Automated Formal Verification of Application-specific Security Properties

Piergiuseppe Bettassa Copet and Riccardo Sisto

Dipartimento di Automatica e Informatica
Politecnico di Torino, Italy
{piergiuseppe.bettassa,riccardo.sisto}@polito.it

Abstract. In the past, formal verification of security properties of distributed applications has been mostly targeted to security protocols and generic security properties, like confidentiality and authenticity.

At ESSOS 2010, Moebius et. al. presented an approach for developing Java applications with formally verified application-specific security properties. That method, however, is based on an interactive theorem prover, which is not automatic and requires considerable expertise. This paper shows that a similar result can be achieved in a fully automated way, using a different model-driven approach and state-of-the-art automated verification tools. The proposed method splits the verification problem into two independent sub-problems using compositional verification techniques and exploits one tool for analyzing the security protocol under active attackers and another tool for verifying the application logic. The same case study that was verified in the previous work is used here in order to show how the new approach works.

1 Introduction

Formal verification of security properties of distributed applications is attracting researchers' attention, especially in recent years, because of the increasing diffusion of applications with important security requirements.

Distributed applications with security requirements generally use cryptographic protocols to communicate over insecure channels. Sometimes the security protocol and the application logic are totally independent: the protocol provides virtual communication channels with standard security properties (mutual authentication, confidentiality, data integrity) and the application is developed in a nearly security-unaware way, security being provided just by application insulation, which is guaranteed by the fact that the application communicates only over secure channels. In other cases, however, protocol and application logic are less independent. For example, custom protocols can be used in order to guarantee application-specific properties, and the application logic may interact more strictly with the protocol in order to achieve the desired security properties. Of course, using an independent security layer realized by standard protocols (for instance TLS) is preferred when possible, because of its simplicity and reliability. However, this is not always possible or convenient, for example because the devices involved do not have

J. Jürjens, F. Piessens, and N. Bielova (Eds.): ESSoS 2014, LNCS 8364, pp. 45–59, 2014.

enough hardware resources or do not have standard connectivity to the Internet, but only limited ad-hoc connectivity.

The techniques and tools for automated formal verification developed so far are mostly targeted to either the analysis of security protocols or the analysis of application code. On the one hand, some tools [1] can formally verify standard security properties of cryptographic protocols under the presence of active attackers. However, these tools can analyze only the bare protocol (message exchanges and related checks) while they are not adequate to also model and analyze the application logic that interacts with the protocol, which can be made of complex programs, without particular constraints. Moreover, generally these tools cannot deal with application-specific security properties. On the other hand, tools for the automated formal verification of arbitrary application source code are available (e.g. software model checkers [2]). In theory these tools even allow to consider active attackers in the system, but a model of those attackers must be supplied by the user and the inclusion of active attackers makes verification very complex.

Actually, the main obstacle to extending existing verification techniques to analyze both protocol and application logic together in the face of active attackers is mainly practical, and is related to the limited scalability of these verification techniques. In fact, the problem of cryptographic protocol verification is itself challenging despite the simplicity of such protocols.

A case study of formal verification of application-specific security properties (i.e. the truth of a predicate involving some variables of the application), taking into account both the protocol and the application logic together, appeared recently in literature [3]. In this case study the application is developed with a model-driven approach and the model is used to generate a formal specification, which afterwards can be verified by an interactive theorem prover. An important limitation of this approach is that it is based on interactive theorem proving, which is not automatic, is very time consuming, and requires a lot of expertise. Moreover, if the application is flawed, interactive theorem proving does not provide counter examples, which can make error diagnosis and correction very difficult.

In this paper we show that a simpler approach can be used to achieve a similar result. In fact, the proposed method is based on verification techniques that are automated, simpler to use, and that can also provide counter examples when the properties to be verified do not hold.

The main idea is to combine two already existing and well-known automated formal verification techniques, *theorem proving* for cryptographic protocol verification and *model checking* for source code verification, according to the principles of assume-guarantee compositional verification. This approach brings, in addition to the above mentioned advantages, better scalability, due to the splitting of the verification problem into simpler sub-problems. The work proposed in this paper, as well as combining the two mentioned verification techniques, also aims at automating the entire process of implementing and verifying distributed

applications. To our knowledge, at present there are no other proposals with the same characteristics in literature.

