DOI 10.1007/s00165-012-0269-9
BCS © 2012
Formal Aspects of Computing (2014) 26: 99–123

Formal Aspects of Computing

Formal verification of security protocol implementations: a survey

Matteo Avalle[1], Alfredo Pironti[2] and Riccardo Sisto[1]

[1] Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129 Torino, Italy
[2] Prosecco, INRIA Paris-Rocquencourt, 23, Avenue d'Italie, 75013 Paris, France

Abstract. Automated formal verification of security protocols has been mostly focused on analyzing high-level abstract models which, however, are significantly different from real protocol implementations written in programming languages. Recently, some researchers have started investigating techniques that bring automated formal proofs closer to real implementations. This paper surveys these attempts, focusing on approaches that target the application code that implements protocol logic, rather than the libraries that implement cryptography. According to these approaches, libraries are assumed to correctly implement some models. The aim is to derive formal proofs that, under this assumption, give assurance about the application code that implements the protocol logic. The two main approaches of model extraction and code generation are presented, along with the main techniques adopted for each approach.

Keywords: Security protocols; Automated formal verification; Software verification; Sound refinement

1. Introduction

Background

Security protocols are communication protocols that aim to reach some goals despite the hostile activity of attackers that interfere with the protocol (e.g. by having access to the public channels used by protocol actors). Typical goals are concealing information to unauthorized parties or giving one actor assurance about the identity of another actor with which it is communicating. The typical means used for this purpose is cryptography.

Security protocols are generally used to protect something valuable. This is why high assurance about their correctness is highly desirable. Unfortunately, despite their simplicity, security protocols are quite difficult to get right. The main difficulties, experienced even by security experts, are not just related to the strength of the cryptographic algorithms employed (even if these problems must be faced too); when designing a novel security protocol it is necessary to take into consideration all possible behaviors of hypothetical attackers, including

Correspondence and offprint requests to: R. Sisto, E-mail: riccardo.sisto@polito.it
M. Avalle, E-mail: matteo.avalle@polito.it; A. Pironti, E-mail: alfredo.pironti@inria.fr

violations of the protocol rules, and any possible forgery of messages. The number of these behaviors is typically unbounded or at least huge, because an attacker can forge and inject a new message at each protocol step in a number of ways that is typically unbounded. This fact adds extra complexity to the already complex concurrent interactions that a communication protocol must normally manage. Thus, despite the existence of best practices and recommendations [AN96], the manual design of a novel security protocol remains a very error-prone and challenging task. The difficulty of defining security protocols right is witnessed by stories like the one of the Needham–Schroeder public-key protocol [NS78], which was believed secure for 17 years before Lowe discovered it was affected by a flaw [Low95]; another witness is the recent discovery of a logical flaw[1] in the renegotiation feature of the widely used TLS protocol [DR08], 13 years after the first version of the protocol was published (under the SSL 3.0 name).

Due to the inherent complexity, developing security protocols right demands rigorous, mathematically based methods for reasoning about their correctness. It is significant, for example, that the above mentioned flaw affecting the Needham-Schroeder public-key protocol could be found by applying formal methods [Low96].

The rigorous methods that have been developed so far for reasoning out security protocols belong to two main lines of research. One is based on quite abstract, symbolic modeling: it was originated from the seminal paper by Dolev and Yao [DY83], and was developed mainly in the formal methods community. The other one was originated from the papers by Goldwasser and Micali [GM84] and by Yao [Yao82], and is based on more detailed computational models, involving complexity and probability theories. Both approaches have made much progress in the last few years, pushed by a growing interest in secure computing. The symbolic approach, being more abstract, enables better automation in developing proofs, but gives results that are more complex to relate to real world security goals. On the other side, the computational approach, being closer to reality, gives more realistic security assurance at the expense of increased difficulty in proof automation. After the seminal paper of Abadi and Rogaway [AR02], researchers have been trying to define a relation between the two approaches, either by proving computational soundness for the symbolic model, or by applying reasoning techniques that proved successful in the symbolic model to the computational model. An extensive account of the recent progress in this direction can be found in [CKW11].

Motivation

Both symbolic and computational approaches provide rigorous proofs based on abstract models, albeit at different levels of abstraction. Despite this, a large gap still exists between these models and a real-world protocol implementation and its execution. This gap may be responsible for final unsatisfactory security levels, even when correctness proofs have been developed for a model of the protocol. One important component of this gap is the usually big difference between an abstract protocol model on which proofs are developed and the real code that implements the protocol, written in programming languages. For example, the real control flow and data types of a protocol implementation are generally more complex than the ones of the abstract models. Moreover, when deriving an implementation from a model or specification, programmers may introduce logical and coding errors, and some of these errors may not be detected by testing and may make the behavior of the implementation not corresponding to the model or specification. In practice, widely spread implementations of security protocols, such as OpenSSL and OpenSSH, receive several security patches per year, due to low-level implementation bugs. Notoriously, in OpenSSL an error condition returned by a cryptographic function was incorrectly interpreted by the function caller, making the application accept corrupted data;[2] such a fault cannot be found if a formal model that has no relation with the implementation code is analyzed, because the semantics of the model itself defines the (correct) interpretation of the results of cryptographic functions, and the way the code handles return values is neglected.

Additionally, each programming language has its own mechanisms for accessing data and its own libraries for performing basic operations. Of course, these details cannot be considered by language-agnostic abstract models like the ones that are usually analyzed in a rigorous way, and may be responsible for program bugs that affect security.

On the basis of such considerations, in recent years some researchers have started working towards methods that reduce the gap between models and implementations, bringing formal security proofs closer to real protocol implementations.

[1] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3555.
[2] http://www.openssl.org/news/secadv_20090107.txt.

Focus

This survey focuses on research work aimed at automatically getting formal security proofs that apply to models close to the source code of real protocol implementations, and presents the current state of the art.

The problem of whether security properties are preserved when the source code is compiled into executable code is generally not considered by the work surveyed here. Compilers and run time environments are thus considered part of the trusted computing base.

Moreover, the focus of the survey is on the implementation of protocol logic rather than on the implementation of cryptography or the implementation of the operating system, along with its basic communication primitives and device drivers, which are considered here part of the trusted computing base as well. In particular, the research work that is surveyed aims at ensuring that some security properties hold when an application program source code that implements the protocol logic is executed, under certain environmental assumptions, including the one where the cryptographic libraries used by the code behave as expected.

In the papers surveyed here, the attacker model is such that the attacker can interact with the protocol only through the network, as a separate computing entity. In some works, compromised actors are considered: in these cases the protocol is proved secure for honest actors, even in the presence of compromised actors on which the attacker has complete control. However, what is not covered in the surveyed work is an attacker model where the attacker runs as a parallel process on the same machine as one of the honest actors, and can thus access its memory or its trace of system calls. A formal account of such an attacker model is given for example in [AP12a].

Preview

The naïve approach to prove security properties directly on the source code, interpreted according to some formal semantics, is not practical because of its complexity. A common practical approach, also used in other fields, is to perform formal analysis on a simplified model of the program using state-of-the-art verifiers, and prove that there is a formal link between the program and the abstract model on which the proof has been derived. The formal link must be such that a useful conclusion about the program can be drawn. Such a link is generally expressed by soundness theorems that informally sound like this: if the model is proved free from certain attacks under some assumptions, then the program is also free from the same attacks under some corresponding assumptions.

This is the kind of approach that this survey intends to present.

The following ingredients are essential in order to be able to apply such an approach: a formal abstract modeling language and a programming language, both with formal semantics, a formal definition of the relationship between the abstract model and the program, and a soundness proof. If any of them is missing, the formal chain is not complete. Since the most commonly used programming languages lack standard formal semantics, a modified approach has been explored by some researchers where, in place of the real protocol implementation code, formal models that are quite close to it are used (e.g. code translations to similar languages having formal semantics). In this way, even if the formal proof does not apply to the actual code, the gap between the formally verified model and the actual code is small. This survey will also consider techniques taking this approach. However, approaches that develop artificial languages from scratch with no relation with existing programming languages are neglected here.

The link between model and program can be enforced in several ways. There seems to be no widely acknowledged taxonomy on this. Nevertheless, two main approaches can be identified, namely automated model extraction [BFG06, BFGT08, CD09, O'S08] and automated code generation [ABB+10, BCK+08, BCPM+09, CJK05, PPS12, PS10, PSD04, SPP01, TH04, BBH12]. The typical workflows used by these methodologies are illustrated in Fig. 1. Some variations of these approaches will also be discussed here.

With model extraction, the starting point is source code, possibly annotated with semantic information. An automatic tool builds the abstract model out of the code. During this operation those code details that are not relevant for proving the property of interest are discarded.

Code generation works in the opposite way: the starting point is an abstract model from which an automatic tool builds source code. User-provided implementation choices drive the code generation process and provide information that is missing in the abstract model.

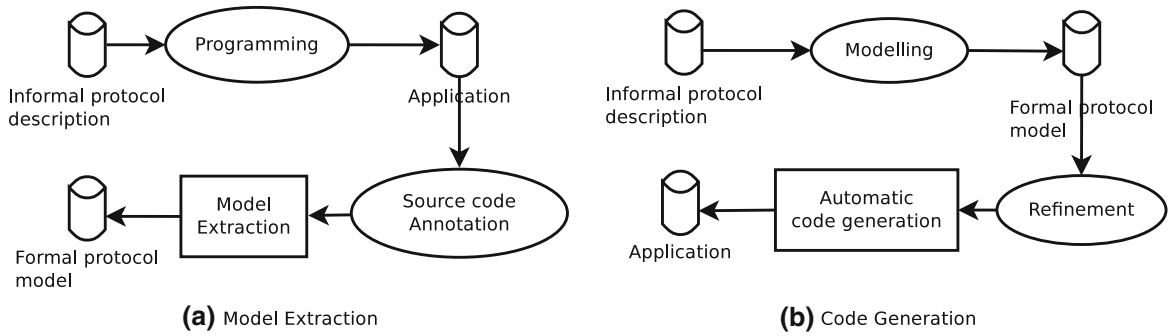

Fig. 1. Typical workflows used by **a** model extraction; **b** code generation

Both approaches have pros and cons.

In principle, model extraction can work on legacy implementations or on implementations that use legacy libraries, which is impossible for code generation. In practice, however, state-of-the-art model extraction approaches cannot deal with arbitrary legacy code, but introduce some requirements on how the code should be written. For aspects that are not captured formally (e.g. timing or memory footprint), an experienced developer can carefully write the security critical code in a way to mitigate attacks based on information leakage, while other parts of the code can be optimized for efficiency. Then, when using model extraction, it is possible to start from code that is already optimized and carefully developed by experienced programmers. Another advantage of model extraction over code generation is that in principle it allows hiding abstract models to the user, who can see model extraction and verification tools as black boxes. This way the user is not required to know abstract modeling languages. Moreover, editing and debugging tools for programming languages are more advanced than corresponding tools for high-level models.

On the other hand, in principle code generation can avoid information leakage via side channels (like timing and memory footprint) by construction, although state-of-the-art approaches normally do not take such aspects into account in the code generation phase. Low-level programming errors, such as for example the ones that may cause buffer overflows in C programs, can be avoided by construction as well. These errors may affect hand-written code and may be the vector leading to security vulnerabilities. While it is true that code generation requires knowledge of abstract modeling languages, at the same time it lets the user focus attention on simpler artifacts (the models), leaving out the error-prone and time-consuming coding phase, according to the well recognized model-driven software engineering practice.

