology. These examples (MANA III, WEP-SKA and CHAT-SRP) are representative, since they are real protocols that have been presented to the community and, specially in the case of WEP-SKA, widely used. Note that the first two examples underline already known security flaws. However, our aim is not to detect new flaws, but to show how our methodology can be applied and the benefits of doing so.

We would like to remark that, when designing and evaluating communication protocols, the application of such a methodology helps to detect and avoid flaws that can lead to attacks on the protocols, like we have shown. It is important to note that both phases of the methodology are required: if the first phase is skipped, a wrong contextualization may render the formalization invalid; if, on the other hand, we skip the second phase, most probably we will not realize several involved subtleties that, in turn, can lead to security flaws. By using what we called procedural methods, like formal protocol verifiers, or the computational model for the verification of protocols, we can prevent much of the damage caused by flawed designs. Nevertheless, the human factor also intervenes in the procedural analysis of protocols, and a wrong formalization can make some flaws to go unnoticed. Hence, like in every engineering

process, this does not provide a 100% success rate, but it does help to avoid many flaws.

Acknowledgments

This work was supported by the UAM project of Teaching Innovation and the Spanish Government projects TIN2010-19607 and TIN2012-30883. The work of David Arroyo was supported by a Juan de la Cierva fellowship from the Ministerio de Ciencia e Innovación of Spain.

