UNIVERSITÀ DEGLI STUDI DI MILANO

CORSO DI DOTTORATO IN INFORMATICA
CICLO XXXVI

DIPARTIMENTO DI INFORMATICA

APROVER: A Framework For The Development Of Security Protocols

Settore Scientifico disciplinare INF/01

Tesi di Dottorato di:
Mario Lilli

Supervisore: Prof. Elvinia Riccobene

Co-supervisore: Dr. Chiara Braghin

Coordinatore: Prof. Roberto Sassi

A.A. 2023-2024

Acknowledgements

I would like to express my heartfelt gratitude to my supervisor, Elvinia Riccobene, for providing me with the invaluable opportunity to engage in research within this exceptional environment. Her guidance and unwavering support have played a pivotal role in shaping my research journey.

In addition, I extend my thanks to all my co-authors for their collaborative efforts in this research endeavour. Their contributions have enriched the project and made it a harmonious synthesis of diverse perspectives and expertise.

Furthermore, I want to acknowledge and appreciate my parents' unwavering support and patience throughout every phase of this journey, from its inception to its completion. Their steadfast encouragement has been an invaluable pillar, and I am grateful for their enduring support. Their encouragement, alongside the mentorship of Elvinia Riccobene, has been instrumental in the successful completion of this research project.

Abstract

In the dynamic landscape of emerging technologies such as IoT and 5G, the integration of advanced cryptographic protocols becomes essential for fostering secure and data-driven communication. The reliability of cryptographic protocols plays a pivotal role, enabling secure communication channels, privacy preservation, real-time responses, energy-efficient operations, enhanced security measures, personalised user experiences, and optimized network performance. However, a critical concern emerges due to the limited presence of formalised communication protocols, posing security risks in our interconnected daily lives. The lack of formalisation can be traced back to the mathematical intricacy of formal methods languages and their complex integration into the tight development timeframes of companies. To ensure that mathematical proofs of correctness, secrecy, authentication and integrity are guaranteed through all the developing phases of security protocols, we perfuse our effort in creating a framework to close the gap between the designers' needs and the formal methods tools. The project introduces APROVER, an "Automatic Protocol Verifier" designed to simplify the formal specification, verification, and development of security protocols. The initiative focuses on two main aspects: i) bridging the usability gap by developing a web UI based on a sequence diagram for message exchange and a versatile language (KANT) for power users, helping them with syntactic and semantic validation of their models; ii) unifying the verification methodology by creating a multi-level approach that targets both the communication and the internal device logic.

Contents

List of Figures

List of Tables

Glossary

development refers to a set of computer science activities that are dedicated to the process of creating, designing, deploying, and supporting software.

verification entails the procedure of checking using a mathematical formalism whether a design meets certain specified requirements.

language validation refers to the process of ensuring that the input conforms to the syntax, semantics, and constraints defined within the specific domain.

model validation refers to the process of creating scenarios to test the adherence of the model to the requirements.

formalisation refers to the process of representing systems or specification using precise, well-defined mathematical or logical frameworks.

1 Introduction

This thesis aims to bridge the gap between the complex mathematical formalism of formal methods and the community of designers who develop security protocols. We present the motivation and two research questions in Section 1.1. To address the previously outlined challenges, we detail our approach to solving these two questions in Section 1.2.

1.1 Motivation

Security protocols can be characterised as distributed algorithms specifically crafted to attain essential security objectives, such as ensuring confidentiality, authentication, and integrity, particularly in communication across networks that may be untrusted. Despite these protocols' crucial role in fortifying information systems, the design and implementation process is notably error-prone, as evidenced by the many flaws observed in published and deployed security protocols. The presence of such flaws underscores the need for a methodology that produces functional specifications and formal security proofs when designing security protocols for modern networks and business infrastructures. Despite the importance of formal verification in the design of protocols and their proven effectiveness in uncovering vulnerabilities, its implementation remains limited. This is because formal verification approaches are mathematically complex, and designers often prioritise quick feature development over the time-consuming process of modelling and verifying the requirements of protocols. The lack of integration of formal methods during the security protocol development process cannot be blamed solely on designers. Most of the tools recognised as state-of-the-art for security protocol verification do not integrate well with the development flow since they are not very intuitive (both in their languages and findings presentation) and would require companies to hire highly skilled personnel to perform this task. Furthermore, tool integration is challenging to achieve [50] since each tool has its own input language and modelling primitives; thus, using different tools often requires starting to model the same protocol from scratch each time.

Organisations often view security as a financial liability rather than a strategic investment, as highlighted in [74]. Specifically, in the realm of companies engaged in the development of Internet of Things (IoT) devices, a prevalent tendency is to incorporate security features only in response to a security breach. This reactive stance neglects the fundamental principle of cost-effectiveness, wherein the expenses associated with rectifying a vulnerability identified during the production phase far surpass those incurred through the establishment of secure and verified requirements in advance. Because of this, millions of devices are abandoned without further patching, exposing organisations and individuals to security risks.

The primary objective of this thesis is to assist designers in integrating formal methods into the security protocol's development cycle. By doing so, we can reduce the barrier to the adoption by allowing even developers with low knowledge of formal methods to analyse the absence of logical flow in the requirements of the security protocol. This motivates our first research question.

Research Question 1: How can the usability of formal methods be enhanced to facilitate their effective integration into the design process of security protocols, ensuring that designers, who may lack formal methods expertise, can leverage these tools more intuitively and efficiently?

The second research question arises from the observation that formal verification tools used in literature for verifying security protocols (e.g., TAMARIN, PROVERIF, AVISPA) usually capture different protocol details and retrieve different attack traces. Moreover, in an environment such as the corporate environment where development time is tight, the verification results must be returned to the designer as quickly as possible. Hence our question:

Research Question 2: How can we exploit the strength of various formal methods to speed up the verification process and check the security of different aspects of a security protocol?

1.2 Approach Taken

In the following, we describe our approach to solving our two research questions.

Research Question 1

To answer the first question, our research concerted in hiding the complexity of formalisation by creating an intuitive and expressive language. The language we designed is called KANT [34]. It has traits in common with the language adopted by the VERIFPAL [57] tool. However, it adds validation primitives that allow conceptual errors to be eliminated already at the validation stage, thus reducing both the verification time and supporting protocol designers with continuous feedback during development. In addition to KANT, we have developed a graphical interface that allows less experienced users to build a protocol with a drug-and-drop mode. The GUI was designed to act as a web front-end so that corporations can easily access and integrate it into the protocol development process. Furthermore, The GUI will be able to aggregate the verification results, making the error traces found more readable for designers.

Research Question 2

Our goal is not to create new tools for formal protocol verification but to establish a working framework validated on real-world security protocols that demonstrates the effectiveness of our approach. We focused on minimising the designer's involvement in the verification phase. To achieve this result, we have explored the integration of formal protocol verification in the ASMETA framework that, thanks to formal verification via model checker, does not require user involvement to help the verification termination. To exploit the various potential of the verification tools used in the field of security protocol in the literature (e.g. TAMARIN [65] and PROVERIF [24]), we choose to use METACP [13] , which allows starting from a protocol model written in JSON to obtain two models in the target languages with a set of security properties that can be tested. During the development of our approach, a new tool that combines the TAMARIN and PROVERIF syntax called SAPIC+ [38] was presented. However, our testing of this tool has shown that its implementation is still in the development phase, and because of this, we have preferred to use the target tools directly, which are more mature and stable. To conclude the answer to the second research question on how it might be possible to capture different aspects of a security protocol, we focused on an approach that envisaged exploiting the combination of results from two different verification methods. In more detail, our proposed approach involves a feedback loop between the verification methods chosen to finish the incremental models and increase the designer's confidence that they have covered the totality of the specification of the protocol requirements.

1.3 Contribution

We identify three main contributions in our work.

- We have designed a Domain Specification Language (DSL) named KANT, which has a user-friendly syntax to capture the requirements of a security protocol. KANT provides useful feedback to designers about the common errors found in literature, which helps them avoid incorporating those errors into their designs. By eliminating errors during the validation phase, KANT makes the verification process faster. Additionally, KANT enables designers to capture the transitions of agents' internal states participating in the security protocol's messaging exchange. This feature is useful in the verification phase to ensure the accuracy of the agents' internal state machine.

- We have developed a web front-end that simplifies the formalisation of a security protocol using a drag-and-drop approach. This front-end is compatible with KANT and aims to reduce errors that a designer could make. The drag-and-drop modules represent cryptographic primitives that can only be combined following specific logical constraints, ensuring a proactive approach to preventing errors. The front-end is built on a JSON Schema and can also show the attack trace found during the verification stage.

- We have created a set of primitives and CTL formulas that can be used to model and verify security protocols using the ASMETA framework. The reusable CTL formula patterns for the security properties and primitives that describe both the message structure, the cryptographic functions and the internal logic of the agents make it easier to model a security protocol, even for designers who do not have much knowledge of formal methods. This is because the Abstract State Machine is very similar to the Finite State Machine, known to any programmer. We tested the effectiveness of our approach by formalising the Z-Wave protocol and discovering a zero-day vulnerability that was confirmed by Silicon Labs, the company that developed the protocol.

- We have proposed a methodology for unifying the security protocol verification, enabling the capture of different levels and details of a security protocol. More specifically, we formalised the WPA3 SAE protocol of the IEEE 802.11 standard, using PROVERIF to standardise message exchange and ASMETA to authenticate the machine to agent states. We discovered 20 new attacks, which helped us understand how a multi-tool approach with a feedback loop will enable us to capture more details than those caught with a single verification tool.

The contributions listed above all pertain to the APROVER framework. In particular, this thesis deals with creating the front-end and back-end of the framework, with a particular focus on the pi-calculus and abstract state machine formalisms. In Fig. 3, the structure of the framework is visible. The boxes highlighted in green will be described in this thesis; those highlighted in yellow are obtained through other tools (i.e., for conversions to PROVERIF and TAMARIN, we used METACP). The connections between the front-end conversion tools are in the process of being refined; at the moment, the conversion still requires a step of human intervention to achieve the end-to-end conversion.

1.4 Overview

This thesis consists of two parts. The first part details our effort in bridging the usability gap between the formal verification tools and the designer by developing a user-friendly web interface and Domain Specific Language KANT to validate models syntactically and semantically. The first part is

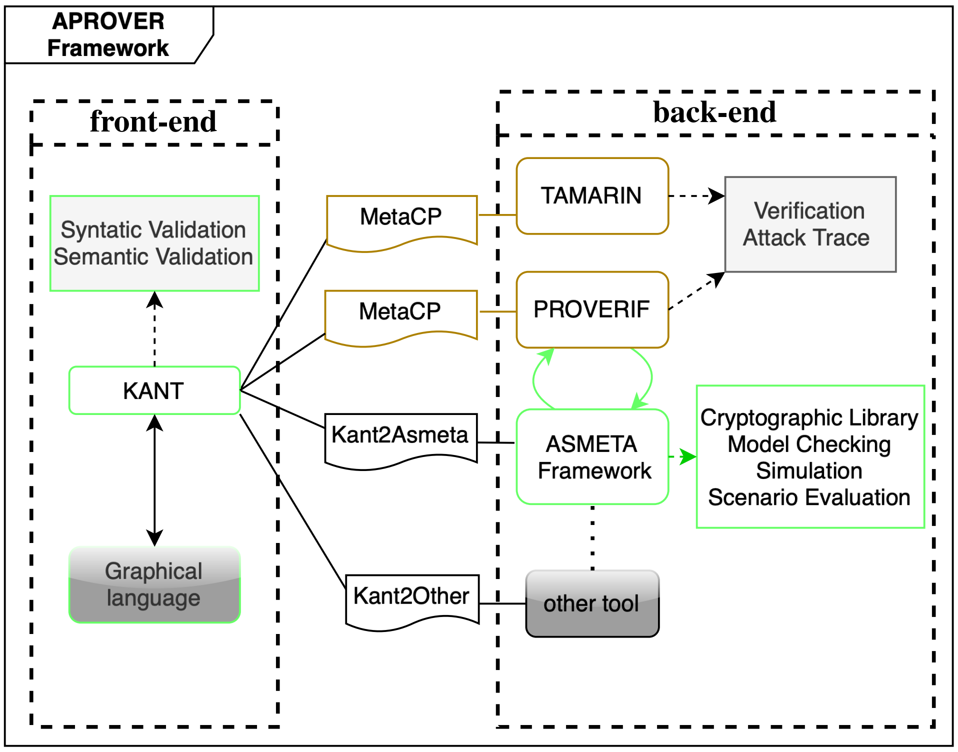


Figure 1.1: APROVER Framework

based on the publication [34], which is a joint work with Chiara Braghin and Elvinia Riccobene. The second part aims to make the verification more within the reach of inexperienced users and, simultaneously, allow them to verify the requirements of the security protocol, capturing even more details of the classical verification methods. The approach we propose involves a multi-level verification that will enable us to verify both the communication layer (a layer usually analysed by tools such as TAMARIN or PROVERIF) and the device layer, which captures the internal operating logic of each agent involved in the message exchange. The second part is based on the publications [60, 35, 33, 61], which are a joint work with Chiara Braghin, Elvinia Riccobene, Roberto Metere and Luca Arnaboldi. We introduce the notation for security protocols in 2.1. In Section 2.2, we explain the framework that we used to verify security protocols. Section 2.2.3 provides an overview of the verification tools used for verifying security protocols. Section 2.2.4 analyses related work that adopts a user-centric approach and approaches combining multiple verification tools. The last two sections of the Background chapter present the verification tools used in our experiments.

They are the ASMETA framework and the Abstract State Machine (ASM) on which ASMETA is based, presented in Section 2.3, and PROVERIF and the Applied π-calculus described in section 2.4. The last two chapters are self-contained, and we outline them below.

Bridging the Usability Gap

Chapter 3 describes the creation of a user-friendly development environment aimed at modelling a security protocol. The environment includes two main components: a graphical specification language that captures the message flow between security protocol agents (Section 3.1) and a domain-specific language (DSL) called KANT (Section 3.2).

In Section 3.1, we present the APROVER Front-End GUI, which is a web graphical interface that provides novice users with a clean modelling environment for developing security protocols. The GUI aims to mitigate error-prone steps and guide users in making the correct choices. We explore the usability of the GUI in Section 3.1.1, showing the various pages that compose it and their use. Additionally, we discuss how the GUI stores data that would be converted into the various back-end languages in Section 3.1.2.

KANT, a DSL for validating syntactically and semantically the security protocol models, is detailed in Section 3.2. The development process of KANT is elaborated in Section 3.2.2, covering its objectives and structure. The section also details the syntax of KANT, including principal definition, type and function definitions, property definition, knowledge definition, principals' communication, security checks, and the KANT model of the reference scenario. In Section 3.2.4, we discuss validation rules for KANT, encompassing both syntactic and semantic checks. Syntactic checks (Section 3.2.4.1)cover aspects such as syntax well-formedness, knowledge declaration, naming convention, and more. Semantic checks (Section 3.2.4.2)cover type compatibility, knowledge scoping, function inversion, list access, and other prudent engineering practices. In Section 3.2.5, we evaluate the effectiveness of KANT in detecting errors within the security protocol specification. The KANT language presentation concludes with an exploration of related work (Section 3.2.6), examining existing efforts to develop domain-specific languages for security protocol modelling.

Unifying Verification Methodology

In Chapter 4, a method for formally verifying security protocols is presented. This method analyses the protocol at two levels: communication and device. At the communication level, the protocol is modelled statelessly, which is the traditional way used in the literature to verify security protocols formally. On the other hand, the device level describes the internal state machine of the agents involved in the protocol and their transitions during the message exchange.

Section 4.1 presents a verification approach that takes advantage of the simplicity of ASM syntax to perform formal verification covering both the communication and device levels. This new approach involved developing a library for ASMETA that would capture cryptographic primitives (Section 4.1.4.1) and security properties (Section 4.1.4.2). The remaining sections will show the application of the approach to the Z-wave protocol by demonstrating how novel vulnerabilities could be captured with it.

Section 4.2 presents a formal verification approach that involves both levels (communication and device) and allows us to retrieve important feedback to further detail the model and discover vulnerabilities that a simple analysis of the communication level could not have captured. Section 4.2.3 presents the WPA3-SAE protocol that was used as a running example. Section 4.2.4 describes how ProVerif and ASMETA are used in multi-level modelling. Section 4.2.5 presents the two models developed for the ProVerif and ASMETA back-ends. Finally, section 4.2.6 describes the results obtained from using the tools disjointly, combined, and results common to both.

2 Background

2.1 Security protocol

A communication protocol is a set of defined rules governing how entities exchange information. In the context of this thesis, these entities primarily consist of two computers, although they may also encompass interactions between individuals or a person and a computer. The transport medium, or communication channel, is integral to the communication process. In specific scenarios, the communication channel cannot be deemed trustworthy. For example, sharing sensitive information over unsecured public Wi-Fi networks can put that information at risk of being intercepted and potentially accessed by unauthorised individuals. In situations involving untrusted channels, the necessity for security protocols arises. These protocols are designed to uphold specific properties even in the presence of an adversarial entity. An early example of a security protocol is using wax seals to close and authenticate letters. The wax seal served the dual purpose of authenticating the sender and protecting the letter from unauthorised modification. The capabilities of an adversary are contingent upon the circumstances. Sometimes, assuming that the adversary can only observe communication may be reasonable. At the same time, in other cases, one must consider the possibility that the attacker may control one or more parties involved in the protocol. For instance, a protocol utilising a wax seal as a form of authentication assumes that the adversary cannot forge an existing seal. In computer-mediated communication, cryptography achieves properties analogous to those provided by wax seals for physical letters. Our focus primarily centres on cryptographic primitives: nonces, hashes, and encryption. Nonces, as random values, prevent message reuse and thwart an attacker's ability to discern if the same encrypted message has been sent multiple times. Hash functions, characterised as one-way deterministic functions, facilitate the verification of whether two hash values originate from the same message. Encryption, a pivotal component, conceals the contents of a message during transmission. The subsequent sections of this thesis delineate properties relevant to a security protocol, enumerate potential characteristics of a communication channel, and expound upon various attacks that an adversary may deploy against a protocol.

2.1.1 Security properties

In the realm of security protocols, a spectrum of properties may be deemed requisite. While we enumerate some of the most common properties herein, it is imperative to acknowledge the existence of additional properties pertinent to specific security contexts. Noteworthy exclusions from consideration in this thesis are properties such as privacy and observational equivalence, which are not addressed succinctly due to their complexity.

Secrecy: Secrecy stands as one of the fundamental properties of a security protocol. It asserts that an adversary should be precluded from acquiring knowledge of any information intended to remain confidential. An illustrative example is the concealment of a key generated from a key exchange protocol.