It is important to note here that some of the classical refinement techniques used in model-driven software engineering practice may not preserve some security properties, as it has been explained for example in [Jür01]. For this reason, refinement techniques can be applied safely only if they have been proved to preserve the intended security properties.

In addition to the pure model extraction and code generation approaches, slightly different ways are also possible. For example, one is to manually write both program code and abstract model and then automatically prove that (i) the abstract model fulfills certain properties and that (ii) the program code is a correct implementation of the verified model. This approach can be considered as a variation of the model extraction technique, when the "source code annotation" phase becomes so extended, that almost all the model information has been given by the user, by means of annotations.

In other cases, the user writes an intermediate, abstract version of the protocol, which is not refined enough to run directly as an implementation, and is not yet a fully abstract model. On this intermediate version, on one hand an abstract compiler is run to derive the fully abstract model, on which formal verification is performed. On the other hand, a concrete compiler transforms the intermediate version into a concrete implementation of the protocol. This approach stands in the middle between model extraction and code generation, leveraging techniques from both approaches.

Besides classifying methods according to whether they adopt the model extraction or the code generation approach, another possible classification could be based on the kind of abstract models they use: symbolic versus computational models. As the work on automating proofs on computational models is quite recent, not so much work has already been done on formally linking computational models to real code. Accordingly, this survey is oriented towards symbolic models, and just mentions the first attempts made in the direction of computational models [BCFZ08, BMU10, FKS11, KTG12].

There is still another approach that has been investigated by some researchers as a means for ensuring or verifying the correspondence between an abstract security protocol model and its implementation: runtime verification [Jür05, JYB08, PJ10, BJ10, BJY11]. According to this approach, the protocol model is formal and proved correct. The protocol code, instead, has no particular restrictions. At runtime, the behavior of the protocol implementation is monitored and any deviation from the behavior described in the abstract model is reported as a potential security problem. This survey does not drill into this approach because attention is focused on getting provably secure code rather than on discovering attacks or non-conformance at runtime.

The rest of the survey is organized as follows. Section 2 gives some basic background and notation. Then, Sect. 3 presents the model extraction approach, with a survey of the most relevant research work falling in that category. Section 4 presents recent work based on an approach that can be considered as a variation of model extraction, namely security by typing. Section 5 follows presenting the code generation approach and surveying the most relevant research work based on it. Section 6 presents some results about the non-relevance of certain implementation details for proving some security properties. These results are presented separately, because they equally apply to both model extraction, code generation and their variations. Finally, Sect. 7 discusses the state of the art and points out some directions for future research. Section 8 concludes.

2. Background and notation

This section introduces some notation and basic concepts that will be used throughout the survey.

2.1. Security protocols and their vulnerabilities

A security protocol involves two or more communicating actors (also called principals). Each actor plays a protocol role and is usually associated with an identity (e.g. the identity of a human user). Each protocol role defines the rules that actors playing that role have to stick to. Communication among actors may occur on communication channels of various kinds (a public network, dial-up lines, etc.), by message exchanges. Multiple sessions of the protocol can run concurrently. In addition to honest actors, there may be dishonest actors or attackers aiming to subvert the protocol, i.e. to prevent protocol sessions from reaching their security goals. Of course, attackers are not constrained to follow the protocol rules.

Attackers can be broadly divided into two classes: passive attackers can just intercept, record and analyze protocol messages; while active attackers can also interfere with the protocol, by altering, deleting, redirecting and reordering protocol messages, as well as by forging and inserting new protocol messages into the conversation.

As shown in [GS97b], [Car94] and [PPS11], it is also possible to categorize the attacks on security protocols depending on the weaknesses they exploit. For completeness, a non exhaustive list of the most common types of attacks follows.

Attacks based on *cryptographic flaws* are those that try to break ideal cryptographic properties by exploiting weaknesses of the cryptographic algorithms. Analyzing an encrypted message to extrapolate the secret key is an example of a cryptographic flaw attack. These attacks can be refined to take into account collateral information leaked by a cryptographic primitive or by its particular usage within a security protocol. For example, the measure of the power consumed or the time taken to encrypt the same message with different secret keys can be used to infer some information on the secret keys themselves.

Other attacks exploit the absence of some operation that is crucial to guarantee a security property. This kind of weakness falls within the so-called *internal action flaw* class and may be due to protocol design errors or to implementation mistakes. An example of this type of vulnerability was found in the Three Pass Protocol definition [SRA78], where the absence of a check on the message received in the third phase of the protocol enables the attack.

Other similar attacks are based on the absence of proper message type checking, in which case the attacks are known as *type flaw attacks*. In this kind of attack, the attacker sends a protocol actor a message of different type than what expected, and the actor fails to detect the type mismatch, so misinterpreting the message contents or behaving in an unexpected way. An example of this type of vulnerability can be found in the Otway-Rees protocol [OR87].

When some actor is not able to distinguish between a fresh message and an old message that is being re-used, then *replay attacks* (or, in other words, attacks exploiting *freshness flaws*) are possible. For example, the Needham-Schroeder secret key protocol [NS78] has this type of vulnerability: the attacker can force server and client to re-use an old secret key—that the attacker may already have compromised—in a fresh session.

Other attacks, like the one possible on the Needham-Schroeder public-key protocol, are called *man-in-the-middle* attacks because the attacker stands in the middle between two honest actors and breaks the protocol while relaying messages from one actor to the other.

Some of the vulnerabilities that are most difficult to spot are those related to possible bad interactions among multiple sessions of the same protocol. For example, an attacker could be able to use a protocol actor as an oracle to get some information that the attacker could not generate on its own. Then, this information can be used by the attacker to forge new messages that get injected into another parallel protocol session. This type of attack is known as *oracle attack*. An example of a protocol vulnerable to this type of attacks is again the Three Pass Protocol [SRA78].

As already remarked, the vulnerabilities exploited by attackers may depend on logical flaws or ambiguities in the protocol specification itself or on divergences between the protocol specification and its implementation, caused by programming errors.

The code that implements a protocol may also be affected by other more general errors (buffer overflows, memory leakages, and so on) that may affect not only the security of the implemented protocol itself but also the security of the system where the protocol is executed. For this reason, verifying that the implementation code does not diverge significantly from the intended model and that it is not affected by critical bugs is as important as verifying the security properties of the protocol specification.

2.2. Symbolic protocol models

Symbolic protocol models are also known as Dolev–Yao models, after the names of the researchers who introduced them [DY83].

In these models, data are represented symbolically as terms of a free-term algebra, and cryptographic functions are represented as operations in the same algebra. The ideal properties of real cryptographic functions are captured by the algebraic properties of the symbolic operations.

For example, if $symenc(x, k)$ symbolically represents the symmetric encryption of x with key k, the term algebra is such that the only way to get back the plaintext x from the ciphertext $symenc(x, k)$ is by applying the inverse function $symdec$, i.e. $symdec(symenc(x, k), k) = x$. This algebraic property captures the ideal property of symmetric encryption whereby the plaintext can be recovered from the ciphertext only by decrypting it with the corresponding encryption key.

Channels can be either public or private. Attackers are assumed to be able to access public channels but not private channels. On public channels attackers have full control. They can eavesdrop, remove and inject messages. Furthermore, attackers are assumed to be unable to guess secrets. The only way for an attacker to learn some data is to derive them from the messages exchanged on public channels. This can be done by combining data coming from different protocol sessions at different times. Similarly, an attacker can influence the behavior of honest actors only by interfering with their communications. This means, for example, that it is assumed that the attacker has no control on the scheduling of internal honest actors' operations but it has control on the scheduling of network operations (e.g. by delaying the delivery of a message).

2.3. Computational protocol models

Computational models of security protocols are mathematical models closer to reality. In such models data are represented as bit strings and cryptographic primitives as probabilistic polynomial time algorithms working on bit strings. These algorithms are assumed to fulfill some positive functional properties (e.g. $symdec(symenc(x, k), k) = x$) and some negative security properties expressing what the adversary cannot do (e.g. distinguish between two implementations of the encryption scheme). An attacker is represented by any polynomial time algorithm having access to public communication channels but also to certain oracles that are introduced in order to model some assumptions about extra information the attacker may have access to. A run of the protocol is finally modeled as a probabilistic machine and a security property is defined to computationally hold if the probability that it is violated by a polynomial time attacker in a run of the probabilistic machine is negligible in the size of the protocol secret.

2.4. Protocol implementation models

Formal reasoning about a program (source code) is possible as long as formal semantics are associated to the programming language in which the program is written. Even if the standard semantics of the most common programming languages like C and Java is given only informally, some formalizations of the semantics of (subsets of) these languages have been provided in literature (e.g. [BPP03]). Although such formalizations are not standard, they typically express the most common way compilers and runtime environments interpret these languages. Hence, they can be considered as a fairly accurate basis for formal reasoning about programs written in these languages. Other languages, such as SML, already come with their formal semantics, which can be directly used for formal verification.

In order to formally prove security properties of security protocol implementations, it is not enough to formally model the behavior of the programs run by protocol actors. The behavior of the underlying communication infrastructure and the possible interactions of the executions of these programs with potential attackers (via messages exchanged on public communication channels) have to be considered and formally modeled.

Let P be a program (or set of programs) written in a language with formal semantics, and let E be a set of formal assumptions on the environment in which P is executed. For example, E may formally specify how attackers behave, how communication works, or the fact that attackers have physical access to some communication channels while they have no access to other communication channels. From P and E, an overall closed formal model that represents the behavior of the whole system (protocol actors, attackers, and execution and communication infrastructures) can be built. Let us denote $M = \mathcal{M}(P, E)$ an overall formal model built from P and E. Security properties about M can be formulated as predicates using an appropriate logic. If ϕ is a formula representing a security property, $M \models \phi$ means as usual that formula ϕ is true in M.

The same notation can be used for abstract descriptions of the protocol. For example, let P_a be an abstract, symbolic model of protocol roles, given in an abstract modeling language with formal semantics, such as the applied pi-calculus [AF01], and E_a be a set of formal assumptions on the environment in which the roles described in P_a are executed. $M_a = \mathcal{M}(P_a, E_a)$ denotes the overall formal model, and abstract security properties can be expressed using an appropriate logic.

For example, when derivatives of the pi-calculus such as the spi-calculus [AG98] or the applied pi-calculus [AF01] are used to abstractly model protocol actors, the environmental assumptions that are normally made are that an attacker may be any process described in the same language, running in parallel with protocol actor processes, with a given initial knowledge (where, for example, some secrets are assumed to be initially not known by the attacker). If P_a is the process that cumulatively represents all honest protocol actors, the overall system behavior is formally modeled by any process taking the form $P_a | Q$, where Q is any process expressible in the language and satisfying the assumptions made.

3. Automated model extraction

As already mentioned, and shown in Fig. 1a, model extraction works on already existing implementation (source) code. In some model extraction techniques, annotations can be added to the original code, in order to drive the model extraction process or in order to specify the desired properties the code should have.

A model extractor builds an abstract model that is then analyzed by a formal prover.

According to the classical theory of abstractions [CGL94], model extraction can be formally described as an abstraction mapping $\alpha(\cdot)$ that maps a concrete protocol model $M = \mathcal{M}(P, E)$ onto a corresponding $\alpha(M)$ abstract program model, and a concrete property ϕ onto a corresponding $\alpha(\phi)$ abstract property.

The mapping is a sound abstraction with respect to property ϕ if

$$\alpha(M) \models \alpha(\phi) \Rightarrow M \models \phi. \tag{1}$$

In practice, if the security property $\alpha(\phi)$ has been proved on the abstract model and $\alpha(\cdot)$ has been proved sound with respect to ϕ, then ϕ is proved on the concrete model.