The proposed development approach is based on the principles of model driven design: it starts by defining a high-level formal model of the communication protocol, where also the expected security properties of the protocol are formally specified. An automated formal verification of those properties is performed by the protocol verifier ProVerif [4], and the Java implementation of the protocol is automatically generated by the model driven development framework JavaSPI [5], which guarantees the preservation of the intended security properties. The resulting protocol implementation must then be integrated within the application logic (client and server), which can be developed in any way (hand written or developed using other code generation techniques). Then, the application-specific properties are formulated and verified on the application logic using a Java source code verifier, such as Java Pathfinder (JPF)[2], but taking the results of the protocol formal verification into account. This is achieved by replacing the code that implements the protocol with a stub that enforces the properties already verified on the protocol model. If compared to a separate and independent use of the theorem prover and the model checker, the main advantage of the methodology proposed here is the reduction of verification complexity, made possible by leveraging compositional verification in the assume-guarantee reasoning style.

The whole process evolves in a largely automated workflow, which reduces the probability of introducing errors significantly, and enables quick error diagnosis (both the tools used for formal verification can provide counter examples, i.e. the execution traces that violate the intended properties).

The remainder of the paper is organized as follows. Section 2 discusses related work and Section 3 gives some background about the tools that are exploited in this work (ProVerif, JavaSPI and Java Pathfinder). Then, Section 4 explains some new features that have been added to JavaSPI in order to support the approach proposed in this paper and Section 5 introduces the case study example and describes how it was developed and verified. Finally, Section 6 concludes.

2 Related Work

In the last decades many automated techniques have been developed for the formal analysis of security protocols, as recently surveyed in Patel et al. [1]. These techniques analyze high-level abstract models, in order to prove the correctness of the protocol logic. More recently, some researchers have started working on techniques that bring automated formal proofs closer to real implementations of security protocols [6]. Among these are the model-driven development approaches, like the one exploited in this paper [5].

All the above mentioned techniques are focused on security protocols rather than on whole applications, and address the generic security properties enforced by such protocols (e.g. authentication, secrecy and integrity), rather than the application-specific security properties.

Some papers have addressed the formal verification of security protocols for specific applications, such as for example electronic commerce, with their related

application-specific properties. For example, Bella et al. [7] presented the formal verification of some application-specific properties of the suite of protocols "Electronic Secure Transaction", used for e-commerce. However, this work is substantially different from the one presented here because verification is not automatic (being based on the interactive theorem prover Isabelle [8] which requires human assistance), and what is formally verified is only an abstract model of the application rather than its final implementation.

Besides the work by Moebius et al. [3] that was already mentioned in the introduction, and a related publication [9] that presents exactly the same methodology but applied to a service-oriented application, some other papers have addressed the problem of developing distributed applications with formally verified security properties. A recent paper [10] extends the previous approach by integrating the AVANTSSAR [11] model checker into SecureMDD. As a result, it is possible to automatically generate a formal specification for the model checker from a UML model. However, only some application-specific properties can be verified using AVANTSSAR. For example, differently from the work presented here, which enables the verification of arbitrary properties, it is not possible to compare numeric values inside the model checker.

Jürjens [12] proposed a UML-based technique for the specification of distributed applications and automated formal verification of application-specific security properties. The technique was applied to the Common Electronic Purse Specifications regarding payment via smart-card. One of the properties that were verified is, for example, that the amount of money in the system is every time the same, that is the total sum of budgets of smart-card holders is always equal to the sum of the earnings of all merchants. However, this technique provides formal verification of UML models only, whereas a formal link with the application implementation is missing. Moreover, differently from our approach, verification is performed in a single step on the whole model, without using compositional verification.

Gunawan et al. [13] proposed a method to integrate some standard security mechanisms (for protecting information transfer) into distributed applications automatically. The paper includes a proof that the security mechanisms are integrated into the application so as to fulfill some generic properties. However this approach does not target the verification of application-specific properties.

The idea of using compositional verification to formally verify application-specific security properties of distributed applications already appeared in Gunawan and Herrmann [14]. In that work, however, formal verification is done by a general-purpose model checker, without considering active network attackers and the properties of cryptographic operations.