Authentication: Authentication is foundational to a security protocol, ensuring the verification of the true identity of the message sender. Authentication exhibits varying degrees, encompassing weak-authentication, wherein one party can ascertain that the other party executed the protocol at some point, and injective authentication, assuring that the other party recently engaged in the protocol exclusively with the present entity. A more exhaustive exploration of authentication is available in [62].

Freshness: Parameter freshness denotes the assurance within a protocol that a utilised parameter, such as a key, has not been employed previously. This is particularly critical, as numerous cryptographic primitives hinge on the premise that a key is utilised for a limited number of instances.

Forward Secrecy: Forward Secrecy ensures that the compromise of long-term keys does not jeopardise the confidentiality of past communications. In other words, even if an adversary gains access to the long-term secret keys at some point in the future, it should not be able to decrypt past communications that were securely transmitted. This property provides an additional layer of security, especially in the face of evolving threats and potential long-term key exposures.

Comprehending the inherent characteristics of a security protocol demands a clear understanding of its properties. It is imperative to note that these properties represent only a subset, and the considerations may evolve depending on an application's specific security requirements and context.

2.1.2 Common Attacks

In this section, we enumerate various types of attacks on protocols as outlined in the book "Protocols for Authentication and Key Establishment" [32]. Notably, we omit consideration of the Certificate Manipulation attack, as it pertains to public key encryption with certificates and does not apply to the current setting.

Eavesdropping: It constitutes a passive attack wherein the adversary endeavours to gain knowledge by clandestinely intercepting messages without active interference. This tactic is often coupled with other attacks, such as replay. A straightforward example is an individual surreptitiously listening to a private phone conversation.

Modification: In a modification attack, the adversary alters parts of existing messages to construct another message that can be utilised to compromise the protocol. For instance, if an adversary can manipulate the amount of money in a financial transaction message, this constitutes a modification attack.

Replay: A replay attack occurs when the adversary duplicates and retransmits a previous message. This tactic can be exemplified by illicitly reusing a coupon to obtain another discount. A specific instance of replay is reflection, where messages sent are returned to the sender, inducing a false belief that communication is with a valid party when, in reality, it is with itself.

Preplay: Preplay denotes a scenario in which the adversary initiates the protocol before the involved parties formally initialised it. A more general form is message insertion, wherein the adversary can introduce a message during the protocol, potentially acquiring knowledge to complement other attacks.

Typing Attacks: A typing attack transpires when a message of one type is employed as another type of message. An example is the decryption of an encrypted nonce without proper verification, potentially leading to the unintended decryption of an encrypted secret.

Denial of Service: Denial of Service entails the adversary impeding legitimate users from completing a protocol run. Common instances include flooding a server with excessive requests to overwhelm its capacity or physically severing the communication line of a residence equipped with a dialling alarm system.

Cryptanalysis: Cryptanalysis involves the adversary leveraging knowledge about the encryption employed in the protocol to compromise security properties. For instance, the encryption key might be calculable if the adversary possesses both the encrypted message and its plaintext content for multiple messages.

Cross-Protocol Attack: A cross-protocol attack occurs when vulnerabilities emerge through the collaborative interaction of two protocols. For instance, if the same encryption key is employed across both protocols, the system may be susceptible to a typing attack, wherein a message from one protocol is illegitimately utilised in the other.

2.1.3 Communication Channels

Protocols employ various types of channels for communication, and the nature of these channels can significantly impact the security properties associated with message transmission. In [64], four distinct channel types are delineated, emphasising the insecure channel. When considering a scenario with a single sender (Alice) and a single receiver (Bob), the following channel classifications are identified:

Authentic Channel: In an authentic channel, Bob can reliably ascertain that Alice indeed sent the message and that it was explicitly intended for him. This channel type emphasises the assurance of message authenticity and source verification.

Confidential Channel: Within a confidential channel, Alice can depend on the assurance that only Bob can decipher and comprehend the contents of the messages transmitted by Alice. This channel type underscores the confidentiality of message content.

Secure Channel (or Private Channel): The term "secure channel" is used interchangeably with "private channel" in this thesis. Both terms denote a communication channel that ensures the authenticity of messages from the sender (e.g., Alice) to the receiver (e.g., Bob) and maintains the confidentiality of message contents. The term "private channel" emphasises security without encryption, relying on the adversary's lack of knowledge about the channel for protection.

Insecure Channel (or Public Channel): The term "insecure channel" is used interchangeably with "public channel" in this thesis. Both terms describe a communication channel lacking assurances of message authenticity and confidentiality. Despite its inherent insecurity, cryptographic methods may be applied to enhance the security properties of messages transmitted over a public channel.

2.2 Formal verification of security protocols

Securing communication systems requires formal verification of security protocols to ensure resilience against potential threats. In protocol analysis, two primary models exist: symbolic and computational. The symbolic model, with its abstraction and high-level representations, offers a conceptual framework that facilitates comprehensive protocol analysis. In contrast, the computational model delves into cryptographic algorithm intricacies, providing granular protocol execution scrutiny. We prefer the symbolic model, as it can handle protocol complexities with a reasonable balance between fidelity and tractability. Additionally, it is more suitable for inexperienced users who target our framework. While the computational model excels in capturing cryptographic nuances, its susceptibility to undecidability challenges requires a knowledgeable user to invest time and effort to exploit its capability.

The following sections delve into the Dolev-Yao model, a symbolic formalism widely employed for security protocol analysis. We explore how this model, through strategic abstraction, mitigates undecidability challenges inherent in cryptographic protocol verification. Our aim is to illuminate pragmatic strategies to strike a reasonable compromise between computational intricacy and symbolic abstraction rigorous formal verification pursuits.

2.2.1 Dolev-Yao Model

The Dolev-Yao model is conceptualised as a framework where the adversary wields control over all communication channels but is devoid of cryptanalytic capabilities. In this thesis, we adhere to the notion of an adversary in-capable of launching attacks on cryptographic primitives. Furthermore, we impose additional constraints by restricting the adversary from interacting with private channels. The adversary, within the constraints of our model and as outlined in [43], is endowed with four actions tailored to our notion of public channels:

Read Action: The adversary can read a message on a public channel and remove it from the channel.

Split Action: The adversary can split a message into smaller parts and retain them in memory.

Generate and Combine Action: The adversary can generate new data and combine various parts to form novel data constructs.

Send Action: The adversary is empowered to send messages generated from known data to a public channel.

2.2.2 Undecidability and state space explosion

The challenge of state space explosion is pervasive in model checking, and protocol verification is not exempt from it. Due to the myriad and potentially infinite ways in which a protocol can be executed, exhaustive examination of all concurrent execution paths becomes infeasible. If protocol verification mandated the scrutiny of every execution path, it would render verification an undecidable problem, particularly for protocols characterised by an infinite set of traces. Indeed, studies have established the general undecidability of protocol verification [73, 3, 44]. Specifically, the verification of secrecy, an extensively studied aspect, has been demonstrated to be undecidable for protocols that impose restrictions on various factors, including the number of actors and message length [73]. Given the inherent undecidability, it becomes apparent that no tool can universally verify all protocols. In practical terms, this implies that a verification tool may either fail to terminate as it navigates through an infinite number of traces or yield a result indicating uncertainty in the face of undecidability.

2.2.3 Verification tools

In security protocol verification, numerous tools are available, distinguished by their approaches to both bounded and unbounded verification. Bounded verification involves imposing limits on certain protocol parameters, ensuring the protocol is secure within those bounds. These bounds are typically set to mitigate state space explosion, such as placing a cap on the maximum message length. The rationale behind bounded verification lies in achieving decidability and guaranteeing termination. If the property to be proven is decidable within the specified bounds, termination can be ensured. However, it is crucial to note that bounded verification does not provide an absolute guarantee against all attacks, as potential vulnerabilities may exist beyond the specified bounds. Despite this limitation, bounded verification is considered effective, particularly when the bounds align with those inherent in the protocol specification. This aligns with the argument that practical attacks on realistic protocols tend to be small and discoverable through bounded verification [18]. One notable tool for bounded verification is SATMC [9], which boasts an expressive language and allows the analysis to closely mirror protocol implementations. SATMC utilises a bound on the number of protocol runs, iteratively testing the protocol for attacks until the specified bound is reached. It has been employed in various projects, including AVISPA [10] and AVANTSSAR [11]. Unbounded verification, in contrast, imposes no limits during the verification process, yet it faces challenges due to the inherent difficulty of use. Tools for unbounded verification rarely guarantee termination, given the undecidability of general properties like secrecy. These tools assume the decidability of a property for a specific protocol and attempt to construct proof validating the property. However, non-termination is a plausible outcome exemplified by scenarios where tools get stuck while attempting to find attacks, potentially by adding an infinite number of participants to a protocol run. To enhance the likelihood of termination, some tools employ approximations. These approximations expand the adversary's capabilities, often by relaxing the order in which protocol operations must be executed. Analogously, this concept can be likened to the mortality problem [52], an undecidable problem that becomes more manageable with approximations. Two state-of-the-art tools for protocol verification, PROVERIF and TAMARIN, have been used to verify large and complex protocols, demonstrating their goodness even without being able to guarantee termination. PROVERIF [24] is a tool that internally converts a protocol specification into a simplified representation for property verification. While this translation introduces false positives (false attacks), denoting proofs that an adversary could exploit, it effectively identifies potential vulnerabilities. TAMARIN [65] employs a backwards-search approach for property proof on a protocol specification, eliminating issues related to false positives but introducing challenges with termination. An extension to TAMARIN, as outlined in [58], translates protocol specifications in the applied π-calculus into a model understood by TAMARIN, a more intuitive description of protocols. Although proven sound, this translation may require user guidance for successful proof construction.

2.2.4 Tools Improving Verification Usability

In recent years, there have been various attempts to make formal methods more user-friendly and integrated. The first category involves the creation of more accessible front-end languages for non-expert users. Two examples of this category are METACP and VERIFPAL.

METACP [13] is a tool that simplifies the creation and verification of security protocols. It achieves this through a user-friendly graphical interface akin to modern database editors. This interface lets users craft complex security specifications intuitively, making protocol design accessible to a broader audience. At the core of METACP lies its structured data storage system, utilising XML format to store protocol information. While XML lacks inherent meaning, METACP employs plugins to imbue the data with precise semantics. These plugins ensure that the graphical designs are accurately translated into various back-end verification languages like TAMARIN and PROVERIF.

VERIFPAL [57] is a tool that tries to address the challenges of adopting formal verification to verify security protocols. VERIFPAL seeks to make formal verification more approachable for real-world practitioners, students, and engineers while maintaining comprehensive features crucial for rigorous analysis. At its core, VERIFPAL introduces a new language for modelling protocols, offering users a user-friendly environment that mirrors informal conversations. This approach, coupled with the internal logic of deconstructing and reconstructing abstract terms, aims to enhance the intuitiveness of protocol modelling without compromising on precision. VERIFPAL takes a unique stance by disallowing the definition of custom cryptographic primitives. Instead, it provides a set of built-in cryptographic functions. This standardised approach ensures that fundamental cryptographic operations are defined correctly, contributing to model clarity and reliability.

However, the limiting set of primitives could hamper the ability of the user to capture specific details of the protocol specification. Moreover, VERIFPAL is mapped into a library defined for the Coq theorem prover [39]. This integration allows VERIFPAL models to be automatically translated within Coq, enabling it to verify the classical security properties. After the verification, the user can view the obtained traces in the interface.

The second category concerns the integration of the use of verification tools with each other. In this line of research, we can find the SAPIC+ tool.

SAPIC+ [38] is a big step forward over Sapic [58]. The SAPIC+ conversion is proven sound and ensures correct PROVERIF, DEEPSEC, and TAMARIN models. These translations, covering both protocol and property specifications, yield consistent verification outcomes. Integration is seamless, with SAPIC+ becoming an integral part of TAMARIN. This integration streamlines the verification process, offering direct translations and enhancing visual feedback in manual, interactive proofs. An optional PROVERIF-like type system detects modelling errors, simplifying development through a unified approach. Capabilities are extended as SAPIC+ introduces support for destructor symbols, let-bindings, and macro declarations in the translation procedure to TAMARIN. A compression technique reduces model size while maintaining correctness, broadening the tool's applicability to larger and more realistic protocol models. However, the implementation still appears not perfectly stable and lacks the documentation needed to enable broader tool adoption. Also, the issues with the use of mathematical formalism remain since it is still based on the TAMARIN syntax, preventing novice users from using its capabilities.

2.3 Abstract State Machines and ASMETA framework

2.3.1 Abstract State Machines

Abstract State Machines (ASMs), formerly referred to as Evolving Algebras, constitute a robust system engineering methodology that guides the holistic development lifecycle of systems, including software and hardware. This method orchestrates the progression from initial requirements capture to the ultimate code implementation. ASMs have garnered commendable success across diverse domains, exemplifying versatility and efficacy. Real-world applications of ASMs:

- *Definition of Industrial Standards*: ASMs contribute significantly to formulating industrial standards for programming and modelling languages, affording a precise and unambiguous foundation for specifications.
- *Design and Re-engineering of Industrial Control Systems*: Within industrial control systems, ASMs provide a systematic approach for capturing, refining, and evolving complex control logic.
- *Modelling E-commerce and Web Services*: ASMs furnish a formal methodology for modelling interactions and processes in e-commerce and web services, ensuring a structured representation of online transactions and service-oriented architectures.
- *Design and Analysis of Protocols*: ASMs find utility in designing and analysing communication protocols, ensuring correctness and reliability in systems involved in data exchange.
- *Architectural Design*: ASMs contribute to architectural design by providing a systematic framework for defining and refining system architectures, encompassing the specification of structural components and dynamic interactions.
- *Verification of Compilation Schemas and Compiler Back-ends*: ASMs play a pivotal role in verifying compilation schemas and compiler back-ends. They facilitate the formal representation of the translation process, ensuring the correctness of the conversion from high-level source code to machine-executable instructions.

Abstract State Machines (ASMs) are a type of transition system that utilises the concept of state to represent the instantaneous configuration of a system during development. The change in state is described by transition rules. ASMs offer a more sophisticated and expressive method for modelling transitions than Finite State Machines (FSMs).

2.3.1.1 Abstract States

In the ASM (Abstract State Machine) framework, the concept of an ASM state refers to multi-sorted first-order structures. These structures consist of domains of objects that have functions and predicates defined on them.

More specifically, a state in the ASM framework is represented by a set of couples, each consisting of a location and its associated value.

The ASM locations in the ASM framework refer to pairs consisting of a function name and a list of paracharacterised. These pairs represent basic object containers, such as memory units, in the abstract ASM concept. Location updates are the fundamental units of state change in the ASM framework and are expressed as assignments of the form *loc* := *v*, where *loc* is a location and *v* is its new value. To provide a more precise definition of terms, we formally denote the following:

Definition 1 (Signature). *A Signature (or Vocabulary)* Σ *is a finite set of function names. Each function, denoted by* f*, is associated with an arity, a non-negative integer representing the number of arguments it takes. Functions in the signature can be categorised as static or dynamic. Nullary functions, those with arity zero, are often called constants.*

It is important to note that, as explained later, the interpretation of dynamic nullary functions may vary from state to state, akin to programming variables. Moreover, every ASM vocabulary is assumed to include the static constants undef, True, and False.

Definition 2 (Location). *A Location can be defined as a pair* $(f,(1,\dots,U_n))$ *consisting of a function name* f *of arity* $n \geq 0$*, fixed by the signature, and an argument* $(V_1,\dots,U_n)$ *(empty if* $n = 0$*), formed by a list of dynamic parameter values* v_i *of whatever type.*

The concept of locations can be regarded as an abstraction of memory units, where the mechanisms of memory addressing and object referencing are not explicitly defined [31].

Definition 3 (State). *An ASM state* (A) *of a signature* (Σ) *consists of a non-empty set X (called the superuniverse of A) and interpretations for the function names in* Σ*.*

An interpretation of an n-ary function f, where $n > 0$, is a function $f_A : X^n \to X$. For a 0-ary function c, an interpretation c_A is an element of X.

ASM functions may be partial and not completely defined. They can be viewed as total functions, where the interpretation of an unspecified location ($f(v_1,\ldots,U_n)$) is equivalent to the interpretation of the constant *undef* ($f_A(U_1,\ldots,U_n) = undef_A$). The superuniverse ($X$) is typically divided into smaller domains. A domain D can be described using a characteristic function (g) that indicates the elements of the superuniverse that make up the domain, i.e., for all $x \in X$, $x \in D$ if and only if $g(x) =$ true.

2.3.1.2 Transitions

ASM transition rules are used to express changes in function interpretations from one system state to another. These rules describe how the system configuration is modified. The basic format of an ASM transition rule is a guarded update, which takes the form of "**if** Condition **then** Updates". The rule executes a set of function updates (or update rules) simultaneously when the Condition is true. The function updates are of the form $f(t_1,\ldots,t_n) := t$, where f is an arbitrary n-ary function and $t_1,\ldots,t_n,t$ are first-order terms.

Definition 4 (**Update**)**.** *Updates are location-value pairs, represented by* (loc, v)*, where* $(f,(v_1,\ldots,v_n),v)$ *represent the basic units of state change.*

Definition 5 (**Update set**)**.** *The update set represents the set of all the applicable updates in a state.*

The update set may contain several updates for the same function name, f. In this case, the updates must be consistent; otherwise, the execution stops. In its most basic form, the transition rule is characterised by a guarded update.

Definition 6 (**Consitent Update set**)**.** *An Update set, U, is considered consistent if it meets the following requirements:*

$$\text{If } (f,(a_1,\ldots,a_n),b) \in U \text{ and } (f,(a_1,\ldots,a_n),c) \in U, \text{ then } b = c.$$

This means that a consistent update set contains at most one value for each function and argument tuple.

Definition 7 (Run). *A run (or computation) of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n$ of states of the machine, where s_0 is an initial state and each s_{i+1} is obtained from s_i by simultaneously applying all of the transition rules which can fire in s_i.*

2.3.1.2.1 Transition rules

A transition rule can be expressed as a guarded update:

if *cond* **then** *updates*

The rule applies when a first-order formula without free variables called *cond* is true, and a set of function updates called *updates* of the form $f(1,\ldots,t_n) := t$ are simultaneously executed. Here, f is an arbitrary n-ary function, and $t_1,\ldots,t_n$ are first-order terms. To apply this rule in a state s_i, all terms $t_1,\ldots,t_n$ are evaluated at s_i to their values, say $v_1,\ldots,v_n,v$. Then the value of the location $(f,(v_1,\ldots,v_n))$ is updated to v, representing the value of the location in the next state s_{i+1}. ASMs provide a rich set of transition rules that allow us to describe complex guarded updates quickly and concisely. These transition rules are highly expressive and can be used to express a variety of scenarios. In the following, we will provide a brief description of the transition rules that are available in the ASMs.