Table 1. The main model extraction approaches

References	Programming language	Abstract model language	Model kind	Security properties	Formal soundness
[BMU10]	F#	RCF	Symbolic and computational	Trace properties	Yes
[BCFZ08, BFG06] [BFGT08, BFGS08]	F#	Applied pi-calculus	Symbolic	Secrecy, authentication	Yes
[BCFZ08]	F#	CryptoVerif	Computational	Secrecy, authentication	No
[CD09]	C	ASPIER	Symbolic	Secrecy, authentication	Yes
[GLP05, GLP09]	C	Horn clauses	Symbolic	Secrecy, information flow	Yes
[Jür08, Jür09]	Java	First-order logic (FOL)	Symbolic	FOL properties	No
[O'S08]	Java	LySa	Symbolic	Secrecy, authentication	No

3.1. Overview of existing approaches

Two major features can be considered when analyzing a model extraction approach for security protocols: (i) existence of formal proof of soundness for the abstraction function $\alpha(\cdot)$; (ii) degree of coverage of source programming language features, i.e. how significant is the subset of the source language that is actually handled by the method.

Table 1 presents the main features of the main model extraction approaches that have been proposed so far. The last but one column reports the security properties that can be verified on the abstract model, while the last column indicates the existence of a formal soundness proof that relates the abstract model language and a language with formal semantics close to the source programming language.

Although several takes have been tried on extracting abstract models from concrete implementations written in different languages, only some of them provide formal soundness proofs. For example, in [O'S08] no soundness proof is provided, although a formal definition of the abstraction mapping is given.

Moreover, due to the complex nature of each programming language, none of the listed methods covers the full syntax of a source programming language. Instead, proper subsets of the standard programming languages F#, C and Java have been defined in order to enable the extraction of formal models.

For example, model extractors that use C implementations, like [CD09] and [GLP05, GLP09], are promising because in principle they can be used to validate widely deployed existing protocol implementations, written in this popular language. The drawback, however, is that it is very hard to correctly model such a low level programming language, formally guaranteeing, at the same time, soundness between model and implementation. This difficulty has brought the developers of model extraction approaches for this language to limit the features of the C language admitted for model extraction.

Some limitations regard minor aspects of the code, and are mainly motivated by implementation limits: as an example, in [CD09] floating point operations are not modeled exactly (but they are approximated as operations on integers) and bitwise operations are left as uninterpreted functions. In [GLP05, GLP09] instead, explicit casts and negative array indexes are not supported, restricting verification to a"well-typed" code subset.

Besides these minor limitations, a major aspect of the C code that is left out is pointer aliasing and pointer algebra. So, neither solution can provide a sound technique to model all pointer aspects, thus restricting their application field or the significance of their results.

Using a higher-level programming language as a starting point, like Java or one of the several available ML dialects, enables model extractors to admit a lager subset of the original code expressiveness. It is still necessary, however, to limit the original syntax in order to automatically recognize particular operations, and this is usually performed by forcing the user to link to a particular ad-hoc API for the cryptographic and networking functionalities or to adopt related programming disciplines. On one hand this can be seen as a facilitation, as these libraries usually greatly simplify code and development, but on the other hand this makes existing legacy protocol implementations hard to verify with these approaches.

The usage of higher-level programming languages also brings a relevant advantage: in fact, the code is less error prone, which also facilitates the formal verification task. For example, buffer overflow vulnerabilities are a serious problem that can affect protocol implementations written in the C language. Model extractors for the C language usually delegate the detection of this problem to other tools or remove the problem by forbidding pointers, so further limiting language expressiveness. By using Java as source language, instead, model extractors do not even have to care about this problem as the runtime execution machine automatically handles it. Of course, when using higher level programming languages, the security results depend on the further assumption that the language interpreters (e.g. the Java virtual machine) do not introduce vulnerabilities themselves. Another advantage of higher level programming languages is that they sometimes present strict similarities to the target formal languages. This is the case for F#, whose first-order subset is not so far from formal languages derived from pi-calculus. This similarity greatly facilitates soundness proofs.

Only two of the existing model extraction approaches deal with computational models [BMU10, BCFZ08]. In [BCFZ08], extraction of computational models is cited as preliminary work, where an F# program is taken to generate a corresponding CryptoVerif [BP06] model. However, soundness proofs are left for future work.

The other approach [BMU10] works on a subset of the F# language. Here, the formal semantics of the source language is expressed by means of the RCF calculus and security trace properties are expressed as first-order logic formulas. The approach of this work is to deduce security trace properties defined in the computational model from corresponding symbolic security properties, exploiting a computational soundness theorem. This theorem states that, under some assumptions, security trace properties defined in the computational model are implied by corresponding symbolic properties. As a consequence, a property can be proved in the computational model by proving the corresponding symbolic property. However, the assumptions under which the computational soundness implication has been proved are rather strict for the moment: the work in [BMU10] considers only encryption and digital signatures as cryptographic primitives and assumes they are implemented by very strong algorithms (IND-CCA2 encryption algorithms and strongly existentially non-forgeable signature algorithms), which does not apply to most current real protocols. Moreover, in order to avoid the key cycle and key commitment problems that are typically encountered in computational soundness proofs, security protocols that can be verified by this approach are further restricted so that, for example, messages sent on public channels never include secret keys.

The next subsections give a more detailed view of the most relevant model extraction approaches developed to date. The work in [BFGT08] targets F# programs, and was the first method to appear, while the one in [CD09] targets C code and is more recent.

3.2. From F# to pi-calculus

The approach described in [BFGT08] applies to protocol implementation code written in F#, a dialect of ML targeting the .NET framework. A formal semantics for F# is available, which is used as a basis for formal proofs.

The method described in [BFGT08] is not general enough to work on any code written in F#, but it applies only to code adhering to a given subset of the possible syntax, called F. Essentially, in order to be acceptable for the model extraction tool described in [BFGT08], a F protocol implementation has to perform cryptographic and communication operations exclusively by invoking functions through appropriate interfaces. Moreover, the F syntax forbids high-order functions and some imperative features available in the F# semantics.

The definition of well-known interfaces used by the application code to access cryptographic and communication operations makes it possible to decouple the protocol application code P_A from the implementation of cryptography and communication.

A key point is that the cryptographic operations defined in the interface operate on an abstract type called *bytes*, and two distinct implementations of the cryptographic library are provided, one symbolic and one concrete. The concrete implementation PC_C implements *bytes* as regular byte arrays, and cryptographic operations as regular cryptographic algorithms finally provided by the .NET framework. PC_C is not verified, rather it is assumed to correctly implement cryptography and networking: for this reason PC_C does not need to comply to F syntax, but it can exploit the full F# expressive power. The symbolic implementation PC_S instead implements *bytes* as algebraic expressions in the Dolev–Yao modeling style.

When P_A executes linked to PC_S it abstractly simulates the protocol, while when the same code executes linked to PC_C it behaves as a regular implementation of the protocol.

If $M = \mathcal{M}(P, E)$ is the formal implementation model to which formal security proofs are targeted, P includes P_A and PC_S. Note that PC_S, like P_A is constrained to be F code, so that the whole P is written in F. Note also that P is quite close to the real implementation code, with which it shares the *same* application code.

For the environmental assumptions E, it is assumed that the F protocol code P can be extended with additional code P_O that implements the behavior of an opponent. P_O can be any F code that uses exclusively a particular predefined set of interfaces. These interfaces are appropriately crafted so as to let P_O interact with P, with a power matching the one of Dolev–Yao attackers.

Security properties about this model are formally expressed as predicates about event logs generated by the protocol application code (this requires that appropriate event logging code be included in the protocol code when the code is developed). A simple logic is introduced for this purpose. For example, an authentication property can be expressed as a correspondence $ev : Accept(x) \Rightarrow ev : Send(x)$, meaning that each execution that logs event $Accept(x)$ also previously logged the corresponding $Send(x)$ event.

Finally, $M \models \phi$ means that for any opponent program P_O written according to the above restrictions, all the event logs generated by the program $P\ P_O$ satisfy ϕ (the program $P\ P_O$ is the concatenation of P and P_O).

In order to formally verify a security property about a protocol implementation, a simpler, more abstract protocol model expressed in a variant of the pi-calculus can be automatically extracted from the F code P using a tool named fs2pv. The variant of the pi-calculus into which fs2pv converts the F input program is a subset of the modeling language accepted by the automatic ProVerif [Bla01, ABF08] tool. In addition, the opponent concept is very similar to the typical environmental assumptions made by ProVerif in order to verify security properties, and correspondence properties defined on event logs are directly mapped to similar properties in the ProVerif model.

A formal proof of the soundness property for this abstraction step can be found in [BFGT08] along with the formal syntax and semantics of F and of the target pi-calculus language, and the formal definition of the abstraction function.

Practical experience with the combined use of fs2pv and ProVerif has shown that this approach can be applied with success to automatically prove secrecy and authentication properties on real protocol implementations. However, the outcome of the ProVerif tool changes in a rather unpredictable way when making small changes in the source code, which requires careful tweaking of the implementation, before such results can be obtained. Nevertheless, implementations of some standard Web Services Security protocols have been developed and formally verified by translating them to pi-calculus and by verifying the output model with ProVerif [BFG06, BFGT08].

3.3. ASPIER

The ASPIER model extraction and verification tool [CD09] operates on protocol implementation code written in C and uses a model checker for verification. ASPIER works in a way similar to the previous approach, but the starting language and the verification engine are different.

Since the C language has no standard formal semantics, C protocol programs are first translated into a C-like language called the ASPIER concrete protocol language, which has formal semantics. This formalized version of the protocol code is close to the original program and includes abstract models of cryptographic operations. More precisely, the ASPIER concrete language divides variables into two disjoint classes: message variables and numeric variables. While on numeric variables ASPIER operates just like the C language, message variables can only be manipulated in ASPIER by specification state machines (SSM). The latter abstractly specify the behavior of corresponding library functions that are called in the C code for message processing. For example, if the C code calls a function $symenc(x, k)$ in order to encrypt message x using key k, when the C code is translated to the ASPIER language each call to this function is substituted with an instance of a corresponding SSM which simply computes the result as an algebraic expression according to the Dolev–Yao modeling style.

The ASPIER language is a simplified version of C, where, for example, features such as floating-point numbers, bitwise operators and some features of pointers are absent. This entails several restrictions on the C code that can actually be fed to the tool.

The formal semantics of the ASPIER language expresses the meaning of a program by a Labeled Transition System (LTS) where each state may include multiple components, corresponding to multiple concurrent threads, each one executing a protocol role. The same LTS also tracks the evolution of the attacker's knowledge, modeled according to the Dolev–Yao style as a set of messages known by the attacker. Thus, the formal semantics of the concrete ASPIER language incorporates environmental assumptions.

Events are defined onto an LTS execution trace as statements labeled by the identity of the thread that executed the statement. Properties such as secrecy and authentication are formally expressed as predicates on event traces.

Since a protocol coded in the ASPIER language results in an infinite-state complex LTS, a second translation from this code to a more abstract model is performed, in order to facilitate formal verification. By using a model checking verification technique, the approach is inherently limited to verification of a finite number of concurrent sessions.

The abstract model to which the concrete ASPIER program is translated is built using predicate abstraction [GS97a]. All numeric variables are substituted by predicates about their values. According to how such predicates are built and to how many predicates are used, different levels of abstraction can be obtained.