References

Abadi, M., Rogaway, P., 2002. Reconciling two views of cryptography (the computational soundness of formal encryption). In: IFIP TCS, pp. 3–22.
Aizatulin, M., Dupressoir, F., Gordon, A.D., Jürjens, J., 2011a. Verifying cryptographic code in c: some experience and the csec challenge. In: Formal Aspects in Security and Trust, pp. 1–20.
Aizatulin, M., Gordon, A.D., Jürjens, J., 2011b. Extracting and verifying cryptographic models from c protocol code by symbolic execution. In: ACM Conference on Computer and Communications Security, pp. 331–340.
Anderson, R.J., 2008. Security Engineering – A Guide to Building Dependable Distributed Systems, 2nd ed. Wiley Publishing, Inc., Indianapolis, IN.
Andrew, C.H.A.J., Gordon, D., 2002. Cryptyc: cryptographic protocol type checker. http://cryptyc.cs.depaul.edu/
Barthe, G., Grégoire, B., Béguelin, S.Z., 2009. Formal certification of code-based cryptographic proofs. In: POPL, pp. 90–101.
Bellare, M., Boldyreva, A., Micali, S., 2000. Public-key encryption in a multi-user setting: security proofs and improvements. In: EUROCRYPT, pp. 259–274.
Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S., 2011. Refinement types for secure implementations. ACM Transactions on Programming Languages and Systems 33, 8.
Bhargavan, K., Fournet, C., Gordon, A.D., 2010. Modular verification of security protocol code by typing. In: POPL, pp. 445–456.
Blanchet, B., Cadé, D., 2012. Cryptoverif Cryptographic protocol verifier in the computational model. http://www.cryptoverif.ens.fr/
Blanchet, B., 2010. ProVerif Automatic Cryptographic Protocol Verifier User Manual. CNRS, Département d'Informatique École Normale Supérieure, Paris.
Borisov, N., Goldberg, I., Wagner, D., 2001. Intercepting mobile communications: the insecurity of 802.11. In: MOBICOM, pp. 180–189.
Boyd, C.A., Mathuria, A., 2003. Protocols for Authentication and Key Establishment. Information Security and Cryptography, Springer.
Brown, M., 2013. Toward a taxonomy of communications security models. Journal of Cryptographic Engineering, 1–15.
Chaki, S., Datta, A., 2009. Aspier: an automated framework for verifying security protocol implementations. In: CSF, pp. 172–185.
Díaz, C., Seys, S., Claessens, J., Preneel, B., 2002. Towards measuring anonymity. In: Privacy Enhancing Technologies, pp. 54–68.
U. di Genova, C. group, I.S.G. (ETHZ), S. AG, Avispa: Automated Validation of Internet Security Protocols and Applications, 2006.
Diaz, J., Arroyo, D., Rodriguez, F., 2011. An approach for adapting moodle into a secure infrastructure. In: Herrero, A., Corchado, E. (Eds.), Computational Intelligence in Security for Information Systems, volume 6694, of series Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 214–221.
Diaz, J., Arroyo, D., Rodriguez, F., 2012. Formal security analysis of registration protocols for interactive systems: a methodology and a case of study. http://arxiv.org/abs/1201.1134
Diaz, J., Arroyo, D., Rodriguez, F., 2013. Code for the protocols analyzed in "A formal methodology for integral security design and verification of network protocols". http://www.ii.uam.es/gnb/secmethod.tgz
2007. Digital Privacy: Theory, Technologies, and Practices, 1st ed. Auerbach Publications.
Dolev, D., Yao, A.C.C., 1983. On the security of public key protocols. IEEE Transactions on Information Theory 29, 198–207.
Farrell, S., 2009. Why don't we encrypt our email? IEEE Internet Computing 13, 82–85.
Garfinkel, S.L., 2003. Email-based identification and authentication: An alternative to pki? IEEE Security & Privacy 1, 20–26.
Gehrmann, C., Mitchell, C.J., Nyberg, K., 2004. Manual authentication for wireless devices. RSA Cryptobytes 7, 2004.
Goubault-Larrecq, J., Parrennes, F., 2005. Cryptographic protocol analysis on real c code. In: VMCAI, pp. 363–379.
Hernan, S., Lambert, S., Ostwald, T., Shostack, A., 2006. Uncover security design flaws using the stride approach, 9.aspx.
Houmani, H., Mejri, M., Fujita, H., 2009. Secrecy of cryptographic protocols under equational theory. Knowledge-Based Systems 22, 160–173.
IEEE, Wireless lan medium access control (mac) and physical layer (phy) specifications, IEEE Standard 802.11, 1999.
ISO27001, 2005. Information Security Management System (ISMS) standard.
Janvier, R., Lakhnech, Y., Mazaré, L., 2004. Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries, Technical Report number19, Verimag Technical Report.
Jurjens, J., 2003. Secure Systems Development with UML. Springer, Berlin/Heidelberg/New York, ISBN 3-540-00701-6.
Küsters, R., Truderung, T., 2011. Reducing protocol analysis with xor to the xor-free case in the horn theory based approach. In: ACM Conference on Computer and Communications Security, pp. 129–138.
Kemmerer, R.A., 1987. Using formal verification techniques to analyze encryption protocols. In: IEEE Symposium on Security and Privacy, pp. 134–139.
Lamport, L., Shostak, R.E., Pease, M.C., 1982. The byzantine generals problem. ACM Transactions on Programming Languages and Systems 4, 382–401.
Matsuo, S., Miyazaki, K., Otsuka, A., Basin, D.A., 2010. How to evaluate the security of real-life cryptographic protocols? The cases of iso/iec 29128 and cryptrec. In: Financial Cryptography Workshops, pp. 182–194.
Mohammad, M., Alagar, V., 2011. A formal approach for the specification and verification of trustworthy component-based systems. Journal of Systems and Software 84, 77–104.
Paulson, L., Nipkow, T., Wenzel, M., 2012. Isabelle.
Paulson, L.C., 1998. The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6, 85–128.
Swigart, S., Campbell, S., 2008. Sdl series – article #1: Investigating the security development lifecycle at microsoft. http://www.microsoft.com/security/sdl/resources/publications.aspx
2009. Common Criteria for Information Technology Security Evaluation – Part 3: Security assurance components, Technical Report.
Weldemariam, K., Kemmerer, R.A., Villafiorita, A., 2011. Formal analysis of an electronic voting system: an experience report. Journal of Systems and Software 84, 1618–1637.
Wong, F.L., Stajano, F., 2005. Multi-channel protocols. In: Security Protocols Workshop, pp. 112–127.