Skip rule. It does not result in any updates.

skip

Update Rule. The update rule specifies the process by which the location $f(v_1,\ldots,v_n)$ is updated to the value v in the current state. This involves evaluating the terms $t_1,\ldots,t_n$ and t to obtain the values $v_1,\ldots,v_n$ and v, respectively. These values are then used to update the location f in accordance with the rule.

$$f(t_1,\ldots,t_n) := t$$

Block Rule. It executes rules R_1 to R_n in parallel.

```
    par
        $R_1$
        ...
        $R_n$
    endpar
```

Conditional Rule. The rule R_1 is executed upon the satisfaction of the boolean condition *cond*, whereas rule R_2 is executed otherwise.

```
    if $cond$ then
        $R_1$
    else
        $R_2$
    endif
```

Forall rule. It executes rule R in parallel for all x values where the boolean condition *cond* is true. The variable x can appear in both the boolean condition and rule R.

forall x **in** D **with** *cond* **do** R

Choose rule. It non-deterministically chooses a value for x for which the boolean condition *cond* is true and executes the rule R with the selected value. Nothing is done if no value for x satisfies the condition. The variable x can occur in the boolean condition and rule R.

choose x **in** D **with** *cond* **do** R

Extend rule. It extracts a new element e from the reserve (i.e., a subset of the superuniverse X, containing new elements), removes e from the reserve and adds it to the domain D. After that, it executes rule R, which allows the new value e to be utilised.

extend x **in** D **with** *cond* **do** R

Let rule. It assigns the value of t to x and executes R. The variable x can occur in the rule R.

let x=t **in** R

Call rule. It invokes the rule R using as parameters $t_1, \ldots, t_n$.

$$R(t_1, \ldots, t_n)$$

2.3.1.3 ASM Definition

Having defined the notion of signature, state, and transition rule in the section above, we can formally define an *ASM* using the notion of *rule declaration*.

Definition 8 (**Rule definition**). *A rule definition for a rule name r of arity n is an expression:*

$$r(x_1,\ldots,x_n) = R_i$$

Let R_i be a transition rule. When calling a rule, denoted by $r(t_1,\ldots,t_n)$, the variables x_i in the body R_i of the rule definition are replaced by the corresponding parameters t_i.

Definition 9 (**ASM**). *An Abstract State Machine M is a triplet (Σ,A,R):*

- a vocabulary Σ,
- an initial state *A* for Σ,
- a set *R* of rules name consisting of
  - a rule name of arity zero called the main rule name of the machine
  - a rule definition for each rule name.

2.3.1.4 ASM Agents

Multi-agent ASM models involve agents interacting and executing moves according to their programs, which are sets of rules outlining their behaviour.

Multi-agent ASMs can be:

- **Synchronous:** All agents execute their programs in parallel, which means that all agents are synchronised using an implicit global system clock [31]. A synchronous multi-agent ASM allows a complex single-agent ASM to be decomposed into sub-agents. Each sub-agent interacts with the others, capturing parts of the complex single-agent ASM, including the signature, behaviours and transition rules.

- **Asynchronous:** The moves of the agents can be scheduled in any desired order. A run of asynchronous multi-agent ASM is a partially ordered set $(M,<)$ that guarantees three conditions:

    1. *Finite History:* Each move $m \in M$ has finitely many predecessors.
    2. *Sequentiality of Agents:* The set of moves of every agent is linearly ordered by $<$.
    3. *Coherence:* Given an initial finite segment X of $(M,<)$, there exists a state $\sigma(X)$ that is the result of applying any maximal element $m \in M$ to a state $\sigma(X-m)$.

2.3.1.5 Functions Classification

ASMs functions are categorised based on their read and update characteristics, as illustrated in Fig. 2.1. The primary distinction is between basic and derived functions. Basic functions constitute the fundamental signature, while derived functions are defined in terms of other basic functions. Basic Functions can be categorised into:

1. **Static Functions:** Functions that remain constant throughout any execution of the machine. A particular case of static functions is the 0-ary functions that can be interpreted as constants.

2. **Dynamic Functions:** Functions that may be modified by the environment or through machine updates. Similarly to 0-ary static functions, they have a particular interpretation. The 0-ary dynamic functions can be viewed as variables.

Dynamic Functions can be further divided into:

1. **Monitored Functions:** Functions updated by the environment or another agent in multi-agent systems. They are read-only for the ma-

chine, which means they cannot appear on the left-hand side of an update rule. The monitored functions specify the portion of the dynamic state controlled by the environment.

2. **Controlled Functions:** Functions read and updated by the machine through transition rules. They cannot be updated by the environment or other agents. The controlled functions identify the portion of the dynamic state directly controlled by the machine.
3. **Shared Functions:** Functions read and updated by both the machine and the environment, representing a combination of monitored and controlled functions. They require a policy to ensure consistent updates by multiple agents.
4. **Out Functions:** Functions updated but not read by the machine (only appear on the left-hand side of an update rule). They can be read but not updated by the environment and other agents.

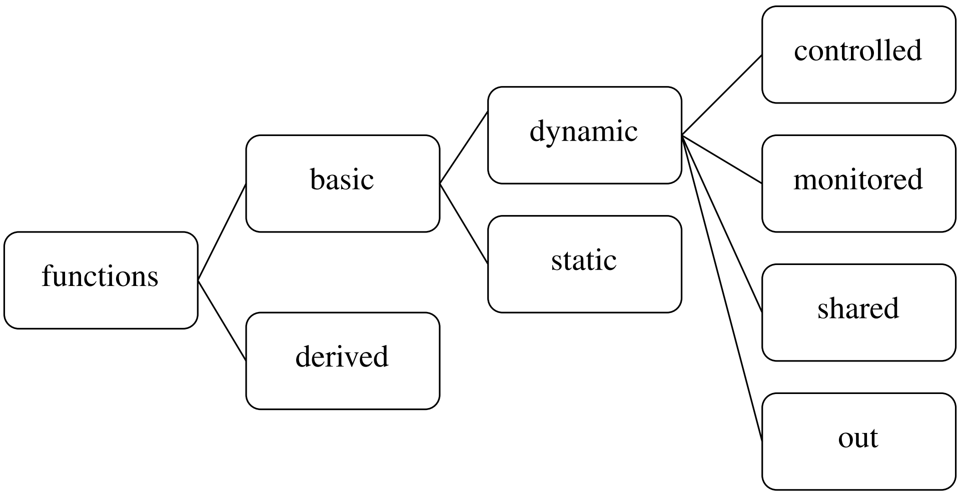


Figure 2.1: ASMs functions classification

2.3.2 ASMETA framework

2.3.2.1 Concrete Syntax and Language Artifacts

ASMETAL [78] is a platform-independent syntax that allows users to create ASM models in a way that is easy to understand for humans. Along with this syntax, the user can use a text-to-model compiler called ASMETALC to parse ASMETAL models and ensure consistency with the AsmM metamodel OCL constraints. Furthermore, users can save ASM models in the XMI interchange format, and Java APIs are available to represent ASMs using Java objects.

2.3.2.2 Simulator

For basic model validation, the ASM simulator ASMETAS [48] can be employed to simulate ASM models. This process helps verify that a system model aligns with the desired behaviour, ensuring that the specification accurately reflects user needs. ASMETAS supports invariant checking, consistent updates checking, random simulation with environmental input, and interactive simulation where inputs are provided interactively during simulation.

2.3.2.3 Scenario-Based Validation

A more robust validation approach is scenario-based validation using the ASM validator ASMETAV [37]. Built upon the ASMETAS simulator and the Avalla modelling language, ASMETAV allows the expression of execution scenarios through interaction sequences. These sequences consist of actions executed by the user actor to set the environment, check the machine state, request the execution of transition rules, and guide the machine through one or more steps as reactions to actor actions.

2.3.2.4 AsmetaRE

For validating system requirements, ASMETARE [79] automates the mapping of use case models (following the Restricted Use Case Modeling approach) into ASM models written in ASMETAL. The resulting executable ASM specification serves as the basis for requirements validation using the SAPIC+ toolset. Ad-hoc transformations also enable the generation of Avalla scenarios from use cases for scenarios-based validation with the ASMETAV tool.

2.3.2.5 Model Review

Model review is a validation technique aimed at assessing the quality of a model early in the system development process. The AsmetaMA [5] tool facilitates an automatic review of ASMs, identifying common vulnerabilities and defects introduced by developers during the modelling activity.

2.3.2.6 Model Checking

Formal verification of ASM models is supported by the AsmetaSMV [4] tool. It takes ASM models written in AsmetaL and translates them into specifications for the NuSMV model checker. AsmetaSMV supports both Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) formulas for verification.

2.3.2.7 Runtime Verification

Runtime verification, as facilitated by the CoMA [6] tool, involves checking whether a running system conforms to given correctness properties. CoMA uses abstract state machines to monitor Java software during runtime, identifying undesirable behaviours and verifying the conformance of the concrete implementation with its formal specification.

2.3.2.8 ATGT (Model-Based Testing)

The ATGT [47] tool is designed for testing ASM models. It generates test suites from abstract models of the system under test, employing adequacy criteria defined for ASMs to measure test suite coverage. ATGT uses the SPIN model checker for automatic test case generation based on counterexamples produced by the model checker.

2.4 π-calculus and ProVerif

2.4.1 Applied π-calculus

Process calculus is a formal framework for describing concurrent computation, elucidating how programs communicate and interact in an environment where multiple programs may execute simultaneously. The primary construct in a process calculus is process parallelisation, akin to instantiating new threads and communication with other processes. The Applied π-calculus, originally defined in [2], extends the π-calculus [68, 67], introducing the symbolic application of functions and equations. A process in the Applied π-calculus is characterised by a finite set F of functions and their arity, an infinite set of names N, an infinite set of variables V, and an equational theory Σ. Terms can be constructed from names, variables, and functions. The equational theory establishes relations between terms. Symbolic application of functions involves applying a function f of arity one to a term t, resulting in the term $f(t)$. Functions without equations can be considered one-way hash functions, generating a deterministic random value for the input without disclosing the arguments. To model a non-hash function, a destructor must be defined by adding equations to the equational theory. For example, if $(g(f(a)) = a) \in \Sigma$, then the term $g(f(t))$ can be created to retrieve t. Here, g acts as a destructor for the function f. Another example of a destructor is a decryption function.

A process in the Applied π-calculus cab be defined as sequence of operations. The grammar for all operations is presented in Figure 2.2. The null process signifies inactivity. Parallel composition runs two processes concurrently as separate threads, and replication denotes a process that runs in parallel with itself an unlimited number of times. Name restriction associates a variable in the process with a fresh name. This action can be interpreted as the process of generating a new, essentially random, value or channel. Additionally, the *event* operation from PROVERIF is included. This operation does not alter the process but is appended to the process trace, a sequence of all executed operations. For instance, the process trace can be utilised to demonstrate the reachability of a specific point in a process or as part of a counterproof of secrecy.

Symbol	**Meaning**
M,N	Terms
a,b,c	Names
z,y,z	Variables
$f(M_1,\ldots,M_n)$	Functions
P,Q,R	Processes
0	Null processes
$P\|Q$	Parallel composition
$!P$	Replication
$\nu n.P$	Restriction
if $M=N$ then P else Q	Conditional
$c(x).P$	Message input
$c\langle N\rangle.P$	Message output
$event(e(M))$	Log event e(M)

Figure 2.2: The grammar notation of applied π-calculus

2.4.1.1 Operational Semantics

Every process runs according to the rules specified in the operational semantics, as presented in Fig. 2.3. Each rule defines an initial process configuration and its resulting counterpart. Some rules may have side-conditions dictating when they are applicable. A process configuration is denoted by (E,A,P), where E is the set of all names bound by the ν operation (with νx binding the variable x to a distinct new name), A is a set of terms representing the adversary's knowledge, and P is a multiset of processes executed concurrently.

Including adversary knowledge in the operational semantics clarifies when an output or input operation can occur and how the adversary influences the process. The adversary's knowledge is reflected in the set A, indicating what the adversary knows or can generate through term creation or equation application. Notably, if the adversary knows a term used for communication, it is considered a public channel; otherwise, it is a private channel. A prerequisite for output on a private channel is the existence of another process with an input operation on that channel to allow the continuation of processes. This constraint is not applicable to output and input on a public channel, where all communication is assumed to be direct with the adversary.

$$
\begin{array}{ll}
(1) & (E,A,P \cup \{0\}) \rightarrow (E,A,P) \\
(2) & (E,A,P \cup \{P|Q\}) \rightarrow (E,A,P \cup \{P\} \cup \{Q\}) \\
(3) & (E,A,P \cup \{!P\}) \rightarrow (E,A,P \cup \{P\} \cup \{!P\}) \\
(4) & (E,A,P \cup \{\nu n.P\}) \rightarrow (E \cup \{n'\},A,P \cup \{P\{n'/n\}\}) \quad \text{if } n' \notin E \\
(5) & (E,A,P \cup \{c(M).P\}) \rightarrow (E,A \cup \{M\},P \cup \{P\}) \quad \text{if } c \in A \\
(6) & (E,A,P \cup \{c(x).P\}) \rightarrow (E,A,P \cup \{P\{M/x\}\}) \quad \text{if } c,M \in A \\
(7) & (E,A,P \cup \{c(M).P_1\} \cup \{c(x).P_2\}) \rightarrow (E,A,P \cup \{P_1\} \cup \{P_2\{M/x\}\}) \quad \text{if } c \notin A \\
(8) & (E,A,P \cup \{if M = N\, then\, P\, else\, Q\}) \rightarrow (E,A,P \cup \{P\}) \quad \text{if } M = N \\
(9) & (E,A,P \cup \{if M = N\, then\, P\, else\, Q\}) \rightarrow (E,A,P \cup \{Q\}) \quad \text{if } M \neq N \\
(10) & (E,A,P \cup \{event(e(M)).P\}) \rightarrow (E,A,P \cup \{P\})
\end{array}
$$

Figure 2.3: Operational Semantics Rules

In this thesis, we establish the convention that the initial process configuration always includes a public name, 'pub.' This name explicitly communicates information to the adversary, such as new terms intended to be public. Formally, the initial configuration is represented as (E,A,P) where pub $\in E$ and pub $\in A$.

A protocol run trace is a sequential record of applied rules and the initial state. This trace provides a detailed account of how the protocol unfolds, including when each message is sent. Such traces are valuable for demonstrating the existence of attacks or verifying the capability to transmit messages within the protocol.

2.4.2 ProVerif

PROVERIF is an automated verification tool for security protocols, employing the applied π-calculus as its input language. Internally, it generates Horn clauses to verify specified security properties. With origins dating back to 2001, PROVERIF is a well-established tool with comprehensive documentation. The verification process involves the transformation of the input, expressed in the applied π-calculus, into Horn clauses. These Horn clauses are subsequently utilised to establish and validate various properties within the security protocol. The verification process encompasses the identification and analysis of potential false attacks. In cases where PROVERIF detects an attack within the Horn clause representation, it endeavours to pinpoint a valid attack trace within the original process description. This entails a backward analysis starting from the Horn clause solution. If a trace is successfully identified, the attack is deemed valid, and PROVERIF generates a sequence of steps to reproduce the attack. Conversely, if no trace is found, PROVERIF outputs "cannot be proved," signifying a lack of conclusive evidence for or against the existence of the specified property. This outcome corresponds to a "don't know" response regarding the protocol's verifiability. Fig. 2.4 provides a schematic representation of PROVERIF's internal workings to aid comprehension of this verification process. The figure illustrates the translation of input to Horn clauses, their application in property verification, and the subsequent analysis to identify and validate potential attacks.

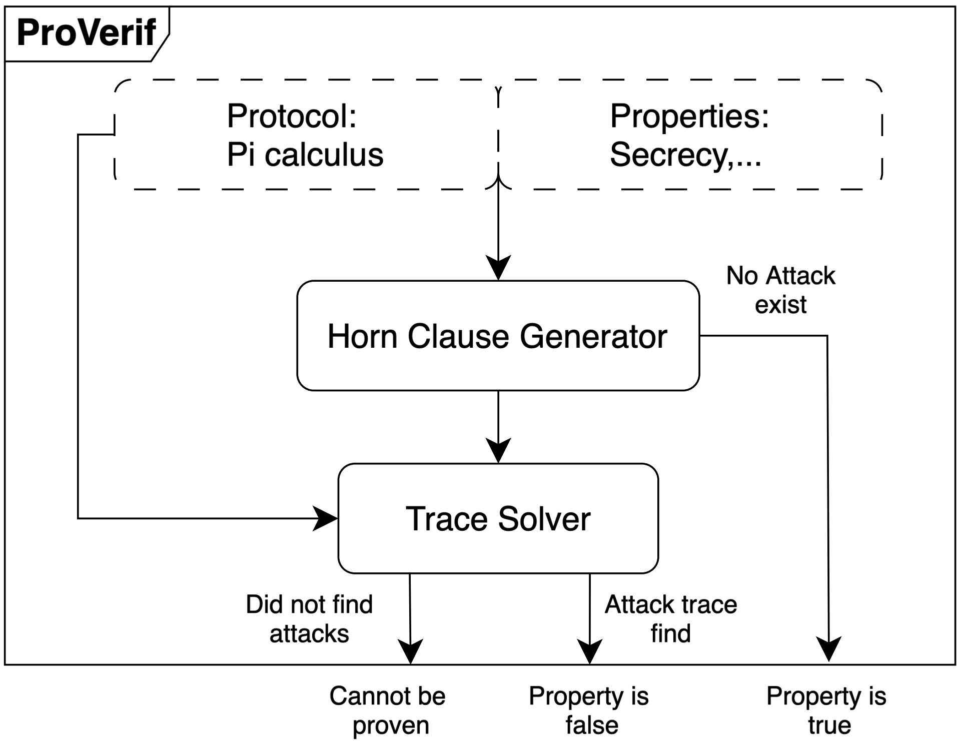


Figure 2.4: The internal logic of PROVERIF

2.4.2.1 ProVerif Input

A protocol is formally defined by specifying a primary process, a set of functions, equations, and queries. Additionally, PROVERIF enables users to articulate predicates using Horn clauses to enhance control over the translation process. Queries serve as a pivotal component in the verification process, delineating the properties that PROVERIF is tasked with proving. These properties may include checks for adversary knowledge or the reachability of specific events. It is crucial to formulate queries meticulously, as inadequately defined queries may result in the verification of a protocol even in the presence of vulnerabilities. Equations play a pivotal role in articulating relationships among functions. A common application involves describing exponentiation, where a function, denoted as $\exp(n,m)$, signifies $n^m \mod p$. For instance, the relation $g^{xy} \mod p = g^{yx} \mod p$ can be succinctly captured through the equation $\exp(\exp(g,x),y) = \exp(\exp(g,y),x)$. Functions are categorised into two types: constructors and destructors. In the absence of a specified destructor, a function is considered a one-way hash function. As discussed in the applied π-calculus section, a destructor is a function facilitating the extraction of a term using equations. For example, the equation $\mathrm{dec}(k,\mathrm{enc}(k,m)) = m$ defines decryption, where the dec function serves as a destructor. Events are conceptualised as elemental log messages seamlessly integrated into the protocol's run trace without exerting any influence on the protocol itself. These events are frequently employed as debugging tools to validate the protocol's ability to complete a session and are instrumental in proving the freshness property within a protocol. Predicates sanctioned by PROVERIF empower users to define lists and abstract concepts such as random selection from a list. These predicates are subsequently employed to mitigate the risk of false attacks during the encoding of state information. In summary, the formalisation of a protocol involves meticulously specifying its main process, functions, equations, and queries and employing PROVERIF-supported constructs like predicates to enhance precision and control during the verification process.