The abstract model has a formal semantics defined in the same way as for the concrete ASPIER language, by LTSs defined over the same alphabet. This implies that properties can be defined in exactly the same way on the concrete and abstract models. A soundness theorem proves that the abstract model LTS simulates the concrete model LTS, thus finally proving that if M is the concrete model, $\alpha(M)$ is the corresponding abstract model, and ϕ is a security or authenticity property expressed as a trace property, then $\alpha(M) \models \phi \Rightarrow M \models \phi$.

The model checker used for proving properties on the abstract model (COPPER) is integrated with a CEGAR (Counter Example Guided Abstraction Refinement) approach. This means that initially a quite abstract version of the program is built, using few predicates. Since this is an over-approximation of program behavior, it is possible that the model checker will find out false property violations. In this case, the model checker produces a counterexample, which is an error trace. This trace is replayed on the concrete model, in order to discover if it is a real error or not. If the error is false, the counterexample is used to automatically build a new predicate that is added to the existing predicates, thus obtaining a more refined abstract model. The predicate construction procedure guarantees that the refined model will not present the same counterexample. After successive refinements, either the property will be proved, or a real counterexample will be found or the model will grow so much to become intractable.

This method was applied on the OpenSSL implementation. From the whole protocol code, only the core handshake implementation code was manually extracted, and some further manual editing was necessary before the code could be fed to the model extraction tool. Secrecy and authenticity properties for a bounded number of concurrent sessions could be finally proved on this implementation.

4. Security-refined types

Security-refined typing is a further means to ensure correctness of implementations. Essentially, an existing implementation is enriched with refined types that formally express the security requirements of the application. If the application type-checks against these security-refined types, then the application is proved to fulfill some security properties.

This technique has similarities with the model extraction approach, because the user provides the protocol implementation enriched with annotations. However, in this case annotations are not just a way to direct model extraction. They could be considered as the formal abstract model itself, built by the security-refined typing activity. Type checking implements the formal verification strategy, by ensuring conformance between model and implementation. Furthermore, the techniques associated with security-refined type checking present significant peculiarities that set them apart from the model extraction approaches that were presented in the previous section. For this reason they are treated separately. For example, when using security-refined type checking each function can be type-checked in isolation, making verification modular. Also, the way security properties are expressed is tightly coupled with the security-refined types paradigm.

Two main works addressed correctness of security protocol implementations by means of refined types [AS05, BBF+11]. They are detailed in the next sections, and summarized for reference in Table 2.

Table 2. The main security-refined type checking approaches

References	Programming language	Type Checking architecture	Model kind	Security properties	Formal soundness
[AS05]	Java	Jif	Symbolic	Information flow (Secrecy)	No
[BBF+11]	F#	RCF	Symbolic	Authentication, Authorization, Information flow (Secrecy)	Yes

4.1. Jif types for Java

In [AS05], the Jif (Java Information Flow) framework is exploited to ensure an information flow property (which ultimately implies a kind of secrecy) on a multi-player on-line poker protocol that does not require a trusted third party (TTP).

The Jif framework has primarily been conceived to express and verify by typing standard information flow properties on Java code. The work in [AS05] represents the first documented experiment of applying security-refined types to a real security protocol implementation.

The work consists of three main steps: (i) Java application development as a monolithic application embedding all protocol actors; (ii) application tweaking so that it fits into the Jif framework; (iii) Jif application transformation in order to split protocol actors.

In the first step, the on-line poker protocol is developed as a standard Java application. Each actor is indeed implemented as an instance of a "Player" class, which in fact makes each actor's code already neatly split from each other. However, all actors run together as part of the same application.

In the second step, the code is refactored up to the point where Jif types can be assigned to the code. This refactoring step required significant effort, and was far from linear, calling for several refactoring iterations. Unfortunately, these non-trivial refactoring iterations are not formally proved to preserve the semantics of the original Java application. Indeed, some refactoring implies much finer handling of Java run-time exceptions, since an uncaught exception could in fact leak some information. Thus, this step breaks the formal link between the original application and the verified one, although a good confidence about secrecy preservation is still achieved informally.

In the third step, the code is finally modified to make each protocol actor run on a different process, like a real distributed application.

On one hand, the number of modifications required by step (ii) suggests that the Jif framework is probably not able, in the current form, to handle arbitrary already developed Java applications. On the other hand, the refactoring made on the original code gives an insight on which programming patterns are most effective in tracing information flow and improving understandability of a program, and can be used as a reference in future application development.

The case study in [AS05] mainly focused on proving that the cards held by a player remain secret until the end of a game. This can be modeled as an information flow property, where a policy enforces that no one except the holder of the cards can read their value.

In Jif, policies are expressed through labeled Java types. A Jif labeled type takes the form of

$$type\{Owner \rightarrow Readers\}$$

where *type* is a standard Java type, and $\{Owner \rightarrow Readers\}$ is the label expressing the policy for that variable, which respectively names the owner of the data and the allowed readers. For example

```
boolean{Alice → Bob}
```

is a valid Jif type where Alice owns values of that type and Alice and Bob can read them.

As hinted above, if a Jif-labeled program type checks, it means that the policies specified in the program are enforced. In the online poker protocol example, it means that the cards remain secret.

In [AS05] cryptography is treated symbolically, by giving axiomatic annotations to the Java methods implementing the cryptographic algorithms. As in other symbolic approaches that abstract computational hardness problems of cryptography away, soundness is proved assuming the cryptographic implementations behave according to a symbolic algebra.

4.2. F7 types for F#

In [BBF+11], a refinement type framework named F7 is developed and applied to express and verify security properties of security protocol implementations written in F#. Essentially, while a standard F# program is composed of interfaces declaring input and output types of functions and their implementations, the F7 framework adds a layer of more refined interfaces, where pre- and post-conditions of such functions can be more finely specified, in the form of refined types.

A refinement type is a type of the form $x : T\{C\}$ where x is a value of type T, such that the formula C holds, where x is bound in C. For example, given the type *int* of all the integers, it is possible to define the refinement-type of the natural numbers by $x : int\{x \geq 0\}$.

A function type $T1 \rightarrow T2$ can be refined as $x : T1\{C1\} \rightarrow y : T2\{C2\}$, where x is bound in $C1$ and $C2$, and y is bound in $C2$. Type checking a function of this type means assuming that the conjunction of the formulas of the input values hold, and proving that all possible return values of that function satisfy the formula refining the output type. That is, a refined type checking function will always satisfy its post-conditions, provided the pre-conditions were true. For example, the `inc : int → int` function that returns the given argument incremented by one can be refined to

$$\texttt{inc} : \texttt{x} : \texttt{int} \rightarrow \texttt{x}' : \texttt{int}\{\texttt{Suc}(\texttt{x}', \texttt{x})\}$$

where the predicate $Suc(a, b)$ indicates that a is the logical successor of b. The implementation of the function will refinement-type check if the returned value is in fact the successor of the input value, and refinement-type checking will fail otherwise (while the function may still successfully type check with non-refined types).

As illustrated by the example above, since in a refined function type a variable is bound in all subsequent appearing formulas, it becomes possible to express the relation existing between input and output of a function, or the constraints that must hold among the input parameters of a function.

By defining security-related predicates, and by using refinement types in a particular pattern, it is possible to express some security properties of a program, such as secrecy and authentication. By refinement type checking its functions, it is possible to prove that such security properties hold in the implementation.

Refinement type checking as described in [BBF+11] uses type inference algorithms to carry formulas from one statement to the next one, mapping F# code into the RCF concurrent λ-calculus language. The F7 type checker can handle a significant portion of the F# language, not imposing any fundamental limitation to the code that can be analyzed. However, experience showed that the refined type inference algorithm works best with specific programming patterns, failing in other cases: although this limitation does not reduce the expressiveness of the implementation, it narrows the number of existing applications that can be analyzed without modification. Moreover, F# is not a very popular language among common developers, which makes it more challenging to find widely deployed applications to validate and test scalability of the approach.

The work in [BBF+11], mainly focuses on the classical secrecy and authentication properties in the Dolev–Yao symbolic model.

Secrecy is expressed by using an information-flow approach, although not expressed in classical information-flow terms. Essentially, if a special $Pub(\cdot)$ predicate can be proved to hold on some data, then these data are "public" (low confidentiality), and thus they can be made available to the attacker through the network; otherwise, they are "private" (high confidentiality). Hence, like in classical information-flow, the sets of low and high confidentiality data are disjoint.

Classification of data is always possible, because no special predicate has to be proved for the data to be classified. Conversely, encryption functions fulfill the role of de-classifying data: their input (plaintext) is any data (crucially including private data for which $Pub(\cdot)$ cannot be proved); and their output (ciphertext) is some data for which no relation with the input is expressed and on which the $Pub(\cdot)$ predicate holds. Leveraging the modularity of the assume-guarantee approach, the encryption functions are not refinement type checked, rather they are assumed to behave as specified, and their refined interfaces express the Dolev–Yao model.

Since interaction of the application with the attacker only happens by means of communication channels, the API implementing such channels is carefully refined. In particular, functions sending data over the network will only accept public data, so that only data for which an explicit proof of low confidentiality exists will be given to the attacker; conversely, the $Pub(\cdot)$ predicate will hold for any data returned by functions reading data from the network.

Authentication can be obtained by defining authentication related predicates and using them with a particular pattern, which in fact makes them equivalent to correspondence events often used to express authentication with agreement on data.

In [BBF+11], it is assumed that authentication can be achieved through MAC (Message Authentication Code) functions. In particular, encryption, although expressed in a Dolev–Yao style and thus being injective, does not give any authentication assurance, to take into account malleable encryption schemes.

Essentially, the pattern used for authentication consists of defining a special predicate, *MACSays*, such that it can be considered an event. Basically, this predicate must be carefully defined to hold if and only if the application is in a "valid" state where data can be transmitted, and correct data have been computed. The MACing functions only accept data for which the *MACSays* predicate can be proved to hold. Conversely, if MAC verification succeeds, then the *MACSays* predicate holds for the MACed data. Finally, by the definition of *MACSays*, it follows that if MAC verification succeeds, then authentic data have been received (or the MACing key was compromised, in case compromised actors are considered). Like for secrecy, the implementation of the MACing and MAC verification functions is trusted, and the interfaces of these functions express the Dolev–Yao model.

A soundness theorem proves that if a security protocol implementation that uses the network and cryptographic libraries as annotated above type checks, then it is safe for secrecy and authentication against a Dolev–Yao attacker. Here, instead of directly referring to the abstract and concrete models $\alpha(M)$ and M, the proof focuses on the application and the opponent types. The attacker is given a special *Un* type, which is a super- and a sub-type of any other type, so that it accounts for all the behaviors typical of a Dolev–Yao attacker.

Since refinement types can be naturally recursive, they offer an effective way to model security protocol features that are inherently recursive and open ended such as certificate chain checking, or concatenated protocol sessions, where secrets agreed in a previous session are used in a subsequent session as a basis to agree new session secrets [BBF+11, BCPM+09, BFG10]. These protocol features cannot usually be verified by model checking, and even state-of-the-art automatic theorem provers such as ProVerif often fail in proving this kind of properties.

5. Automated code generation

Code generation fits the model-driven approach for software development, where a high-level model is first developed and analyzed (e.g. in order to prove that it fulfills some desired properties, such as the security properties of a security protocol). When good confidence on the correctness of the high-level model has been achieved, the model is refined into a concrete implementation, written in a programming language. This second step can be automated, using automatic code generation tools. Since the high-level model does not include all implementation details, further details have to be provided by the user during code generation.