3 Background

3.1 ProVerif

ProVerif [4] is an automated theorem prover for cryptographic protocols. In ProVerif, the protocol and the attacker are modeled according to the Dolev-Yao

[15] symbolic approach, which substantially means representing data and cryptographic operations symbolically and assuming the attacker has complete control over public communication channels, thus being able to read, delete, and modify messages in transit or forge new messages using the knowledge the attacker has achieved so far. The symbolic representation of data and cryptography entails that cryptography is assumed to be ideal. For example, an encrypted message can be decrypted only if the correct decryption key is known. Differently from model checkers, ProVerif can model and analyze an unbounded number of concurrent sessions of the protocol, thus providing results that hold for any number of parallel sessions. However, like model checkers, ProVerif can reconstruct a possible attack trace when it detects a violation of the intended security properties. ProVerif may report false attacks, that is attacks that in reality are not possible, but at the same time if a security property is reported as satisfied then it is true in all cases, so it is necessary to analyze the results carefully when attacks are reported.

3.2 The JavaSPI Framework

JavaSPI [5] is a framework for modeling, formally verifying and implementing cryptographic protocols, according to the paradigm of model-driven development. Initially, the user defines an abstract formal model of the protocol according to the Dolev-Yao modeling approach. This model, being abstract, does not include implementation details such as, for example, hash algorithms and length of cryptographic keys. This model can be formally verified by ProVerif in order to check that it satisfies some security properties. These properties are generally expressed either as secrecy requirements (the attacker must not be able to know some data) or as correspondence requirements referred to events specified in the abstract model. The latter requirements can be used to express authentication or data integrity properties; for example an authentication requirement could be expressed as $terminate(A, B) \Rightarrow start(B, A)$, which means that each time actor A terminates a session of the protocol apparently with B (i.e. event $terminate(A, B)$ occurs), B has previously started a session of the protocol with A (i.e. event $start(B, A)$ has occurred).

When the user is satisfied with the model and confident about its logical correctness, the missing implementation details can be specified and a Java implementation of the protocol can be automatically generated. JavaSPI is very similar to Spi2Java [16], the main difference being the modeling language: while with Spi2Java a protocol is modeled directly in the formal specification language spi-calculus, JavaSPI lets the user develop the protocol model in the form of a Java application, written with some restrictions on the Java language and making use of a custom library (JavaSpiSim), which offers the same expressiveness as the spi calculus language. In fact, a formal specification of the protocol compatible with ProVerif can be generated automatically from the Java code. Using Java as the modeling language facilitates users who are familiar with object oriented programming and Java. Moreover, this approach lets the user simulate the execution logic of the protocol by means of a normal Java debugger.

Figure 1 shows an excerpt of an abstract model written with JavaSPI. Each model is composed of a number of processes, each one specified by a Java class that extends the `spiProcess` library class. The behavior of a process is specified by defining the `doRun` method, which takes as arguments objects belonging to classes of the `JavaSpiSim` library. These classes represent the data types admitted in a security protocol model and include methods for performing common operations, such as for example encrypting or decrypting data or sending or receiving data on channels. The occurrence of an event is specified by calling the `event` method which can have any number of arguments (e.g. `event("start",A,B)` generates event $start(A, B)$.

The implementation details that are necessary for generating the final implementation code are specified as Java annotations added to the abstract model. JavaSPI shares with Spi2Java the same code generation mechanism, which has been proved to preserve a large class of security properties [17]. This means that if a security property has been proved to hold on the formal model, then that property holds on the automatically generated Java implementation too.

3.3 Java Pathfinder

Java Pathfinder [2] (JPF) is a software model checking tool for the Java language. Java Pathfinder can directly analyze the bytecode of Java multithreaded applications, checking the truth of assertions or LTL formulas. Java Pathfinder consists of a particular Java Virtual Machine (JVM) which executes the bytecode by exploring all possible execution paths (when nondeterministic choices are possible in the execution, each one of them is explored by backtracking execution).

JPF includes several optimizations that automatically reduce the number of states to be visited (avoiding those whose inspection is redundant) and thus the complexity of the analysis.

