Formal Modeling and Security Analysis of Security Protocols

Paolo Modesti and Rémi Garcia

This is an accepted manuscript of a book chapter published by CRC Press in
Handbook of Formal Analysis and Verification in Cryptography
pp. 213–274, 1st edition, CRC Press, 2023
eBook ISBN: 9781003090052
published on 19 September 2023

Available online:
https://doi.org/10.1201/9781003090052-5 (Chapter)
https://doi.org/10.1201/9781003090052 (Book)

Cite as:

```
@incollection{books/crc/23/Modesti023,
author = {Paolo Modesti and R{\'{e}}mi Garcia},
editor = {Sedat Akleylek and Besik Dundua},
title = {Formal Modeling and Security Analysis of Security Protocols},
booktitle = {Handbook of Formal Analysis and Verification in Cryptography},
pages = {213--274},
publisher = {{CRC} Press},
year = {2023},
doi = {10.1201/9781003090052-5},
}
```

1 Formal Modeling and Security Analysis of Security Protocols

1.1 INTRODUCTION

The Covid-19 pandemic has offered an incredible opportunity for cybercrime to thrive with millions of people working from home using digital devices secured by accidental circumstances rather than by design. Many organizations were notoriously under-prepared. Cyberattacks are more than just technical events, they are attacks on the very foundations of the modern information society. The Center for Strategic and International Studies estimated the worldwide monetary loss from cybercrime to be around $945 billion in 2020 [102]. This has clearly exposed the gap between the practices adopted by the software industry and the stark reality. Crucially, designing secure and dependable systems is fundamental to protect digital resources and reduce the attack surface. Therefore, experts [110, 51] recommend investing more resources on enhancing the design of secure systems rather than just fixing vulnerabilities in existing systems when they are discovered.

In particular, security protocols are critical components for the construction of secure Internet services and distributed applications, but their design and implementation are difficult and error-prone. Some vulnerabilities in very popular protocols like TLS and SSH [97, 55], which are both expertly built and rigorously tested, have been undetected for years. The core reasons of this situation need to be investigated and novel approaches explored. In this context, techniques for formal modeling and analysis of security protocols could support software engineers, but there is reluctance to embrace these tools outside the research settings because of their complexity [59]. Most practitioners find such methodologies complex and incompatible with their work requirements (e.g. difficulty to write the formal specification, steep learning curve, need to understand the underlying advanced theoretical computer science notions [108, 99]). Therefore, vulnerable software is still deployed, with tens of thousands of new vulnerabilities being discovered in major products each year. Such defective software has a serious negative impact on both individuals and organizations utilizing vulnerable systems every day.

In this chapter, we focus on an approach to formal modeling and verification that can be adequate to the practitioner's skills and needs. In fact, we can bridge the gap between formal representation and actual implementation with a framework adopting a conceptual model aligned with the level of abstraction used for the symbolic (high-level) representation of cryptographic and communication primitives. This would

make formal methods and tools more accessible to students and practitioners [90].

Outline of the chapter

In Section 1.2, we consider modeling and verification in the context of security protocol development, and we introduce the fundamentals like the attacker model and security properties. Then, in Section 1.3, we discuss the theoretical and practical challenges in automated verification. Different specification languages and verification tools are considered in Section 1.4 to address different levels of user expertise and complexity of the protocols under analysis. Section 1.5 includes case studies, demonstrating the practical applicability of these tools in two different application fields: e-payments and blockchain.

1.2 MODELING FUNDAMENTALS

This section details the foundational concepts used in security protocol verification. We present how protocols are modeled in a formal framework, and what a dishonest entity represents when searching for vulnerabilities. The different properties that are commonly checked appear in this section, as well as how the verification tools can capture the behavior of cryptographic primitives. We also discuss the problem of abstraction and its pedagogic implications.

A typical workflow for the development of security protocols is shown in Figure 1.1. Similar workflows have been proposed for example in [58, 90]. The process usually starts by identifying a set of software requirements specifying the expected behavior of the protocol, including the security goals that the application aims to achieve. There may be two different scenarios: 1) *New protocols*: the definition of the requirements is entirely under the control of the protocol designer; 2) *Existing protocols*: the requirements, and sometimes a reference implementation, already exist; the developer needs to consult the documentation to extract the relevant information.

The specification of a protocol typically includes the roles of *agents* involved in the protocol, the information they possess prior to the protocol run (*initial knowledge*), the specification of the messages exchanged between agents (*actions*) during the protocol execution and the *security goals* (e.g. secrecy, authentication) that should hold at the end of the protocol run.

While in software engineering having both explicit and implicit requirements is generally common, in formal modeling implicit requirements are particularly problematic. In fact, even something that may be obvious and unambiguous for a human needs to be defined precisely and explicitly to be interpreted by a machine. Therefore, implicit requirements can be a source of ambiguity or incompleteness, that can lead to the development of a formalized model that does not reflect the reality of the system under consideration.

Requirements can be expressed in different ways: informally, formally or semi-formally. Informal requirements are usually described in a natural language, and though common in human communication, may be ambiguous or refer to implicit concepts that may lead to different interpretations by different people. This is quite

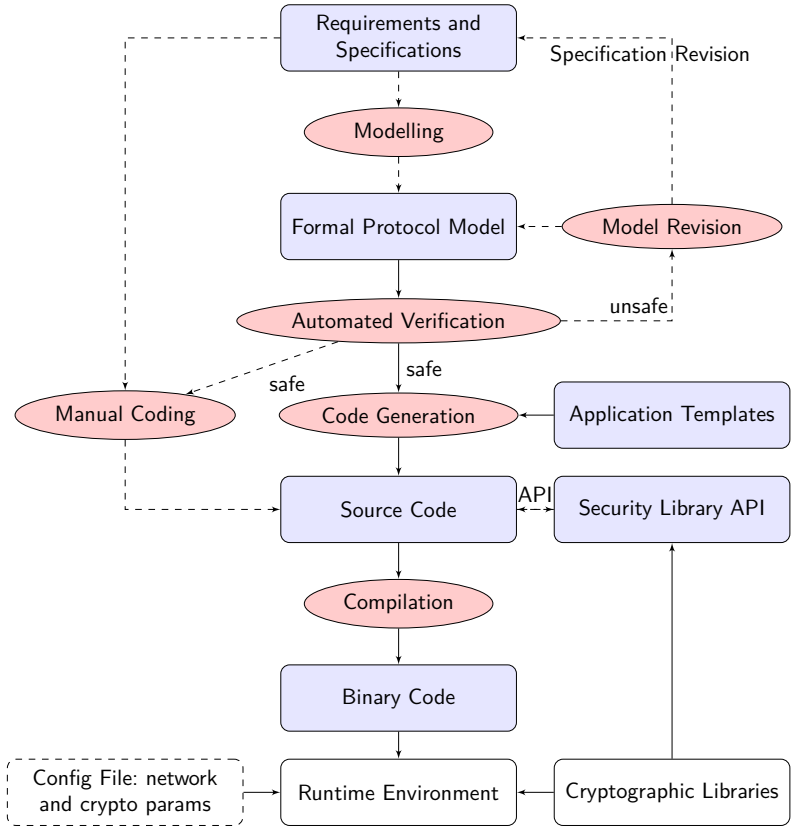

Figure 1.1 Development of security protocols workflow

usual in software engineering as end-users and developers need not only to build a common understanding of the application domain, but also a mutually intelligible communication language. Instead, formal requirements use mathematics-based specification languages to describe in rigorous terms the desired properties of the system under development. Unfortunately, the required formalism is often too complex, not only for the end-users but also for the software developers. Therefore, the adoption of specification languages that can be used by practitioners is extremely important as we will see in Section 1.4.

In the initial phase of the modeling process, there are a few essential aspects that need to be considered. First of all, the specification of the model requires defining not only the actions performed by the agents, but also their initial knowledge. If this is omitted, the protocol may be ambiguous or not even executable. Another crucial aspect is the specification of the security goals that protocols are meant to achieve. This is an often neglected activity as many informal descriptions of protocols used in the industry fail to give a rigorous definition of the security goals. Instead, they use the natural language to describe them. This is the case, for example, of the ISO/IEC

9798 standard for entity authentication [72] and the EMV payment protocol [58].

It is worth to mention that there are attempts at automatic formalization of requirements from natural languages. Such approaches include Natural Language Processing, AI-based interpretation of specification documents and computational linguistics. However, their adoption by the industry is extremely limited [59].

Once the model is formalized, it may be tested with one or more verification tools. Various tools are available, and they differ not only from the specification language but also from the different security properties they can verify. Section 1.3 presents an overview of such tools, along with theoretical and practical challenges in automated verification.

The verification process, if it terminates, will indicate if the protocol satisfies the expected security goals (*safe protocol*) or not (*unsafe*). In the latter case, the tools usually provide an *attack trace*: a list of actions that the *attacker* (aka *intruder*) has to perform to violate one or more security goals. Additionally, the tools can report about the *reached state*, *i.e.* the knowledge of agents at the end of the attack trace.

This output can lead the next design and implementation steps. Unsafe protocols may require an iterative process of revision and verification of the model until the protocol is successfully verified. In some cases, even the requirements may need a revision if they are ambiguous or ill-formed. Although users can employ a verification tool as a black-box security oracle, the attack trace provides elements that help to understand why the protocol fails in satisfying the security goals. Such iterative process is quite standard in formal design of security protocols, and it is aimed at capturing design errors in the very early phases of software development.

After modeling and verification, manual coding or automatic code generation can be used to build an implementation of the protocol. Code generation can be very effective, as this is a phase where implementation errors typically occur [55]. Additionally, code generation often relies on application templates to define the common structure of a program, and on an existing security library API that allows the generated source code to be compiled. Application execution support is provided by the runtime environment of the target platform, including relevant cryptographic libraries and configuration files.

1.2.1 THE DOLEV-YAO MODEL

To model communications in an adversarial environment, the Dolev-Yao model [53] has been proposed to provide a formal framework of the intruder's capabilities over the network and the honest agents, also called honest principals. In this model, the intruder has complete control over the network. It can overhear and intercept any message, as well as generate its own, following specific rules allowed in a case-by-case basis.

Figure 1.2 shows the intruder rules, and the fact $\mathsf{ik}(m)$ denotes that the intruder knows the term m. Since every communication between honest agents is assumed to be mediated by the intruder, it happens through $\mathsf{ik}(\cdot)$ facts. The model assumes the existence of a set of function symbols (with an associated arity) partitioned into two subsets of *public* and *private* symbols. The first rule describes both asymmetric

$$
\begin{array}{rcll}
\mathsf{ik}(M).\mathsf{ik}(K) & \Rightarrow & \mathsf{ik}(\{M\}_K) & \\
\mathsf{ik}(\{M\}_K).\mathsf{ik}(\mathsf{inv}(K)) & \Rightarrow & \mathsf{ik}(M) & \textit{Asymmetric Encryption} \\
\mathsf{ik}(\{M\}_{\mathsf{inv}(K)}) & \Rightarrow & \mathsf{ik}(M) & \\
\mathsf{ik}(M).\mathsf{ik}(K) & \Rightarrow & \mathsf{ik}(\{|M|\}_K) & \\
\mathsf{ik}(\{|M|\}_K).\mathsf{ik}(K) & \Rightarrow & \mathsf{ik}(M) & \textit{Symmetric Encryption} \\
\mathsf{ik}(M).\mathsf{ik}(N) & \Rightarrow & \mathsf{ik}(M,N) & \textit{Tupling} \\
\mathsf{ik}(M,N) & \Rightarrow & \mathsf{ik}(M).\mathsf{ik}(N) & \textit{Projection} \\
\mathsf{ik}(M_1).\cdots.\mathsf{ik}(M_n) & \Rightarrow & \mathsf{ik}(f(M_1,\ldots,M_n)) & \textit{Function Application}
\end{array}
$$

Figure 1.2 Dolev-Yao intruder rules

encryption and signing ($\{\cdot\}$), while the second one models that a ciphertext can be decrypted if the corresponding decryption key is known. $\mathsf{inv}(\cdot)$ is a *private* function symbol representing the secret component of a given key-pair. The third rule allows the attacker to learn the payload of any known signed message. Symmetric encryption ($\{|\cdot|\}$) can be modeled similarly to the first two rules, but in this case the same key is employed for both encryption and decryption. Additionally, there are rules for tupling and projecting tuple elements, and a rule for the application of *public* function symbols to known messages. Constants, including agent identities, are modeled as public functions with zero-arity.

1.2.2 SECURITY PROPERTIES

We introduce here the most common security properties considered in protocol verification: *secrecy* and *authentication*. We also discuss *equational theories* that allow to model important functions and their algebraic properties, like exponentiation used in the Diffie-Hellman key agreement or the *xor* operator used in various cryptographic protocols.

1.2.2.1 Secrecy

When honest agents communicate, the confidentiality of the core payloads is essential to avoid eavesdropping on sensitive information. A term is said to be secret for a given agent when it cannot be derived from the intruder's knowledge at any point in the protocol execution. It should be noted that secrecy is subjective: an agent receiving a message which has been produced by the intruder may accept it as a legitimate one. This can be, for example, if a simple message is sent encrypted with a public key. The intruder just needs to encrypt its own forged message with the same public key and fool the receiving agent. The term sent at first will still be confidential, but the received one will not, for the honest receiver.

Stronger definitions can include *forward secrecy*. With this property, any message exchanged between non-compromised agents will remain secret after they fall victim

to an attack. A common way to implement it is to use one-time keys, preventing information leakage of message history when long-term keys are compromised.

1.2.2.2 Authentication

Authentication properties follow Lowe's definition [77], where they correspond to *agreement* observations. Two kinds of authentication can arise: *non-injective* and the stronger *injective*.

With non-injective agreement, we can have two honest agents A and B, with B wanting to agree with A on a certain message Msg. A is then supposed to send Msg to B and be authenticated by Msg. The agreement is successful when there exists two observations in the protocol execution: B completed a protocol run as receiver, getting Msg from apparently A, and A completed it as well as initiator, sending Msg to apparently B.

Injective agreement introduces a notion of freshness to avoid replay attacks. The above conditions need to be satisfied, but for each run of B as receiver of Msg from apparently A, there must exist only one run of A as initiator sending Msg to apparently B. With this guarantee, an intruder could not reuse Msg in another session in order to impersonate A.

1.2.2.3 Equational Theories

In every formal system, some axioms and symbols are defined, as well as the logic of the relations every symbol implies. The set of formulas that can be coherently derived from the axioms form a global theory of the system.

On a local level, every operator or symbol can have its own theory. Equational theories are expressed in quantifier free first-order logic with an equality relation. Formally, an *equation* is an atomic formula of the form $x = y$. Taking the example of the classical addition, a purely additive equation admits commutativity and associativity for its terms, *i.e.* $x+y = y+x$ and $(x+y)+z = x+(y+z)$.