The relationship that should exist between the abstract model and the concrete implementation can be informally stated as "The generated implementation has the same security properties that were proved for the starting abstract model". It can be formalized in the same way as already shown for model extraction.

More precisely, code generation can be formally described as a refinement mapping $\rho(\cdot)$ that maps an abstract program model M_a with some associated implementation choices C onto a corresponding $\rho(M_a, C)$ concrete program model, and an abstract property ϕ_a onto a corresponding concrete property $\rho(\phi_a)$. We say that $\rho(\cdot)$ is a sound refinement with respect to an abstract property ϕ_a if, for all implementation choices C that satisfy some implementability constraints

$$M_a \models \phi_a \Rightarrow \rho(M_a, C) \models \rho(\phi_a). \tag{2}$$

In practice, if a security property ϕ_a has been proved on the abstract model and $\rho(\cdot)$ has been proved sound with respect to ϕ_a, then $\rho(\phi_a)$ is proved on the concrete model.

5.1. Overview of existing approaches

Two major features can be considered when analyzing a code generation approach for security protocols: (i) the existence of a formal proof of soundness for the code generation function $\rho(\cdot)$; (ii) the extent to which implementation choices (such as for example protocol message formats) can be freely specified by the user, so that the generated implementation can interoperate with third party existing implementations.

Table 3. The main code generation approaches

References	Abstract Model language	Programming language	Model kind	Security properties	Formal soundness	Interop
[ABB+10] [BCK+08]	Custom notation	C, Java, LaTeX	Computational	ZK-PoK	Yes (PoK only)	N/A
[BCPM+09]	Custom notation	F#, Ocaml	Symbolic	Secrecy, authentication	Yes	N/A
[CJK05]	Casper	C#	Symbolic	Secrecy, authentication	No	Yes
[GBS+08]	ASM	Java card	Symbolic	Custom	Yes	Yes
[HOP03]	Casper + JML	Java	Symbolic	Secrecy, authentication	No	Yes
[KOT08]	XML	C	Symbolic	Secrecy, authentication	No	No
[MEK+10]	Custom notation	C++	Computational	ZK-PoK	No	N/A
[PSD04]	Spi calculus	Java	Symbolic	Secrecy, authentication	Yes	Yes
[SPP01]	Custom notation	Java	Symbolic	Secrecy, authentication	Yes	N/A
[TH04]	Spi calculus	Java	Symbolic	Secrecy, authentication	No	No
[BBH12]	Proverif input	Java	Symbolic	Secrecy, authentication	No	Yes
[O'S10]	LySa	Java	Symbolic	Secrecy, authentication	No	No

In the literature, several automated code generation approaches for security protocols have been proposed. Table 3 summarizes the main features of the main code generation approaches that have been proposed so far.

Many of them do not address all the considered features [TH04, KOT08, CJK05, HOP03, BBH12, O'S10]. For example, in [KOT08] the focus is to provide a fast C source code security protocol generator, so that mobile devices can replace or update current implementations on-the-fly. In that work the novelty and main contribution stands in the realization of a fast protocol generator. However, formal soundness of the generated implementations is not considered, and although the user can loosely specify the intended security properties on the starting model, no means are provided to verify them.

As another example, in [HOP03] a formal CSP model of a security protocol used by smart cards is derived and refined from its informal description, and then a Java implementation of the refined model is manually derived. Finally, the source code is manually enriched with JML [LBR06] annotations to check whether the implementation satisfies its specification. Although all the steps are performed manually, the work in [HOP03] documents a general way to refine formal models of security protocols into running implementations. Unfortunately, this approach does not come with a formal proof ensuring that the starting CSP model matches the JML annotations on which some formal analysis is finally done.

The code generation approach described in [O'S10] does not provide a formal soundness proof, but it is originally combined with a corresponding model extraction tool [O'S08] already mentioned in Sect. 3. The code generation tool, named Hajyle, generates a Java implementation from a protocol model expressed in a process algebra named LySa. The Java code can then be modified by the programmer and converted back to a LySa model, by the model extraction tool named Elyjah, to be formally verified again.

AGVI [SPP01] is a framework for the design and implementation of custom security protocols starting from their security requirements. It was the first documented approach of security protocol synthesis and verification. Given some security requirements, AVGI generates a family of custom security protocols that fulfill those security requirements. Then the user can generate implementations of a chosen custom protocol. The framework lacks a formal proof showing that the generated code preserves all the security properties that hold in the protocol model. Clearly, with a protocol synthesis approach, interoperability of the generated implementation cannot be discussed, because each synthesized protocol is custom, and so there can be no existing implementation to interoperate with.

Recently, protocol synthesis has become an emerging trend with Zero-Knowledge Proof-of-Knowledge (ZK-PoK) protocols. A ZK-PoK protocol allows a verifier to be convinced by a prover that a fact is true (PoK), without getting any further knowledge except the fact itself (ZK). For example, users might want to get authorization to access a resource from a server, without disclosing their identities to the server. Two main approaches have been proposed, CACE [BCK+08, ABB+10] and ZKPDL [MEK+10]. Despite some differences in implementations, essentially both approaches start from a ZK-PoK specific language, in which the intended security properties are specified. A compiler then transforms the given specification into a Σ-protocol, which is in turn translated into implementation code. A fine comparison of their implementation differences can be found in [BKSS10]. In [ABB+10] a certifying compiler is presented, which means that the generated code comes with a certificate of correctness that can be proved in the Isabelle/HOL interactive theorem prover. Currently, the soundness proofs only cover the PoK property for the generated implementation, while the ZK formal proof is left for futurework.

Being protocol synthesis approaches, neither CACE nor ZKPDL can be evaluated about the interoperability of the generated implementation, as discussed above. Indeed, none of them takes the problem of handling a custom marshaling layer into account.

In [BCPM+09], a similar protocol synthesis approach is developed to address multi-party session protocols. A custom notation is defined to describe the graph of protocol actors and the protocol control flow, expressed as a sequence of asynchronous messages exchanged among the protocol participants. From this graph representation, a custom compiler infers the cryptographic functionalities needed to achieve the committed security requirements and it generates an API in F# with a full protocol implementation. The generated implementation is also enriched with refinement types. In this way, assuming the refinement types added by the compiler correctly express the intended security properties, a refinement type checker can be used to check whether the generated code satisfies the required security properties.

The work in [GBS+08] starts from an abstract state machine (ASM) model of the Mondex [Inc] protocol and generates an interoperable Java Card implementation by refinement. The generated implementation is proved to be correct by showing that it is a sound refinement of the original ASM, where the custom Mondex security requirements are proved to hold. By sound refinement, it is intended that any possible behavior of the Java Card implementation can be mapped to a behavior of the verified ASM. Technically, the sound refinement is proved by simulating symbolic Java execution steps in a calculus expressed in the KIV interactive theorem prover [Ste04]. The main drawbacks of the approach lay in the fact that most of the proofs require manual interaction, and thus a high level of expertise. Moreover, the methodology and results are not totally general, rather specifically tailored to the analyzed Mondex case, which makes the results not so reusable in other works.

The work in [PSD04] presents a code generation framework, called Spi2Java, which starts from spi calculus specifications of security protocols, and semi-automatically generates Java implementations of the protocol roles. The framework comes with a formal proof of soundness related to symbolic models [PS10, Pir10], and the user is allowed to provide enough refinement details, so as to enable the generation of an implementation that interoperates with third party existing implementations [PPS12]. The framework has been used mainly to implement classical client-server security protocols, where secrecy and authentication are proved under a Dolev–Yao attacker. Among the limitations of this approach it is worth to point out that only the protocol core logic is automatically generated, while the functions for serialization and de-serialization have to be written manually. Hence, soundness is ensured under the assumption that serialization functions have been implemented correctly. Another limitation is that the language for specifying the input abstract model is restricted to a version of the spi-calculus where each protocol role is a sequential program. This does not prevent to have multiple concurrent sessions of the protocol, but each role in each session is restricted to be a sequential program. It is also important to note that the spi-calculus comes with a fixed number of cryptographic primitives. Although this number could be extended in principle, the current implementation can only be used for those protocols that use the provided primitives.

Some of these limitations, i.e. the restriction to sequential code and the fixed set of built-in cryptographic primitives, have been overtaken by Expi2Java [BBH12, Bus11]. Expi2Java extends the Spi2Java approach by modeling cryptographic operations by constructors and destructors, using the ProVerif input language [Bla01]. This facilitates the introduction of new cryptographic primitives. The transformation from the extended formal language to Java has been formalized with the language of the Coq [BC04] proof assistant, which enables the development of machine-checked proofs. However, up to now a formal soundness proof has not yet been developed for Expi2Java; at the moment, a simpler theorem about well-formedness of the generated code has been proved [BBH12, Bus11].

5.2. Formally linking abstract models to generated implementation code: the Spi2Java example

This section describes in more details how the formal link between an abstract protocol model and the corresponding implementation code written in a programming language can be defined and proved sound. To this extent, the Spi2Java framework is taken as a representative example, because it comes with a formal proof of soundness, and its implied methodology is general enough, and not tailored to a specific protocol.

Taking into account that many classical security properties, including secrecy and authentication, are safety properties that can be defined as predicates on single traces (or protocol executions), the soundness property expressed by (2) is a corollary of a trace refinement relation between the implementation code and the abstract

model. A trace refinement relation means that each protocol execution trace that can be executed by the concrete program, can also be executed by the abstract model. So, if the concrete program can execute a trace that represents an attack (i.e. that leads to the violation of a security property), the same trace can also be executed in the abstract model, and thus the potential security violation can be detected during the formal analysis of the abstract model.

The approach taken in [PS10] for proving the soundness of the spi calculus to Java transformation is to prove that a weak simulation relation exists between the generated Java code and the corresponding spi calculus model. Briefly, a weak simulation relation binds the transitions between external states of an abstract process to the transitions between external states of a concrete process, but each process is still allowed to perform any internal step in between two external states. This form of relation is needed, because a Java program will in general compute many internal steps before exhibiting the next external state that will match a corresponding spi calculus state. More details about the weak refinement referred to here can be found in [Sch05]. A weak simulation relation implies trace refinement.

The refinement mapping ρ transforms a spi calculus model into a Java program, i.e. $\rho : Spi \rightarrow Java$, where *Spi* is the set of spi calculus models that can be translated into Java,[3] and *Java* is the set of the generated Java programs. The implementation choices do not occur explicitly as arguments of ρ. Instead, they are implicit parameters of ρ.

An LTS is defined for spi calculus, and one for Java, so that the two systems can be formally related by matching traces. The Java LTS is built according to a formal semantics that has been defined for a subset of the language in [BPP03]. This subset includes all the language features used by the generated code, so that the choice of this restricted language semantics is not restrictive for this application.

A generic spi calculus state can be written as $P\sigma$ where P is a spi calculus process expression and σ is a (possibly empty) substitution that binds variables in P. Using this representation for processes, a generic state transition can be written as

$$P_1\sigma \xrightarrow{\mathcal{L}} P_1'\sigma\sigma'$$

where $\mathcal{L}$ is the transition label, which indicates an input of data on a channel or the output of data on a channel or an internal step τ.

In the LTS for the spi calculus, all states are defined as external.