4 The Extended JavaSPI

To achieve the final goal of this work the JavaSPI framework has been extended in order to enable increased interaction between the generated protocol code and the application that uses the protocol. With the original JavaSPI, only a simple interaction mechanism was possible, where the application starts a protocol session, passing input arguments, and, upon termination of the protocol session, the application gets the outputs. With the extended JavaSPI version, the application can be called back by the protocol code when some events defined in the model occur. In this way, the application can receive outputs from the protocol at intermediate stages of a protocol session. The `@EventsInterface` annotation enables this new mechanism. When the annotation is present, the code generator generates a Java interface that contains the methods associated with the events generated by the process and has the name specified in the annotation. When a session of the protocol is started by the application, a callback object that

```
public class p_Card extends spiProcess {
...
  @EventsInterface("p_Card_Interface")
  public void doRun(Channel cTermCard, Nonce passphrase,
            Identifier LOAD, Identifier PAY, Identifier TERMAUTH,
            Identifier RESAUTH, Identifier TERMLOAD, Identifier TERMPAY,
            Identifier RESPAY) throws SpiWrapperSimException{

    Message xIn = cTermCard.receive(Message.class);

    if(xIn.equals(TERMAUTH)){
      Nonce challenge = new Nonce();
      Pair<Identifier,Nonce> _w0 = new Pair<Identifier, Nonce>(RESAUTH,challenge);
      cTermCard.send(_w0);

      Pair<Message,Hashing> _p0 = cTermCard.receive(Pair.class);
      Pair<Identifier,Integer> xTermLoad_xValue = (Pair<Identifier, Integer>) _p0.getLeft();
      Hashing xHash = _p0.getRight();
      Identifier xTermLoad = xTermLoad_xValue.getLeft();
      Integer xValue = xTermLoad_xValue.getRight();

      if(xTermLoad.equals(TERMLOAD)){
        Pair<Identifier,Nonce> _w1 = new Pair<Identifier, Nonce> (LOAD,passphrase);
        Pair<Message,Nonce> _w2 = new Pair<Message, Nonce>(_w1,challenge);
        Pair<Message,Integer> _w3 = new Pair<Message, Integer>(_w2,xValue);
        Hashing h = new Hashing(_w3);

        if(h.equals(xHash)){
          event("addToBalance",xValue);
...
```

Fig. 1. Excerpt of a sample model code

implements the generated interface must be passed as argument. This extension does not affect the validity of the ProVerif model that is generated from JavaSPI, because the methods called on event occurrence cannot alter the protocol behavior as modeled by ProVerif. As detailed in Section 5.6, when performing the verification of the application code, the protocol code is substituted by stubs that enforce exactly the event orderings that are made possible by the protocol.

5 The Case Study Application Development

The case study is the development of a smart-card based application that implements a sort of electronic purse. The application lets the user load credit onto the smart card and use the loaded credit to get some services. In Moebius et al. [3], a copy service offered by a University Campus to students is considered, but which specific service is offered by the application is not relevant. In the description of the case study, we stick to the copy service example.

The users of the application are some customers and a manager. Each customer owns a smart-card where Java code can run, on which credit can be loaded. The manager provides a set of terminals where customers can go with their smart-card in order to buy or spend credit. The current balance of credit is stored on the smart card and is updated at each operation performed. For simplicity, the example considers one unit of credit corresponding to one copy.

Finally, all terminals and all smart-cards store internally the same secret key, shared by all trusted and original components. The secret keys are assumed to be not accessible, both in the smart-cards and in the terminals (the smart-card is assumed to be tamper-proof while the terminal is assumed to be secured so that only the manager can access its internals for maintenance).

The security goal that is considered in this case study is "the manager does not lose money", that is the total amount of issued copies does not exceed the total credit bought previously by all users during their loading operations on their smart-cards. This property must be satisfied even in the presence of potential active attackers who may intercept/alter/delete messages transmitted between the actors (smart-cards and terminals), or create new ones, following the definition of attackers of the Dolev-Yao model.

5.1 The Development Workflow

The key idea of the proposed development approach (depicted in Figure 2) is to divide the application into two distinct parts, to be developed and verified separately: the protocol, and the application logic.

The protocol is developed according to the JavaSPI model-driven methodology. It includes all communication activities and must satisfy some security properties, specified by the developer.

The application logic can be developed in any way, but it must properly interact with the protocol, by starting protocol sessions and reacting to events.