2.4.2.2 Horn Clauses

Horn clauses constitute a fundamental element in logical expressions, embodying a set of predicates interconnected by the logical disjunction (OR) operation. Within this framework, three distinct types of Horn clauses emerge: Facts, Rules, and Goals.

Facts: A Fact is characterised by a singular non-negated predicate denoted as P. This fundamental form represents a straightforward assertion without any negations.

Rules: Rules, on the other hand, present a more intricate structure. They involve multiple predicates, where more than one is negated and precisely one is non-negated. Symbolically, a Rule is expressed as $(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow Q$, which is equivalent to $\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee Q$. This formula captures the essence of a rule, where the conjunction of negated predicates implies a non-negated predicate.

Goals: Goals represent negative clauses devoid of any positive predicates. In a formal sense, a Goal is articulated as $\neg P$, where P is a predicate to be disproven.

The utilisation of these Horn clauses finds application in resolution-based reasoning. In the context of PROVERIF, the construction of clauses involves a combination of rules and facts, where predicates incorporate arguments. Notably, clauses are crafted with a focus on adversary knowledge. Predicates labelled 'attacker' typically feature a single argument, signifying a term known to the adversary. Furthermore, variables employed as arguments within predicates may exhibit nested structures. While these nested arguments are not actively employed during the Horn clause resolution process in PROVERIF, they play a crucial role in the tool's capacity to construct concrete attacks on the protocol. An illustrative example of nested arguments includes their application in nonces, specifying the process by which a nonce was generated.

2.4.2.3 Termination

PROVERIF models may encounter two types of termination problems. One of them arises due to the transformation to Horn clauses, which can result in an infinite loop, causing PROVERIF not to terminate. In the event of non-termination caused by this, it is difficult to predict what output PROVERIF would produce if it were to terminate. The other cause for non-termination is unification. When PROVERIF identifies a trace in the Horn clause representation, it must verify that this corresponds to a trace in the model. This verification process can be time-consuming and memory-intensive. If we encounter this problem, we know that PROVERIF either finds a trace or outputs "cannot be proven". It never returns the result that no attack exists. The uncertainty lies in whether this cause of non-termination is genuine or if the algorithm would eventually conclude, given enough time.

3 Bridging the Usability Gap

This chapter describes our effort to create a user-friendly development environment to model a security protocol. It consists of two parts: (1) a graphical specification language based on the message flow between the agent of the security protocol and (2) a domain-specific language (DSL) called KANT, which would provide first aid in detecting common errors from the security protocol specification. We conducted usability testing for the front end (targeting both the GUI and the KANT language) with participants from the Cybersecurity Master's degree program at Università degli Studi di Milano. The study involved students with a cybersecurity background to ensure relevance and practical insights into the tool's usability within the specific domain.

3.1 AproVer Front-End GUI

We can see two main approaches to developing graphical user interfaces (GUI) for formalising protocols in literature. The first approach involves interfaces that help users understand the tool's output, such as SCYTHER and TAMARIN's traces output [41]. The second approach involves interfaces allowing users to input and specify protocols like SPAN [28], the companion graphical tool for the HLPSL language of AVISPA.

However, for a user unfamiliar with modelling security protocols, it might help to have both possibilities in one tool. Out of this need came the GUI for the APROVER framework, which will allow the flow of a protocol to be modelled similarly to SPAN but with the addition of constraint to avoid introducing basic errors. All information entered during the protocol modelling phase is collected in an interchange format (JSON Schema), allowing a straightforward mapping to back-end verification tools. Also, due to its flexibility, the proposed JSON Schema can be used as a format to encode error traces found by verification tools to be shown to the user in the APROVER GUI.

When deciding between a web-based graphical user interface (GUI) and a native GUI for a verification tool, there are several factors to consider. Opting for a web-based GUI has several advantages over a native GUI. Firstly, web-based GUIs allow for cross-platform accessibility, meaning users can access the verification tool from any device or operating system without needing specific installations. This feature is particularly useful in collaborative settings where stakeholders may use different platforms.

Secondly, the process of updates and maintenance is streamlined with web-based GUIs. Since the GUI is on the web, updates can be applied centrally on the server, ensuring that all users can benefit from the latest features and improvements. This eliminates the need for users to manually update their native applications.

Thirdly, web-based GUIs offer a collaborative potential. Multiple users can simultaneously interact with the verification tool, allowing real-time collaboration and data sharing. This collaborative environment is conducive to teamwork, facilitating joint analyses and discussions.

Fourthly, web-based GUIs offer inherent scalability and flexibility. These apps can adapt effortlessly as the verification tool evolves or experiences increased user demand. This scalability is critical in dynamic environments where the tool's usage patterns fluctuate.

Lastly, a web-based GUI can be more cost-effective from a development perspective. The shared codebase and platform compatibility can lead to resource optimisation, reducing the efforts required to maintain separate native versions.

3.1.1 Usability Features of the GUI

On the above advantages, the choice for the APROVER GUI has fallen on a web app. The landing page of APROVER is shown in Fig. 3.1. Upon opening the interface, it is possible to find the two classic actors of AnB notation (Alice and Bob) arranged on a message sequence graph. On Alice's lifeline, there is a blue rectangle representing her initial state; next to the initial state, it is possible to notice a small letter icon that, if clicked, will open the procedure for message creation.

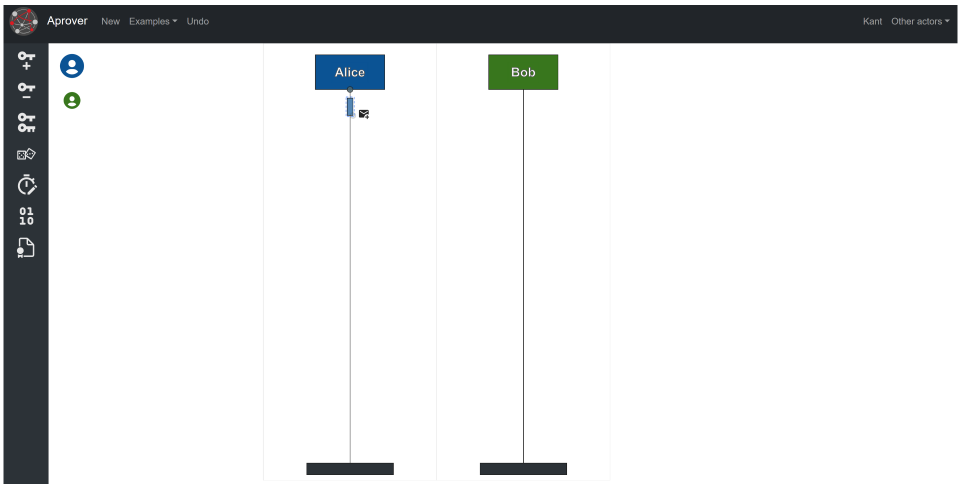


Figure 3.1: AproVer website landing page

On the left side, we find two badges (colour-coded with the colours of the actor labels) that allow us to create, modify or delete the associated actor's knowledge. To the left of the two badges, we find a menu bar containing the categories of knowledge that can be made available to each actor. In Fig. 3.2, it is possible to better appreciate the association between the various knowledge concepts and the icons chosen to represent them; also shown in the above image is how the GUI renders the Needham-Schroeder public-key protocol.

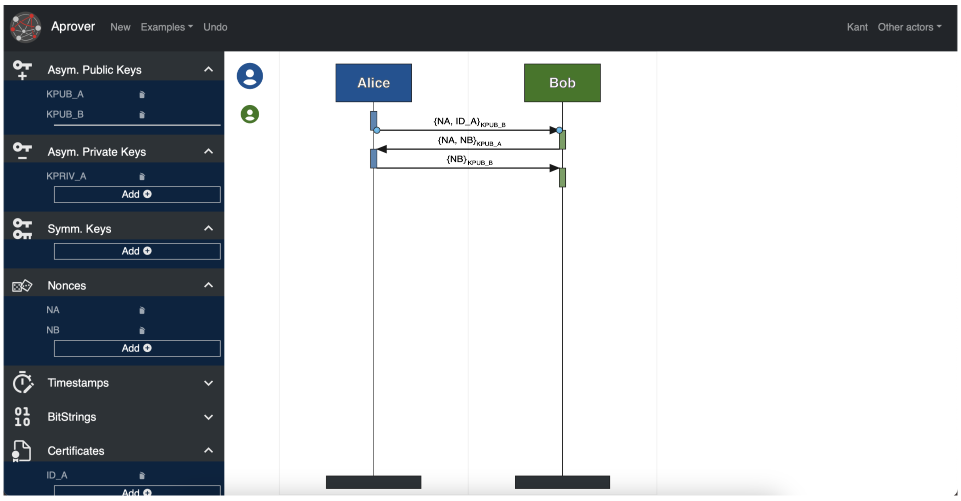


Figure 3.2: AproVer GUI showing the Needham-Schroeder protocol

Once the user clicks on the letter envelope icon in Fig. 3.1, he/she will be redirected to the message creation page, which will look as shown in Fig. 3.3.

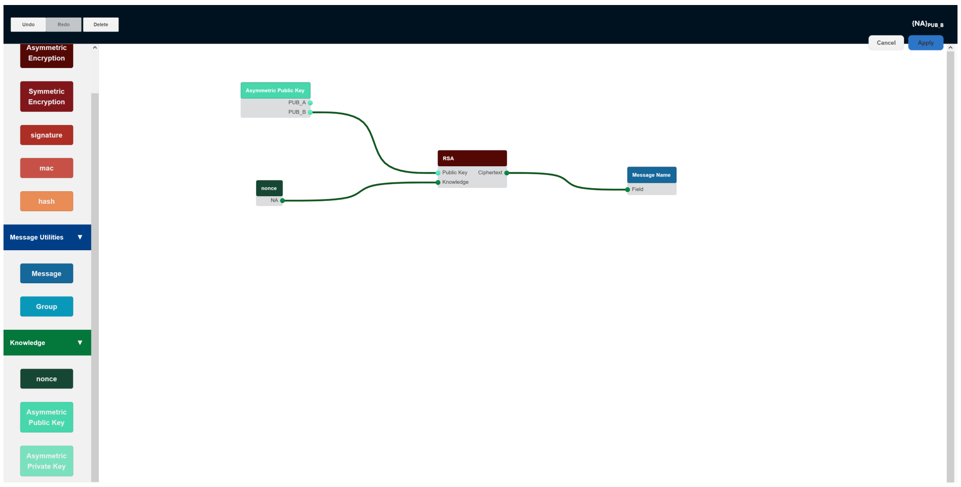


Figure 3.3: AproVer GUI that shows the drag-and-drop interface to build messages

The message creation page provides on the left side a series of boxes which can be dragged into the main space to compose the Message. The boxes are divided into three colour-coded sections. The first section, represented by shades of red, encloses all the cryptographic primitives made available by the interface, which the user can also find in KANT. Cryptographic primitives can take as input the knowledge present in the section represented by the shades of green. However, the input of cryptographic primitives is not limited to a direct application and can be nested (the output of one cryptographic primitive is used as the input of another). The nesting process might require grouping knowledge or the results of diverse cryptographic primitives; this can be achieved through the use of the blue box called Group. Once the message fields have been obtained in the desired manner, they can be linked to the respective nodes of the blue box called Message. A key feature to aid in error reduction present on the message creation page is the semantic control of connections that are allowed between boxes. More precisely, each box has input nodes on the left and output nodes on the right. When a user selects a knowledge to be used in a cryptographic primitive, it must correspond to the label next to the input node to be used. To make this task even user-friendlier for the user, the interface highlights, when a user selects a knowledge or drowning input, all the nodes to which that information can be linked. The page shown in Fig. 3.3, as mentioned above, is generated from a given state of one of the agents participating in the protocol, which means that the agent Alice, in the case of the image, will only possess the knowledge given to her by the user. In subsequent messages, Alice will have access to further knowledge derived from the content of messages received from other agents. The last features for improving usability can be found in the top bar, where the user can undo and redo actions, select boxes to delete them and view the preview of the created message. Once the apply button is pressed, the message creation is completed, and the user will be redirected to the message sequence diagram page. On this page, in addition to entering new knowledge and visualising the flow of the messages, it is also possible to see which knowledge is exposed in case an attacker is eavesdropping on the channel. This view, shown in Fig. 3.4, can be enabled via the button at the top right called other options.

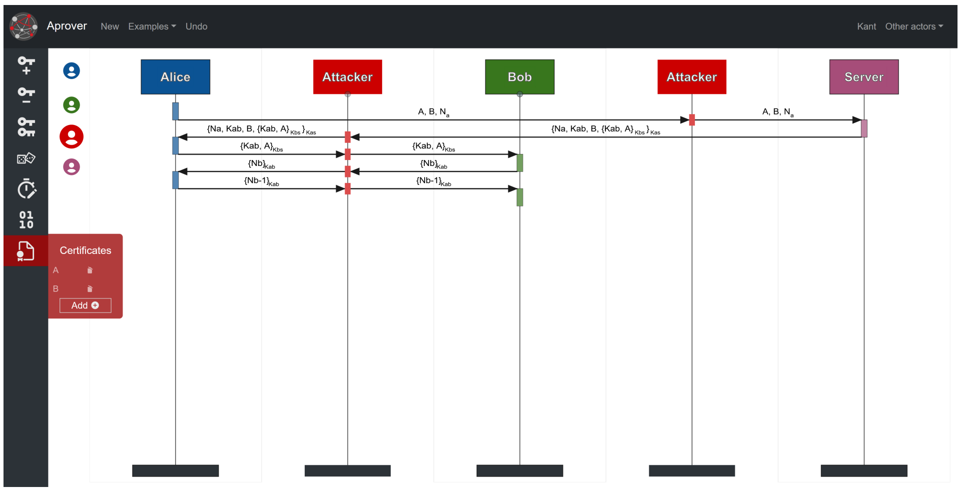


Figure 3.4: AproVer GUI showing an attacker eavesdropping on channels

Lastly, it is possible to import attack traces found by the verification tools and display them on the sequence diagram, as shown in Fig. 3.5. The diagram shows an attack sequence generated by executing an authentication query on the Needham-Schroeder public-key protocol.

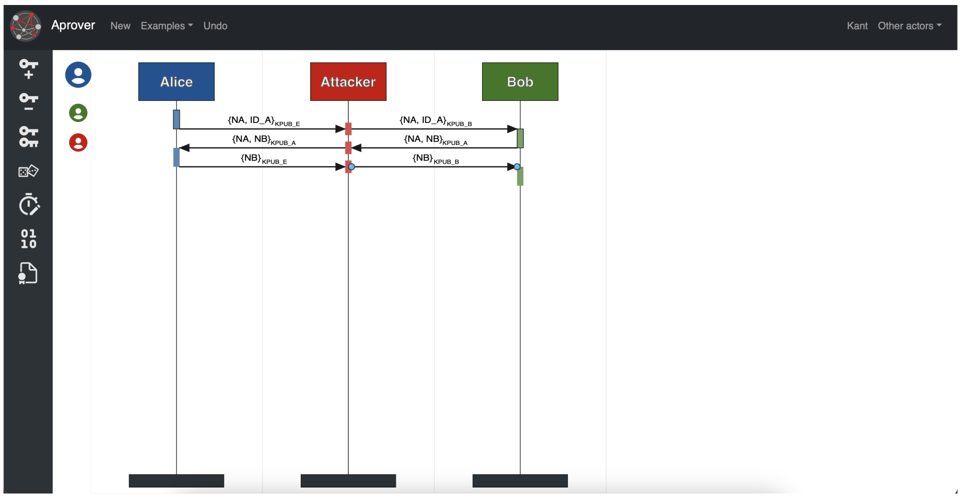


Figure 3.5: AproVer GUI showing an attack trace

3.1.2 GUI Data Handling

As introduced in Section 3.1, the data entered in the interface is handled by a JSON Schema saved in the session cookie of the browser chosen by the user. In this preliminary version of the interface, the cookie is saved in plain text. However, it is planned in the subsequent development phase to add an encryption layer to keep the data secure on the user's local machine. We have created two schemas to validate the correctness of the data passed to the interface; the first schema checks the correctness of the knowledge attributed to a protocol actor, while the second schema deals with validating the structure of the messages.

The first JSON Schema is used to validate the principals' knowledge contained in the cookie against seven types of knowledge: public and private keys, symmetric keys, nonce, timestamp, bitstring, and identity certificates. In trying to limit the introduction of errors by the user, we decided to tie the creation and assignment of the public keys to the generation of the corresponding private keys so that the user does not have to create a posteriori link between the two. The cookie is updated each time a message is received, and the knowledge is added to the receiver only if it has the keys available to reverse the contents of the cryptographic primitives in the message. The cookie containing the principal knowledge can also be updated by the user adding new knowledge, as shown on the left side of Fig. 3.2.

The second JSON Schema validates the cookie that stores the message content. This cookie contains the types defined earlier in the knowledge schema and their application within the cryptographic primitives. The Schema validates the cryptographic primitives to ensure that only acceptable arguments are passed to them. For instance, in Code 3.1's JSON Schema excerpt, line 27 mandates that for symmetric encryption, the key parameter type must be a knowledge of the symmetric key type. We have also incorporated the ability to include metadata in the Schema, as seen in line 10. Symmetric encryption metadata could include the chosen algorithm type. This metadata can be used in a verification phase to provide more information to the verification tools or in a validation phase to warn the user of potential vulnerabilities and suggest more secure algorithms.