There are however limits to what can be admissible with this kind of theories in a verification tool. For example, an equational theory must have the *Finite Variant Property* [44], stating that given any term *t, t* must be rewritable into a finite number of most general *variants* $t_1, \ldots, t_n$. A variant can be considered as a pattern describing the canonical form of instances of a term. With this property, an infinite number of possible rewritings for a given term can be expressed using finitely many terms and substitutions.

1.2.3 ABSTRACTIONS AND PEDAGOGY

When applying formal methods, particularly in a practitioner context, we should consider how such techniques can be learned and used correctly. This is a general problem in Computer Science Education (CSE), and learning theories can be useful to understand the challenges newcomers may face. The *constructivist learning theory* [109, 35] claims that knowledge is acquired by combining sensory data (gathered from experience) with existing knowledge to create new cognitive structures. This

process is applied recursively to generate new knowledge. Additionally, new knowledge is built reflecting on the existing one. To be effective, learning must be active, and the teacher must guide and support students in their endeavours.

Authors promoting a constructivist perspective in CSE [69, 27] indicate the importance of understanding both abstract and concrete concepts. Ben Ari [27] underlines that the application of constructivism must consider that "a (beginning) computer science student has no effective model of a computer", and that "the computer forms an accessible ontological reality". The latter concept manifests itself when students interacting with a computer can immediately realize the effect of the application of their mental models. In other words, the results of misconceptions are discovered instantly. Rightly, Ben Ari [27] notes that "there is not much point negotiating models of the syntax or semantics of a programming language". Tools used by learners, like compilers used in programming activities, have implications for building mental models.

In formal methods for security, abstractions play a fundamental role. The formalization of a model requires not only a good understanding of the concrete system, but also the ability to identify what elements are relevant, and what can be neglected for the purpose of formal verification. This requires a viable abstract representation of the system, suitable for verification.

There is no general consensus on when abstractions need to be introduced. Object-Oriented Programming is a good example. Adams [5] opposes the idea of postponing teaching OOP until late in the course of studies because at that time it is difficult to have an impact on the learners' low-level model, but he also believes that OOP should not be taught too early when students are not mature enough to assimilate properly the related concepts. Ben Ari [27] remarks that "advocates of an objects-first approach seem to be rejecting Piaget's view that abstraction (or accommodation) follows assimilation". This seems to be confirmed by the fact that practitioners who use abstractions usually have a fairly good knowledge of the underlying concrete model.

An initial crucial question is: what is the appropriate model of the "system"? While in the domain of security protocols there are two main approaches, computational and symbolic, the latter one seems to be more appropriate from a pedagogic point of view. Haberman and Kolikant [68] found that a blackbox-based approach can be used effectively to introduce programming concepts to novices. An important characteristic of the symbolic model is its simplicity, and according to the constructivist approach, the model must be taught explicitly [28, 74].

Along with the adversary model (e.g. Dolev-Yao), it is necessary to learn how to represent cryptographic primitives in the symbolic model and model their properties, to allow the learner to understand the actions that honest agents perform during the protocol execution. At this point, we should recall the recommendation given by [27] regarding the need to explicitly present a viable model one level beneath the one we are teaching. Therefore, learners of tools and techniques for modeling and verification of security protocols cannot ignore the construction of such protocols in a real programming language.

We can exemplify this with a conceptual framework (Figure 1.3). It consists of

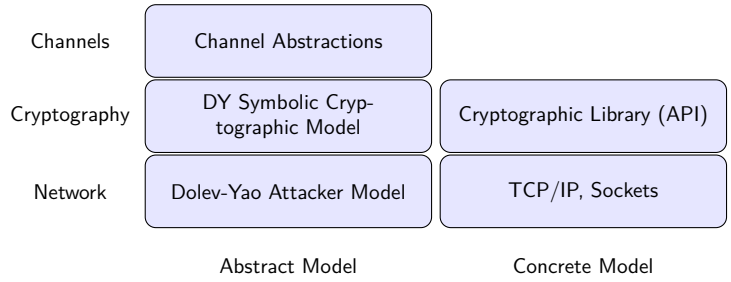

Figure 1.3 Abstract and concrete models

both an abstract and concrete models with three different layers: Network, Cryptography and Channels. At the network level, the concrete network protocols run onto, where the adversary has full control of the communication medium, is abstracted by the Dolev-Yao adversary model. The concrete cryptographic primitives are often available through an application programming interface (API), simplifying the access to a set of standard network and cryptographic primitives required to build security protocols implementations. In the abstract model, the symbolic representation of Dolev-Yao intruder rules provides an abstraction of the concrete cryptographic functions.

Abstracting from low-level details where most implementation errors occur [104], the developer can focus on the application design and its security properties. For this reason, the formal methods research community [3, 38, 19] has advocated for the specification of security protocols with high-level programming abstractions, suited for security analysis and automated verification. Alexandron *et al.* [7] suggest that when programmers can work with a less detailed mental model, it becomes easier to work with high-level abstractions. Concretely, we can abstract from cryptographic details at the channel level, using for example a language like *AnBx* [37], an extension of the *AnB* language, where actions are presented in the popular *Alice and Bob* [83] narration style.

Users can model protocols and reason about their security properties using tools for the verification of security protocols in the symbolic model. In a nutshell, the user can specify the security protocol and its security goals and then verify whether the protocol satisfies these goals, or if an attack may occur, with an attack trace being provided in this case.

It should be noted that, with the specification of security goals, the users describe the expectations regarding the security properties of the protocol that reflects their mental model. Running the verification tools provides feedback on the correctness of their mental model and helps to build their own knowledge autonomously. The analysis of errors is therefore an opportunity for individual reflection.

1.3 THEORETICAL AND PRACTICAL CHALLENGES

To precisely formulate a conclusion on whether a security property holds or not, verification tools have to face certain challenges and inherent limitations. Giving a definite and trustworthy answer about non-trivial protocol verification is not always possible, or unreasonably resource-intensive in some cases. The interactions between such processes raise additional concerns in the proof effort, as well as how to reproduce abstract attacks in the real world.

1.3.1 DECIDABILITY

In the protocol verification effort, a major obstacle is the undecidability of security properties. Requirements for a program usually include knowing whether a result will eventually arise, which is unfortunately often equivalent to the undecidable halting problem [105]. In practice, a tool will fail to reach a conclusion or will not terminate in this case. Despite this general limitation, protocol verification can be successfully completed in many cases, under very broad conditions. It has been proven that the security problem is decidable in co-NP time in a Dolev-Yao model of intruders with a bounded number of concurrent sessions [98], including when the intruder is allowed to guess low-entropy messages [6] like passwords instead of random numbers. However, the general case with an infinite number of sessions is undecidable [57], even with a bounded message length [82]. Another undecidability factor with unbounded sessions is the presence of nonces in a protocol. They are fresh values generated by honest agents to prevent replay attacks between concurrent sessions. If the same value appears in two different sessions, it can safely be ignored since we have either an intruder reusing this value, or an honest agent not following the protocol guidelines. Their existence adds complexity to the verification problem, making it undecidable even with a bounded number of nonces [10].

The decidability of a security property also heavily depends on how the cryptographic primitives are considered. In the classic Dolev-Yao model, we assume perfect cryptography and abstract the cryptographic operations as black-box functions. An encrypted message is then akin to random data, and the algebraic properties it exhibits are ignored. While it is true that many attacks exploit design flaws in a protocol more than its underlying cryptographic schemes, a concrete implementation might be vulnerable. Attacks relying on some properties of the encryption function might be missed under the perfect cryptography assumption, and existing results are summarized in a survey [46]. The authors highlight how recent works investigate possible refinings on cryptographic primitives abstraction, in order to allow for a more thorough protocol verification.

1.3.2 VERIFICATION TRUSTWORTHINESS

Whenever a formal result is provided, the trustworthiness of the underlying process needs to be assessed. In fact, a poorly modeled system would lead to an unusable verification result, but the verification in itself should be questioned too. As the effort nowadays is largely automatic, how can we be sure that the verification tools

we use are reliable? BAN-logic [40] and its assumptions over the intruder's behavior was found insufficient by Lowe on the Needham-Schroeder protocol [76]. Model checkers then introduced different logics, putting the emphasis on automation.

Doing so, they are complex tools with a large and often complicated codebase, which itself is not necessarily free from bugs. We have then to rely on the programmer's skill and the regular testing of the tool over time to bring us confidence about the verification correctness. Even with a supposedly correct proof system, Gödel second incompleteness theorem [62] states that such a system could not prove its own correctness. *Soundness*, *completeness* and level of certification are key elements in the verification process.

Formally, let Σ be a set of hypotheses and Φ a statement. $\Sigma \models \Phi$, means that Σ logically implies Φ, *i.e.*, in every circumstance in which Σ is true, Φ also holds. Another relation, $\Sigma \vdash \Phi$, states that we can derive Φ starting from Σ, or that Φ is provable from Σ.

A formal deduction system is said *sound* where every conclusion that can be reached in the proof system derives from its premises. Nothing that violates Σ can be proven as true in a sound system. Formally, if $\Sigma \vdash \Phi$, then $\Sigma \models \Phi$.

A *complete* system allows every statement to be provable with our hypotheses. Here, $\Sigma \models \Phi$ implies $\Sigma \vdash \Phi$. If Φ is true given Σ, then we can prove Φ from Σ.

In practice, some tools are sound but not complete, like ProVerif [33]. Here, if the tool states that a security property is true then the property is indeed true in the Dolev-Yao attacker model. However, not all true properties may be provable by the tool.

A way to confidently verify a set of properties is to implement a very small amount of verified mathematics. A *theorem prover* takes logical statements as an input and tests them against an axiomatically accepted set of inference and equivalence rules that can be defined in a trusted kernel. This is the *Logic for Computable Functions* (LCF) [81] paradigm, where *theorems* are an abstract data type, and constructors enforce the underlying logic. With a strong typing enforcement, every user-specified proof step is essentially a composition of the kernel's rules. Modern proof assistants like HOL4 [101] or Isabelle/HOL [93] opt for *Higher-Order Logic* (HOL). HOL allows passing functions as arguments and returning functions, with arbitrary nesting. This extends *First-Order Logic*, where only quantified variables and sets of variables can be considered. Some assistants also apply the *De Bruijn criterion* [23], and generate independently checkable proof objects.

1.3.3 STATE EXPLOSION

Usually, a verification tool explores the realm of possibilities for a given protocol execution. Due to the interleaving between actions and possible messages, the total number of possible traces can be unacceptably high [48] or even infinite. Most systems are represented as state-transition, where the knowledge of an agent is given at each step of a protocol. Great execution complexity can cause the *state explosion* problem, along with potentially enormous time or memory resources needed for protocol verification, which can be limited with various approaches.

Bounded Model Checking is one of the most popular techniques for critical systems verification. Here, the state-transition system is finite and there is a bound on how many transitions the model checker is allowed to explore. Some states are called *error* or *attack* states, and the purpose of the model checker is to try to find a series of transitions which leads to such states. This approach then works by finding a counterexample or bug in the system. Given a bound k, the model checker will typically encode the problem in a propositional Boolean formula in conjunctive normal form $(A \vee B) \wedge (C \vee D \ldots) \wedge \ldots$. With respect to the bound k, the formula is only allowed to contain k conjunctions at most, which denotes a state-space exploration of paths of maximal length k. A SAT solver will in turn test if there is a possible valuation of the literals such as the formula holds true, as does the SATMC [15] tool. If no path of length k in the transition system leads to an unsafe state, then the tested security property is true, given this bound k. Even if it loses precision compared to an exhaustive state-space exploration, a bounded proof can generalize to the unbounded case [78], like when the bound is greater or equal to the *diameter* of the transition system, *i.e.* the smallest number of transitions needed to reach all reachable states.

Binary Decision Diagrams (BDDs) [36] are other representations of interest to explore a state-space efficiently. Instead of listing every possible state individually, this approach allows for state grouping, yielding potentially exponential time and memory savings. A BDD represents a Boolean function with a rooted and directed acyclic graph, where redundant tests of Boolean variables are omitted. Only two leaves exist as Boolean values, and every other node is a Boolean variable. Every non-terminal node represents a variable and has two children, denoting the outcomes of its *true* and *false* valuations. With a brute-force checking process for a Boolean function, every input has to be tested, which equates to 2^n cases with n the number of variables. For example, the expression $A \wedge B$ has four possible valuations. However, having A valued as *false* makes the expression unsatisfiable, and testing B would be useless. In this case, we can omit a B node between A and the *false* Boolean value, so only two variable nodes are required. Tools like OFMC [84] or Tamarin [80] can greatly benefit from this search space reduction.

When the complete model to verify is composed of several asynchronous processes, some of them might be independent of each other. The interleavings between them are then not relevant to the property at hand and could be ignored safely. This simplification is called *Partial Order Reduction* [61]. It makes the precedence relations between events explicit in a dependency graph. If it is found that, from a given state, several paths leading to another state denote equivalent behaviors, then only one *representative* path must be explored. With this technique, paths can be tested for independence on the fly, one local component at the time, saving considerable memory resources. Recent studies show that independence can be tested quasi-optimally in polynomial time [47], avoiding the execution of all possible traces. Similarly to BDDs, this technique can be applied to state-transition-based verification.

Another technique to reduce the state-space size is to work on an abstraction of our model. Distinct levels of abstraction exhibit different levels of detail, and the goal is to find the most abstract model that does not omit relevant details with respect to the tested property. The search of the adequate level of abstraction is then con-

ducted by successive refinements, where a highly abstracted model is verified first. This technique is named *counterexample-guided abstraction refinement* (CEGAR) [43]. A counterexample denotes a bug or vulnerability, which is executed against the concrete system once found. If the execution fails, then the counterexample is called *spurious*, and more detail must be added to the model. The first *non-spurious* counterexample will then embody a genuinely problematic behavior in the concrete model. Hajdu and Micskei introduced the efficient THETA framework [70], along with a survey of other state-of-the-art CEGAR implementations.

1.3.4 COMPOSITIONALITY IN VERIFICATION

As we have seen, in many cases verification tools can automatically provide an answer about the security of a protocol in isolation, even when multiple concurrent sessions are considered. However, in the real-world, protocols do not run in isolation. They are typically components of more complex systems, where they are executed along with different protocols. Several examples have demonstrated that protocol composition is not secure in general (e.g. [11, 65]), even if individual components are secure in isolation. Therefore, researchers have investigated both the verification of complex systems and the identification of sufficient conditions for the safe composition of protocols which are secure in isolation.

While Partial Order Reduction introduces a formalism of asynchronous components, only their execution order is used to devise a simpler proof. In complex protocols, ordered small components can be truly valid in regard to their security properties. Proving the correctness of a multitude of small models would lead to a greatly simplified proof effort, but demonstrating the compositionality of a protocol is non-trivial. A divide and conquer framework was introduced in [11] where large protocols are expressed as multiple smaller ones. In the same spirit, identifying how encryption between protocols can be non-overlapping is presented in [67]. A non-decomposable process is referred to as *atomic* [42]. When protocols are allowed to interact between each other in a stateful way *i.e.* having access to information from previous sessions, secure composition is still possible as outlined in [71]. When components can be isolated and tested separately, we talk about *sequential*, *parallel* or *vertical* compositionality.