The verification process is compositional. The security properties of the protocol are verified on the abstract protocol model using ProVerif and assuming a generic scenario with an unbounded number of parallel protocol sessions. The same properties are guaranteed to hold on the Java code that implements the protocol by the code generation algorithm. Application-specific security properties are specified and formally verified using an automated formal verification tool capable of analyzing Java code directly (Java Pathfinder in our case). When performing this verification step, it is possible to avoid the explicit modeling of the protocol part, by substituting it with a stub that describes the security properties proved by ProVerif. The stub can be automatically generated from the protocol properties.

The rest of this section details the various steps with reference to the case study.

5.2 Developing the JavaSPI Abstract Protocol Model

The protocol designed for this application is based on challenge interactions. Figure 3 shows the interaction between a terminal and a card during the load operation.

Once a card is plugged into the terminal, the user can enter money into the terminal, which triggers the start of the load operation. This operation then proceeds as shown in Figure 3, where *value* is the amount of credit to be loaded. The terminal starts the operation generating the $addToIssued(value)$ event and

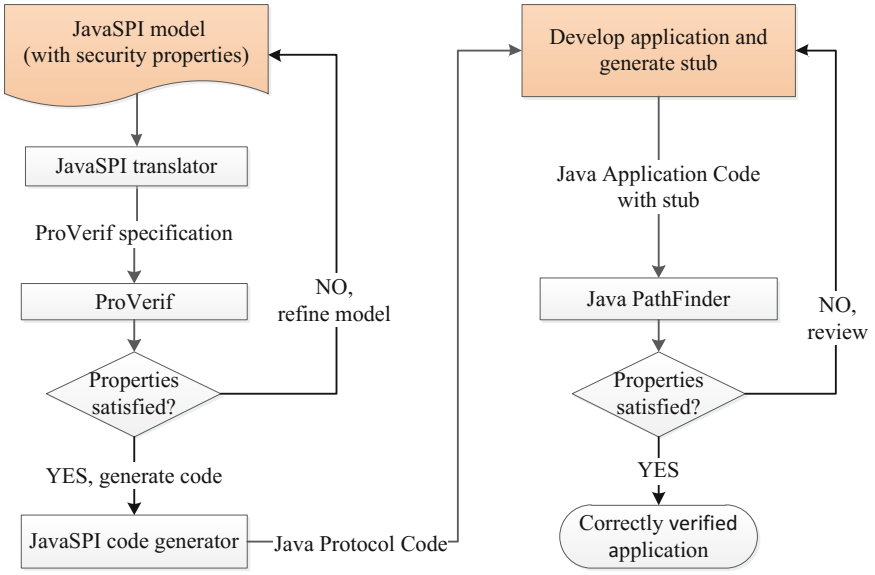

Fig. 2. Workflow of the verification process

sending the card the TERMAUTH message. The card responds issuing the challenge message, composed of the RESAUTH tag and a nonce (a randomly generated number). The terminal responds to the challenge by sending the last message, which includes the TERMLOAD tag followed by the value to be loaded and a hash value, computed on a 4-tuple that includes the shared secret key, the nonce and the value. Finally, the card re-computes the hash value using its own copy of the secret key and nonce and the value received in the message, and if the result matches the received hash value it concludes successfully the operation, by generating the $addToBalance(value)$ event.

The two events will correspond to operations in the application logic that record, respectively, the amount of money earned and the amount of credit spent.

The JavaSPI specification of the card behavior during the load operation is the code excerpt shown in Figure 1.

The JavaSPI model can be simulated in order to check that it behaves as expected.

This security protocol is expected to satisfy two main security properties. The first one is that the secret shared by all the original components cannot be known by an attacker, who has access to the communication channel between the terminal and the card. The second one is the correspondence of the protocol events. For the load operation, each time some credit is actually loaded onto a smart-card (event $addToBalance(credit)$), the corresponding amount of money must have been previously entered into one terminal (event