Code 3.1: JSON Schema excerpt of the cryptographic primitive symmetric encryption

```
"symmetricEncryption": {
     "type": "object",
     "properties": {
       "id": {
         "type": "string"
       },
       "securityConcept": {
         "const": "symmetricEncryption"
       },
       "algorithm": {
         "type": "string"
       },
       "argument": {
         "allOf": [
           {
             "$ref": "#/definitions/knowledge"
           }
         ]
       },
       "encryptionKey": {
         "allOf": [
           {
             "$ref": "#/definitions/knowledge"
           }
         ],
         "required": [
           "symmetricKey"
         ]
       }
     },
     "required": [
       "id",
       "securityConcept",
       "algorithm",
       "argument",
       "encryptionKey"
     ]
   }
```

3.2 KANT Language

We here present KANT (Knowledge to ANalyzing Trace), a DSL explicitly designed as the front-end language of APROVER for the specification of security protocols. KANT is a high-level language with primitives for protocol modelling and assumption expression. It has been conceived as a sort of *lingua franca* that allows simple textual descriptions to express protocols and easy translation into the input languages of various tools for protocol analysis. Since KANT has been engineered and developed by using the Langium platform[1], its grammar definition comes with an editor and several other tools that allow checking KANT models not only for syntactic correctness but also for consistency concerning a given semantic model. The latter is expressed in terms of a set of validation rules describing constraints and best practice principles in designing security protocols. Such a checking mechanism is extremely important at design time to avoid common security errors (e.g., incorrect use of encryption keys) or to warn the designer regarding choices that might lead to protocol vulnerabilities (e.g., a principal's name not mentioned explicitly in the message). Note that in Fig. 1.1, blocks in grey are under development, and the mapping from KANT to validation and verification tools is out of the scope of this work.

The Section is subdivided as follows. Section 3.2.1 recalls the concept of security protocol and presents a protocol that we use as a reference scenario throughout the paper. Section 3.2.2 introduces the Langium platform used to develop KANT as DSL, while KANT modelling primitives and constructs are given in Section 3.2.3. Section 3.2.4 presents the semantic model of KANT and shows the application of the validation rules. In Section 3.2.5, we show the use of KANT and the language validation mechanism on the reference scenario.

3.2.1 Reference scenario

Although there is a wide range of protocols, differing by the number of principals (or actors), the number of messages, and the protocol's goals (that may often be expressed with a list of desired security properties), they all

[1]`https://langium.org/`

share a common structure. Indeed, a communication protocol consists of a sequence of messages between two or more principals. Each message may be written by using the classical Alice&Bob notation in the form:

$$\mathsf{M1.}\ A \rightarrow B : \mathit{message\_payload}$$

which specifies:

- The *principals* (or *actors*) exchanging messages (in general, symbols A and B represent arbitrary principals, S a server). In particular, the direction of the arrow specifies the sender and the receiver of the message.

- The order in which messages are sent, and their specific *payload*. In particular, M1 is a label identifying the message, whereas *message_payload* specifies the actual content of the message.

In secure protocols, payloads can be partially or totally ciphered, either by symmetric-key encryption (in this case, a key shared between actors is used both to encrypt and decrypt), or by asymmetric-key encryption (here, K_B and $K_B{}^{-1}$ is used to specify a public and private key of B, to encrypt and, respectively, decrypt). Message payloads can contain other information, such as nonces (N), timestamps (T), etc.

The security goals are often defined with respect to CIA (Confidentiality, Integrity, Authentication) triad. The most common are *confidentiality* or *integrity* of message payloads, or *entity authentication* (i.e., the process by which one entity is assured of the identity of a second entity that is participating in the same session of a protocol. Thus, they share the same values of the protocol parameters, such as session keys, nonces, etc.).

Consider the classic Needham-Schroeder public-key protocol (NSPK, for short) that will be used throughout the paper as a running example to introduce the KANT notation.

$$\boxed{\begin{array}{ll} \mathsf{M1.}\ A \rightarrow B : & \{A, N_A\}_{K_B} \\ \mathsf{M2.}\ B \rightarrow A : & \{N_A, N_B\}_{K_A} \\ \mathsf{M3.}\ A \rightarrow B : & \{N_B\}_{K_B} \end{array}}$$

It was introduced in 1978 for mutual authentication (here, we omit the exchanges with the certification authority to get the public keys). It consists of three messages: in the first message, principal A sends to B a message containing her identity, A, and a nonce, N_A, to avoid replay attacks (i.e., reuse of old messages, often called a *challenge* message), that only B can decrypt with his private key. B's answer (message M2) is ciphered with A's public key and contains nonce N_A to authenticate B (he is the only one able to decrypt message M1 and obtain N_A in clear), and a nonce N_B to authenticate A with B. Since message M2 is encrypted with A's public key, she is the only one who can decrypt it, thus if B receives message M3 containing nonce N_B encrypted with his public key, A is authenticated, too.

3.2.2 KANT Development

KANT (Knowledge ANalysis of Trace) is the domain-specific language we explicitly designed and implemented in Langium for the specification of security protocols. Using a tool to engineer a DSL offers several advantages that can streamline the development process and enhance the utility of the DSL, such as syntax highlighting autocompletion, and debugging support.

In particular, Langium represents an innovative tool in the context of language engineering, enabling DSL development in a web-based technology stack. When used in the context of a desktop app (e.g., VS Code, Eclipse IDE, etc.), Langium runs on the Node.js platform. Additionally, it can run in a web browser to add language support to web applications with embedded text editors (e.g., Monaco Editor). The interface between Langium and the text editor is the Language Server Protocol (LSP), allowing languages based on Langium to seamlessly interact with a range of popular IDEs and editors supporting LSP.

A *grammar language* is provided to specify the syntax and structure of the language. The grammar rules describe the concrete syntax by instructing the parser how to read input text, and also the abstract syntax in terms of meta-classes and their properties. For a given text document, Langium creates a data structure called Abstract Syntax Tree (AST): every grammar rule invocation leads to a corresponding node in the AST that is a JavaScript object, and Langium generates a TypeScript interface for every rule to provide static typing for these nodes. As programs written in the language are parsed, Langium automatically generates Abstract Syntax Trees based on these interfaces, making efficient manipulation and analysis of the parsed content possible. In particular, Langium allows to implement also custom validation without the need to involve complex external tools, thus ensuring that validation is defined alongside the grammar specification. It provides a cutomizable Validation Registry mapping validation functions with specific nodes in the AST, enabling targeted validation checks at relevant points in the AST (e.g., either internal or leaf nodes). Designers only need to define the validation functions encapsulating the desired checks on both the structural and semantic aspects of the language captured by the AST, and to update the Validation Registry with the new functions. This approach is great for rapid prototyping, when the focus is on designing the syntax of a new language.

Beyond parsing, Langium extends its capabilities to establish connections between different language elements, enabling cross-referencing and linking. These relationships play an important role, and Langium is capable of resolving them automatically thanks to built-in scoping and indexing.

Although Langium's grammar declaration language is similar to Xtext, they are built upon different open-source libraries and tools: Xtext is built upon Eclipse and ANTLR, while Langium is built upon Visual Code and Chevrotain. Moreover, Xtext is heavily based on the Eclipse Modeling Framework, whereas Langium uses TypeScript interfaces, enabling language engineering in TypeScript, the same technology that is used for VS Code extensions. In contrast, building a tool that uses an Xtext-based language server with VS Code or Theia means creating a hybrid technology stack with some parts implemented in Java and others in TypeScript. Developing and maintaining such a mixed code base is more demanding, and long-term maintenance is likely more difficult than Langium's coherent technology stack.

3.2.3 KANT Syntax

In this section, we present the abstract and concrete syntax of the KANT language and show how it encompasses all the essential aspects of security protocols. In particular, it integrates useful features that can be helpful for the verification process and provide valuable insights for protocol correctness at the design phase.

The syntax includes constructs of the notations commonly used to express security protocols (e.g., the Alice&Bob notation); however, it has been extended with the concept of *knowledge exchange* between the parties involved in the security protocol session to help reasoning and analysing the flow of information during the communication.

KANT has been developed by exploiting the MDE approach for a DSL definition, so its abstract syntax is given in terms of a metamodel from which a concrete textual notation is derived. KANT's meta-model (see Fig. 3.6) defines six mandatory meta-classes and two optional ones are necessary to specify a valid *protocol model* in KANT. The protocol must contain specific elements, including the definition of *principals* involved in the protocol (such as Alice, Bob, and Server), the *types* used in the function definitions and the *definition* of cryptographic *functions* and their inverse functions, the *knowledge* that each *principal* has during the message exchange, and the definition of the *communication mechanism* for messages exchange.

In addition to these mandatory model sections, two optional meta-classes can be used to define *shared knowledge* between principals, and to help the user writing, in a human-readable style, either *properties* or constraints on the model elements, and security *checks*. These statements are arguments for the validation and verification of the model.

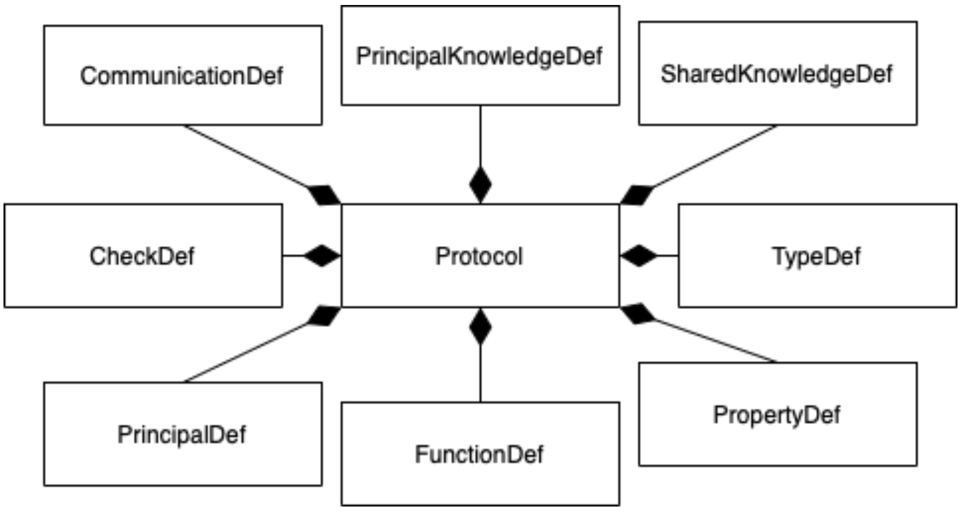


Figure 3.6: Protocol Meta-model

3.2.3.1 Principal Definition

A protocol model (instance of the meta-model) written in KANT starts with the name declaration of the `principals` involved in the message exchange. Two examples of declarations follow (the user is free to select the most suitable name):

```
principal Alice
principal Alice, Bob, Server
```

3.2.3.2 Type and Function Definitions

We designed the language with static analysis in mind, so we included types in it. The user can use built-in types that are declared in the language *prelude*, but he/she can also define other types. An example of `type definition` is the following:

```
type SymmetricKey, PublicKey, BitString, Group
```

where `SymmetricKey` and `PublicKey` are types of cryptographic keys used for symmetric and asymmetric encryption, respectively; `Bitstring` is a special type we call 'sink type' since a function that accepts a Bitstring as a parameter can accept any other type; `Group` represents elements of an Abelian group, i.e., a set of elements with commutative operations.

Types are used to categorize information in the principal's knowledge and allow the validation of the usage of a piece of information through the protocol. Moreover, since some input languages of the back-end tools are typed, using types at the top level of the KANT model removes the need for a conversion step.

The `FunctionDef` meta-class enables users to define custom functions in agreement with the symbolic model [25] to describe both invertible and one-way functions. A function is defined by a `name`, one or more `parameters`, and one or more `return values`. Each function component requires an `identifier` and a type, either pre-defined or custom-defined to suit the particular use case. Cryptographic functions, in addition, require the definition of a `key` used to encrypt data (which are the function parameters).

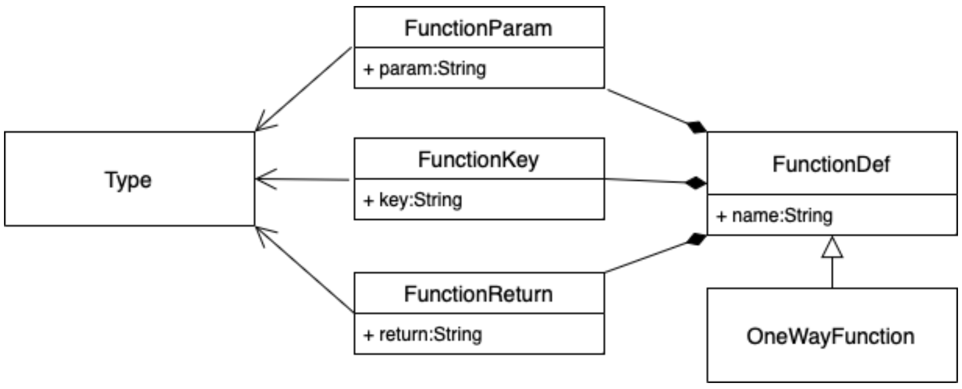


Figure 3.7: Function Definition Meta-model

The separation of the `FunctionDef` in the composing meta-classes (`FunctionParam`, `Functionkey` and `FunctionReturn`, see Fig. 3.7) enables us to define validation rules (see Section 3.2.4) that are specific to each class and, in turn, helps to streamline the validation process.

Examples of function definitions are given below for the function `ENC` to encrypt data with a given key and the one-way cryptographic `HASH` function.

```
function ENC(content:BitString)
   with k:SymmetricKey -> [ enc:Ciphertext ]

function HASH(value:BitString)
                  -> [ hash:Digest ] one way
```

3.2.3.3 Property Definition

Properties can be added to a model to express constraints on model elements (e.g., on functions and their parameters) or equivalence properties (e.g., Diffie-Hellman exponentiation, as well as other equational theories). For example, the following property states the identity function as the result of applying description on encrypted data by using a symmetric key.

```
property forall x:BitString, k:SymmetricKey |
           DEC(ENC(x) with k) with k -> [ x ]
```

An example of a property stating function equivalence is the following that guarantees the Diffie-Hellman equivalence between two private keys `a` and `b` on a generator `g` in a finite cyclic group . [2]

```
property forall a:PrivateKey, b:PrivateKey,
             g:Group | DF(EXP(g,b),a)
             equals DF(EXP(g,a),b)
```

3.2.3.4 Knowledge Definition

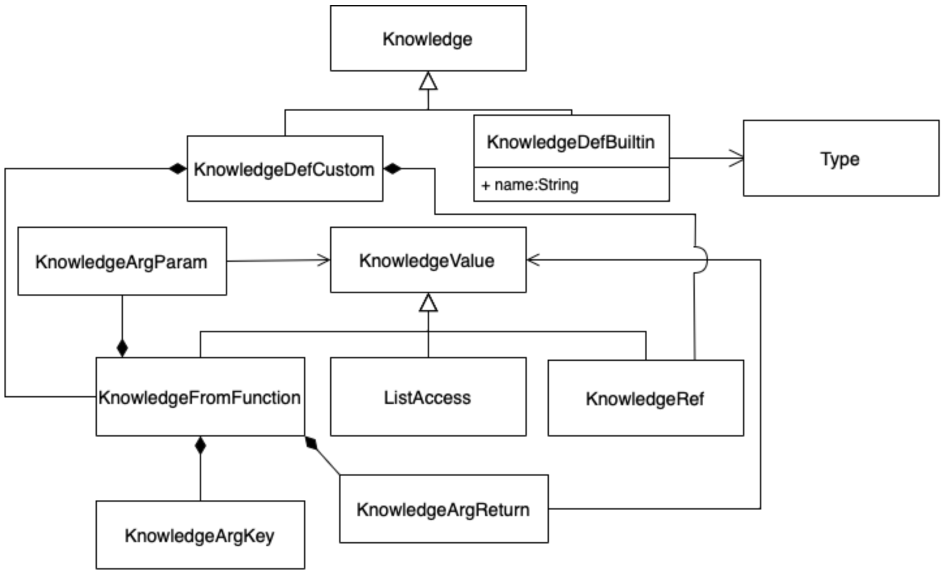


Figure 3.8: Knowledge Meta-model

KANT allows declaring the knowledge of the principals. There are two ways to accomplish this: (*i*) at the beginning of the protocol model, define the knowledge that is *shared* by selected principals in the initial state of the protocol run (by the construct `share` of the meta-class `SharedKnowledgeDef`); (*ii*) define *private* principal's knowledge at any point of the protocol model, but before sending a message (by the construct `know` in the meta-class `PrincipalKnowledgeDef`). In both cases of knowledge definition, two qualifiers can be used to distinguish between knowledge that is regenerated every time the protocol is executed (`fresh`

[2]According to the Diffie-Hellman key exchange protocol, the shared secret between two parties is given by the formula $K = (g^a)^b \mod p = (g^b)^a \mod p$, being p a prime number and g a primitive root modulo p on which the two parties agree and that are public values, a and b secret values chosen by Alice and Bob, respectively.

knowledge at any protocol session), and knowledge that remains the same (constant knowledge during a protocol session) – both qualifiers are defined in the meta-class `KnowledgeDefBuiltin` on top of the meta-classes `SharedKnowledgeDef` and `PrincipalKnowledgeDef`.

Private knowledge of a principal can include the specification of its own Finite State Machine (FSM): principal's initial state is specified as

```
        state initial_state;
```

at the beginning of the protocol model; information on a reached state and its enabled transition can be specified at any point of the protocol model as:

```
        state reached_state;
        transition previous_state => reached_state;
```

Information relative to principal's FSM is primarily used in verification, but it is also useful in validation to analyze the knowledge flow of principals[3].

Two examples of shared and private knowledge specification follow:

```
    Bob,Alice share{
        const key:SymmetricKey;}

    Alice know{
        state second_state_Alice;
        transition first_state_Alice => second_state_Alice;
        fresh priv_a:PrivateKey;
        pub_a = PUB_GEN(priv_a);
        enc_mess = ENC(m) with key;}
```

The knowledge block is used to specify what a principal knows at the application of a given protocol rule. Fig. 3.8 shows the relation among the meta-classes for knowledge representation.

Each information is identified by a reference `priv_a` of a given type `PrivateKey` in `priv_a:PrivateKey` and can be constant or fresh (from the meta-class `KnowledgeDefBuiltIn`). Additionally, information of meta-class `KnowledgeDefCustom` is of the form `ref= ...` and can refer to the result of a function application (e.g., `puba=PUB_GEN(priva)`), or it

[3]Since not a mature feature yet, KANT allows for drawing the FSM of each principal from his/her knowledge block.

can reference some saved information derived from the received messages (e.g., `dec_na = PKE_DEC(enc_na)`) or built to be sent as a message (e.g., `enc_na = PKE_ENC(na)`) (see the reference scenario model, in the following).