Sequential Composition In a sequentially composed security protocol, one component's output is another one's input. To ensure a secure execution, processes are annotated with pre- and post-conditions. When such invariants are satisfied, the protocols' steps are considered securely usable in their composed form. A common application of this philosophy is to have one key-establishment protocol, and another one after it that uses this key for message exchange over a secure channel like in [85]. A sequence of message transmissions and receptions for an agent is called a *strand*, and the relationships between strands for every agent form a graph: the *strand space* [103]. Figure 1.4 shows the strand space for a challenge-response where agents A and B agree on a number N using a private symmetric key KAB. The protocol is then considered correct when every subgraph, or *bundle*, contains at least one honest agent

behavior, and when injecting an intruder leads to a *strand space* that is isomorphic to the original one.

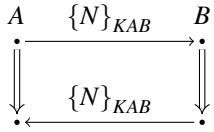

Figure 1.4 Example of strand space in communications

Parallel Composition Parallel composition is the situation when different protocols run in parallel. The first notion to be considered is protocol independence. According to Guttman and Thayer [67], two protocols are independent if the achievement of goals in one protocol does not depend on the other protocols being in use. They proved a theorem that holds even if the protocols share public key certificates and secret key "tickets" (e.g. Kerberos). In general, interdependence can arise when protocols share a piece of data, like long-term keys. Furthermore, if an honest agent is confused about what protocol is being executed, the intruder might exploit it. To ensure that the received messages are sufficiently distinguishable, Cortier *et al.* opt for a *tagging* approach [45]. A tagged message includes an identifier of the protocol it comes from. For example, adding the protocol's name to an encrypted payload avoids using a ciphertext that was intended for another protocol. This way, even if the two messages are encrypted with the same key, an intruder could not make an honest agent running a protocol accept a message intended for another protocol. Generalizing beyond secrecy goals to the entire geometric fragment proposed by Guttman [66], it is possible to perform secure parallel composition without tagging. This result was presented in [8] where two main syntactic conditions are checked: if a given protocol is type-flaw-resistant and if the protocols in a given set are pairwise parallel-composable. Moreover, requirements are more relaxed: it is sufficient that non-atomic subterms are not unifiable unless they belong to the same protocol and have the same type.

Vertical Composition The last two compositions can be named *horizontal*, as *vertical* composition [65] focuses on protocol encapsulation, like when an application runs on top of a secure TLS channel. While such a composition is reasonable most of the time with disjoint protocols at each layer [85, 42], the self-composition case is more complicated. If the same channel is to be re-used in the protocol stack, stronger conditions must be satisfied. According to Mödersheim and Viganò, message formats disjointness and uniqueness of each payload usage context constitute sufficient conditions [86] for such vertical composition, satisfied in practice by a large class of protocols. A stronger result is presented in [63], where the number of transmissions a channel is allowed to carry is unbounded, and with support for stateful protocols.

1.4 SPECIFICATION LANGUAGES AND TOOLS

To tackle the challenges of security protocols verification, several approaches and tools have been adopted in the past decades. Popular notations like *Alice and Bob* narrations enable high-level specification, while other languages trade simplicity for expressivity. Every concrete tool handles a specific set of behavioral properties and this section presents some of the most widespread or promising frameworks.

1.4.1 ANB LANGUAGE

In the most classical *AnB (Alice and Bob)* notation [83], a protocol is specified by a list of steps (or actions), each representing a message exchange between two honest agents like $A \rightarrow B : Msg$. This notation makes the encoding of security protocols considerably simpler (and more compact) than their equivalents in other formal languages (e.g. process calculi [4, 1]) or real-world programming languages. This intuitive language allows learners to build their own models of security protocols and experiment with them using tools that support such a notation like the model checker OFMC [84] and Tamarin [100].

An example of an *AnB* protocol is displayed in Figure 1.6. The main goal of the protocol is to achieve authentication (precisely the injective agreement defined in [77]) on the message Msg. The recipient B should have evidence that the message has been endorsed by A and is a fresh message. The goal is achieved using asymmetric encryption and a challenge–response technique with a nonce exchange. The abstract functions pk and sk are used to model asymmetric encryption, mapping agents to their public keys for encryption and signature respectively, while inv is a private function modeling the private key of a given public key. It should be noted that private function symbols are used to symbolically represent a notion. They are not concrete functions that can be computed [83].

The succinctness of the *AnB* notation comes with an expressivity drawback. The specification relies on implicit assumptions on how a receiver processes a message or how agents handle errors with unexpected messages. Therefore, it is potentially ambiguous for a novice unaware of such implicit assumptions. Some research focuses on extending its expressivity with explicit message formats [9] and user-specified equational theories [24].

1.4.2 ANBX LANGUAGE

In this spirit, the *AnBx* language (formally defined in [37]) is built as an extension of *AnB* [83]. The main peculiarity of *AnBx* is to use channels as the main abstraction for communication, providing different authenticity and confidentiality guarantees for message transmission, including a novel notion of *forwarding* channels, enforcing specific security guarantees from the message originator to the final recipient along a chain of intermediate forwarding agents. The translation from *AnBx* to *AnB* can be parametrized using different channel implementations, by means of different cryptographic operations. It also supports private function declarations,

```
Protocol: Example_AnBx
Types:
    Agent A,B;
    Certified A,B;
    Number Msg;
    SymmetricKey K;
    Function [Agent,Number -> Number] log
Knowledge:
    A: A,B,log;
    B: B,A,log
Actions:
    A -> B, @(A|B|B): K
    B -> A: {|Msg|}K
    A -> B: {|hash(Msg),log(A,Msg)|}K
Goals:
    K secret between A,B
    Msg secret between A,B
    A authenticates B on Msg
    B authenticates A on K
    B authenticates A on Msg
```

Figure 1.5 *AnBx* protocol example

useful for example when two honest agents want to share a secret shared key function. It can be explicitly excluded from the intruder's knowledge with a signature as $[ParamTypes \rightarrow\!\!* \ ReturnType]$.

Figure 1.5 shows an example protocol in which two agents want to exchange securely a message *Msg*, using a freshly generated symmetric key *K*, *i.e.* a key that is different for each protocol run. If *K* is compromised, neither previous nor subsequent message exchanges will be compromised, only the current one. This is similar to what happens in TLS where a symmetric session key is established (using asymmetric encryption) at the beginning of the exchange. It should also be noted that this setting is more efficient, as symmetric encryption is notoriously faster than the asymmetric kind. Therefore, if the size of the message is significant, using symmetric encryption should be preferable.

The *Types* section includes declarations of identifiers of different types and functions declarations, while the *Knowledge* section denotes the initial knowledge of each agent. An optional section, *Definitions*, can be used to specify macros with parameters. In the *Actions* section, the action $A \rightarrow B, @(A|B|B) : K$ means that the key *K* is generated by *A* and sent on a secure channel to *B*. The notation $@(A|B|B)$ denotes the properties of the channel: the message originates from *A*, it is freshly generated (@), verifiable by *B*, and secret for *B*. How the channel is implemented is delegated to the compiler. The designer can choose between different options, or simply use the default one. This simplifies the life of the designer, who does not need to be in charge of low-level implementation details. A translation to *AnB* is shown in Figure 1.6: in this case the channel is implemented using a challenge-response technique, where *B* freshly generates a nonce (the challenge) encrypted with *pk(A)*, the public

```
Protocol: Example AnB
Types:
    Agent A,B;
    Number Msg,Nonce;
    SymmetricKey K;
    Function pk,sk,hash;
    Function log
Knowledge:
    A: A,B,pk,sk,inv(pk(A)),inv(sk(A)),hash,log;
    B: A,B,pk,sk,inv(pk(B)),inv(sk(B)),hash,log
Actions:
    A -> B: A
    B -> A: {Nonce,B}pk(A)
    A -> B: {{Nonce,B,K}inv(sk(A))}pk(B)
    B -> A: {|Msg|}K
    A -> B: {|hash(Msg),log(A,Msg)|}K
Goals:
    K secret between A,B
    Msg secret between A,B
    A authenticates B on Msg
    B authenticates A on K
    B authenticates A on Msg
    inv(pk(A)) secret between A
    inv(sk(A)) secret between A
```

Figure 1.6 *AnB* protocol example

key of A along with the sender name ($\{\cdot\}$ denotes the asymmetric encryption). This guarantees that only A would be able to decrypt the incoming message.

The response, along with the challenge, includes the symmetric key K. The response is digitally signed with *inv(sk(A))*, the private key of A and then encrypted with *pk(B)*, the public key of B. This will allow B to verify the origin of the message and that K is known only by A and B.

It should be noted that in *AnBx*, we abstract from these cryptographic details, and we simply denote the capacity of A and B to encrypt and digitally sign using a Public Key Infrastructure (PKI) with the keyword *Certified*. This reflects the customary practice of a Certification Authority to endorse public keys of agents, usually issuing X.509 certificates, allowing every agent to verify the identity associated with a specific public key. Moreover, in *AnBx*, keys for encryption and for signature are distinguished by using two different symbolic functions, *pk* and *sk* respectively.

Once the symmetric key K is shared securely between A and B, then B can send the payload *Msg* secretly ($\{|\cdot|\}$ denotes the symmetric encryption). Finally, A acknowledges receipt, replying with a digest of the payload computed with the *hash* function (a predefined function available in *AnBx*), and with a value computed with the *log* function.

The section *Goals* denotes the security properties that the protocol is meant to satisfy. They can also be translated into low-level goals suitable for verification with various tools. Supported goals are as follows:

1. *Weak Authentication* goals have the form *B weakly authenticates A on Msg* and are defined in terms of non-injective agreement [77].
2. *Authentication* goals have the form *B authenticates A on Msg* and are defined in terms of injective agreement on the runs of the protocol, assessing the freshness of the exchange.
3. *Secrecy* goals have the form *Msg secret between* $A_1, \ldots A_n$ and are intended to specify which agents are entitled to learn the message *Msg* at the end of a protocol run.

In the example protocol (Figure 1.5), the desirable goals are the secrecy of the symmetric key K and of the payload *Msg* that should be known only by A and B. There are also authentication goals: B should be able to verify that K originates from A and that the key is freshly generated. Finally, two goals express the mutual authentication between A and B regarding *Msg*, including the freshness of the message. In summary, this protocol allows two agents to securely exchange a message, with guarantees about its origin and freshness.

1.4.3 OFMC MODEL CHECKER

In the cases studies presented in Section 1.5, the OFMC model checker [84] is used for the verification of abstract models. OFMC employs the AVISPA Intermediate Format IF [21] as "native" input language, defining security protocols as an infinite state-transition system using set-rewriting. Notably, OFMC also supports the more intuitive language *AnB* [83], and the *AnB* specifications are automatically translated to IF.

OFMC performs both protocol falsification and bounded session verification by exploring the transition system in a demand-driven way. If a security goal is violated, an attack trace is provided. The two major techniques employed by OFMC are the *lazy intruder*, a symbolic representation of the intruder, and *constraint differentiation*, a general search-reduction technique that combines the lazy intruder with ideas from Partial Order Reduction.

1.4.3.1 Channels as Assumptions

In general, along with the usual insecure channel, the *AnB* language supported by OFMC allows specifying three other types of channels: *authentic*, *confidential*, and *secure*, with variants that allow agents to be identified by a pseudonym rather than by a real identity. The supported standard channels are:

1. $A \rightarrow B : M$, an insecure channel from A to B, under the complete control of a Dolev-Yao intruder [53].
2. $A \bullet\!\!\rightarrow B : M$, an authentic channel from A to B, where B can rely on the facts that A has sent the message M and meant it for B.
3. $A \rightarrow\!\!\bullet B : M$, a confidential channel, where A can rely on the fact that only B can discover the message M.
4. $A \bullet\!\!\rightarrow\!\!\bullet B : M$, a secure channel (both authentic and confidential).

Pseudonymous channels [85] are similar to standard channels, with the exception that one of the secured endpoints is logically tied to a pseudonym instead of a real name. The notation $[A]_\psi$ represents that an agent A is not identified by its real name A, but by the pseudonym ψ. Usually, ψ can be omitted, simplifying the notation to $[A]$, when the role uses only one pseudonym for the entire session, as it is in our case and in many other protocols.

For example, $[A] \bullet\!\!\rightarrow B : M_1$ denotes an authentic channel from A to B, where B can rely on the facts that an agent identified by a pseudonym has sent a message M_1 and that this message was meant for B. If during the same protocol run, another action like $[A] \bullet\!\!\rightarrow B : M_2$ is executed, B can rely on the facts that the same agent (identified by the same pseudonym) has also sent M_2 and again that the message was meant for B.

Assuming that B does not already know the real name of A, the execution of these two actions does not allow B to learn the real identity of A (unless this information is made available during the protocol execution), but B has a guarantee that it was communicating with the same agent during both message exchanges. The term *sender invariance* is used to refer to this property, and the most common example is the TLS protocol without client authentication.

1.4.4 ANBX COMPILER AND CODE GENERATOR

The *AnBx* Compiler and Code Generator [89] is an automatic Java code generator for security protocols specified in *AnBx* or *AnB*. The *AnBx* compiler can be used in the context of Model Driven Development. Provided that a model has been validated with OFMC, the user can automatically generate a Java implementation. Along with this immediate benefit, in a learning context [90] this is useful to familiarize oneself with the software engineering approach of Model Driven Development, but also to compare a manual implementation with a generated one. The main features of the compiler are:

- Automatic computation of the defensive checks that an agent has to perform on incoming messages.
- Optimization of cryptographic operations in order to minimize the number of computational steps and reduce the overall execution time [88].
- Mapping of abstract types and API calls to the concrete ones provided by the AnBxJ library.
- A set of template files is used to generate the code. Template files can be customized, for example, to integrate the generated application in larger systems.

Since the compiler translates the intermediate format to the applied pi-calculus [1], the verification of the protocol logic used for the code emission phase can be performed with the protocol verifier ProVerif [30].