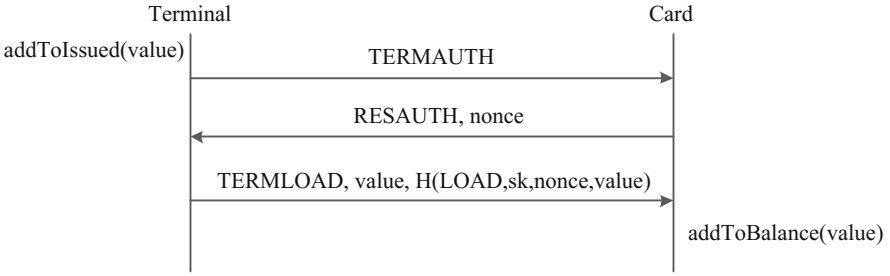

Fig. 3. The load operation

$addToIssued(credit)$). Moreover, the correspondence between these events must be injective, i.e. any $addToBalance(credit)$ event must have its own corresponding $addToIssued(credit)$ event. Injectivity is necessary in order to avoid replay attacks (i.e. a duplicated load credit message, which would result in an addition of unpaid credit on the smart-card, must be avoided). A similar property can be specified for the operation of spending credit.

Note that a tool like ProVerif cannot model integer arithmetic and precedence comparisons between integers. Hence, it does not allow to specify more complex properties, e.g. the ones related to the sum of credit loaded or spent, nor it allows to describe the application logic that processes the events and updates integer counters.

5.3 Formal Protocol Verification

The model generated in the previous step is automatically converted by JavaSPI into the input syntax accepted by ProVerif. The resulting code is ready to be formally analyzed, but first the information on the multiplicity of processes must be added, in order to indicate that there may be an unbounded number of instances of processes.

ProVerif succeeds in proving that the intended properties of the protocol hold on the model. ProVerif takes 15ms to complete the proof on a computer equipped with Intel Core2 Quad Q9450 running at 2.66GHz, 8 GB of DDR2 RAM and Ubuntu 12.04 64-bit operating system and ProVerif 1.86p13.

5.4 Protocol Code Generation

After having verified the model with ProVerif, the generation of the Java code that implements the protocol can take place, by means of the code generator provided by JavaSPI. The result is a set of Java packages, one for each process in the model, which implements the behavior defined in the model.

5.5 Application Logic Development

The generated protocol code must now be integrated with the application code that uses it. In our case study, the application code has been kept simple, but it includes all the fundamental aspects of the application that are necessary for its verification. More precisely, only the functionalities related to the management of the credit system have been implemented on the card software.

5.6 Checking the Application Code

The last step of the workflow is the verification of the application-specific properties using Java Pathfinder. As already anticipated, in order to reduce the complexity of this verification task, the protocol code generated by JavaSPI is replaced with a stub that just reproduces any possible behavior of the protocol sessions, as seen by the application, without really executing the protocol. Of course, the behavior of the stub must be constrained so as to satisfy the security properties that have already been verified by ProVerif. In principle, this constraint can be enforced in one of two different ways: either the constraint is enforced when generating the stub, or the stub is generated without any constraint but the application-specific security property P to be verified is rewritten in the following form

$$C \Rightarrow P$$

where C is the constraint (i.e. the property verified by ProVerif). This second approach is more difficult, because of the difficulty of expressing C. Then, the first approach (generation of a stub that incorporates the constraints coming from the properties verified by ProVerif has been selected for our case study).

As the application processes interact with each other only through the protocol, having replaced the protocol implementation with the stub makes it possible to avoid considering the behavior of active attackers any more during the verification of the application code. In fact, the behavior of potential active attackers has already been considered when analyzing the protocol by ProVerif, and it is already incorporated in the stub behavior itself.

Based on the architecture of the developed application, the only possible interactions between the protocol and the application logic are those that occur at the start and at the end of each session, as well as at the occurrence of one of the intermediate events described in the model. For this reason, it is enough for the stub to include the statements corresponding to these interaction points. All the other statements that make up the protocol implementation can be safely omitted.

The stub can be built by creating multiple Java threads, each one playing the behavior of a single actor in a single protocol session. In order to include the constraints deriving from the security properties verified by ProVerif, it is enough to synchronize these threads in such a way that the security properties proved for the protocol are enforced.

In our case study, the stub includes threads that play the role of the terminal and threads that play the role of the card. The threads that play the role of the terminal learn the kind of operation and the amount of credit to be loaded or spent at their startup (this information is an input coming from the user when the application starts the session). Instead, the threads that play the card role are ready to perform either a load or a spend operation, which in the real protocol is selected by the first message received.