Particular importance is given to the two built-in functions that we have defined to facilitate list management (and that we use to model our reference scenario). The language prelude contains the basic types and functions: CONCAT and SPLIT of the meta-class `ListAccess`; the former yields the reference of a list concatenating a sequence of elements; the latter returns the list of elements from a given list reference. The property that derives from the two functions definition (see below) guarantees that concatenating a sequence of elements into a referenced list and then sliting such referenced list, holds the original list.

```
function CONCAT(...values: BitString)
                    -> [ value: BitString ]
function SPLIT(value: BitString)
                    -> [ values: BitString ]
property forall v: BitString |
                SPLIT(CONCAT(...v)) -> [ v ]
```

3.2.3.5 Principals communication

A principal `Alice` builds within its knowledge a piece of information, `enc_mess`, that it wants to send as a protocol message to one of the other participants, `Bob`. To send the message, we exploit the concrete syntax of the meta-class `CommunicationDef`:

```
Alice->Bob: enc_mess
```

3.2.3.6 Security Checks

KANT grammar allows the user to specify security properties, which are verified once the model is transformed into a valid model for the back-end verification tools. The validation rules introduced in Sect. 3.2.4 can guarantee a lightweight form of static analysis of the information flow. Dynamic analysis can be performed only by back-end verification tools.

We provide three meta-classes, `ConfidentialityCheck`, `EquivalenceCheck`, and `AuthenticationCheck`, to express different kinds of security checks. The concrete syntax follows:

- *ConfidentialityCheck*: an information `m` must be known `only` by principals `Alice` and `Bob`

  ```
  only Alice,Bob should know m
  ```

- *EquivalenceCheck*: this is used to check whether two pieces of information are equal or not; this is relevant for a security protocol when communication happens in an insecure channel and some information can be stolen or altered during the protocol run.

  ```
  g_ab, g_ba should be equal
  ```

- *AuthenticationCheck*: the nounce `nb` allows `Alice` to `authenticate` principal `Bob`.

  ```
  Alice should authenticate Bob with nb
  ```

3.2.3.7 KANT Model of the Reference Scenario

The following KANT model [4] is the specification of the NSPK security protocol described in Sect. 3.2.1: Alice and Bob have an associated FSM; they do not share any knowledge; each principal has private keys, uses a public key (generated from the corresponding private one) to encrypt messages, decrypts messages by using its own private key, generates fresh nonces and builds private knowledge according to the protocol rules.

```
principal Alice, Bob

Alice know {
    state sending_puba;
    const priva: PrivateKey;
    puba = PUB_GEN(priva);}

Alice -> Bob : puba

Bob know {
    state waiting_puba;
```

[4] All the KANT language artefacts (grammar, models, validation rules, etc) are available at `https://github.com/Aprover/KANT` on GitHub.

```
    const privb: PrivateKey;
    pubb = PUB_GEN(privb);}

Bob -> Alice : pubb

Alice know {
    state waiting_pubb;
    transition sending_puba=>waiting_pubb;
    fresh na: Nonce;
    enc_na = PKE_ENC(na) with pubb;}

Alice -> Bob : enc_na

Bob know {
    state waiting_enc_na;
    transition waiting_puba=>waiting_enc_na;
    fresh nb: Nonce;
    dec_na = PKE_DEC(enc_na) with privb;
    nb_na = CONCAT(nb, dec_na);
    enc_nb_na = PKE_ENC(nb_na) with puba;}

Bob -> Alice : enc_nb_na

Alice know {
    state waiting_enc_na_nb;
    transition waiting_pubb => waiting_enc_na_nb;
    dec_na_nb = PKE_DEC(enc_nb_na) with priva;
    rec_na_nb = SPLIT(dec_na_nb);
    enc_nb = PKE_ENC(rec_na_nb[1]) with pubb;}

Alice -> Bob : enc_nb

Bob know {
    state waiting_enc_nb;
    transition waiting_enc_na=>waiting_enc_nb;
    dec_nb = PKE_DEC(enc_nb) with privb;}

check nb, dec_nb should be equal
check Bob should authenticate Alice with nb
```

3.2.4 KANT Validation Rules

A model written in a DSL is an instance of the language meta-model. Thanks to Langium, a model can be validated according to certain validation rules and constraints, covering both syntactic and semantic aspects. These rules have to be defined at the meta-model level. Error and warning messages can be reported upon model validation.

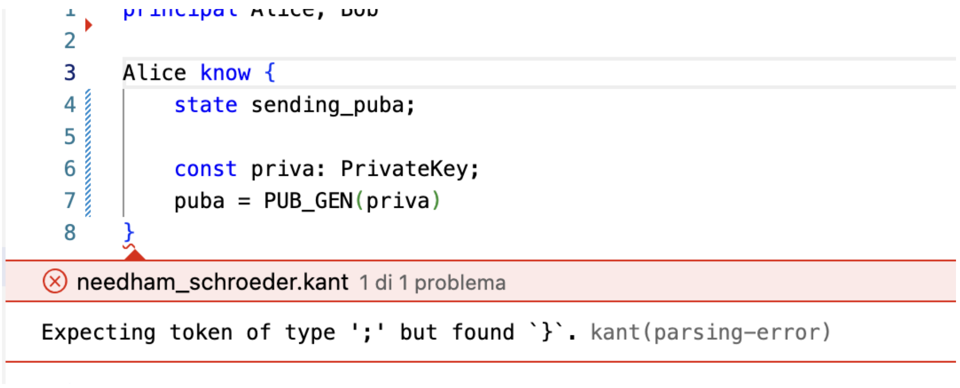
In KANT, the model validation phase helps the protocol designer to avoid common mistakes such as incorrect use of encryption keys or definition of incorrect knowledge flows. Moreover, KANT grammar has some restrictions on the use of undeclared names of functions, types, and principals, and these restrictions are captured by the use of *cross-references* in Langium (see Sect. 3.2.2). Thus, it is possible to eliminate incorrect spelling of terms during model writing and to suggest, by means of auto-completion, only what is allowed.

Besides the advantages offered by the cross-reference mechanism in terms of revealing mistakes and making suggestions, in order to improve KANT model validation, we defined and implemented three sets of validation rules whose violations result in errors or warnings. Rules for *syntactic checks* (see Sect. 3.2.4.1) are used to reveal syntactic errors that must be corrected. Rules for *semantic checks* (see Sect. 3.2.4.2) can reveal potential errors or violations of security properties. Rules for *prudent engineering practices* (see Sect. 3.2.4.3) work as guidelines for security protocol specification following some of the principles outlined in [1]; they help to identify and prevent common error patterns found in the literature, which might lead to attacks. Especially for semantic and prudent practice rules, a warning serves as a suggestion to improve the clarity of the model or to adopt good practices in writing the protocol.

All validations are performed in real-time and are triggered by the user entering new knowledge. In Section 3.2.4.1 we show an example taken from the GUI of Visual Studio Code, subsequent error reports are shown in plain text for better readability. We convey to mark incorrect elements by a *red* underline and a warning by a *yellow* underline. We use fragments of the KANT model of the NSPK protocol to show the application of the validation rules.

3.2.4.1 Syntactic Checks

Syntax Well-formedness This set of validation rules checks for the correct formatting of parentheses, comments, and delimiters when writing a protocol in KANT. The example in Fig. 3.9 catches the error of a missing semicolon, which is used to declare separate knowledge.



```
principal Alice, Bob

Alice know {
    state sending_puba;

    const priva: PrivateKey;
    puba = PUB_GEN(priva)
}
```

needham_schroeder.kant 1 di 1 problema

```
Expecting token of type ';' but found `}`. kant(parsing-error)
```

Figure 3.9: Semicolon placement error

Knowledge declaration New knowledge must be either fresh, constant or the result of a function. When declaring fresh or constant knowledge, the type must also be specified.

```
Alice know {
    state sending_puba;
    const priva;
    }
```

The error message specifies the expected token sequence:

Expecting: one of these possible token sequences:

1 `[ID : [Type:ID]]`

2 `[ID : [Type:ID], ID : [Type:ID]]`

but found: 'priva'.

Naming Convention The naming convention in KANT requires that principals' names start with capital letters, function names are capitalised, and variables are lowercase.

```
principal  alice , Bob
```

In this case, it is raised a warning since this is a stylistic convention and not a real error:

Principal name alice should start with a capital letter.

Keyword Usage Keywords can only be used in the way specified by the syntax (similarly to other programming languages).

```
Bob know {
     state state;}
```

This means it is impossible to use "state" as a name for a FSM state. The error produced is:

Expecting token of type 'ID' but found 'state'.

Functions Definition A function in KANT is declared as a name with typed parameters and typed results. Invocations must follow the same structure as the definitions.

```
function PKE_ENC(content: BitString)
    with k: PublicKey -> [pke_enc: Ciphertext]
   Alice know {
    ...
    fresh na: Nonce;
    fresh nx: Nonce;
    enc_na = PKE_ENC(na,nb) with pubb;}
```

The validator produces the following error because the declaration has only one parameter:

"PKE_ENC" requires "1" argument, but "2" arguments are provided.

A similar error would be produced if the number of keys entered does not correspond to the number of keys declared.

Unused Knowledge or Principal In order to increase the clarity of the model, we warn the user of all principal and knowledge declarations that remain unused in the writing of a protocol:

```
principal Alice, Bob, Server
```

Principal Server is declared but never used.

Check Fields User-entered checks in the protocol may only target knowledge and principals, but not functions.

```
    only Alice,Bob should know CONCAT(na,nb)
```

Knowledge check should target only knowledge references and list access.

3.2.4.2 Semantic Checks

Type Compatibility The types of parameters used in the function invocation are inferred and checked to be congruent with those of the declarations.

```
   Bob know {
    ...
    dec_na = PKE_DEC(enc_na) with pubb;
    ...}
```

Incorrect key type: "PublicKey", the invoked function requires a key of type "PrivateKey".

Knowledge Scoping The knowledge declared by a principal is only accessible by that principal or the ones it has shared it with. Furthermore, the names of new knowledge must be unique across the entire protocol.

```
   Bob -> Alice : nx
```

Principal "Bob" doesn't know "nx".

Function Inversion Only functions that are not one-way can be inverted, and a property must be specified for the inversion.

```
function PKE_ENC(content: BitString)
    with k: PublicKey
    -> [ pke_enc: Ciphertext ]one way
function PKE_DEC(pke_enc: Ciphertext)
    with k: PrivateKey
    -> [ content: BitString ]
property forall x: BitString, k: PrivateKey
| PKE_DEC(PKE_ENC(x) with PUB_GEN(k))
    with k -> [ x ]
```

PKE_ENC is a one-way function that cannot be inverted.

List Access It is possible to access a list by using only the names resulting from the application of a SPLIT. The validation of the SPLIT requires it to be used only on the results of a CONCAT:

```
Alice know {
 ...
 dec_na_nb = PKE_DEC(enc_nb_na) with priva;
 rec_na_nb = SPLIT(enc_nb_na);
 ...}
```

The "SPLIT" function is called on a parameter that is not the result of "CONCAT".

3.2.4.3 Prudent engineering practices

We have implemented additional validation rules to complement the standard semantic and syntactic checks. These new rules are based on the principles outlined in [1] and help identify and prevent common error patterns found in the literature. By catching these errors early on, users are able to correct them before the verification phase.

Same Key for Encryption and Authentication The usage of the same key for symmetric encryption and signing can allow an attacker to use the signing algorithm to decrypt messages, so it is crucial to use a different key for each method. In this example, Alice uses the same key `priva` both for decrypting the message `enc_bit` and to sign the plaintext `dec_mess`. This causes a potential vulnerability when the signature is sent out.

```
Alice know {
    fresh bit:BitString;
    fresh priva:PrivateKey;
    puba = PUB_GEN(priva);}
...
Alice know {
    dec_mess = PKE_DEC(enc_bit) with priva;
    sign_mess = SIGN(dec_mess) with priva;}
```

Hash used as Encryption A message should not contain a signed hash of a plaintext as it could have been generated by a third party and sent to another principal. An instance of the problem can be shown in the example below, which specifies the following message:

$$Alice \rightarrow Bob : \{X\}_{K_b}, \{HASH(X)\}_{K_a^{-1}}$$

where $\{X\}_{K_b}$ represents the asymmetric encryption of secret X (in plaintext) using Bob's public key K_b and $\{HASH(X)\}_{K_a^{-1}}$ represents the signature of the hash function applied to secret X using Alice's private key.

```
Alice know {
    enc_x = PKE_ENC(x) with pubb;
    hash_x = HASH(x);
    sign_hash_x = SIGN(hash_x) with priva;
    mess = CONCAT(enc_x,sign_hash_x);}

Alice -> Bob: mess
```

Encrypt then Sign The authentication pattern *encrypt then sign* is vulnerable to attacks: the attacker can remove the signature and replace it with its own, claiming ownership of the message that the recipient receives. Thus, this pattern is generally insecure and should be avoided. To ensure greater security, it would be better to use the sign-then-encrypt method and add the identity of the recipient to the signature. The example below shows a message `mess` built by using asymmetric encryption with a public key `pubb`. The resulting cyphertext `enc_x` is signed and concatenated to the encryption.

```
Alice know {
    enc_x = PKE_ENC(x) with pubb;
    sign_enc_x = SIGN(enc_x) with priva;
    mess = CONCAT(enc_x,sign_enc_x)}
Alice -> Bob: mess
```

Add Recipient Identity to Signature According to [1], it is crucial never to infer the principal's identity from the content or the sender of a message, as this can lead to *impersonation attacks*. To prevent such attacks, it is essential to mention the identity of the receiver in the message's signature. Additionally, although it can be deduced from the key used to sign the message, the identity of the sender should also be included. In the example below, Alice sends a message `enc_sign` encrypted with asymmetric encryption and containing a signature `sign_kab_ta` that does not include `Bob`'s identity.

```
Alice know {
    fresh kab:SymmetricKey;
    fresh ta:Nonce;
    kab_ta=CONCAT(kab,ta);
    sign_kab_ta = SIGN(kab_ta) with priva;
    enc_sign = PKE_ENC(sign_kab_ta) with pubb;}

Alice -> Bob: enc_sign
```

Avoid Double Encryption Double encryption, also known as *cascading encryption*, is a practice that has fallen out of favor in modern cryptographic security. While it may seem like a logical way to enhance data security, it is often counterproductive. Double encryption introduces complexity, consumes additional computational resources, and poses key management challenges. Rather than enhancing security, it can lead to marginal improvements while increasing the potential for implementation errors and vulnerabilities. The following example uses double encryption. The first symmetric encryption with key `kab` is applied on `bit`, and the result is encrypted using public encryption with the key `pubb`.

```
Alice know {
 fresh bit:BitString;
 enc1 = ENC(bit) with kab;
 enc2 = PKE_ENC(enc1) with pubb;}
```

3.2.5 KANT Effectiveness

In order to evaluate the ability of KANT language to capture the common concepts and primitives of security protocols, besides the NSPK protocol, we modelled classical security protocols such as SSL, Needham-Schroeder, Needham-Schroeder-Lowe, Woo-lam, Yahalom, BAN-Yahalom, Otway-Rees, Denning-Sacco, and other protocols are under development. KANT's models of all the case studies have been validated by using the set of validation rules and, as expected, some warnings were raised on those protocol aspects that might cause vulnerabilities – not surprisingly since the rules on prudent engineering practice (see Sect. 3.2.4.3) were defined following the guidelines suggested in [1] as measures to prevent known attacks against classical protocols.

Besides modelling constructs in common with other notations for security protocol description, KANT also has primitives to model the principal's knowledge and the FSM of its execution of a given protocol session. Such concepts are relevant for protocol verification by back-end tools, and the advantage of being already specified at the level of the KANT model facilitates the translation of these models into models suitable for verification and allows better integration of such crucial activity into the development process of robust protocols. Typically, verification is undertaken after the protocol has already been released [19, 40, 60], and the effort of making changes in the protocol is expensive. Our idea in developing KANT was to facilitate the integration of verification early in the development process. In this way, not only does one develop consistent documentation across iterations, but also avoids the introduction of vulnerabilities when adding new features or making changes to cryptographic primitives.

To assess the potentialities of KANT in modelling knowledge information flow among parties, we selected the WPA3 Simultaneous Authentication of Equals (SAE) protocol[5], a password-based authentication and password-authenticated key agreement method. The choice was due to the fact that this protocol includes in its documentation an FSM of the protocol instances, and it is one of the first protocols to expose this feature. In addition, it provides advanced security features such as Forward Secrecy, Eavesdropping, and Dictionary attack resilience.

The overall picture of the WPA3-SAE protocol execution is given in the official documentation in terms of the FSM reported in Fig.3.10. The path in green describes the transitions performed by each principal (and are captured by our model) involved in the protocol for getting authentication; the other transitions are performed by other entities described in the WPA3 documentation, and we here abstract from them.

The crucial steps of the protocol message exchanges depicted in Fig. 3.11 are mainly two: (1) the *commitment exchange*, to move from state *Committed* to state *Confirmed*, where each party commits to a single password guess; (2) the *confirmation exchange*, to move from state *Confirmed* to state *Accepted*, which validates the correctness of the guess. The state *Nothing* is the initial state where a principal is when it is created and immediately moves to state *Committed* when starts the protocol exchange.

[5]802.11-2020 - IEEE Standard for Information Technology

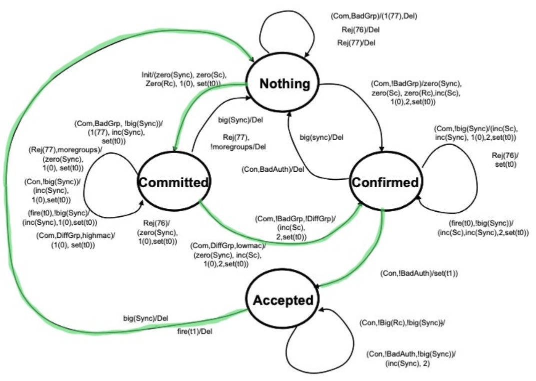

(Com,BadGrp)/(1(77),Del)
Rej(76)/Del
Rej(77)/Del
Init/(zero(Sync), zero(Sc), Zero(Rc), 1(0), set(t0))
Nothing
(Com,!BadGrp)/zero(Sync), zero(Sc), zero(Rc),inc(Sc), 1(0),2,set(t0))
(Com,BadGrp, !big(Sync))/ (1(77), inc(Sync), set(t0))
big(Sync)/Del
Rej(77), !moregroups/Del
big(sync)/Del
(Com,!big(Sync)/(inc(Sc), inc(Sync), 1(0),2,set(t0))
Rej(76)/ set(t0)
(Rej(77),moregroups)/ (zero(Sync), 1(0), set(t0))
(Con,BadAuth)/Del
Committed
Confirmed
(Con,!big(Sync))/ (inc(Sync), 1(0),set(t0))
(fire(t0),!big(Sync)/ (inc(Sync),1(0),set(t0))
(Com,DiffGrp,highmac)/ (1(0), set(t0))
Rej(76)/ (zero(Sync), 1(0),set(t0))
(Com,!BadGrp,!DiffGrp)/ (inc(Sc), 2,set(t0))
(fire(t0),!big(Sync))/ (inc(Sc),inc(Sync),2,set(t0))
(Com,DiffGrp,lowmac)/ (zero(Sync), inc(Sc), 1(0),2,set(t0))
(Con,!BadAuth)/set(t1))
Accepted
big(Sync)/Del
fire(t1)/Del
(Con,!Big(Rc),!big(Sync))/
(Con,!BadAuth,!big(Sync))/ (inc(Sync), 2)

Figure 3.10: WPA3 SAE FSM

The main rules of the protocol are the following:

- A party can commit at any point during the exchange.
- Confirmation can only be done after a party and its peer have committed.
- Authentication is only accepted when a peer has successfully confirmed.
- The protocol ends successfully when each participating party has acknowledged and accepted the authentication process.