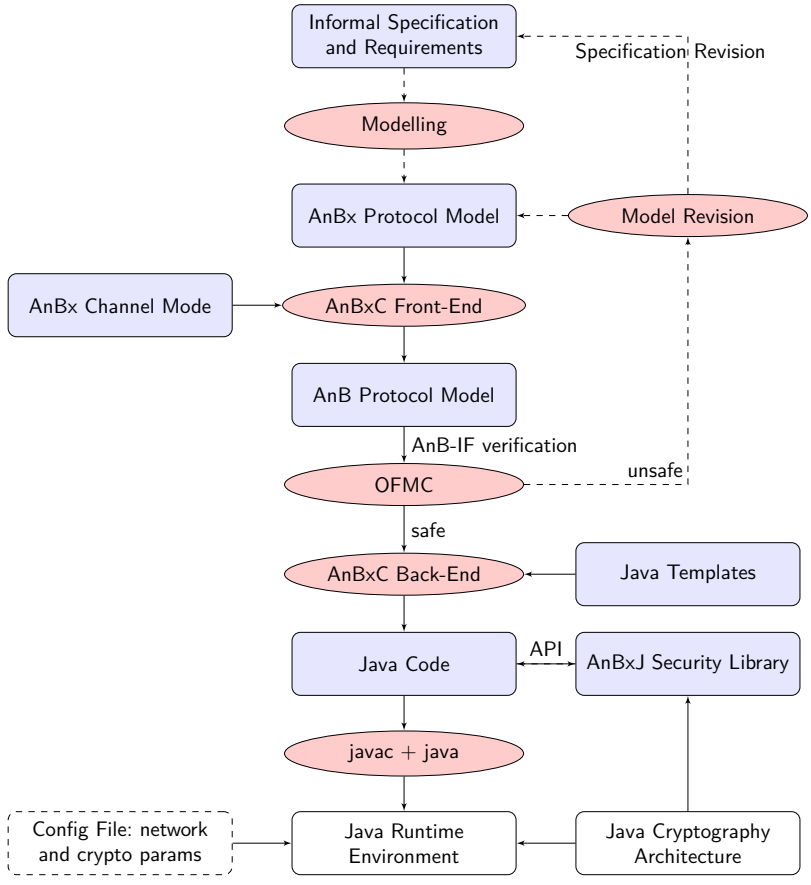

Figure 1.7 Model driven development with *AnBx* (- - - - - manual ——— automatic)

1.4.4.1 AnBx Compiler Architecture

In this section, we introduce the development methodology (Figure 1.7) and provide an overview of the architecture of the *AnBx* compiler (Figure 1.8), a tool built in Haskell, which is one of the key components of this methodology. Other tools, included in the toolkit, are the OFMC symbolic model checker [84] and the cryptographic protocol verifier ProVerif [30]. All components are integrated in the *AnBx* IDE [60], an Eclipse plug-in for the design, verification and implementation of security protocols. Along with the integration of the back-end tools, the IDE includes many features meant to help programmers to increase their productivity, like syntax highlighting, code completion, code navigation and quick fixes.

1.4.4.2 Development Methodology

As seen in Section 1.2, the work of a developer usually begins from gathering the available specification documentation and build a model that can be verified and then used to construct an implementation. Expressing requirements in a simple but formalized language for the specification of security protocols that is amenable for the verification of the model is a key aspect of the approach presented here. Although formal languages like the SPI [4] and applied pi-calculus [2] have been created to model and verify security protocols, their usage among software developers in the industry remains limited due to their complexity. On the contrary, protocol narrations in the *Alice & Bob* style are much closer to the familiar way software engineers use to describe security protocols. Therefore, this methodology adopts the simple *AnB* notation [83] and its extension *AnBx* [37].

One of the main advantages of such languages, which share common traits but are syntactically different, is that they can be quickly learned by developers as they are rather intuitive. Their syntax is similar to the informal or semi-formal languages used in the documentation software engineers are familiar with, but their semantics are formally defined so there are no ambiguities in the way the system interprets them. We have direct experience of students developing a few person-month projects, being able to learn effectively the specification language in a few days or less.

Once the protocol specification has been completed, it is possible to use the *AnBx* compiler to translate the input file to *AnB*, a format which can be verified with the OFMC model checker [84]. It should be underlined that the translation from *AnBx* can be parametrized using different channel implementations that convey the security properties specified at the channel level, by means of different cryptographic operations [37]. The compiler can also directly process protocols in *AnB*. In all cases, the compiler will perform a number of sanity and type checks to ensure that the input provided to OFMC is fully sanitized. This is important, because OFMC lacks an *AnB* type checker, and there are situations where the model checker accepts (and might deem secure) protocols which are ill-typed.

If OFMC finds an attack, *i.e.* a security goal can be violated by the intruder, the developer can manually (and iteratively) revise the model until the model checker concludes on a safe model. If necessary, changes can be backported to the original standard. It should be noted that if the user is developing a brand-new protocol, this is an effective way to iteratively prototype security protocols. In that respect, *AnBx*, given the channel abstractions, can also be considered as a design language along to its nature of specification language.

When OFMC verifies the protocol as secure, the *AnB* specification can be passed to the compiler back-end (described later in this section) and generated to Java source code. In practice, the compilation is fully automated from *AnBx* to Java (Figure 1.7). The compiler uses application template files, customizable by the user and written in the target language, to integrate the generated code in the end-user application. The templates are instantiated by the compiler with the information derived from the protocol specification.

The run-time support is provided by the cryptographic services offered by the

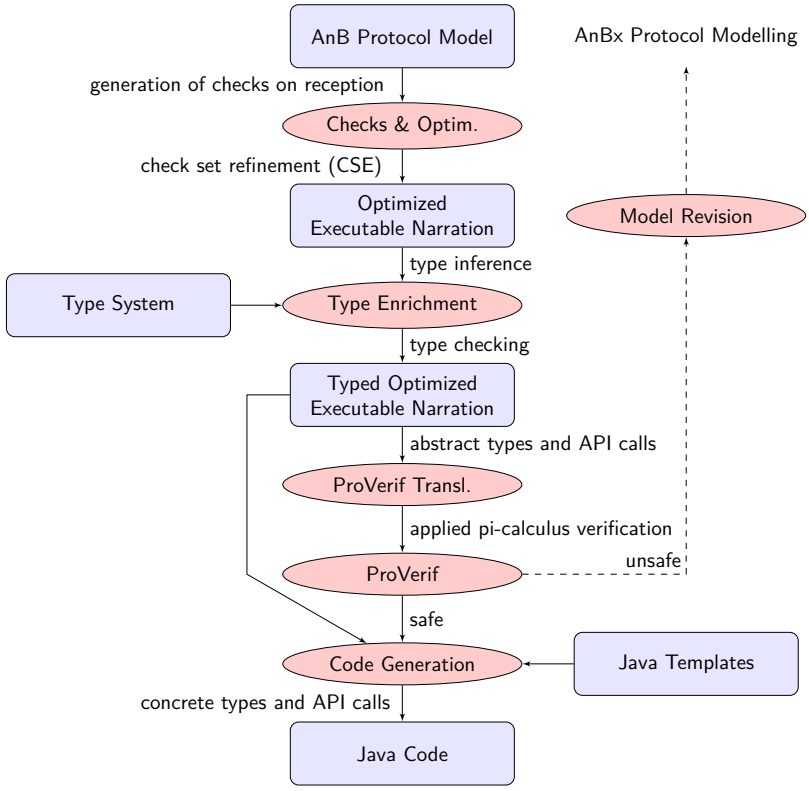

Figure 1.8 Compiler back-end: Type system, code generator, verification (- - - - - manual ——— automatic)

Java Cryptography Architecture (JCA) [64, 96]. In order to connect to the JCA, a security API (called *AnBxJ* library) wraps, in an abstract way, the JCA interface and implements the custom classes necessary to encode the generated Java programs. The *AnBxJ* library guarantees a high degree of generality and customization, since the API allows to write code that does not commit to any specific cryptographic solution (algorithms, libraries, providers). This code can be instantiated at run-time using a configuration file that allows the developer to customize the deployment of the application at the cryptographic (key store location, aliases, cipher schemes, key lengths, etc.) and network level (IP addresses, ports, etc.) without needing to regenerate the application. The library also gives access to the communication primitives used to exchange messages in the standard TCP/IP network environment, including secure channels like TLS. Communication and cryptographic run-time errors are handled at this level, and exceptions are raised accordingly.

1.4.4.3 *AnBx* Compiler Back-End

The second phase of the compilation process (Figure 1.8) begins with the translation of *AnB* to an *(Optimized) Executable Narration*, a set of actions that operationally encodes how the different agents are expected to execute the protocol. The core of this step is the automatic generation of the consistency checks derived from the static (implicit) information included in the model specification. The checks are defined as consistency formulas; some simplification strategies can be applied to reduce the number of generated formulas and speed up the application. A further step in this direction is the application of other optimization techniques [88], including common sub-expression elimination (CSE), which in general are useful to generate efficient code. In particular, our compiler considers a set of cryptographic operations, which are computationally expensive, to reduce the overall execution time by storing partial results and reordering instructions with the purpose of minimizing the overall number of cryptographic operations.

The next stage is the construction of the *Typed Optimized Executable Narration*, a typed abstract representation of the security-related portion of a generic procedural language supporting a rich set of abstract cryptographic primitives. For that purpose, a type system infers the types of expressions and variables, ensuring that the generated code is well-typed. It has the additional benefit of detecting at run-time whether the structure of the incoming messages is equal to the expected one, according to the protocol specification. This step is necessary, as the type system of *AnB* is too simple, and unsuitable to represent the complexity of a complete implementation of a protocol, in Java for example.

The *Typed Optimized Executable Narration* can be translated into applied pi-calculus and verified with ProVerif [30]. To generate applied pi-calculus, *AnB* security goals need to be analyzed at the initial phase and specific annotations modeling the security properties need to be generated and preserved along the compilation chain. The verification with ProVerif is preliminary to the code emission which is performed by instantiating the protocol templates. It is worth noting that only at this final stage, the language-specific features and their API calls are actually bound to the typed abstract representation of the protocol built so far. To this end, two mappings are applied by the compiler. One between the abstract and the concrete types, the other one between the abstract actions and the concrete API calls.

In summary, the *AnBx* compiler allows for a one-click code generation of a widely configurable and customizable ready-to-run Java application from an *AnBx* or *AnB* specification. In addition to the Java classes and the configuration file, an Ant [13] build file is also generated to easily build and run the application. It is also important to underline that the application templates are generic, *i.e.* independent of the specific protocol, and can be modified by the user in order to integrate the generated code in the required application domain.

1.4.5 PROVERIF

ProVerif [30] is an automated verifier for cryptographic protocols, modeling the protocol and the attacker according to the Dolev-Yao [53] symbolic approach. Differ-

ently from model checkers, ProVerif can model and analyze an unbounded number of parallel sessions of the protocol, by using Horn clauses to model a protocol. However, like model checkers, ProVerif can reconstruct a possible attack trace when it detects a violation of the intended security properties. It can check equivalence properties, which consists in determining that two processes are indistinguishable, from a third-party point of view.

ProVerif may report false attacks, but if a security property is reported as satisfied, then this is true in all cases, so it is necessary to analyze the results carefully when attacks are reported. The checks honest agents should perform on received messages must be explicitly stated, increasing the risk of user error. A recent development [32] enables support for axioms, lemmas and restrictions, allowing users to declare intermediate properties helping ProVerif to complete proofs. Those axioms can now be used to handle global mutable states, particularly useful for contract signing protocols [14]. To verify protocols in the computational model, a variant called CryptoVerif [31] exists.

1.4.5.1 Horn Clauses and Applied pi-calculus

Horn clauses were firstly used by Weidenbach [111] in protocol verification as a sound abstraction technique, overestimating the attacker's possible knowledge. Usually, trying to prove a security property is convenient with Horn clauses, as one would have to test its negation and check for a contradiction later in the proof. A Horn clause is a disjunction of terms such as at most one might be true. There are three types of Horn clauses:

- Definite clause with exactly one true term among false ones. The clause would read as $\neg p \vee \neg q \vee r$ which, rewritten with an implication would be $p \wedge q \Rightarrow r$. This means that r holds as long as p *and* q are true. This clause is then used for deduction.
- Fact clause with a true term only. A lone variable p is equivalent to assuming that p is true.
- Goal clause with only false terms. $\neg p \vee \neg q \vee \neg r$ would mean that the properties encoded by all three variables hold when the expression is unsatisfiable.

As an example, we can take a Dolev-Yao attacker, which then knows the network and what transits on it. We obtain an initial knowledge fact *attacker(network)*. A plaintext message exchange is then performed as $A \rightarrow B : Msg$ and *Msg* is sent over the network, which is another fact we can write as *send(network,Msg)*. The classical eavesdropping attacker rule follows as $send(x,y) \wedge attacker(x) \Rightarrow attacker(y)$. With this rule, we can derive the attacker's knowledge *attacker(Msg)*, proving that *Msg* is not secret in this case.

ProVerif uses the applied pi-calculus [1], suited to specify the behavior of concurrent processes, emphasized around communicating agents. In applied pi-calculus, each participant of the protocol is represented by a process and the messages exchanged by processes are the messages of the protocol. For example, the exchange:

"The first process sends *a* on channel *c*, the second one inputs this message, puts it in variable *x* and sends *x* on channel *d*" is encoded by *out(c, a)* ‖ *in(c, x).out(d, x)*. The core feature of ProVerif is to translate it to Horn clauses, but only an approximation of a translation is possible sometimes, which can lead to non-termination.

1.4.6 IDENTIFICATION OF ATTACKS

A protocol can have multiple desired properties that must be verified, as expressed in the *AnB Goals* section. A verification tool or framework should be able to tell the user exactly what failed and in what circumstances. OFMC, while providing one of the simplest attack traces, only outputs one failing goal, the one which failed first. ProVerif prints every one of them, but it is easy to have an unverified goal passing the verification because of some user specification error. To have an intuitive visualization of failing goals, the *AnBx* IDE [60] provides a convenient workflow for individual goal verification.

1.4.7 OTHER TOOLS AND COMPARISON

Tamarin [80] is a state-of-the-art symbolic verifier supporting both automatic and interactive modes. The user then has the option to manually guide the proof process with user-specified lemmas.

The Scyther tool [50] shares a backward reasoning approach based on patterns with Tamarin and has guaranteed termination. Maude-NPA [56] shares a lot of characteristics with Tamarin, with a notable exception of not supporting global mutable states and the fact that Maude-NPA models protocols by strands other than multiset rewriting.

Verifpal [73] is a recent tool which focuses on providing an easy to use and to understand design framework. Its specification language resembles an *AnB* protocol narration, and it uses heuristics to limit the state explosion.

In the AVISPA tool suite [17], OFMC is used in conjunction with two other back-ends: CL-AtSe [106] and SATMC [15]. CL-AtSe takes as input a service specified as a set of rewriting rules, and applies rewriting and constraint solving techniques to model all reachable states. SATMC uses SAT solvers to perform a bounded analysis with propositional formulas. Those three tools are now part of the AVANTSSAR platform [16], extending AVISPA capabilities to *Service-Oriented Architectures*.

A comprehensive survey of recent advances in verification can be found in [22], of which we use the comparison criteria in Table 1.1.

1.4.7.1 Supported Properties

All tools except the back-ends from the AVISPA/AVANTSSAR suite support unbounded verification, accepting undecidability in some cases. ProVerif, Tamarin and Maude-NPA can check diff-equivalence properties [52], testing if protocols have the same structure and differ only by the messages they exchange. Verifpal can test some equivalence queries, but limited to any protocol scenario which can be derived such that some shared secrets are not equivalent to one another.