If we want to constrain the behavior of these threads so as to enforce the correspondence properties that have been verified by ProVerif, we have to synchronize the events of each card thread with the events of a corresponding terminal thread. More precisely, before performing a load or spend event, a card thread has to synchronize with a terminal thread that has just performed a corresponding event. This means that the terminal thread enters a synchronization state after having generated an event while a card thread enters a synchronization state before proceeding with a load or spend event.

Model checking does not allow to analyze systems with an unbounded number of states. For this reason, a necessary condition is that the number of parallel protocol sessions (i.e. the number of threads in the stub) is kept bounded. In our case study, this corresponds to having bounded numbers of users and terminals (as each user has one card, the number of cards equals the number of users), with the assumption that no more than one session at a time is possible on each card or on each terminal.

In addition to bounding the number of threads, as with any software model checking problem, abstractions in the application code may be necessary, in order to make the number of states finite and reasonably small.

In our case study, the application-specific property to be checked is given by the fact that in every instant (or for every state reached and analyzed by the model checker) the value of an integer field (named "balance", which represents the difference between the current paid copies and those issued) is always greater than or equal to zero. This property details the more general property "the manager does not lose money". Since the instantaneous value of the balance field depends on the field additions and subtractions performed by the application itself, it is not possible to introduce a layer of abstraction on it. Nevertheless, it is still affordable to run the model checker over a reasonable number of possible cases.

To check if it is satisfied there are two possible ways.

The first one is to use a plugin for Java Pathfinder that enables the verification of LTL formulas during the state exploration performed by JPF. The plugin [1] used in this case study is not maintained directly by the JPF development team and is subject to discontinuity of development over the years. Other plugins that support LTL verification are available. However, the one used in this case study was chosen because it supports the verification of class field values, and not only method calls sequences. In this case, the LTL property to verify is specified through the following annotation:

[1] Available at `https://bitbucket.org/petercipov/jpf-ltl`

```
@LTLSpec("[] ( it.polito.javaSPI.test.CSJPF.balance>=0)")
```

where `it.polito.javaSPI.test.CSJPF` is the class that includes the balance field. This formula simply means that the balance is always greater than or equal to zero.

The second way is to introduce assertions within the application code. In this case, since the example application requires that the "balance" is always non-negative, it is sufficient to place an "`assert balance >= 0`" at any point in the code where the value of the "balance" is set or modified. As this is a private field, it is very simple to identify the only places where it can be set or modified.

Results show that both methods work well for our case study. No violations of the specified properties are detected, thus proving, by exhaustive state exploration, that the properties hold on the application code. Furthermore, the method that uses assertions occupies less memory (RAM) and takes less time, compared to the LTL formula verification.

Verification with JPF was performed on a computer equipped with Intel i7-3770 CPU running at 3.40GHz and 11GiB of DDR3 RAM. The software components relied on an Ubuntu 13.04 32-bit operating system, Java HotSpot(TM) Server VM (Java version 1.7.0_21, build 23.21-b01, mixed mode).

The initial JavaSPI model is composed by 250 lines of Java code and annotations. The size of the ProVerif model is 150 lines, and the size of the protocol code is about 450 lines of Java code. Both are generated by the JavaSPI generator starting from the initial model. The final application requires about 200 additional lines of Java code.

Table 1 and Table 2 report the time and memory required for the verification of the case study example, in the cases of assertions and LTL formula respectively. With the computational resources specified above, in this case it has been possible to analyze a scenario with a maximum of 4 users and 4 terminals when the LTL formula verification is performed. Conversely, the verification of the *assert* conditions requires fewer resources, and can handle efficiently systems with up to 5 users and 5 terminals. It is important to note, however, that, in general, assertions are not always enough for expressing application-specific properties. Therefore, in other case studied the use of LTL formulas can be unavoidable.

Although it is not possible, with a model checker, to formally infer that the properties hold with any number of users and terminals, the results obtained with a small number of participants are sufficient to give reasonable confidence that this is true. In fact, if a distributed application is flawed, usually the error can be detected even with small numbers of parallel sessions.