The protocol uses modular arithmetic primitives to calculate numbers sA and sB by summing:

$$sA = (a + A) mod\, q$$

$$sB = (b + B) mod\, q$$

where a, A, b, B are random numbers, and q is the (prime) order of the group. Password Equivalent (PE) is a hashed value of the password that Bob and Alice know and is raised to the power of $-A$ for Alice and $-B$ for Bob. Once the commit messages have been exchanged, the two agents calculate the value k as:

$$k = (PE^{sB} * PE^{-B})^{a} = (PE^{sA} * PE^{-A})^{b} = PE^{ab} \tag{3.1}$$

The two principals compute KCK as the hash of the concatenation between k and the sum modulo q of sA and sB. Finally, they hashed all received

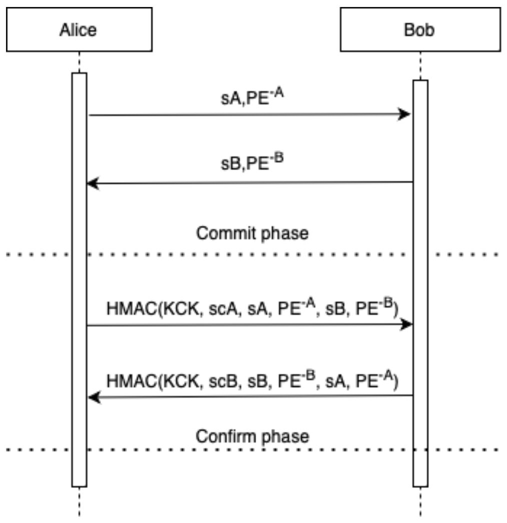

Alice
Bob
sA,PE-A
sB,PE-B
Commit phase
HMAC(KCK, scA, sA, PE-A, sB, PE-B)
HMAC(KCK, scB, sB, PE-B, sA, PE-A)
Confirm phase

Figure 3.11: WPA3 SAE key exchange

information together with *KCK* and a counter called *send-confirm* (*scA*, *scB*) to build the commit messages.

To model in KANT the above computations, we defined the following functions:

```
function SCALAR_OP(r:Number,y:Group)->[z:Group]
function ELEMENT_OP(x:Group,y:Group)->[z:Group]
function INV_OP(h:Group)->[o:Group]
function SUM_MOD(r:Number,n:Number)->[t:Number]
property forall r:Number, t:Number, p:Group |
    ELEMENT_OP(SCALAR_OP(SUM_MOD(r,t),p),
    INV_OP(SCALAR_OP(t,p)))-> SCALAR_OP(r,p)
property forall x:Group, y:Group |
    ELEMENT_OP(x,y) equals ELEMENT_OP(y,x)
property forall r:Number, t:Number |
    SUM_MOD(r,t) equals SUM_MOD(t,r)
```

In both Finite Field Cryptography (FFC) and Elliptic curve cryptography (ECC) groups, WAP3 SAE employs three arithmetic operators: the element function `ELEMENT_OP` that produces a group from two groups, the scalar function `SCALAR_OP` that generates a group from an integer and a group, and the inverse function `INV_OP` that produces a group from a group. The protocol uses modular arithmetic; we added the function `SUM_MOD` has been added for this purpose (we omit the module as an argument since it is of public domain and not relevant to the analysis). We then added various properties to address the commutativity of functions `ELEMENT_OP` and `SUM_MOD`, and a property that allows us to perform the mathematical simplification as described in the Formula 3.1.

The model of the WPA3 protocol in KANT, upon checking syntactic and semantic validation rules, is available at WPA3_SAE. This case study shows the expressive potential of the language in modelling a protocol having very high mathematical complexity. Moreover, thanks to the capability to express the state-end transition in the knowledge of a principal, the model can reflect the FSM structure in Fig. 3.10.

3.2.6 Related Work

Formal verification of security protocols has been a critical area of research and development in computer security in the last twenty years. Many techniques and tools have been proposed for verifying protocols. Depending on the specific tool, either the protocol has been translated into the tool input language or a new language has been defined to model the protocol. In most of the cases, the language was not user-friendly, requiring expertise in formal methods and protocol analysis. There are indeed some examples of domain-specific languages that inspired us when deciding which features needed to be included in the language, such as the knowledge notion, or the state. For example, in the seminal paper [36] introducing one of the first formalisms designed to reason about protocols, the authors recognise the importance of making explicit the assumptions (called principal's beliefs and assumptions) taken before the execution of the protocol, which we expressed with the principal's knowledge.

AVISPA (Automated Validation of Internet Security Protocols and Applications) [12] supports the editing of protocol specifications and allows the user to select and configure the different back-ends of the tool, similar to our long-term goal. The protocol is given in the High-Level Protocol Specification Language HLPSL [57], an expressive, modular, role-based, formal language, thus not really user-friendly; HLPSL specifications are then translated to the so-called Intermediate Format (IF), an even more mathematical-based language at an accordingly lower abstraction level and is thus more suitable for automated deduction.

Also, SAPIC+ [38] aims at exploiting the strengths of some of the tools that reached a high degree of maturity in the last decades, e.g., TAMARIN and PROVERIF, offering a protocol verification platform that permits choosing the tool. However, the input language is an applied π-calculus similar to PROVERIF, thus it requires high expertise in equational theories and rewrite systems.

In [55, 69], the authors define and give the semantics of AnB, a formal protocol description language based on the classical Alice&Bob notation we introduced in Sect. 3.2.1. However, part of the readability is lost in the translation since a mathematical notation is used.

The work most related to ours is VERIFPAL [57], which uses a user-friendly high-level language that allows users to model cryptographic protocols and security properties rather intuitively. The tool uses symbolic analysis techniques and can automatically generate formal verification code in the TAMARIN prover's input language, making it compatible with TAMARIN for further in-depth analysis and verification. However, they provide only a text editor with a syntax highlighter, whereas in our case, we do both syntax and semantic checking.

4 Unifying Verification Methodology

This chapter explores the analysis using various back-ends and proposes a unifying proof methodology to achieve a multi-level verification framework. It is structured into two main parts: In the first part, a verification approach is presented, leveraging Abstract State Machines (ASMs) and their user-friendliness to comprehensively capture the intricacies of a security protocol across multiple levels. This approach extends from the communication to the device level, offering a detailed examination of the protocol's functioning. Notably, the effectiveness of this approach is demonstrated through its application to the Z-Wave protocol, where its performance is rigorously evaluated. The second part introduces a method for integrating verification tools to enhance their capabilities and expressible details during the formalisation of a security protocol. This method harnesses the ASMETA framework and PROVERIF, combining their functionalities to achieve a more comprehensive analysis. The method's efficacy is showcased through its application to the well-known WPA3-SAE protocol, which successfully uncovers vulnerabilities and provides valuable insights into protocol security.

4.1 ASMETA Back-end

4.1.1 Overview

Connected IoT (Internet of Things) devices (e.g., alarms, door locks, lights, sensors, etc.) are extensively used in many software applications to provide users with *smart* services, which go from domotic home services (e.g., home door opening, lights and heating control) to monitoring services (e.g., gas and water monitoring) and managing services (e.g., appliances managing).

These applications are *security*- and *safety- critical*, since installing vulnerable or unsafe devices might have serious consequences in terms of user's privacy and safety [77].

Many alliances of corporations are adopting security procedures to protect the traffic generated by their protocols, and often costly patches are required to protect against attacks exploiting previously unknown vulnerabilities; however, the approaches adopted to meet security requirements have often been proven to be insufficient [81], and many connected devices do not use standardised communication protocols with a provable security guarantee.

For this reason, the *Security By Design* approach [63] is increasingly adopted as a development process where security assurance has to be guaranteed "*a priori*", already at the design phase, instead that "*a posteriori*", at operation time. The security-by-design paradigm aims at identifying, as early as possible, potential security threats and vulnerabilities and designing secure components and interfaces since the beginning of the software development life cycle. Therefore, security-by-design approaches require adopting software security assurance processes at design time, reasoning on an abstract model representation, which allows for validation and verification of the security solutions in a rigorous and provable way.

Model-based security assurance is strictly interconnected with protocol verification, which has been very active in the last thirty years. It reached a fairly mature state. However, most of the tools are not widely adopted by practitioners as one would expect, mainly due to the complexity of the notations and the lack of skills in verification techniques [42]. In order to reduce the gap between designers' backgrounds and practical usage of formal methods in security protocol design, our long-term research goal is the development of a user-friendly but rigorous design approach based on the use of formal methods and allowing security assurance at the model level. In our case, we exploit the symbolic model approach [25]. This paper is an extension of [60] and a step forward in this direction.

Starting from a preliminary work [35], where we introduced a minimal set of templates to formalise, in the Abstract State Machine formal method [30, 31], common behavioural patterns in security protocols to facilitate the protocol formal verification, here we provide a more complete set of specification primitives and develop a more concise, but complete and better scalable model of the Z-Wave protocol. We use this model to validate and verify the protocol security requirements by instantiating a set of property schemas reflecting the common confidentiality, integrity and authentication protocol properties.

Among the numerous IoT security protocols, the Z-Wave protocol [87], designed primarily for home automation, is claimed to be one of the most secure protocols for IoT device communication. It became publicly available only in 2016 (this is the reason why it has not been so widely analysed), and then it was updated with a new security layer (S2 Security), overcoming the shortcomings of the previous S0 security layer, which was found vulnerable [46]. Before the results in [60], which revealed a feasible Man-In-The-Middle (MITM) attack confirmed by Silicon Labs and Z-Wave Alliance, the protocol lacked a rigorous approach to verification. This work completes and improves the protocol analysis.

We can summarise the contribution of the paper as follows:

- We provide a *library of primitives* useful to model communication protocols: functions and domains to model protocol principals, their knowledge, protocol messages and their communication, cryptographic functions, etc.

- We define a set of *schema for temporal logic formulas* that specify common security goals of cryptographic protocols to be instantiated according to the specific protocol information to verify confidentiality, integrity and authentication properties.

- We give a rigorous description of the Z-Wave *joining procedure* using the most updated specification of the S2 Security class and exploiting the library of primitives.

- We perform *scenario-based validation* to check our model against the protocol requirements and to better understand the vulnerability trace retrieved. Validation is performed in the presence of passive and active attacker models.

- We provide an exhaustive *verification of the protocol security goals* obtained through the evaluation of confidentiality, integrity, authentication properties and other protocol-specific properties.

- The QR code-based security solution designed by Z-Alliance to prevent MITM vulnerability exploitation is modelled and analysed.

The rest of the section is organised as follows. Section 4.1.2 introduces the Z-Wave protocol. Section 4.1.3 presents, in a concise way, the theory behind the ASM formal method and the ASMETA framework with its analysis approaches. Section 4.1.4 introduces the modelling primitives for specifying the protocol and the security properties to be verified. Section 4.1.5 presents the ASM models of the protocol, their extension with the attacker models and the solution of the QR code. We report only some excerpts of the available models online[1]. Section 4.1.9 presents the validation and verification processes of the model of the Z-Wave protocol; we discuss the results in the two possible scenarios, with passive and active attackers. Section 4.1.10 discusses some threats to the validity of our approach, while Section 4.1.11 briefly reviews existing results on formal methods in the context of IoT security protocols.

4.1.2 Running case study: the Z-Wave IoT protocol

Z-Wave is a wireless radio frequency based communication protocol and uses a proprietary protocol stack [86]. A Z-Wave network is a mesh network with nodes communicating via low-energy radio waves having frequencies from 865 to 926 MHz. Nodes can be either *controllers* or *slaves*: controllers are in charge of adding and removing nodes from the network, as well as maintaining the routing table for the entire system, whereas slaves are the actual “smart" components of the network. Nodes within range of the controller can interact directly, while remote nodes must rely on intermediary nodes for retransmission. In Fig. 4.1, controller coverage is shown with a grey ellipse, and slave coverage using empty ellipses.

Since the protocol was designed primarily for home automation, slave nodes can be residential appliances or other devices, such as lighting controllers, thermostats, security systems, windows, locks and garage door openers.

[1]All models are available at: `https://github.com/mariolilliresearch/Aprover.git`.

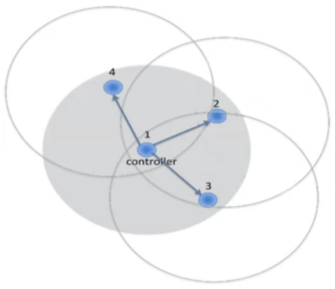


Figure 4.1: Z-wave network topology

Security Feature	Technique
Confidentiality Authentication	AES CCM mode 128 bit key
Integrity	AES CMAC mode 128 bit key
Freshness	Pre-Agreed Nonces (PAN) Multicast Pre-Agreed Nonces (MPAN)

Figure 4.2: Security features of Z-Wave protocol

In order to minimise battery usage, nodes are either active or in sleeping mode. Obviously, only active nodes can relay a message.

Since the beginning, Z-Wave was designed to guarantee secure interactions among devices, thus all messages are sent encrypted. However, [46] found a vulnerability in the Z-Wave first-generation security framework (S0 Security) that has been dismissed, although many devices still have backward compatibility. The second-generation security framework (S2 Security) was released in 2016, offering significant improvements over S0. The new security layer uses a combination of symmetric encryption and message authentication code (MAC) to guarantee integrity, confidentiality, authentication, and data freshness during communication (refer to Tab. 4.2 for the specific techniques).

In addition, the S2 security framework provides four security classes (S0, S2-AccessControl, S2-Authenticated and S2-Unauthenticated) that are used to group devices and segment the network. Door locks, garage door openers, and controllers are assigned to the S2-AccessControl class, while the majority of end devices, including window blind motors, switchers, other sensors and secondary controllers, belong to the S2-Authenticated class. In order for a new device to join the network, the controller and the new device must first agree on the subset of security classes that the controller is capable of handling. The controller then transmits a key for each security class it has been granted. In this way, a Z-Wave network is segmented by grouping nodes into security classes that communicate securely using the same AES 128-bit key. The improved security is clear: access control devices are only

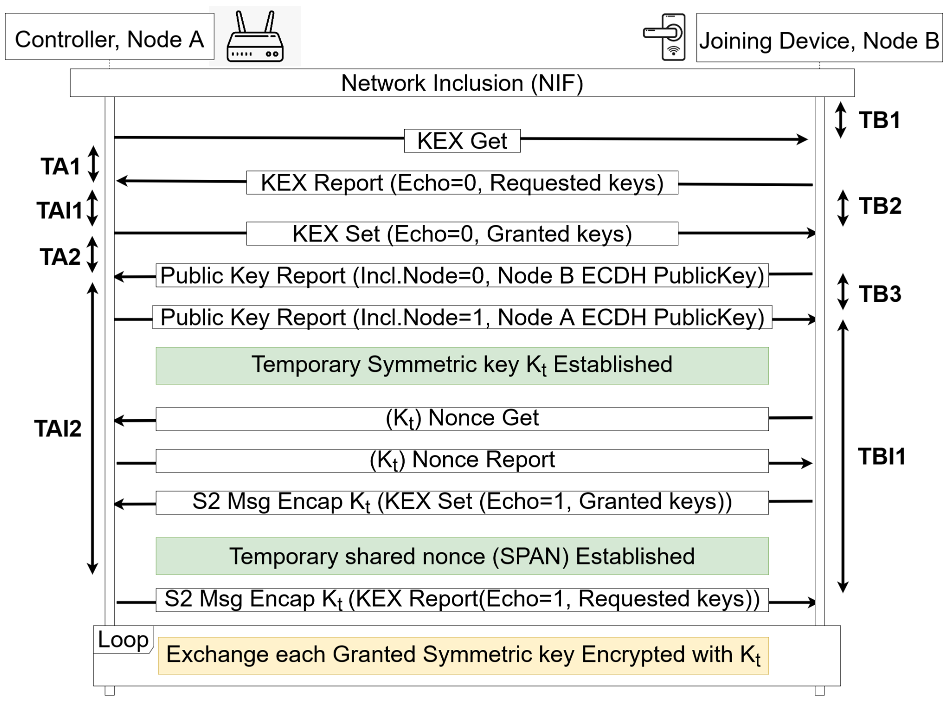

Controller, Node A
Joining Device, Node B
Network Inclusion (NIF)
TB1
KEX Get
TA1
KEX Report (Echo=0, Requested keys)
TAI1
TB2
KEX Set (Echo=0, Granted keys)
TA2
Public Key Report (Incl.Node=0, Node B ECDH PublicKey)
TB3
Public Key Report (Incl.Node=1, Node A ECDH PublicKey)
Temporary Symmetric key Kt Established
TAI2
(Kt) Nonce Get
(Kt) Nonce Report
TBI1
S2 Msg Encap Kt (KEX Set (Echo=1, Granted keys))
Temporary shared nonce (SPAN) Established
S2 Msg Encap Kt (KEX Report(Echo=1, Requested keys))
Loop
Exchange each Granted Symmetric key Encrypted with Kt

Figure 4.3: Z-Wave *joining procedure*

accessible by controllers that need to control them and not by any device in the network. Moreover, when one of the granted keys is compromised, the network is protected by the segmentation of the security class, which reduces the number of devices using the compromised key.

The *joining procedure*, occurring when a new device joins the network and negotiates the AES-128 symmetric key to be used for all communications, is the core of S2 security layer. The procedure consists of two phases (see Fig. 4.3): during the first phase, the two participants agree on a temporary AES key; in the second phase, the temporary key is used to encrypt the symmetric key granted to the new joining device. As a result, key establishment is dependent on the secrecy of the temporary key K_t: if the key is compromised, all of the symmetric keys exchanged in the second phase of the protocol are compromised. Therefore, in our formal analysis of Z-Wave protocol, we discuss and analyse only the messages exchanged between the controller *A* and the new slave device *B* during the protocol's first phase, which consists of three steps:

1. First step: the new device reveals its device type via a *Node In-*

formation Frame (NIF). Then, controller *A* and slave *B* negotiate and agree on which security classes to share, by exchanging `KEX Get`, `KEX Report`, and `KEX Set` messages.

2. Second step: the two participants exchange their public keys to generate the temporary encryption key K_t by using Elliptic Curve Diffie-Hellman (ECDH) on Curve25519 with a 256-bit public key length.