Tool	Unbounded	Equiv	Eq-thy	State	Link
ProVerif [30]	●	●	◐	●	○
Tamarin [80]	●	●	●	●	○
Verifpal [73]	●	◐	◐	●	◐
OFMC [84]	○	○	◐	●	○
CL-AtSe [106]	○	○	●	●	○
SATMC [15]	○	○	○	●	○
Maude-NPA [56]	●	●	●	○	○
Scyther [50]	●	○	○	○	○

Table 1.1
Comparison between tools for symbolic security analysis

Tamarin, CL-AtSe and Maude-NPA are the only ones to allow for user-defined equational theories with the Finite Variant Property in general [54], CL-AtSe being however limited to subterm-convergent equational theories [20]. ProVerif and OFMC only permit such theories without associative-commutative axioms, Verifpal is limited to specifying simple equations for signature or public-key establishment like with Diffie-Hellman, and the other ones have fixed or no equational theories.

A global mutable state is useful to model databases like key servers or shared memory in general [75, 106], but Maude-NPA and Scyther lack support for it.

The last criterion is about linking the model to an implementation, providing executable protocols exhibiting symbolic security properties. Verifpal can generate Coq implementations of its models, with Go code generation planned as well, while now being usable as a Go library.

1.5 CASE STUDIES

We present here some practical examples of verification and re-engineering of security protocols. The purpose of these case studies is to give a practical demonstration, within the context of an approach of formal methods for security, of modelling and design techniques that can be applied by practitioners working with real-world protocols. We first consider the formal modelling and verification of the Bitcoin payment protocol BIP70 (originally presented in [91]) and then we present the modelling, verification and re-engineering of the *iKP* e-commerce protocols (originally presented in [39, 87]). A similar approach (originally presented in [37]) can be used for *SET*, an e-commerce protocol that for its complexity is considered as a benchmark for protocol analysis.

The specification languages considered here are *AnB* and its extension *AnBx*. These are intuitive languages that can be learned relatively quickly by practitioners. The verification tools are the model checker OFMC (for iKP, SET and BIP70) and the cryptographic protocol verifier ProVerif (for iKP and SET), that can accept the aforementioned languages directly or by translation using the *AnBx* compiler.

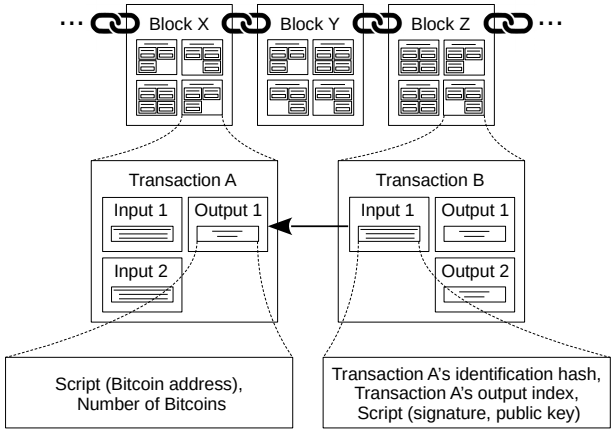

Figure 1.9 Information stored in Bitcoin transactions (courtesy [91])

1.5.1 BITCOIN PAYMENT PROTOCOL

Before describing the formal modelling technique used in [91] to specify the payment protocol, let us introduce some key concepts of the Bitcoin cryptocurrency.

A *Bitcoin address* in a Bitcoin network is the hash of an Elliptic Curve (EC) public key used as an identifier. The address is a pseudonym associated to the user possessing the corresponding private key. Such private key can be used to claim bitcoins sent to a user and authorize payments to other parties using ECDSA, the Elliptic Curve Digital Signature Algorithm. As the probability of collision is negligible considering the length of the output of the hash function, it is possible to safely assume that these identifiers are unique within the network.

A *transaction* records the transfer of bitcoins. It includes one or more inputs, specifying the origin of bitcoins being spent, and one or more outputs, specifying the new owner's Bitcoin address and the amount of the transaction (see Figure 1.9). To authorize the payment, the sender must specify an input consisting of the previous transaction's identification hash and an index to one of its outputs, and provide the corresponding public key and a valid digital signature. The inputs and outputs are controlled by means of scripts in a Forth-like language specifying the conditions to claim the bitcoins. The *pay-to-pubkey-hash* is the most common script to authorize the payment, requiring a single signature from a Bitcoin address.

The *blockchain* holds the entire transaction history of the network with a secure time-stamp [92] and is organized in blocks of transactions. The *ledger* is an append-only data structure stored in a distributed way by most users of the network. Solving a *proof of work* puzzle allows to append a new transaction to the blockchain. The nodes that solve proofs of work are called *miners,* and they are rewarded in bitcoins for their computational effort. A *proof of work* problem is computationally difficult to solve but easy to verify when a solution is provided.

1.5.1.1 Formal Modelling Approach and Formalization

The formal modelling and security analysis of the BIP70 Payment Protocol presented in [91] involves the symbolic model checker OFMC, and the specification language *AnB*. An important reason for adopting this approach is that this toolkit allows modelling communication channels as abstractions conveying security goals like authenticity and/or secrecy, without the need to specify the concrete implementation used to enforce such goals. It is therefore possible to rely on the assumptions provided by such channels. The result is a simpler model that is tractable by the verifier and can be analysed more efficiently, mitigating the state explosion problem.

Interestingly in *AnB* channels, agents can be identified by pseudonyms (e.g. ephemeral public keys) rather than by their real identities, as in secure channels like TLS without client authentication. Therefore, the capability of OFMC to verify a range of different channels specified in *AnB* makes the tool suitable for this verification effort. It should be noted that, as discussed in [91], BIP70 runs on top of abstract channels providing security guarantees, and the specific implementation of the underlying protocol is not part of the BIP70 specification. Thus, the analysis should be performed under the assumption that the sufficient conditions for vertical composition (e.g. in [86]) are satisfied.

The BIP70 Payment Protocol [12] was proposed in 2013 by Andresen and Hearn and later adopted by the Bitcoin community as a standard. The goal of the protocol is presented as follows:

> *"This BIP describes a protocol for communication between a merchant and their customer, enabling both a better customer experience and better security against man-in-the-middle attacks on the payment process."*

The communication channel between the customer and the merchant is strongly recommended being over HTTPS with the merchant authentication based on a X.509 certificate issued by a trusted Certificate Authority.

The actions performed and messages exchanged during the protocol run are shown in Figure 1.10. The protocol initiates with the customer clicking on the 'Pay Now' button on the merchant's website. This generates a Bitcoin payment URI that enables to open the customer's Bitcoin wallet and download the *Payment Request* from the merchant's website. The digital signature on the *Payment Request* can be verified by the wallet application with the public key of the merchant, checking the validity of the associated certificate. Provided the previous step is completed successfully, the bitcoin amount requested is shown to the customer requesting for authorization of the payment, along with the merchant's name extracted from the X.509 certificate's 'common name' field. If the user authorizes the payment, the wallet computes a payment transaction and broadcasts such information to the Bitcoin network. Moreover, transaction and refund addresses within a *Payment* message are sent back to the merchant. The merchant then sends back a *Payment Acknowledgement* to the customer wallet. Finally, the customer receives a confirmation of the payment when the transaction is detected on the blockchain.

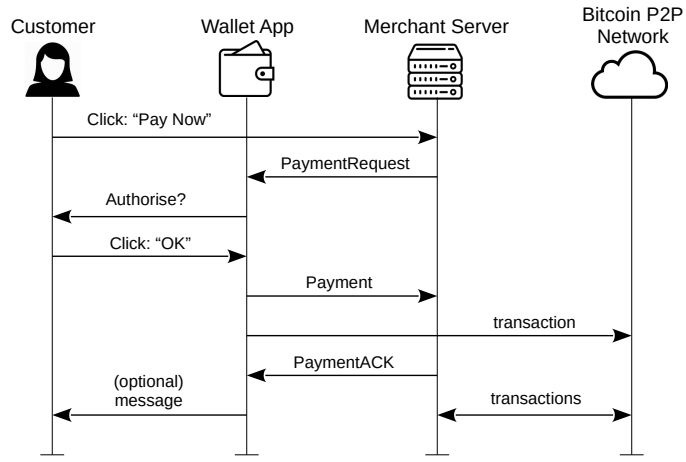

Figure 1.10 BIP70 Payment Protocol overview (courtesy [91], adapted from [12])

Modelling BIP70

This model considers a BIP70 protocol with n ($n \geq 1$) customers $C_1, \ldots, C_n$ and a single merchant M, as the standard specifies that a transaction may involve more customers. These agents should be able to trade over Bitcoin, and they should know the identity of the merchant. Strictly speaking, since multiple customers can cooperate in the payment of a single merchant, the model requires that at least one of the customers knows the merchant's name at the beginning of the protocol. However, it is not required that the merchant knows the identity of the customers. In fact, there is no provision of a communication mechanism between agents and merchants that explicitly discloses the real identity of the client. Such one-way authentication can be customarily achieved using HTTPS, as recommended in BIP70. In this case, the client has the guarantee that messages are exchanged with the authenticated server, but the server is only guaranteed that the communication channel is shared with the same pseudonymous agent. The pseudonym of the agent C_1 during the protocol run is represented by $[C_1]$.

The model also assumes that C_1 is the only agent that communicates directly with the merchant, while other agents communicate with C_1 to jointly set up the order for the merchant, using a secure channel (or out-of band). This is in accordance with the scenario in which payments may be made from multiple pseudonymous Bitcoin addresses, belonging to one or different entities. It is up to the customer communicating with the merchant to compose the payment transaction, coordinating with all the Bitcoin address holders. The model employs two kinds of channels:

- $[C_1] \bullet\!\!\rightarrow\!\!\bullet M$ represents a secure (secret and authentic) channel between the client C_1 and the merchant M; M can bind the other end point to a pseudonym $[C_1]$ rather than to the real identity of C.

Identifier	*Description*
B_M	Merchant Bitcoin address for the current transaction, a public key freshly generated by M with the corresponding private key denoted by $\mathsf{inv}(B_M)$
B_{C_i}	Customer C_i Bitcoin address for the current transaction, a public key freshly generated by C_i, with the corresponding private key denoted by $\mathsf{inv}(B_{C_i})$
R_{C_i}	Refund address of customer C_i
฿	Number of Bitcoins for the current transaction
$฿_{C_i}$	Number of Bitcoins to be refunded to R_{C_i} in case of a refund
t_1, t_2*	Timestamps indicating *Payment Request* creation and expiry times, resp.
m_M* m_C*, m'_M*	Memo messages included in the *Payment Request* (by M), *Payment* (by C), and *Payment Acknowledgement* (by M) messages
u_M*	Payment URL
z_M*	Payment id provided by the merchant

Table 1.2
Identifiers used to denote the data exchanged (* optional parameters)

- $C_i \bullet\!\!\rightarrow\!\!\bullet C_j$ represents a secure channel between the clients C_i and C_j.

The identifiers used in the messages exchanged during the protocol run are shown in Table 1.2. Additionally, the $\mathcal{H}$ symbol represents the hash function used to generate Bitcoin addresses and the following definitions are used in the message specification:

- $\omega_i = ฿_{C_i}, \mathcal{H}(B_{C_i})$: the previous transaction outputs for customer C_i.
- $\tau_{C_i} = \mathsf{tr}(\omega_i)$: the previous transaction for customer C_i. Future transactions depend only on unspent/spendable transaction outputs; the function tr that returns a transaction is parameterized on the output used by C_i in the current transaction.
- $\pi_{C_i} = \mathsf{sign}_{\mathsf{inv}(B_{C_i})}(\mathcal{H}(\tau'_{C_i}), B_{C_i})$: the transaction input endorsed by C_i.
- $\pi = \pi_{C_1}, ..., \pi_{C_n}$: the transaction input, a list of transaction inputs endorsed by the customers.
- $PaymentRequest = \mathsf{sign}_{\mathsf{inv}(\mathsf{sk}(M))}(\mathcal{H}(B_M), ฿, t_1, t_2, m_M, u_M, z_M)$: the *Payment Request*, a message digitally signed with $\mathsf{inv}(\mathsf{sk}(M))$, the private key of M. The associated public key utilized to verify the digital signature, that we denote as $\mathsf{sk}(M)$, is certified by a Certificate Authority and stored in a X.509 certificate.
- $RA_{C_i} = (R_{C_i}, ฿_{Ci})$: the refund address and amount for customer C_i.
- $\tau_C = \pi, (\mathcal{H}(B_M), ฿)$: one or more valid transactions, where π represents the inputs, and $(\mathcal{H}(B_M), ฿)$ represents the output.

Agents' Initial Knowledge

To keep the model simple, the initial knowledge of a single merchant M and two customers C_1, C_2 can be represented as:

- $C_1 : C_1, C_2, M, \mathcal{H}, \mathsf{tr}, \mathsf{sk}, \mathrm{paynow}$;
- $C_2 : C_1, C_2, \mathcal{H}, \mathsf{tr}, \mathsf{sk}$;
- $M : M, \mathcal{H}, \mathsf{tr}, \mathsf{sk}, \mathsf{inv}(\mathsf{sk}(M)), \mathrm{paynow}, t_1, t_2$

Each agent has an identity and access to the hash function $\mathcal{H}$, the symbolic function tr and a symbolic function sk for modelling digital signatures. In particular, the sk function allows customers C_i to retrieve $\mathsf{sk}(M)$ the public key of agent M from a repository, and verify the corresponding X.509 certificate, provided that they know the name of M.

$\mathsf{inv}(\mathsf{sk}(M))$ denotes the private key of M and is known only by M. We should note that in the *AnB* language, inv is a private function. Therefore, neither other agents nor the intruder can use inv to retrieve any agent's private key.

Initially, M does not know the identities of C_1 and C_2, while C_1 and C_2 know each other as they need to collaborate to build the transaction. However, only C_1 knows C_2 since C_1 will be the only customer interacting with the merchant. Finally, various constants (t_1, t_2, paynow) are available to agents.

The initial knowledge can be easily generalized for n customers; it should be noted that a customer does not need to know all other customers prior to the protocol run, but at least one. As customers can coordinate as they wish (including out-of-band communication), only one customer needs to interact with the merchant.

Security Goals

The following security goals are expected to hold after the protocol completion:

- **Goal 1:** *Refund Addresses Authentication*. M has a guarantee that all refund addresses R_{C_i}, for all $i = 1..n$ are provided by and linked to the customers involved in the transaction. In *AnB*, we denote the goal as:
  M *weakly authenticates* C_i *on* $R_{C_i}, ฿_{C_i}$ (for all $i = 1..n$).
- **Goal 2:** *Refund Address Agreement and Secrecy*. All refund addresses R_{C_i} are secret and known only by the merchant and the customers involved in the transaction. In *AnB*, we denote the goal as:
  $(R_{C_1}, \ldots, R_{C_n})$ *secret between* $M, C_1, \ldots, C_n$.