An LTS for a sequential subset of Java which is enough to implement spi calculus processes within the Spi2Java framework has been defined in [PS10]. In order to relate the Java behavior to the spi calculus behavior, the Java LTS uses the same abstract labels used for the spi calculus LTS. Let j be the Java code that is going to be executed, *JavaVar* the set of identifiers that can be used as variables in Java programs, and *JavaObj* the set of object identifiers. Then a generic state (j, Val, Res) is defined by the code j that is going to be executed, plus a partial function $Val : JavaObj \rightarrow SpiTerm$, mapping each Java object that has been created by previously executed code to the spi calculus term the object is implementing, and a partial function $Res : JavaVar \rightarrow JavaObj$ mapping each Java variable in the scope of j to the referenced Java object. For example, $Val(o) = \{M\}_N$ means that the Java object o implements the $\{M\}_N$ spi calculus term; $Res(\texttt{var}) = o$ means that the Java variable `var` references the object o. The intended invariant that should hold is

$$Val(Res(J(M))) = M\sigma$$

where σ is the variable substitution in the corresponding spi calculus process and J is a bijection that gives the name of the Java variable for term M, by mangling it. That is, the object referenced by the Java variable $J(M)$ must implement the $M\sigma$ term, which is the run-time value of the M spi calculus term. A Java state (j, Val, Res) is defined as external iff $j = \rho(P)$ for some spi calculus process P. The transitions of the form

$$j, Val, Res \xrightarrow{\mathcal{L}} j', Val', Res'$$

take from one generic state to another, following an abstract operational semantics for the Java language, as given in [BPP03].

[3] Only a strict subset of all possible spi calculus models can be translated into Java by the Spi2Java framework, namely those spi calculus models that can be translated into a type-safe Java program.

The simulation relation S, that relates external spi calculus states to external Java states, is formally defined as

$$S(P\sigma, (j, Val, Res)) \quad \Leftrightarrow \quad j = \rho(P) \ \wedge \sigma_{|fv(P)} = Val \circ Res \circ J_{|fv(P)} \ \wedge \ \sigma \supseteq Val \circ Res \circ J$$

Informally, a spi calculus state $P\sigma$ and a Java state (j, Val, Res) are S-related, iff the Java state is external, and the invariant $M\sigma = Val(Res(J(M)))$ holds. Note that it is required that the domain of $Val \circ Res \circ J$ contains all the free variables in P (denoted by $fv(P)$); however some compound terms may not (yet) be stored in Java memory: it is enough to require that the invariant holds for the already built terms, which are stored in Java memory.

The main soundness theorem in [PS10] relies on the existence of a trusted Java library, called SpiWrapper, that implements the behavior of spi calculus terms and processes in Java. For example, it is assumed that a spi calculus pair is implemented by a Java class "Pair" that offers the `getRight()` and `getLeft()` methods, so that the pair splitting process can be implemented; or a spi calculus channel is implemented by a Java "Channel" class, offering the `send()` and `receive()` methods to implement the input and output spi calculus processes. In [PS10], a full formal definition of the intended semantics of the SpiWrapper library is given. As shown in [PS10], the correctness of the classes implementing the SpiWrapper library can be proved using the same Java semantics used to prove the simulation relation. Of course, the proof assumes that the underlying Java libraries used by the SpiWrapper layer for communication and for cryptography behave as expected.

The main soundness theorem (code generation soundness) is finally stated as follows: If the SpiWrapper library behaves as formally specified, then, for any external state (j, Val, Res) of the generated Java program and for any spi calculus process state $P\sigma$,

$$S(P\sigma, (j, Val, Res)) \ \wedge \ j, Val, Res \xrightarrow{\tau}{}^{*} \xrightarrow{\mathcal{L}} \xrightarrow{\tau}{}^{*} j', Val', Res' \quad \Rightarrow \quad P\sigma \xrightarrow{\mathcal{L}} P'\sigma' \ \wedge \ S(P'\sigma', (j', Val', Res'))$$

i.e. if the simulation relation S holds between the spi calculus process state and the Java program state, and if the Java program can evolve into a new external state, then the spi calculus process can evolve into a new external state too, and the new external states are still related by the simulation relation S.

In other words, every step that can be done by a generated Java program, can also be done in the starting spi calculus model. As a corollary, it follows that every trace (sequence of steps) that the Java program can execute, is a valid trace in the original spi calculus model, finally implying that every (trace) security property that is proved on the spi calculus model, holds in the generated Java program too.

6. Safe abstraction of data formatting functions

When dealing with techniques that formally link program code to corresponding abstract protocol models, a common issue is knowing if certain code aspects or parts are relevant or not for proving a given security property. If it is possible to prove that a given code aspect is not relevant for some security properties, that aspect can be safely abstracted away during model extraction. Similarly, it can be safely added during code generation, without compromising the validity of proofs related to the abstract model. In both cases, the final effect is a simplification of the models to be formally analyzed, thus facilitating formal automated verification.

A common intuition is that data formatting functions should not be essential to the security of the protocol, and thus it might appear natural to try to abstract them away during verification, to get smaller, and thus easier to verify and more understandable models.

However, arbitrary implementations of such data formatting functions can still break security properties, in very trivial ways. For example, an incorrect implementation of a marshaling function could unwillingly leak secret data, or the function that encodes some data before applying a hash function to them could erroneously transform part of the data (e.g. a nonce) into a constant, thus enabling replay attacks. These errors do not necessarily infringe interoperability, so they may be difficult to discover by testing.

These considerations also apply to the code generation approach, when the user provides custom implementations of such data formatting functions.

Hence, some researchers have focused on formally finding and proving sufficient conditions under which data formatting functions cannot break security properties, so that they can be safely abstracted when analyzing the formal models [Pir10, PS12, KR06].

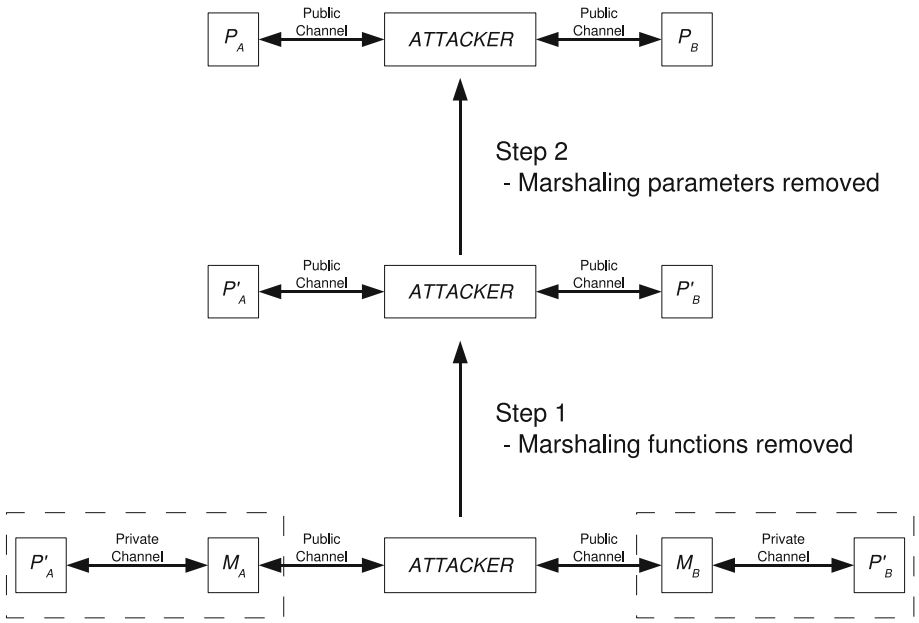

Fig. 2. The 2-steps simplification procedure leading from a refined model including marshaling functions to an abstract one

The most comprehensive results are reported in [PS12], where data formatting functions are partitioned into two categories. (i) "marshaling" functions, that are functions formatting data in packets to be sent over the network, e.g. by adding a header with content type, or packet length. And (ii) "encoding" functions, that are functions formatting data before they are processed by cryptographic operations such as encryption or hashing, e.g. functions that add a padding, or concatenate several items to be processed.

This partition is motivated by a difference in criticality for marshaling and encoding functions. Intuitively, less constraints are required on marshaling functions, because their input data are already protected by cryptography and would be ready to be transmitted over the network anyway; the data are simply in a format which is unsuitable for the chosen protocol. Conversely, encoding functions operate on unprotected data, so much stricter constraints on their implementation correctness are expected to be required. Besides this intuition, formally finding the exact definition of these constraints is a useful step for the formally based approaches considered in this survey.

In [PS12], sufficient conditions for both marshaling and encoding functions abstraction are found with a 2-steps simplification procedure. For example, the 2-steps procedure for marshaling functions is depicted in Fig. 2. The most abstract model at the top of the figure is the one where marshaling functions are completely neglected, only the protocol logic P_x of each actor is represented. In a code generation approach, this model would be the one on which formal verification is done, and from which code generation is started. Conversely, the most refined model at the bottom explicitly includes the marshaling functions M_x, and each protocol logic P'_x is a refined version of its corresponding abstract P_x, where marshaling parameters and interaction with the marshaling functions are taken into account.

The first simplification step soundly removes the model of the marshaling functions. This step is safe for any security property that can be expressed as a safety property on protocol traces, like for instance weak secrecy and authentication.

In the second step, the fault-preserving simplifying transformations (FPST) originally described in [HL01] are used to bring each refined P'_x back to its abstract version P_x. A FPST is a function that transforms a refined model into a more abstract one, preserving secrecy and authentication attacks. That is, if the abstract model is safe with respect to secrecy and authentication, then the refined model is implied to be safe too.

The final result is that in order to abstract marshaling functions away, no assumptions are requested about injectivity, nor about implementation correctness; instead, it is only required that the implementation of encoding and decoding functions cannot access any application data, except the ones that are given as input by the application. This condition can be checked by standard static information-flow analysis techniques. The consequence of this result is that, if such information flow properties are satisfied on implementation code, then even erroneous specifications or implementations of encoding schemes cannot be more harmful than a Dolev–Yao intruder is.

For what concerns instead abstracting away encoding functions applied to key material or to data on which cryptographic operations are applied, much stricter constraints are required. When verifying secrecy, it is required that the implementation of the encoding and decoding functions is correct w.r.t. their specification. If instead authentication is being verified, an additional constraint requires that all actors agree on the same encoding parameters too.

7. Discussion and future directions

As can be seen in the reference tables proposed in this work, most state-of-the-art research providing automated formal verification of security protocol implementations focuses on symbolic models. Certainly, automatic verification in the symbolic domain is a more mature research field, compared to automatic verification in the computational domain, which explains why most of the surveyed approaches deal with symbolic models. Considering computational models would bring advantages, because the formal proofs would rely on more realistic assumptions, at the cost of increased complexity. Nevertheless, before considering the possible research shift towards computational models, it is worth pointing out the main current limitations that are still present at the symbolic level.

The recent purely logical flaw discovered on the renegotiation feature of the TLS protocol could have been fully captured by state-of-the-art symbolic verification techniques. Notoriously, this attack came out after several papers claimed verification of "the TLS" protocol [MSS98, OF05, Pau99, TCCD06, Vig06, WS96], even down to the implementation level [BCFZ08, CD09, Jür09]. Basically, the flaw went undiscovered for years because it exploits the standard renegotiation feature of TLS, which has been widely neglected during formal verifications of the protocol. This happened because usually the verified models represented a simplified version of the protocol, in order to cope with the verification complexity. As a result, advanced features such as renegotiation were neglected. Even verified implementations did not support renegotiation, thereby allowing the flaw to go undetected.