Table 1. Java Pathfinder verification time and memory consumption using the `assert` construct

	Users and terminals				
	1	2	3	4	5
Time	<1s	1s	7s	2m 40s	42m 21s
Memory	61MB	79MB	145MB	275MB	697MB

Table 2. Java Pathfinder verification time and memory consumption of the LTL formula

	Users and terminals			
	1	2	3	4
Time	1s	30s	30m 18s	44h 24m 59s
Memory	61MB	290MB	467MB	952MB

As mentioned above, the characteristics of the application itself have a significant effect on the complexity of model checking, so performance can be very different depending on the application under test.

6 Conclusions

In this paper it has been shown how a distributed application with application-specific security requirements can be developed using a model-driven approach that finally yields a formally verified Java implementation. The formal verification of the security properties takes into account active attackers and is entirely automated. The most critical part of the code, i.e. the implementation of the security protocol, is generated automatically from an abstract model with the guarantee of security property preservation. Moreover, the model is written in Java, instead of using domain-specific formal languages. The adoption of a compositional verification approach splits verification into two separate simpler tasks, which potentially leads to the possibility to handle larger applications.

Up to our knowledge, no other approach was previously proposed with all these features together. Compared to the approach presented in [3], which developed the same case study, our approach has the advantage of being fully automated. Even if model checking does not allow us to get a result that holds for any number of users and terminals, the result gives anyway good security assurance and can be obtained using only automated tools and without requiring excessive expertise.

The results obtained are encouraging because they confirm that it is possible to develop distributed applications with formally verified application-specific security properties using only automated tools.

One drawback that we found is the high quantity of resources that the model checking with JPF requires, in terms of memory and time. This is partially due to the kind of verification that interprets the bytecode of the real Java application. Using other verification tools for Java may improve the performance. Future works will address the verification of generic security properties in the final application code, for example guarantee that a the value of a field added manually remains confidential in the final application.

References

1. Patel, R., Borisaniya, B., Patel, A., Patel, D., Rajarajan, M., Zisman, A.: Comparative analysis of formal model checking tools for security protocol verification. In: Meghanathan, N., Boumerdassi, S., Chaki, N., Nagamalai, D. (eds.) CNSA 2010. CCIS, vol. 89, pp. 152–163. Springer, Heidelberg (2010)

2. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engg. 10(2), 203–232 (2003)
3. Moebius, N., Stenzel, K., Reif, W.: Formal verification of application-specific security properties in a model-driven approach. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 166–181. Springer, Heidelberg (2010)
4. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE workshop on Computer Security Foundations, p. 82 (2001)
5. Avalle, M., Pironti, A., Sisto, R., Pozza, D.: The Java SPI framework for security protocol implementation. In: Sixth International Conference on Availability, Reliability and Security (ARES), pp. 746–751 (2011)
6. Avalle, M., Pironti, A., Sisto, R.: Formal verification of security protocol implementations: a survey. In: Formal Aspects of Computing (to appear)
7. Bella, G., Massacci, F., Paulson, L.C.: Verifying the SET purchase protocols. J. Autom. Reason. 36(1-2), 5–37 (2006)
8. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
9. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Model-driven development of secure service applications. In: Proceedings of the 35th Annual IEEE Software Engineering Workshop (SEW), pp. 62–71. IEEE (2012)
10. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Model checking of security-critical applications in a model-driven approach. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 76–90. Springer, Heidelberg (2013)
11. Armando, A., et al.: The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 267–282. Springer, Heidelberg (2012)
12. Jürjens, J.: Developing high-assurance secure systems with UML: a smartcard-based purchase protocol. In: 8th IEEE International Conference on High Assurance Systems Engineering, pp. 231–240 (2004)
13. Gunawan, L.A., Kraemer, F.A., Herrmann, P.: A tool-supported method for the design and implementation of secure distributed applications. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 142–155. Springer, Heidelberg (2011)
14. Gunawan, L.A., Herrmann, P.: Compositional verification of application-level security properties. In: Jürjens, J., Livshits, B., Scandariato, R. (eds.) ESSoS 2013. LNCS, vol. 7781, pp. 75–90. Springer, Heidelberg (2013)
15. Dolev, D., Yao, A.C.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–207 (1983)
16. Pozza, D., Sisto, R., Durante, L.: Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In: 18th International Conference on Advanced Information Networking and Applications, 2004, vol. 1, pp. 400–405 (2004)
17. Pironti, A., Sisto, R.: Provably correct Java implementations of Spi Calculus security protocols specifications. Computers & Security 29, 302–314 (2010)