As the Payment Protocol is built on top of the core Bitcoin protocol and blockchain, a question that should be considered is whether the Payment Protocol is secure assuming the core Bitcoin protocol is secure. In this exercise, the security goals that are expected to be guaranteed by the core Bitcoin protocol and blockchain, such as the double-spending prevention, are assumed to be satisfied. This is a reasonable assumption as the security properties of the core Bitcoin protocol and blockchain have been formally proven in previous works [18, 41]. Similarly, we do not explicitly consider the security issues at the lower layers of the networking stack since the formalized model only encompasses the application layer and assumes that protocols such

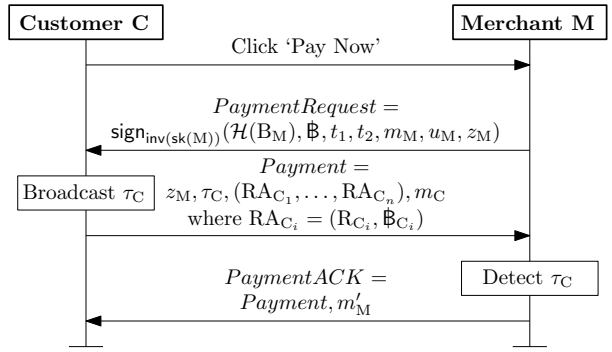

Figure 1.11 Expanded message contents for the Payment Protocol for C and M. Messages are sent over an HTTPS communication channel. $\mathsf{sign}_{\mathsf{inv}(\mathsf{sk}(\mathrm{M}))}(X)$ denotes both the message X and the digital signature on the message by the private key $\mathsf{inv}(\mathsf{sk}(\mathrm{M}))$ (courtesy [91]).

as TLS are secure. The approach of considering the security properties of different layers in isolation is sound, provided that the conditions of the vertical composition theorem [86] (see 1.3.4) are satisfied. It should be noted that the secrecy goal (2) prevents eavesdropping, and that known prediction and fixation vulnerabilities have been addressed by more recent versions of TLS [29, 49].

Protocol Actions

Agents are involved in a sequence of message exchanges over the designated channel. On the sender's side, agents should have enough information to compose the message, based on their initial knowledge and the new knowledge acquired during the protocol execution. On the recipient's side, every agent must decompose the incoming messages (e.g, decrypting the message or verifying a digital signature) according to their current knowledge. For the sake of simplicity, it is assumed that all public keys are available, at a certain point of the protocol execution, to the intruder and protocol participants.

$[\mathrm{C}_1]\bullet\!\!\rightarrow\!\!\bullet\mathrm{M}$	: paynow	C_1 *clicks* 'Pay Now'
$\mathrm{M}\bullet\!\!\rightarrow\!\!\bullet[\mathrm{C}_1]$	: *PaymentRequest*	*Payment Request*
$\mathrm{C}_1\bullet\!\!\rightarrow\!\!\bullet\mathrm{C}_2$	: $\mathrm{R}_{\mathrm{C}_1}\mathrm{M}$, *PaymentRequest*, $\mathrm{B}_{\mathrm{C}_1}$	$\mathrm{C}_1,\mathrm{C}_2$ *cooperate -*
$\mathrm{C}_2\bullet\!\!\rightarrow\!\!\bullet\mathrm{C}_1$	: $\mathrm{R}_{\mathrm{C}_2}, \pi_{\mathrm{C}_2}$	*- to build a transaction*
$[\mathrm{C}_1]\bullet\!\!\rightarrow\!\!\bullet\mathrm{M}$	: $z_{\mathrm{M}}, \tau_{\mathrm{C}}, \mathrm{RA}_{\mathrm{C}_1}, \mathrm{RA}_{\mathrm{C}_2}, m_{\mathrm{C}}$	*Payment*
$\mathrm{M}\bullet\!\!\rightarrow\!\!\bullet[\mathrm{C}_1]$	: $z_{\mathrm{M}}, \tau_{\mathrm{C}}, \mathrm{RA}_{\mathrm{C}_1}, \mathrm{RA}_{\mathrm{C}_2}, m_{\mathrm{C}}, m'_{\mathrm{M}}$	*PaymentACK*

Protocol Message Details

The format of the *Payment Request*, *Payment*, and *Payment Acknowledgement* messages is specified by the BIP70 standard. While the documentation only *recommends* running the protocol over HTTPS, this verification exercise assumes HTTPS is used. Moreover, although the standard supports payment via multiple transactions,

the details of the messages considered here are for the case where the customer pays through a single transaction. The formalization and verification results can be easily extended to the case where a payment is made through multiple transactions. The protocol messages are modelled as follows:

- The *Payment Request* consists of the recipient's Bitcoin address $\mathcal{H}(\mathrm{B_M})$, requested Bitcoin amount ฿, timestamps t_1, t_2 corresponding to the request creation and expiry times, a 'memo' field m_M for a note showed to the customer, the payment URL u_M where the payment message should be sent, and an identifier z_M for the merchant to link subsequent payment messages with this request. All the fields are collectively signed by the merchant using their private key denoted by $\mathsf{inv}(\mathsf{sk}(\mathrm{M}))$, corresponding to their X.509 certificate public key.
- The *Payment* message consists of the merchant's identifier z_M, the payment transaction τ_C, a list of pairs of the form $\mathrm{RA_{C_i}} = (\mathrm{R_{C_i}}, ฿_{\mathrm{C}i})$ each containing the refund address $\mathrm{R_{C_i}}$, the amount to be paid to that address $฿_{\mathrm{C}i}$ in case of refund, and an optional customer 'memo' field m_C.
- The *Payment Acknowledgement* consists of a copy of the *Payment* message sent by the customer and an optional 'memo' m'_M to be shown to the customer.

The Payment Protocol messages are shown Figure 1.11. Note that the *Payment* message, and especially the refund addresses provided therein, are not signed by the customer, and although protected by HTTPS, they can be subsequently repudiated by the customer. This is an underlying weakness that allows the Silkroad Trader attack discovered by McCorry *et al.* [79].

1.5.1.2 Main Results of BIP70 Security Verification

The formal model presented in [91] is encoded in the *AnBx* language [37], an extension of the *AnB* language supported by OFMC, allowing for macro definitions, functions type signatures and stricter type checking. The model was not tested with ProVerif because this tool does not support pseudonymous channels.

The tests were run on a Windows 10 PC, with Intel Core i7 4700HQ 2.40 GHz CPU with 16 GB RAM and the model was verified for a single session in OFMC in the classic and typed mode. As a result, the model demonstrated that both authentication and secrecy goals were violated. The attack was found in 2.34 seconds.

The authentication goal M *weakly authenticates* C_i *on* $\mathrm{R_{C_i}}, ฿_{\mathrm{C}_i}$ states that for all customers ($i = 1..n$), the merchant can have a guarantee of endorsement of the refund addresses and amounts.

In particular, the goal is violated because it is not possible to verify the non-injective agreement [77] between the construction of $\mathrm{RA_{C_i}} = (\mathrm{R_{C_i}}, ฿_{\mathrm{C}_i})$ done by C_i and the corresponding values received by M. This is possible because the customers are not required to endorse the value $(\mathrm{R_{C_i}}, ฿_{\mathrm{C}_i})$ using digital signatures. Therefore, a compromised or dishonest client can easily manipulate the refund address and perform the Silkroad Trader attack [79], where a customer can route Bitcoin payments

through an honest merchant to an illicit trader and later deny their involvement.

The secrecy goal $(R_{C_1}, \ldots, R_{C_n})$ *secret between* $M, C_1, \ldots, C_n$ is also violated. The definition of secrecy used in the model implies that all members of the secrecy set know the secret values and agree on them. But in this case, due to a lack of authentication, the customer who is communicating with the merchant can convince other customers that the refund address it is using is different from the one sent to the merchant. For example, R_{C_2}, the refund address of the second customer can be easily replaced with a different address by C_1 before being communicated to M.

It should be noted that, in general, with the automated verification, it is not possible to validate a specific attack trace known a priori, and the analysis usually aims at assessing the absence or presence of at least an attack trace that leads to a violation of a security goal. In particular, in order to verify the protocol, the model checker OFMC builds a state-transition system, and given the initial configuration, analyses the possible transitions in order to see if any attack state is reachable in the presence of an active attacker. Therefore, the presence of a specific attack trace is not automatically confirmed, rather such automated verification helps to decide whether any attack trace is present or absent, where an attack trace is defined as a sequence of steps leading to a violation of a given security goal. In order to verify a protocol's security, the most important thing is to guarantee the absence of any attack trace.

1.5.2 E-COMMERCE PROTOCOLS

We consider here the specification of a wide and interesting class of protocols, namely e-payment protocols, showing how *AnBx* naturally provides all the necessary primitives to reason about the required high-level security. The case studies we consider are the *iKP* and *SET* e-payment protocols, showing how *AnBx* lends itself to a robust and modular design that captures the increasing level of security enforced by the protocols in the family, depending on the number of agents possessing a certified signing key. Interestingly, as a by-product of these design and verification efforts, a new flaw was identified in the original *iKP* specification and a fix proposed.

1.5.2.1 A Basic e-Payment Scheme

The figure above shows an outline of a bare-bones specification of an e-payment protocol: this abstraction allows us to introduce here many concepts which are common to both of the examples considered.

We assume three agents: a Customer *C*, a Merchant *M* and an Acquirer *A*, *i.e.* a financial institution entitled to process the payment. In the model, each agent starts with an initial knowledge, which may be partially shared with other participants. Indeed, since most e-payment protocols describe only the payment transaction and do not consider any preliminary phase, we assume that the Customer and the Merchant

have already agreed on the details of the transaction, including an order description (*desc*) and a *price*. We also assume that the Acquirer shares with the Customer a customer's account number (*can*) comprised of a credit card number and the related PIN. The initial knowledge of the three parties can thus be summarized as follows: *C* knows *price, desc, can*; *M* knows *price, desc*; and *A* knows *can*.

The transaction can be decomposed into the following steps:

1. $C \rightarrow M$: *Initiate*
2. $C \leftarrow M$: *Invoice*
   (In steps 1 and 2 the Customer and the Merchant exchange all the information which is necessary to compose the next payment messages.)
3. $C \rightarrow M$: *Payment Request*
4. $M \rightarrow A$: *Authorization Request*
   (In steps 3 and 4 the Customer sends a payment request to the Merchant. The Merchant uses this information to compose an authorization request for the Acquirer and try to collect the payment.)
5. $M \leftarrow A$: *Authorization Response*
6. $C \leftarrow M$: *Confirm*
   (In steps 5 and 6 the Acquirer processes the transaction information, and then relays the purchase data directly to the issuing bank, which actually authorizes the sale in accordance with the Customer's account. This interaction is not part of the narration. The Acquirer returns a response to the Merchant, indicating success or failure of the transaction. The Merchant then informs the Customer about the outcome.)

Interestingly, steps (4) and (6) involve forwarding operations, since the Customer never communicates directly with the Acquirer. Still, some credit-card information from the Customer must flow to the Acquirer through the Merchant to compose a reasonable payment request, while the final response from the Acquirer must flow to the Customer through the Merchant to provide evidence of the transaction.

Besides some elements of the initial knowledge, other information needs to be exchanged in the previous protocol template. First, to make transactions unequivocally identifiable, the Merchant generates a fresh transaction ID (tid) for each different transaction. Second, the Merchant associates a *date* to the transaction or any appropriate timestamp. Both pieces of information must be communicated to the other parties. The transaction is then identified by a *contract*, which comprises most of the previous information: if Customer and Merchant reach an agreement on it, and they can prove this to the Acquirer, then the transaction can be completed successfully. The details on the structure of the contract vary among different protocols. At the end of the transaction, the authorization $auth$ is then returned by the Acquirer, and communicated to the two other participants.

We note that two main confidentiality concerns arise in the previous process: on the one hand, the Customer typically wishes to avoid leaking credit-card information to the Merchant; on the other hand, the Customer and the Merchant would not let the Acquirer know the details of the order or the services involved in the transaction. Specific protocols address such an issue in different ways, as we detail below.

Message Formats

In the case studies, a message M may be a name m, a tuple of messages $(\tilde{M})$, or a *message digest*. Much in the spirit of the *AnBx* channel abstractions, in fact, it turns out that we can abstract from most explicit cryptographic operations in the examples. Namely, the only transformation on data which we need to consider is the creation of a digest $[M]$ to prove the knowledge of a message M without leaking it.

We also consider digests which are resistant to dictionary attacks, hence presuppose an implementation based on a hashing scheme that combines the message M with a key shared only with the agent which must verify the digest. We note with $[M{:}A]$ a digest of a message M which is intended to be verified only by A; *hash* and *hmac* functions can be used to implement this in a standard way.

Protocol goals

A first goal we would like to satisfy for an e-payment system is that all the agents agree on the contract they sign. In terms of security goals, this corresponds to requiring that each participant can authenticate the other two parties on the *contract*. Moreover, the Acquirer should be able to prove to the other two parties that the payment has been authorized and the associated transaction performed: in OFMC this can be represented by requiring that M and C authenticate A on the authorization *auth*.

A stronger variant of the goals described above requires that, after completion of a transaction, each participant is able to provide a non-repudiable proof of the effective agreement by the other two parties on the terms of the transaction. In principle, each agent may wish to have sufficient proofs to convince an external verifier that the transaction was actually carried out as it claims. The lack of some of these provable authorizations does not necessarily make the protocol insecure, but it makes disputes between the parties difficult to settle, requiring to rely on evidence provided by other parties or to collect off-line information.

In summary, the authentication goals we would like to achieve are the following:

1. *M authenticates C on [contract]*, to give evidence to M that C has authorized the payment to M.
2. *C authenticates M on [contract]*, to give evidence to C on the terms of the purchase that M has settled with C.
3. *A authenticates C on [contract]*, to give evidence to A that C authorized A to transfer the money from A's account to M.
4. *A authenticates M on [contract]*, to give evidence to A that M has requested the transfer of the money to M's account.
5. *C authenticates A on [contract],auth*, to give evidence to C that A authorized the payment and performed the transaction.
6. *M authenticates A on [contract],auth*, to give evidence to M that A authorized the payment and performed the transaction.

Finally, we are also interested in some secrecy goals, like verifying that the Customer's credit card information *can* is kept confidential, and transmitted only to the

Acquirer. In general, we would like to keep the data exchanged secret among the expected parties. All validated protocol goals are reported for each case study.