In practice, this experience shows how most of the approaches presenting case studies focus on soundness of an incomplete version of the protocol, leaving room for logical attacks that lie in the unverified parts of it. Thus, a significant challenge for the symbolic verification of protocol implementations, is to be able to scale up to real, fully-compliant implementations. In order to achieve these results, much work has to be done on the composability and scalability of formal methods and implementation verification techniques.

On the computational side, sound automatic verification of security protocol models has made significant progress recently. For example, an automatic tool called CryptoVerif has been recently developed [BP06], which takes a spi calculus-like description of a security protocol, and can automatically prove strong secrecy and authentication properties in the computational model.

It is expected that as computational verification techniques are further refined on the models, more approaches will be developed to link implementations to these computational models. In fact, preliminary results in this respect have already been published both in the model extraction and in the code generation domains. For example, the fs2cv tool [BCFZ08] extracts CryptoVerif models from F# protocol implementations, to get a computational proof of correctness. Conversely, the approach described in [CB12] starts from verified CryptoVerif models to automatically generate ML implementations. These results are here considered as preliminary, because a formal link between the implementation code and the model is currently left as future work.

More ongoing work exists and is being developed in this field. For example, a computational soundness proof for refinement types in F# was developed [FKS11], and a similar approach applied to the Java language exists [KTG12]. However, none of these approaches has been applied in large case studies yet.

It is worth mentioning that there are mainly two ways that this shift towards computational models can be approached. One way is to set the proof directly in the computational model, by taking into account the computational complexity of cryptographic problems, and trying to use well-established techniques in the computational cryptography domain, like game theory. The CryptoVerif tool is an example of a tool built in such a way, and so are the code generation and model extraction approaches based on it. The other way is to prove a soundness refinement theorem between a symbolic model and its computationally refined counterpart, so that formal verification of a security property on the symbolic model implies that a corresponding property holds on the computational model [BMU10].

In the model extraction approach, a foreseeable research trend could be moving from functional languages like ML or F#, where formal semantics and type safety are easily achieved, down to imperative andobject oriented

languages such as C or Java, where the model extraction techniques have to be refined, in order to abstract away lower level problems such as memory and type safety, without compromising the security proofs. Preliminary attempts in this direction are shown in [DGJN11, PM12, AGJ11, AGJ12]. In [DGJN11], C implementations of security protocols are annotated with semantic information, so that a general purpose C verifier can prove security properties in the Dolev–Yao model. Currently, a custom cryptographic library has to be used, and only small fragments of ad-hoc written code can be verified. The work in [PM12] builds on [DGJN11] by adding a layered refinement technique: abstract protocol narrations are first linked to binary data formats and finally to the C functions implementing the protocol. This technique helps making the approach more scalable, so that larger chunks of ad-hoc written code can be verified. In [AGJ11, AGJ12], annotated implementations in C are symbolically executed in order to obtain a ProVerif [AGJ11] or a CryptoVerif [AGJ12] model that can be verified. Both translations from C to ProVerif and to CryptoVerif are proved sound, meaning that all attacks present in the C program are preserved into the extracted models. This technique is just a first step in the direction of automatically analyzing real C implementations of security protocols. The main limitation of the current approach is that it can only handle protocols that have a single execution path, without branches and loops. Of course, this applies only to ad-hoc written code or to finite variants of normally unbounded protocols (where loops are unrolled to a finite number of iterations). In general, state-of-the-art approaches targeting common implementation languages are not yet powerful enough to reliably verify widely deployed implementations of security protocols such as OpenSSL or OpenSSH without making any modification to their source code, and at the same time taking into account all protocol features, including re-keying or error message handling.

Moreover, in the model extraction approach the starting point is the source code, however, the verification results refer to the extracted model. In particular, when a proof of correctness fails, the verification tool gives feedback on the model. In order for this feedback to be useful, it must be ported back to the source code from which the model was extracted, so that the source code can be patched, and verification is run again on the updated model. Research work addressing this engineering issue will make these tools more user friendly, potentially broadening the scope and adoption of these methodologies.

On the code generation side, the steep learning curve to handle abstract models makes this approach usually not affordable by non experts. A possible way to overcome this issue is to make abstract models closer to real programming languages. Preliminary results in this direction, which currently lack formalization, are presented in [APPS11], where Java is directly used both as a modeling and as an implementation language.

In addition, in order to keep the soundness proofs simple, the generated code is usually close to a one-to-one mapping with the starting model. While this can help to generate clean and understandable code, it may also lead to inefficient implementations, that the user cannot improve without losing the soundness result. New fundamental approaches geared towards letting the user interact with the code generator, or taking into account code optimization would mitigate this issue.

Finally, many methods focus on classical security properties such as secrecy and authentication, with some others addressing zero knowledge proofs or multi-party session protocols. However, many other kinds of security protocols and properties remain uncovered by such approaches.

For example, e-voting protocols are not explicitly addressed by state-of-the-art approaches, even though researchers are actively proposing new protocols of these kinds, for which no automation is provided in proving soundness between protocol descriptions and their implementations. These kinds of protocols often come with a set of custom security properties which are not trace properties, and require current approaches to be extended in order to take them into account in their soundness proofs. For example, an e-voting protocol may be designed to be coercion-free, or an e-auction protocol may avoid the scenario where a dishonest bidder gets unfair advantage over other bidders or the seller.

Furthermore, the surveyed papers often neglect the so-called side channels, like execution timing and power consumption, letting related attacks go undetected. Nevertheless, popular protocols like SSH and DTLS (a variation of TLS that works over a datagram transport layer) have been found vulnerable to such kind of attacks: in [XDX01] the timing of user keystrokes was exploited to infer the content that the user typed; in [AP12b], the time taken for a server to answer carefully crafted keep-alive messages was used to recover the plaintext of encrypted packets. While some formal work has been done on side channels, the link between this work and protocol implementations, binding results on the formal level down to the implementation level in a general way, is currently missing.

Finally, attacks can come from incorrect usage of encoding operations before cryptography is applied, such as padding [APW09, RD10, Vau02]. In general, these attacks exploit the weaknesses coming from bad interaction between specific encoding procedures applied on bit strings and the way specific cryptographic algorithms

operate on such bit strings. Moreover, these attacks often lead to partial disclosure of the secret (e.g. by making two protocol sessions distinguishable), but they do not break the classical weak secrecy verified in most Dolev–Yao approaches, where bit strings are abstracted as algebraic terms and full disclosure of a session secret is considered. Verifying strong secrecy (e.g. observational equivalence) in the computational domain can often spot these kinds of attacks. Unfortunately, the papers surveyed in Sect. 6 about the conditions for safely abstracting away encoding functions in formal models are currently limited to Dolev–Yao models and weak secrecy. Once again, this calls for a shift towards more realistic models, which would broaden the applicability and validity of the formal verification results.

8. Conclusions

This paper has surveyed state-of-the-art approaches that formally link security protocol implementations to their verified formal models.

It emerges that two main approaches have been recently developed by researchers to address this challenge, starting either from the protocol code and extracting an abstract, simpler model on which security properties can be verified, or starting from the abstract verified model and generating a fully refined implementation. Significant samples for each of these approaches have been analyzed in detail, highlighting the way the formal link between the source code and the model is achieved, and how this can be used to specify and verify security properties.

Not surprisingly, many of the state-of-the-art approaches operate in the mature Dolev–Yao domain, for which automatic verification tools are available. Furthermore, classical security properties such as secrecy and authentication are usually targeted. However, verification tools working in the computational model are becoming available, and security protocols requiring custom properties are becoming widely deployed, which calls for a shift towards computational models and analysis of different security properties.

References

[ABB+10] Almeida J, Bangerter E, Barbosa M, Krenn S, Sadeghi A-R, Schneider T (2010) A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. In: 15th European symposium on research in computer security (ESORICS), pp 151–167

[ABF08] Abadi M, Blanchet B, Fournet C (2008) Automated verification of selected equivalences for security protocols. J Logic Algebr Progr 75(1):3–51

[AF01] Martín Abadi, Cédric Fournet (2001) Mobile values, new names, and secure communication. In: 28th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), pp 104–115

[AG98] Abadi M, Gordon AD (1998) A calculus for cryptographic protocols: The spi calculus. Technical report, Digital System Research Center, Report 149

[AGJ11] Aizatulin M, Gordon AD, Jürjens J (2011) Extracting and verifying cryptographic models from C protocol code by symbolic execution. In: 18th ACM conference on computer and communications security (CCS), pp 331–340

[AGJ12] Aizatulin M, Gordon AD, Jürjens J (2012) Computational verification of C protocol implementations by symbolic execution. In: 19th ACM conference on computer and communications security (CCS) (To appear)

[AN96] Abadi M, Needham R (1996) Prudent engineering practice for cryptographic protocols. IEEE Trans Softw Eng 22(1):6–15

[AP12a] Abadi M, Plotkin G (2012) On protection by layout randomization. ACM Trans Inf Syst Secur 15(2):8:1–8:29

[AP12b] AlFardan NJ, Paterson K (2012) Plaintext-recovery attacks against datagram TLS. In: Network and distributed system security symposium (NDSS)

[APPS11] Avalle M, Pironti A, Pozza D, Sisto R (2011) JavaSPI: a framework for security protocol implementation. Int J Secur Softw Eng 2:34–48

[APW09] Albrecht MR, Paterson KG, Watson GJ (2009) Plaintext recovery attacks against SSH. In: 30th IEEE symposium on security and privacy, pp 16–26

[AR02] Abadi M, Rogaway P (2002) Reconciling two views of cryptography (the computational soundness of formal encryption). J Cryptol 15(2):103–127

[AS05] Askarov A, Sabelfeld A (2005) Security-typed languages for implementation of cryptographic protocols: a case study. In: 10th European symposium on research in computer security (ESORICS), pp 197–221

[BBF+11] Bengtson J, Bhargavan K, Fournet C, Gordon AD, Maffeis S (2011) Refinement types for secure implementations. ACM Trans Program Lang Syst 33(2):1–45

[BBH12] Backes M, Busenius A, Hritcu C (2012) On the development and formalization of an extensible code generator for real life security protocols. In: NASA Formal Methods Symposium (NFM), pp 371–387

[BC04] Bertot Y, Castéran P (2004) Interactive theorem proving and program development. Coq'Art: the calculus of inductive constructions. Texts in theoretical computer science. Springer, Berlin