1.5.3 IKP PROTOCOL FAMILY

mode/step	$\rightarrow$	*1KP*	*2KP*	*3KP*
η_1	$C \rightarrow M$	$(-\,|\,-\,|\,-)$	$(-\,|\,-\,|\,M)$	$@(C\,|\,M\,|\,M)$
η_2	$C \leftarrow M$	$(-\,|\,-\,|\,-)$	$@(M\,|\,C\,|\,-)$	$@(M\,|\,C\,|\,C)$
η_3	$C \rightarrow M$	$(-\,|\,-\,|\,A)$	$(-\,|\,-\,|\,A)$	$(C\,|\,A\,|\,A)$
η_{4a}	$M \rightarrow A$	$(-\,|\,-\,|\,A)$	$(-\,|\,-\,|\,A)$	$(C\,|\,A\,|\,A)$
η_{4b}	$M \rightarrow A$	$(-\,|\,-\,|\,A)$	$@(M\,|\,A\,|\,A)$	$@(M\,|\,A\,|\,A)$
η_5	$M \leftarrow A$	$@(A\,|\,C,M\,|\,-)$	$@(A\,|\,C,M\,|\,M)$	$@(A\,|\,C,M\,|\,M)$
η_6	$C \leftarrow M$	$(A\,|\,C,M\,|\,-)$	$(A\,|\,C,M\,|\,-)$	$(A\,|\,C,M\,|\,C)$
certified agents		A	M,A	C,M,A

Table 1.3
Exchange modes for the revised *iKP* e-payment protocol

	1KP		*2KP*		*3KP*	
Goal	*O*	*R*	*O*	*R*	*O*	*R*
can secret between *C,A*	+	+	+	+	+	+
A weakly authenticates *C* on *can*	–	–	–	–	+	+
desc secret between *C,M*	+	+	+	+	+	+
auth secret between *C,M,A*	–	–	–	–	–	+
price secret between *C,M,A*	–	–	–	+	–	+
M authenticates *A* on *auth*	+*	+	+*	+	+*	+
C authenticates *A* on *auth*	+	+	+	+	+	+
A authenticates *C* on [*contract*]	–	–	–	–	w	+
M authenticates *C* on [*contract*]	–	–	–	–	+	+
A authenticates *M* on [*contract*]	–	–	+	+	w	+
C authenticates *M* on [*contract*]	–	–	+*	+	+*	+
C authenticates *A* on [*contract*],*auth*	+	+	+	+	+	+
M authenticates *A* on [*contract*],*auth*	+*	+	+*	+	+*	+

* goal satisfied only after fixing the definition of Sig_A as in [39]
w = only weak authentication

Table 1.4
Security goals satisfied by Original and Revised *iKP*

The *iKP* protocol family was developed at IBM Research [26, 95] to support credit card-based transactions between customers and merchants (under the assumption that payment clearing and authorization may be handled securely off-line). All protocols in the family are based on public-key cryptography. The idea is that, depending on the number of parties that own certified public key-pairs, we can achieve increasing levels of security, as reflected by the name of the different protocols (1*KP*, 2*KP*, and 3*KP*).

1.5.3.1 Protocol Narration

Despite the complexity of *iKP*, by abstracting from cryptographic details, we can isolate a common communication pattern underlying all the protocols of the family. Namely, a common template can be specified as follows:

1. $C \rightarrow M, \eta_1 : [can{:}A], [desc{:}M]$
2. $C \leftarrow M, \eta_2 : price, tid, date, [contract]$
3. $C \rightarrow M, \eta_3 : price, tid, can, [can{:}A], [contract]$
4. $M \rightarrow A$ (decomposed into two steps to specify different communication modes)
   a. $M \rightarrow A, \eta_{4a} : price, tid, can, [can{:}A], [contract]$
   b. $M \rightarrow A, \eta_{4b} : price, tid, date, [desc{:}M], [contract]$
5. $M \leftarrow A, \eta_5 : auth, tid, [contract]$
6. $C \leftarrow M, \eta_6 : auth, tid, [contract]$

with $contract = (price, tid, date, [can{:}A], [desc{:}M])$.

By instantiating the exchange modes η_j in the previous scheme, one may generate the *AnBx* variants of the different protocols in the *iKP* family, achieving different security guarantees: this is exactly what is shown in Table 1.3. Notice that all the considered protocols rely on blind forwarding at step 4 to communicate sensitive payment information from the Customer to the Acquirer, without disclosing them to the Merchant. Moreover, a forwarding operation is employed at step 6 to preserve the authenticity of the response by the Acquirer.

1.5.3.2 Main Results of *i*KP Security Verification

The *AnBx* protocols described above were verified in [39, 37] and a corresponding analysis of the original specifications of $\{1,2,3\}KP$, as amended in [94], was carried out. Below, we refer to this amended version as the "original" *iKP*, to be contrasted with the "revised" *AnBx* version in Table 1.3. In both cases, the tests were run assuming that the Acquirer is trusted, *i.e.* encoded as a concrete agent a rather than as a role A; this is often a reasonable assumption in e-payment applications. As mentioned earlier, the *AnBx* specifications are not just more scalable and remarkably simpler, but they also provide stronger security guarantees, which are detailed in Table 1.4 and commented further below.

The *AnBx* specifications of *iKP* (and *SET*, see 1.5.4) were compiled into their respective cryptographic implementations with the *AnBx* compiler. We then verified

the generated *CCM* translation with OFMC [84] against the described security goals. The authors also encoded and verified the original versions of *iKP*, and compared the results with those of the revised versions.

The tests with OFMC were done with one and two symbolic sessions. This bounds how many protocol executions the honest agents can engage in, while the intruder is left unbounded thanks to the symbolic lazy intruder technique in OFMC, but with two sessions it was not possible to complete the full verification due to search space explosion. However, by translating the *AnBx* specification with the *AnBx* compiler to ProVerif (see [89]), it is possible to verify the protocol for an unbounded number of sessions.

During the analysis of the original 2*KP* and 3*KP,* a new flaw was found [39]. It is related to the authenticity of the Authorization response *auth* that is generated by the Acquirer and then sent to the other agents at steps 5 and 6. In particular, the starred goals in Table 1.4 are met only after changing the protocol by adding the identities of Merchant and Customer inside the signature of the Acquirer in the original specification. In 2*KP*, since the Customer is not certified, this can be done with an ephemeral identity derived from the credit card number. It is worth noting that, after the completion of the revised and the amended original 3*KP*, each party has evidence of transaction authorization by the other two parties, since the protocol achieves all the authentication goals that can ideally be satisfied, according to the number of certified agents. Moreover, the revised 3*KP*, with respect to the original version, provides the additional guarantee of preserving the secrecy of the authorization response $auth$.

In contrast, the original 3*KP* protocol, the strongest proposed version, fails in two authentication goals: *A* can only weakly authenticate *M* and *C* on [*contract*]. Luckily, if the transaction ID *tid* is unique, this is only a minor problem, since [*contract*] should also be unique, *i.e.* two different contracts cannot be confused.

1.5.4 SET PURCHASE PROTOCOL

Secure Electronic Transaction (*SET*) is a family of protocols for securing credit card transactions over insecure networks. This standard was proposed by a consortium of credit card and software companies led by Visa and MasterCard and involving organizations like IBM, Microsoft, Netscape, RSA and Verisign. The *SET* purchase protocol specification considered here is the one described in [25], where *signed* and *unsigned* variants of *SET* are presented: in the former all the parties possess certified key-pairs, while in the latter the Customer does not. We describe here the *AnBx* models of both variants of the original *SET* protocol, and show how, using the notion of *AnBx* channels, it is possible to re-engineer the protocol and obtain stricter security guarantees than in the original specification.

1.5.4.1 Protocol Narration

Given the complexity of *SET*, to ease the comparison with other works on such a protocol, in this presentation the information exchanged by the agents is denoted with the names commonly used in *SET* specifications. We introduce some basic concepts

of the protocol by simply providing a mapping of the exchanged data to the corresponding information in the bare-bones specification presented in Section 1.5.2: this should clarify the role of most of the elements.

We can identify *PurchAmt* with *price*, *OrderDesc* with *desc*, *pan* with *can* and *AuthCode* with *auth*. The initial knowledge of the three parties can then be summarized as follows: *C* knows *PurchAmt*, *OrderDesc* and *pan*; *M* knows *PurchAmt* and *OrderDesc*; *A* knows *pan*.

During the protocol run, the agents generate some identifiers: *LIDM* is a local transaction identifier that the Customer sends to the Merchant, while the Merchant generates another session identifier *XID*; we denote the pair (*LIDM*,*XID*) with *TID*. Finally, we complete the abstraction by stipulating $OIdata = OrderDesc$ and $PIdata = pan$; we let $HOD = ([OIdata{:}M], [PIdata{:}A])$. The latter contains the evidence (digest) of the credit card that the Customer intends to use, and the evidence of the order description that will later be forwarded to the Acquirer.

In the model, *HOD* plays the role of the *dual signature*, a key cryptographic mechanism applied in *SET*, used to let the Merchant and the Acquirer agree on the transaction without needing to disclose all the details. More precisely, the Merchant does not need the customer's credit card number to process an order, but only needs to know that the payment has been approved by the Acquirer. Conversely, the Acquirer does not need to be aware of the details of the order, but just needs evidence that a particular payment must be processed.

Although many papers on *SET* [25, 34, 107] focus on the signed version of the protocol, we note that both versions expose a common pattern which allows for an easy specification in *AnBx*. The following narration allows to expose the common structure of the protocols:

1. $C \to M, \eta_1$: *LIDM*
2. $M \to C, \eta_2$: *XID*
3. $C \to M$ (decomposed in two steps to specify different communication modes)
   a. $C \to M, \eta_{3a}$: *TID,HOD*
   b. $C \to M, \eta_{3b}$:*TID,PurchAmt,HOD,PIdata*
4. $M \to A$ (decomposed in two steps to specify different communication modes)
   a. $M \to A, \eta_{4a}$: *TID,PurchAmt,HOD,PIdata*
   b. $M \to A, \eta_{4b}$: *TID,PurchAmt,HOD*
5. $A \to M, \eta_5$: *TID,HOD,AuthCode*
6. $M \to C, \eta_6$: *TID,HOD,AuthCode*

Table 1.5 shows the communication modes we specify to instantiate the previous protocol template with the revised unsigned and signed versions of *SET*.

1.5.4.2 Main Results of SET Security Verification

The *AnBx* specifications of the *SET* purchase protocol were verified with OFMC and ProVerif. With OFMC, the models were verified for 2 sessions, by incrementing the depth of the search space, up to the available RAM (16Gb). ProVerif allows

mode/step	$\rightarrow$	*unsigned* SET	*signed* SET
η_1	$C \rightarrow M$	$(-\,|\,-\,|\,M)$	$@(C\,|\,M\,|\,M)$
η_2	$C \leftarrow M$	$@(M\,|\,C\,|\,-)$	$@(M\,|\,C\,|\,C)$
η_{3a}	$C \rightarrow M$	$(-\,|\,-\,|\,M)$	$@(C\,|\,M\,|\,M)$
η_{3b}	$C \rightarrow M$	$(-\,|\,-\,|\,A)$	$(C\,|\,A\,|\,A)$
η_{4a}	$M \rightarrow A$	$(-\,|\,-\,|\,A)$	$(C\,|\,A\,|\,A)$
η_{4b}	$M \rightarrow A$	$@(M\,|\,A\,|\,A)$	$@(M\,|\,A\,|\,A)$
η_5	$M \leftarrow A$	$@(A\,|\,C,M\,|\,M)$	$@(A\,|\,C,M\,|\,M)$
η_6	$C \leftarrow M$	$@(A\,|\,C,M\,|\,-)$	$@(A\,|\,C,M\,|\,C)$
certified agents		M,A	C,M,A

Table 1.5
Exchange modes for the revised *SET* e-payment protocol

	unsigned SET		*signed* SET	
Goal	*O*	*R*	*O*	*R*
pan secret between *C,A*	+	+	+	+
A weakly authenticates *C* on *pan*	–	+	+	+
OrderDesc secret between *C,M*	+	+	+	+
PurchAmt secret between *C,M,A*	–	–	+	+
AuthCode secret between *C,M,A*	–	–	–	+
M authenticates *A* on *AuthCode*	+	+	+	+
C authenticates *A* on *AuthCode*	–	+	–	+
C authenticates *M* on *AuthCode*	+*	+	+*	+
A authenticates *C* on *contract*	w	+	w	+
M authenticates *C* on *contract*	–	+	+	+
A authenticates *M* on *contract*	–	+	–	+
C authenticates *M* on *contract*	+	+	+	+
C authenticates *A* on *contract,AuthCode*	–	+	–	+
M authenticates *A* on *contract,AuthCode*	+	+	+	+

* goal satisfied only after fixing step 5 as in [25]
w = only weak authentication
for Revised *SET*: $contract = PriceAmt, TID, [PIData{:}A], [OIData{:}M]$
for Original *SET*: $contract = PriceAmt, TID, \mathsf{hash}(PIData), \mathsf{hash}(OIData)$

Table 1.6
Security goals satisfied by Original and Revised SET purchase protocol

to verify an unbounded number of sessions, but in some cases the goals cannot be proved, due to the internal mechanisms of ProVerif and the fact that, in general,

verifying protocols for an unbounded number of sessions is undecidable.

The results show that the revised versions of the protocols satisfy stronger security guarantees than the original ones [25], as reported in Table 1.6. It is worth noting, in particular, that the revised versions do not suffer from two known flaws affecting the original *SET* specification. The first flaw [25] involves the fifth step of the protocol, where it is not possible to unequivocally link the identities of the Acquirer and the Merchant with the ongoing transaction and the authorization code. Namely, the original message should be amended to include the identity of the merchant M, otherwise the goal "C authenticates M on $AuthCode$" cannot be satisfied. In the revised version, the exchange at step 5 is automatically compiled into a message including the identities of both the Merchant and the Customer, so the problem is solved.

The same implementation also prevents the second flaw, presented in [34]. In that paper, the specification of the protocol is more detailed than in [25], as it introduces an additional field *AuthRRTags*, which includes the identity of the Merchant. We tested the version of *SET* presented in [34] with OFMC and verified the presence of the flaw, namely an attack against the purchase phase, which exploits a lack of verification in the payment authorization process. It may allow a dishonest Customer to fool an honest Merchant when collaborating with another dishonest Merchant. The attack is based on the fact that neither *LIDM* nor *XID* can be considered unique, so they cannot be used to identify a specific Merchant. Therefore, the customer can start a parallel purchase with an accomplice playing the role of another merchant, and make the Acquirer authorize the payment in favour of the accomplice. Here again, the goal "C authenticates M on $AuthCode$" fails.

During the analysis, we also verified that both the original specifications [25, 34] fail to verify the goals "C authenticates A on $AuthCode$" and "C authenticates M on $contract, AuthCode$". To overcome this problem, the protocol must be fixed in the sixth (and final) step, as already outlined in [107]. This issue arises from the fact that the Customer does not have any evidence of the origin of *AuthCode* by the Acquirer and instead has to rely only on information provided by the Merchant. For example, giving to the Customer a proof that the Acquirer authorized the payment requires a substantial modification of the sixth step of the protocol. In fact, instead of letting the Merchant sign a message for the Customer, we exploit the *AnBx* forward mode to bring to the Customer the authorization of the payment signed directly by the Acquirer. It is worth noticing that, employing a *fresh* forward mode in the sixth step, we can achieve the desired strong authenticity goal on the pair, even though the transaction identifier is not unique.

The results confirm what is outlined in [107], in showing that, while *iKP* meets all the non-repudiation goals, the original specification of *SET* does not. It is important to notice that, to achieve non-repudiation, each participant must have sufficient proofs to convince an external verifier that the transaction was actually carried out as it claims. A way to obtain this is to assume that the authentication is obtained by means of digital signatures computed with keys which are valid within a Public Key Infrastructure and are issued by a trusted third party (Certification Authority). Although this limits the way authentic channels in *AnBx* could be implemented, in practice it does not represent a significant restriction, since in the considered proto-

cols, digital signatures are the standard means of authentication.

1.6 CONCLUSION

In this chapter, we consider the formalization of security protocols with the purpose of automatically verifying if they satisfy their expected security goals. We also discussed theoretical and practical challenges in automated verification, along with different specification languages and verification tools available. The case studies demonstrated the practical applicability of some of these tools and techniques to real-world security protocols.

REFERENCES

1. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115. ACM, 2001.

2. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *ACM SIGPLAN Notices*, 36(3):104–115, 2001.

3. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Information and Computation(Print)*, 174(1):37–83, 2002.

4. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997*, pages 36–47. ACM, 1997.

5. Joel C Adams. Object-centered design: a five-phase introduction to object-oriented programming in cs1–2. In *ACM SIGCSE Bulletin*, volume 28, pages 78–82. ACM, 1996.

6. Pedro Adão, Paulo Mateus, and Luca Viganò. Protocol insecurity with a finite number of sessions and a cost-sensitive guessing intruder is np-complete. *Theoretical Computer Science*, 538:2–15, 2014.

7. Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. Scenario-based programming: reducing the cognitive load, fostering abstract thinking. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 311–320. ACM, 2014.

8. Omar Almousa, Sebastian Mödersheim, Paolo Modesti, and Luca Viganò. Typing and compositionality for security protocols: A generalization to the geometric fragment. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 209–229, 2015.

9. Omar Almousa, Sebastian Mödersheim, and Luca Viganò. Alice and Bob: Reconciling formal models and implementation. In Chiara Bodei, Gian-Luigi Ferrari, and Corrado

Priami, editors, *Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, volume 9465 of *Lecture Notes in Computer Science*, pages 66–85. Springer International Publishing, 2015.

10. Roberto M. Amadio and Witold Charatonik. On name generation and set-based analysis in the dolev-yao model. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings*, volume 2421 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2002.

11. Suzana Andova, Cas Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig Fr. Mjølsnes, and Sasa Radomirovic. A framework for compositional verification of security protocols. *Inf. Comput.*, 206(2-4):425–459, 2008.

12. G. Andresen and M. Hearn. BIP 70: Payment Protocol. *Bitcoin Improvement Process*, July 2013. `https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki`.

13. Apache Foundation. The Apache Ant Project, 2019. `http://ant.apache.org`.

14. Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark Dermot Ryan. Statverif: Verification of stateful processes. *Journal of Computer Security*, 22(5):743–821, 2014.

15. A. Armando and L. Compagna. SATMC: A SAT-based model checker for security protocols. *Lecture Notes in Computer Science*, pages 730–733, 2004.

16. Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–282. Springer, 2012.

17. Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification*, pages 281–285. Springer, 2005.

18. Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. In Meiklejohn and Sako, editors, *Financial Cryptography and Data Security (FC 2018)*, volume 10957 of *LNCS*, pages 541–560. Springer, 2018.

19. Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, 2014.

20. Tigran Avanesov, Yannick Chevalier, Michaël Rusinowitch, and Mathieu Turuani. Intruder deducibility constraints with negation. decidability and application to secured service compositions. *Journal of Symbolic Computation*, 80:4–26, 2017.

21. AVISPA. Deliverable 2.3: The Intermediate Format. Available at `www.avispa-project.org`, 2003.

22. Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 777–795. IEEE, 2021.

23. Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1149–1238. Elsevier and MIT Press, 2001.

24. David A. Basin, Michel Keller, Sasa Radomirovic, and Ralf Sasse. Alice and bob meet equational theories. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 160–180. Springer, 2015.

25. G. Bella, F. Massacci, and L.C. Paulson. Verifying the SET purchase protocols. *Journal of Automated Reasoning*, 36(1):5–37, 2006.

26. M. Bellare, JA Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, E. Van Herreweghen, and M. Waidner. Design, implementation, and deployment of the iKP secure electronic payment system. *IEEE Journal on Selected Areas in Communications*, 18(4):611–627, 2000.

27. Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–74, 2001.

28. Mordechai Ben-Ari and Tzippora Yeshno. Conceptual models of software artifacts. *Interacting with Computers*, 18(6):1336–1350, 2006.

29. Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502. IEEE Computer Society, 2017.

30. Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Computer Security Foundations Workshop, IEEE*, pages 0082–0082. IEEE Computer Society, 2001.

31. Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

32. Bruno Blanchet, Vincent Cheval, and Véronique Cortier. ProVerif with Lemmas, Induction, Fast Subsumption, and Much More. In *43RD IEEE Symposium on Security and Privacy (S&P'22)*, San Francisco, United States, May 2022.

33. Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

34. S. Brlek, S. Hamadou, and J. Mullins. A flaw in the electronic commerce protocol SET. *Information Processing Letters*, 97(3):104–108, 2006.

35. Jerome S Bruner. *The Process of Education*. Harvard University Press, 2009.

36. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

37. Michele Bugliesi, Stefano Calzavara, Sebastian Mödersheim, and Paolo Modesti. Security protocol specification and verification with anbx. *Journal of Information Security and Applications*, 30:46–63, 2016.

38. Michele Bugliesi and Riccardo Focardi. Language based secure communication. In *Computer Security Foundations Symposium, 2008. CSF'08. IEEE 21st*, pages 3–16, 2008.

39. Michele Bugliesi and Paolo Modesti. AnBx-Security protocols design and verification. In *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security: Joint Workshop, ARSPA-WITS 2010*, pages 164–184. Springer-Verlag, 2010.

40. Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

41. Kaylash Chaudhary, Ansgar Fehnker, Jaco van de Pol, and Mariëlle Stoelinga. Modeling and verification of the bitcoin protocol. In van Glabbeek, Groote, and Höfner, editors, *Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS 2015*, volume 196 of *EPTCS*, pages 46–60, 2015.

42. Ştefan Ciobâcă and Véronique Cortier. Protocol composition for arbitrary primitives. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 322–336. IEEE Computer Society, 2010.

43. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

44. Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

45. Véronique Cortier and Stéphanie Delaune. Safely composing security protocols. *Formal Methods in System Design*, 34(1):1–36, 2009.

46. Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

47. Camille Coti, Laure Petrucci, César Rodríguez, and Marcelo Sousa. Quasi-optimal partial order reduction. *Formal Methods in System Design*, 57(1):3–33, 2021.

48. C. Cremers and P. Lafourcade. Comparing state spaces in automatic security protocol verification. In *Proceedings of the 7th International Workshop on Automated Verification of Critical Systems (AVoCS'07), Oxford, UK, September*, pages 49–63. Citeseer, 2007.

49. Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1773–1788. ACM, 2017.

50. Cas JF Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification*, pages 414–418. Springer, 2008.

51. Melissa Dark, Steven Belcher, Matt Bishop, and Ida Ngambeki. Practice, practice, practice... secure programmer! In *Proceeding of the 19th Colloquium for Information System Security Education*, 2015.

52. Stéphanie Delaune and Lucca Hirschi. A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *Journal of Logical and Algebraic Methods in Programming*, 87:127–144, 2017.

53. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

54. Jannik Dreier, Charles Duménil, Steve Kremer, and Ralf Sasse. Beyond subterm-convergent equational theories in automated verification of stateful protocols. In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10204 of *Lecture Notes in Computer Science*, pages 117–140. Springer, 2017.

55. Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

56. Santiago Escobar, Catherine A. Meadows, and José Meseguer. Maude-npa: Cryptographic protocol analysis modulo equational properties. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2007.

57. Shimon Even and Oded Goldreich. On the security of multi-party ping-pong protocols. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 34–39. IEEE Computer Society, 1983.

58. Leo Freitas, Paolo Modesti, and Martin Emms. A Methodology for Protocol Verification applied to EMV. In *Formal Methods: Foundations and Applications - 21th Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 28-30, 2018, Proceedings*, volume 11254 of *Lecture Notes in Computer Science*. Springer, 2018.

59. Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In Maurice H. ter Beek and Dejan Nickovic, editors, *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings*, volume 12327 of *Lecture Notes in Computer Science*, pages 3–69. Springer, 2020.

60. Rémi Garcia and Paolo Modesti. An IDE for the design, verification and implementation of security protocols. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2017, Toulouse, France, October 23-26, 2017*, pages 157–163, 2017.

61. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

62. Kurt Gödel. On formally undecidable propositions of principia mathematica and related systems. *The undecidable. Hew*, 1964.

63. Sébastien Gondron and Sebastian Mödersheim. Vertical composition and sound payload abstraction for stateful protocols. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021.

64. L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security: Architecture, Api Design, and Implementation*. Addison-Wesley, 2003.

65. Thomas Groß and Sebastian Mödersheim. Vertical protocol composition. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 235–250. IEEE Computer Society, 2011.

66. Joshua D. Guttman. Establishing and preserving protocol security goals. *Journal of Computer Security*, 22(2):203–267, 2014.

67. Joshua D. Guttman and F. Javier Thayer. Protocol independence through disjoint encryption. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 24–34. IEEE Computer Society, 2000.

68. Bruria Haberman and Yifat Ben-David Kolikant. Activating "black boxes" instead of opening "zipper"- a method of teaching novices basic cs concepts. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 41–44, New York, NY, USA, 2001. ACM.

69. Said Hadjerrouit. A constructivist framework for integrating the java paradigm into the undergraduate curriculum. In *ACM SIGCSE Bulletin*, volume 30, pages 105–107. ACM, 1998.

70. Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020.

71. Andreas V. Hess, Sebastian Alexander Mödersheim, and Achim D. Brucker. Stateful protocol composition. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 427–446. Springer, 2018.

72. ISO/IEC. ISO/IEC 9798-1:2010. Information technology – Security techniques – Entity authentication – Part 1: General, 2010.

73. Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *Progress in Cryptology - INDOCRYPT 2020 - 21st International Conference on Cryptology in India, Bangalore, India, December 13-16, 2020, Proceedings*, volume 12578 of *Lecture Notes in Computer Science*, pages 151–202. Springer, 2020.

74. Herman Koppelman and Betsy van Dijk. Teaching abstraction in introductory courses. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 174–178, New York, NY, USA, 2010. ACM.

75. Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.

76. G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

77. Gavin Lowe. A hierarchy of authentication specifications. In *CSFW'97*, pages 31–43. IEEE Computer Society Press, 1997.

78. Gavin Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(1):89–146, 1999.

79. Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. Refund attacks on bitcoin's payment protocol. In *20th Financial Cryptography and Data Security Conference*, 2016.

80. Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

81. Robin Milner. Logic for computable functions: description of a machine implementation. 1972.

82. J Mitchell, A Scedrov, N Durgin, and P Lincoln. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*. Citeseer, 1999.

83. Sebastian Mödersheim. Algebraic properties in Alice and Bob notation. In *International Conference on Availability, Reliability and Security (ARES 2009)*, pages 433–440, 2009.

84. Sebastian Mödersheim and Luca Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 166–194. Springer, 2009.

85. Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. In Michael Backes and Peng Ning, editors, *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*, pages 337–354. Springer, 2009.

86. Sebastian Mödersheim and Luca Viganò. Sufficient conditions for vertical composition of security protocols. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 435–446, New York, NY, USA, 2014. ACM.

87. Paolo Modesti. *Verified Security Protocol Modeling and Implementation with AnBx*. PhD thesis, Università Ca' Foscari Venezia (Italy), 2012.

88. Paolo Modesti. Efficient Java code generation of security protocols specified in AnB/AnBx. In *Security and Trust Management - 10th International Workshop, STM 2014, Proceedings*, pages 204–208, 2014.

89. Paolo Modesti. AnBx: Automatic generation and verification of security protocols implementations. In *8th International Symposium on Foundations & Practice of Security*, volume 9482 of *LNCS*. Springer, 2015.

90. Paolo Modesti. Integrating formal methods for security in software security education. *Informatics in Education*, 19(3):425–454, 2020.

91. Paolo Modesti, Siamak F. Shahandashti, Patrick McCorry, and Feng Hao. Formal modelling and security analysis of bitcoin's payment protocol. *Comput. Secur.*, 107:102279, 2021.

92. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, November 2008. `https://bitcoin.org/bitcoin.pdf`.

93. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

94. K. Ogata and K. Futatsugi. Formal analysis of the iKP electronic payment protocols. *Lecture Notes in Computer Science*, pages 441–460, 2003.

95. D. O'Mahony, M. Peirce, and H. Tewari. *Electronic payment systems for e-commerce*. Artech House Publishers, 2001.

96. M. Pistoia, N. Nagaratnam, L. Koved, and A. Nadalin. *Enterprise Java 2 Security: Building Secure and Robust J2EE Applications*. Addison Wesley, 2004.

97. Erik Poll and Aleksy Schubert. Verifying an implementation of SSH. In *WITS*, volume 7, pages 164–177, 2007.

98. Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with a finite number of sessions, composed keys is np-complete. *Theoretical Computer Science*, 299(1-3):451–475, 2003.

99. Rupert Schlick, Michael Felderer, Istvan Majzik, Roberto Nardone, Alexander Raschke, Colin Snook, and Valeria Vittorini. A proposal of an example and experiments repository to foster industrial adoption of formal methods. pages 249–272, 2018.

100. Benedikt Schmidt, Sebastian Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 78–94. IEEE, 2012.

101. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

102. Zhanna Malekos Smith and Eugenia Lostri. The hidden costs of cybercrime. Technical report, 2020.

103. F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1):191–230, 1999.

104. K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy*, 3(6):81–84, 2005.

105. A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of The London Mathematical Society*, 41:230–265, 1937.

106. M. Turuani. The cl-atse protocol analyser. *Lecture Notes in Computer Science*, 4098:277, 2006.

107. E. Van Herreweghen. Non-repudiation in SET: Open issues. *Lecture Notes in Computer Science*, pages 140–156, 2001.

108. J Voas and K Schaffer. Whatever happened to formal methods for security? *Computer*, 49(8):70, 2016.

109. Barry J Wadsworth. *Piaget's Theory of Cognitive and Affective Development: Foundations of Constructivism*. Longman Publishing, 1996.

110. James Walden and Charles E. Frank. Secure software engineering teaching modules. In *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*, InfoSecCD '06, pages 19–23, New York, NY, USA, 2006. ACM.

111. Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In Harald Ganzinger, editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 1999.