[BCFZ08] Bhargavan K, Corin R, Fournet C, Zălinescu E (2008) Cryptographically verified implementations for TLS. In: 15th ACM conference on computer and communications security (CCS), pp 459–468
[BCK+08] Bangerter E, Camenisch J, Krenn S, Sadeghi A-R, Schneider T (2008) Automatic generation of sound zero-knowledge protocols. Technical report, Cryptology ePrint Archive, Report 2008/471
[BCPM+09] Bhargavan K, Corin R, Deniélou P-M, Fournet C, Leifer JJ (2009) Cryptographic protocol synthesis and verification for multiparty sessions. In: 22nd IEEE symposium on computer security foundations (CSF), pp 124–140
[BFG06] Bhargavan K, Fournet C, Gordon AD (2006) Verified reference implementations of WS-security protocols. In: 3rd International Workshop on Web Services and Formal Methods (WS-FM), pp 88–106
[BFG10] Bhargavan K, Fournet C, Guts N (2010) Typechecking higher-order security libraries. In: 8th Asian conference on programming languages and systems (APLAS), pp 47–62
[BFGS08] Bhargavan K, Fournet C, Gordon AD, Swamy N (2008) Verified implementations of the information card federated identity-management protocol. In: ACM symposium on information, computer and communications security (ASIA CCS), pp 123–135
[BFGT08] Bhargavan K, Fournet C, Gordon AD, Tse S (2008) Verified interoperable implementations of security protocols. ACM Trans Program Lang Syst 31(1):1–61
[BJ10] Bauer A, Jürjens J (2010) Runtime verification of cryptographic protocols. Comput Secur 29(3):315–330
[BJY11] Bauer A, Jürjens J, Yu Y (2011) Run-time security traceability for evolving systems. Comput J 54(1):58–87
[BKSS10] Bangerter E, Krenn S, Sadeghi A-R, Schneider T (2010) YACZK: yet another compiler for zero-knowledge. USENIX Security Symposium Posters
[Bla01] Blanchet B (2001) An efficient cryptographic protocol verifier based on prolog rules. In: 14th IEEE computer security foundations workshop (CSFW), pp 82–96
[BMU10] Backes M, Maffei M, Unruh D (2010) Computationally sound verification of source code. In: 17th ACM conference on computer and communications security (CCS), pp 387–398
[BP06] Blanchet B, Pointcheval D (2006) Automated security proofs with sequences of games. In: 26th international conference on advances in cryptology (CRYPTO), pp 537–554
[BPP03] Bierman G, Parkinson M, Pitts A (2003) MJ: an imperative core calculus for Java and Java with effects. Technical report, Cambridge University Computer Laboratory, Report 563
[Bus11] Busenius A (2011) Mechanized formalization of a transformation from an extensible spi calculus to Java. Master's thesis, Saarland University (Germany). Available at http://www.infsec.cs.uni-sb.de/~hritcu/students/busenius/masters_thesis.pdf
[Car94] Carlsen U (1994) Cryptographic protocol flaws: know your enemy. In: 7th computer security foundations workshop (CSFW), pp 192–200
[CB12] Cadé D, Blanchet B (2012) From computationally-proved protocol specifications to implementations. In: 7th international conference on availability, reliability and security (ARES), pp 65–74
[CD09] Chaki S, Datta A (2009) ASPIER: an automated framework for verifying security protocol implementations. In: 22nd IEEE symposium on computer security foundations (CSF), pp 172–185
[CGL94] Clarke EM, Grumberg O, Long DE (1994) Model checking and abstraction. ACM Trans Program Lang Syst 16(5):1512–1542
[CJK05] Choi J-Y, Jeon C-W, Kim I-G (2005) Automatic generation of the C# code for security protocols verified with Casper/FDR. In: 19th international conference on advanced information networking and applications (AINA), pp 507–510
[CKW11] Cortier V, Kremer S, Warinschi B (2011) A survey of symbolic methods in computational analysis of cryptographic systems. J Autom Secur 46(1):225–259
[DGJN11] Dupressoir F, Gordon AD, Jürjens J, Naumann DA (2011) Guiding a general-purpose C verifier to prove cryptographic protocols. In: 24th IEEE symposium on computer security foundations (CSF), pp 3–17
[DR08] Dierks T, Rescorla E (2008) The transport layer security (TLS) protocol version 1.2. RFC 5246
[DY83] Dolev D, Yao ACC (1983) On the security of public key protocols. IEEE Trans Inf Theory 29(2):198–208
[FKS11] Fournet C, Kohlweiss M, Strub P-Y (2011) Modular code-based cryptographic verification. In: 18th ACM conference on computer and communications security (CCS), pp 341–350
[GBS+08] Grandy H, Bischof M, Stenzel K, Schellhorn G, Reif W (2008) Verification of Mondex electronic purses with KIV: From a security protocol to verified code. In: 15th international symposium on formal methods (FM), pp 165–180
[GLP05] Goubault-Larrecq J, Parrennes F (2005) Cryptographic protocol analysis on real C code. In: 6th international conference on verification, model checking, and abstract interpretation (VMCAI), pp 363–379
[GLP09] Goubault-Larrecq J, Parrennes F (2009) Cryptographic protocol analysis on real C code. Technical report, Laboratoire Spécification et Vérification, Report LSV-09-18
[GM84] Goldwasser S, Micali S (1984) Probabilistic encryption. J Comput Syst Sci 28(2):270–299
[GS97a] Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In: 9th international conference on computer aided verification (CAV), pp 72–83
[GS97b] Gritzalis S, Spinellis D (1997) Cryptographic protocols over open distributed systems: a taxonomy of flaws and related protocol analysis tools. In: 16th international conference on computer safety, reliability and security (SAFECOMP), pp 123–137
[HL01] Hui ML, Lowe G (2001) Fault-preserving simplifying transformations for security protocols. J Comput Secur 9(1/2):3–46
[HOP03] Hubbers E, Oostdijk M, Poll E (2003) Implementing a formally verifiable security protocol in Java Card. In: 1st international conference on security in pervasive computing (SPC), pp 213–226
[Inc] MasterCard International Inc. The Mondex protocol. http://www.mondexusa.com
[Jür01] Jürjens J (2001) Secrecy-preserving refinement. Form Methods Increasing Softw Product 2021(1):135–152
[Jür05] Jürjens J (2005) Verification of low-level crypto-protocol implementations using automated theorem proving. In: 2nd ACM/IEEE international conference on formal methods and models for co-design (MEMOCODE), pp 89–98

[Jür08] Jürjens J (2008) Using interface specifications for verifying crypto-protocol implementations. In: Workshop on foundations of interface technologies (FIT)
[Jür09] Jürjens J (2009) Automated security verification for crypto protocol implementations: Verifying the JESSIE project. Electron Notes Theor Comput Sci 250(1):123–136
[JYB08] Jürjens J, Yu Y, Bauer A (2008) Tools for traceable security verification. In: BCS international academic conference on visions of computer science (VoCS), pp 367–390
[KOT08] Kiyomoto S, Ota H, Tanaka T (2008) A security protocol compiler generating C source codes. In: 2nd international conference on information security and assurance (ISA), pp 20–25
[KR06] Kleiner E, Roscoe AW (2006) On the relationship between web services security and traditional protocols. Electro Notes Theor Comput Sci 155(1):583–603
[KTG12] Kuesters R, Truderung T, Graf J (2012) A framework for the cryptographic verification of Java-like programs. In: 25th IEEE symposium on computer security foundations (CSF), pp 192–212
[LBR06] Leavens GT, Baker AL, Ruby C (2006) Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw Eng Notes 31(3):1–38
[Low95] Lowe G (1995) An attack on the Needham-Schroeder public-key authentication protocol. Inf Process Lett 56(3):131–133
[Low96] Lowe G (1996) Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: 2nd international workshop on tools and algorithms for construction and analysis of systems (TACAS), pp 147–166
[MEK+10] Meiklejohn S, Erway CC, Küpçü A, Hinkle T, Lysyanskaya A (2010) ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In: 19th USENIX conference on security, pp 193–206
[MSS98] Mitchell JC, Shmatikov V, Stern U (1998) Finite-state analysis of SSL 3.0. In: 7th USENIX security symposium (SSYM), pp 201–216
[NS78] Needham RM, Schroeder MD (1978) Using encryption for authentication in large networks of computers. Commun ACM 21(12):993–999
[OF05] Ogata K, Futatsugi K (2005) Equational approach to formal analysis of TLS. In: 25th IEEE international conference on distributed computing systems (ICDCS), pp 795–804
[OR87] Otway D, Rees O (1987) Efficient and timely mutual authentication. ACM Operat Syst Rev 21(1):8–10
[O'S08] O'Shea N (2008) Using Elyjah to analyse Java implementations of cryptographic protocols. In: Joint workshop on foundations of computer security, automated reasoning for security protocol analysis and issues in the theory of security (FCS-ARSPA-WITS), pp 221–226
[O'S10] O'Shea N (2010) Verification and validation of security protocol implementations. PhD thesis, School of Informatics, University of Edinburgh (UK). Available at http://hdl.handle.net/1842/4753.
[Pau99] Paulson LC (1999) Inductive analysis of the Internet protocol TLS. ACM Trans Inf Syst Secur 2(3):332–351
[Pir10] Pironti A (2010) Sound automatic implementation generation and monitoring of security protocol implementations from verified formal specifications. PhD thesis, Politecnico di Torino (Italy). Available at http://alfredo.pironti.eu/research/sites/default/files/Pironti_Dissertation.pdf
[PJ10] Pironti A, Jürjens J (2010) Formally-based black-box monitoring of security protocols. In: International symposium on engineering secure software and systems (ESSoS), pp 79–95
[PM12] Polikarpova N, Moskal M (2012) Verifying implementations of security protocols by refinement. In: Verified software: theories, tools, experiments (VSTTE), pp 50–65
[PPS11] Pironti A, Pozza D, Sisto R (2011) Automated formal methods for security protocol engineering. In: IGI global cyber security standards, practices and industrial applications: systems and methodologies, pp 138–166
[PPS12] Pironti A, Pozza D, Sisto R (2012) Formally-based semi-automatic implementation of an open security protocol. J Syst Soft 85(1):835–849
[PS10] Pironti A, Sisto R (2010) Provably correct Java implementations of Spi Calculus security protocols specifications. Comput Secur 29(3):302–314
[PS12] Pironti A, Sisto R (2012) Safe abstractions of data encodings in formal security protocol models. Form Asp Comput (To appear)
[PSD04] Pozza D, Sisto R, Durante L (2004) Spi2java: automatic cryptographic protocol Java code generation from spi calculus. In: 18th international conference on advanced information networking and applications (AINA), pp 400–405
[RD10] Rizzo J, Duong T (2010) Practical padding oracle attacks. In: 4th USENIX offensive technologies (WOOT), pp 1–8
[Sch05] Schellhorn G (2005) ASM refinement and generalizations of forward simulation in data refinement: a comparison. Theoret Comput Sci 336(2–3):403–435
[SPP01] Song DX, Perrig A, Phan D (2001) AGVI—automatic generation, verification, and implementation of security protocols. In: 13th international conference on computer aided verification (CAV), pp 241–245
[SRA78] Shamir A, Rivest R, Adleman L (1978) Mental poker. Technical report, Massachussets Institute of Technology
[Ste04] Stenzel K (2004) A formally verified calculus for full Java Card. In: 10th international conference on algebraic methodology and software technology (AMAST), pp 33–36
[TCCD06] Tobarra L, Cazorla D, Cuartero F, Díaz G (2006) Formal verification of TLS handshake and extensions for wireless networks. In: IADIS international conference on applied computing, pp 57–64
[TH04] Tobler B, Hutchison A (2004) Generating network security protocol implementations from formal specifications. In: 2nd international workshop on certification and security in inter-organizational e-services (CSES), pp 33–54
[Vau02] Vaudenay S (2002) Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS .... In: International conference on the theory and applications of cryptographic techniques—advances in cryptology (EUROCRYPT), pp 534–546

[Vig06] Viganò L (2006) Automated security protocol analysis with the AVISPA tool. Electr Notes Theor Comput Sci 155(1):61–86
[WS96] Wagner D, Schneier B (1996) Analysis of the SSL 3.0 protocol. In: 2nd USENIX Workshop on Electronic Commerce (WOEC), pp 29–40
[XDX01] Xiaodong SD, David W, Xuqing T (2001) Timing analysis of keystrokes and timing attacks on SSH. In: 10th conference on USENIX security symposium (SSYM), pp 25–25
[Yao82] Yao AC (1982) Theory and application of trapdoor functions. In: 23rd annual symposium on foundations of computer science (FOCS), pp 80–91

Received 30 September 2011
Accepted in revised form 3 November 2012 by Eerke Boiten and Steve Schneider
Published online 4 December 2012