   ECDH is a key agreement mechanism that allows two parties, *A* and *B*, owning an elliptic-curve public-private key pair, to establish a shared secret over an unsecured channel. *A* computes the shared secret using her/his private key and *B*'s public key, whereas *B* computes the shared secret using *B*'s private key and *A*'s public key. Unless an attacker can solve the elliptic curve discrete logarithm problem, only *A* or *B* are able to compute the shared secret. Since ECDH is vulnerable to man-in-the-middle attacks, Z-Wave Alliance developed [88] two Out-Of-Band (OOB) authentication methods to prevent eavesdropping and MITM attack vectors. To continue the *joining procedure*, the user can choose one of two options:

   i) Enter as a PIN the first five digits of the Device Specific Key (DSK) printed on the device and obfuscated during the RF transmission, and visually validate the bytes 3 to 16 of *B*'s DSK; ii) Scan a QR code to verify the full DSK (p. 93, requirement CC:009..01.00.11.05F in [88])[2].

3. Third step: *A* and *B* perform a challenge-response protocol based on a nonce that enables them to verify that they are both able to encrypt using K_t. In order to prevent message manipulation, they send `KEX Get` and `KEX Report` messages again.

The protocol then enters a loop in which an AES key is assigned to each granted class. It should be noticed that several protocol sub-phases are timed. Thus, if a message is delayed, the slave or controller will exit the *joining procedure*. The vertical double-edge arrows in Fig. 4.3 show what messages must arrive before a specific timer expires; the timer name is in-

[2]In the last release of the protocol, Z-Wave Alliance updated the authentication process of the *joining procedure* by selecting the QR code as default user's choice, while previously the PIN was the default one and it is still supported for retro-compatibility. Sect. 4.1.8 provides more details.

dicated as a label of the arrow (e.g., timer `TA1` requires that message `KEX Report` takes no more than 10 seconds to arrive).

Consider as an example a smart thermostat with native S2 Security trying to join the network. In a typical protocol execution, the thermostat (node *B*) requests the symmetric keys associated to three security classes (S2 Authenticated, S2 Unauthenticated, and S0). In order to increase network security, if the network consists exclusively of native S2 Security devices, the user may disable class S0 Security in the controller (node *A*). In case the `KEX Report` message delivered from *B* to *A* has been tampered with, the controller gets only requests with the S0 Security class. Therefore, the intersection between the requested and granted security classes is empty, and the error message `KEX FAIL KEX KEY` is generated. This behaviour complies with a security requirement given in the protocol specification.

4.1.3 Abstract State Machines in a Nutshell

Abstract State Machines (ASMs) [31, 30] is the formalism that we use for modelling the Z-Wave protocol.

ASMs are a state-based formal method that extends Finite State Machines (FSMs) by replacing unstructured control *states* with algebraic structures (i.e., domains of objects with functions defined on them). State transitions are performed by firing *transition rules*. At each computation step, all transition rules are executed in parallel by leading to simultaneous (consistent) updates of a number of locations – i.e., memory units defined as pairs (*function-name*, *list-of-parameter-values*)–, and therefore changing functions interpretation from one state to the next one. Location *updates* are given as assignments of the form $loc := v$, where *loc* is a location and v its new value. Among other *rule constructors*, those used for our purposes are constructors for guarded updates (`if-then`, `switch-case`), parallel updates (`par`),

nondeterministic updates (`choose`).

Functions which are not updated by rule transitions are *static*. Those updated are *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine).

An ASM model has a predefined structure consisting of: a *signature*, which contains declarations of domains and functions; a block of *definitions* of static domains and functions, transition rules, state invariants and properties to verify; a *main rule*, which is the starting point of a machine computation; a set of *initial states*, one of which is elected as *default* and defines an initial value for the (controlled) machine locations.

An ASM *computation* (or *run*) is defined as a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing the unique *main rule* which in turn could fire other transitions rules.

ASMs allow modelling different computational paradigms from a *single* agent executing an ASM, to distributed *multiple* agents, which is the computational paradigm we used in our Z-Wave model. A *multi-agent ASM* is a family of pairs $(a, ASM(a))$, where each a of a predefined set *Agent* executes its own machine $ASM(a)$ (specifying the agent's behaviour), and contributes to determining the next state by interacting synchronously or asynchronously with the other agents. A predefined function *program* on *Agent* is used to associate the ASM with an agent.

Since agents of the same kind (e.g., agents representing Z-Wave protocol slaves) have the same behaviour, within transition rules, each agent can identify itself by means of a special 0-ary function $self : Agent$ which is interpreted by each agent a as itself.

Code 4.1 shows an excerpt of the *multi-agent ASM* model for the Z-Wave joining procedure between two nodes (A and B), working as controller and slave, respectively. `r_controllerRule[]` on line 25 is the ASM program associated to an agent of type *Controller*, while `r_slaveRule[]` on line 27 is that of an agent of type *Slave*; `nodeA` and `nodeB` are instantiated as the corresponding type of agents.

An ASM agent can behave according to a *control-state ASM* [31]: transition rules are guarded by a *mode* function (as, for example, a function *state*), which is updated in the rule body and whose values resemble the states of a

```
asm ZWave_join_MITM

signature:
 domain Slave subsetof Agent
 domain Controller subsetof Agent
 domain Intruder subsetof Agent
 .....
 static nodeA: Controller
 static nodeB: Slave
 static nodeE: Intruder
 ....
definitions:
 main rule r_Main =
   par
     program(nodeA)
     program(nodeB)
     program(nodeE)
   endpar
default init s0:
 function slaveState (a in Agent) = if (a = nodeB) then INIT_SLV endif
 function controllerState (c in Agent) = if (c = nodeA)
                                          then INIT_CTRL endif
   ...
 agent Controller:
   r_controllerRule[]
 agent Slave:
   r_slaveRule[]
 agent Intruder:
   r_mitmRule[]
```

Code 4.1: Excerpt of the ASM model of the Z-Wave protocol

Finite State Machine. This machine model has been used for specifying the agent's behaviour (i.e., the actor's actions).

Tools and Validation & Verification Techniques The ASM formal method is supported by the tool-set ASMETA (ASM mETAmodeling) [8] for model editing (with ASMETAL notation), validation and verification. Model construction, especially when taking the system requirements from natural language, can often be error-prone. For this reason, it is essential to be able to *validate a model* against its functional and non-functional requirements. ASM models can be validated in terms of model simulation (by using ASMETAS), animation (by ASMETAA), and scenarios execution (by ASMETAV). In the latter case, each scenario contains a description of the expected system behaviour and the tool checks whether the machine runs correctly. It is also possible to *verify properties* expressed in temporal logic by means of model checking (with ASMETASMV, which maps ASMETAL models to the model checker NUSMV): the tool will check if the property holds during all possible model executions.

Remark. ASMs offer several advantages as formal methods: (1) due to their *pseudo-code format*, they can be easily understood by practitioners and can be used for *high-level programming*; (2) they allow for system specification at any desired *level of abstraction*; (3) they are *executable models*, so they are also suitable for lighter forms of model analysis, such as simple simulation to check model consistency w.r.t. system requirements; (4) they support techniques for mapping models to code (e.g., to C++ [29], or Java [7]); (5) they can be used for *modeling distributed systems*, such as IoT networks; (6) the ASMETA framework allows for an integrated usage of tools for different forms of model analysis; it is well-maintained and under continuous feature improvement.

4.1.4 An ASM-based formal framework for security protocol analysis

This section introduces the fundamentals for formalising the protocol (Sect. 4.1.4.1) and the security properties to be verified (Sect. 4.1.4.2).

4.1.4.1 ASMs Modeling of Cryptographic Protocols

Security protocols share a common structure: they consist of a sequence of encrypted messages (or messages containing encrypted parts) exchanged between two or more principals (or actors) to establish a secure communication.

To facilitate ASM modelling of security protocols, starting from the work in [35], we defined a set of *primitives* corresponding to the typical parts of a protocol in terms of ASM functions and domains in the ASMETAL notation. They are contained in a predefined library called `CryptoLibrary` that can be imported into any ASM model of a specific protocol and contains a symbolic notation to specify actors, the protocol messages and their exchange, and the main cryptographic primitives. In the following, `CryptoLibrary` is described by using snippets of code derived from the Z-Wave case study.

Actors or Principals. The library supports four main types of actors, depending on the role of the actor in the protocol: `Initiator`, the actor starting the protocol; `Receiver`, the other actor of the protocol that replies to the messages received from the initiator; `Intruder`, a principal working as a potential attacker; `Server`, a trusted actor often used with the role of key distributor. They are all ASM agents.

The following excerpt defines the Z-Wave three types of agents and their corresponding nodes. Since no `Server` is used in the running example, we omit it.

```
domain Initiator subsetof Agent
domain Receiver subsetof Agent
domain Intruder subsetof Agent

static nodeA: Initiator
static nodeB: Receiver
static nodeE: Intruder
```

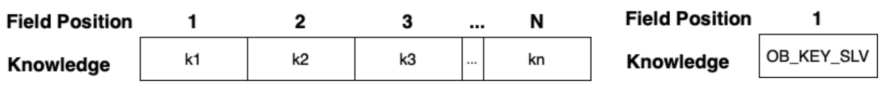
The intruder role is introduced in order to verify the correctness of the protocol in case of attacks. It is a *malicious external actor* controlling the traffic (in accordance with Dolev-Yao model [43]), which can operate in *passive* mode, i.e., as an eavesdropper, or in *active* mode. Also, a legitimate actor of the protocol can have malicious behaviour. This operation mode is not used in the verification of Z-Wave, but examples of use can be found in [35].

Messages. Each protocol message is uniquely identified by a label. Identifiers are contained in the domain `Message` and are protocol-specific (e.g., for the Z-Wave protocol, `Message` contains labels `KEX_GET`, `KEX_REPORT`, `KEX_SET`, etc., identifying the first messages exchanged).

Communication is handled using the function `protocolMessage` that associates the label of the message to the pair of principals (*sender*, *receiver*) exchanging it[3].

```
enum domain Message ={KEX_GET | KEX_REPORT | KEX_SET | ... }
controlled protocolMessage: Prod(Agent,Agent)–> Message
```

[3]`Prod` is the predefined operator for the Cartesian product of sets.



(a) Field structure

(b) Field structure of the message `PUB_KEY_REP_JOIN`

Figure 4.4: Message structure

Knowledge. The message payload is modelled by reflecting the structure depicted in Fig. 4.4a, namely a sequence of “pieces" of information known by actors, called *knowledge*, each contained into a field identified by an integer specifying its position within the message.[4] Fig. 4.4b shows an example of the message structure instantiation of `PUB_KEY_REP_JOIN`. The payload of the message contains a single field with the obfuscated public key of the slave.

The macro domain `Knowledge` is partitioned into sub-domains to group the knowledge depending on the type of information. For example, the domain `KnowledgeAsymPrivKey` identifies private keys and `KnowledgeBitstring` generic binary data. Depending on the protocol, the number of `Knowledge`’s sub-domains can be easily extended to describe more complex case studies. For example, a protocol using the signature would need a sub-domain `KnowledgePubSignKey` of `Knowledge` to store the signature public-keys.

The function `messageField` specifies the content of a specific field of the message. The domain `FieldPosition` is a set of integers used to enumerate the position of the content in a message. The listing below reports part of the declaration of domains and functions used for modelling the Z-Wave protocol. The naming convention used for expressing asymmetric key pairs is `KPUB_ACTOR` and `KPRIV_ACTOR`, where `ACTOR` should be replaced by the actor’s name.

```
enum domain Knowledge = {CSA_0| CSA_1 | SKEX_0 | SKEX_1 |ECDH_0| ECDH_1
        | ACCESS_S2_0 | ACCESS_S2_1 |AUTH_S2_0 |AUTH_S2_1| UNAUTH_S2_0
        | UNAUTH_S2_1 | S0_0 | S0_1
    //Asymmetric Public Key
```

[4]The message structure presented here improves that in [35] to better scale with protocols complexity.

```
| KPUB_SLV | KPUB_CTRL | KPUB_MITM_CTRL | KPUB_MITM_SLV
    | OB_KEY_MITM_CTRL | OB_KEY_MITM_SLV | OB_KEY_SLV | OB_KEY_CTRL
    | KPUB_ERR
//Asymmetric Private Key
| KPRIV_CTRL | KPRIV_SLV | KPRIV_MITM_CTRL | KPRIV_MITM_SLV
//Symmetric Public Key
| KT1 | KT2 | KT3 | KT_ERROR
//Pin used to complete obfuscated keys
| PIN_OK | PIN_ERROR}

domain KnowledgeBitString subsetof Any
domain KnowledgeSymKey subsetof Any
domain KnowledgeAsymPrivKey subsetof Any
domain KnowledgeAsymPubKey subsetof Any
domain FieldPosition subsetof Integer
controlled messageField: Prod(Agent,Agent,FieldPosition,Message)−>Knowledge
```

In general, protocol's actors have some knowledge before starting the protocol run (e.g., the public key of the receiver, a long-term shared key, a nonce, etc.). Then, they increase their knowledge during the execution of the protocol, by means of the exchanged messages. Agent's dynamic knowledge is modeled by controlled functions that store the information known by an agent, grouped by knowledge type.

```
controlled knowsBitString:Prod(Agent,KnowledgeBitString)−>Boolean
controlled knowsSymKey:Prod(Agent,KnowledgeSymKey)−>Boolean
controlled knowsAsymPubKey:Prod(Agent,KnowledgeAsymPubKey)−>Boolean
controlled knowsAsymPrivKey:Prod(Agent,KnowledgeAsymPrivKey)−>Boolean
```

Symmetric encryption primitives. `CryptoLibrary` contains encryption and decryption built-in primitives working with symmetric and asymmetric keys. Here, we present only the ones based on symmetric keys since they are relevant for the Z-Wave case study. The asymmetric ones behave similarly.

Encryption primitives can be applied to the full message payload or part of it. They can also be nested, i.e., applied to already encrypted data. To model these two features, two domains are defined: `Level`, an integer value specifying the nesting level (1 meaning no nesting), and `EncField1` and `EncField2` that specify the indexes delimiting the contiguous fields to which encryption is applied.

The function `symEnc` stores the *encryption key* used to encrypt (part of) a message, the nesting `Level` and the indexes of the encrypted fields. The decryption function `symDec` works similarly. It only adds

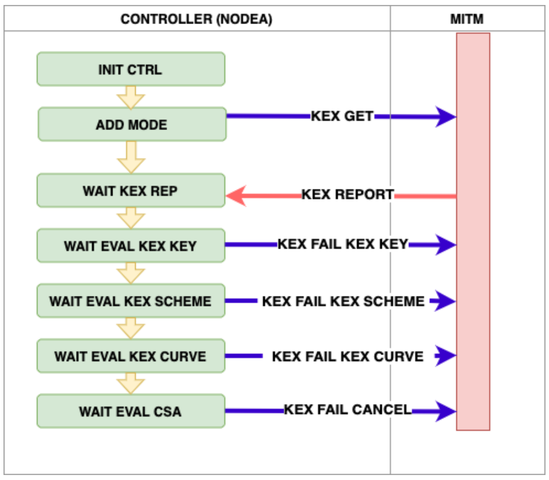


Figure 4.5: Example of the internal state of the controller

the `Agent` parameter to check if the agent performing the decryption knows the correct key, i.e., the key used to encrypt the message[5].

```
domain Level subsetof Integer
domain EncField1 subsetof Integer
domain EncField2 subsetof Integer

controlled symEnc: Prod(Message,Level,EncField1,EncField2)-> KnowledgeSymKey
static symDec: Prod(Message,Level,EncField1,EncField2,Agent)-> Boolean

function symDec($m in Message,$l in Level,$f1 in EncField1,$f2 in EncField2,$d in Agent)=
   if (knowsSymKey($d,symEnc($m,$l,$f1,$f2))=true) then
      true
   else
      false
   endif
```

Actor's Actions. In a communication protocol, principals exchange messages iteratively according to the definition of the protocol. We model honest actors' behaviour as ASM control-state machines: each time an agent receives a new message, the content/knowledge in the message triggers an internal state change and possibly, if foreseen by the protocol, the sending of a new message. Fig. 4.5 shows the state machine for the Z-Wave controller. Blue horizontal arrows pointing to the right denote a message sent by the controller (acting as an honest participant). The red arrow pointing to the left represents a response by the intruder. The labels on top identify the

[5]Note that in the `AsmetaL` notation, a variable x is denoted by `$x`.

names of the messages. Yellow vertical arrows model the internal state change. `Initiator`'s states are defined in the `StateInit` domain, `Receiver`'s states in the `StateRec` domain. The functions `initiatorState` and `receiverState` store the current state of the initiator and of the receiver, respectively.

For readability, in the Z-Wave model, since the controller is the protocol initiator and the slave works as the receiver, we renamed all domains and functions referring to `initiator/receiver` in terms of `controller/slave` to better customize the model signature.

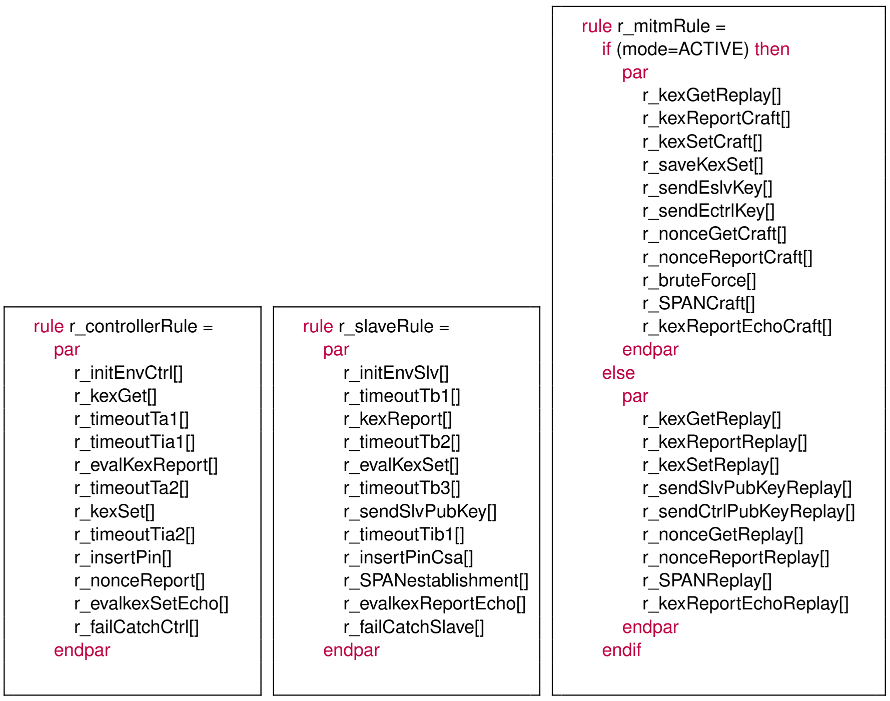
```
enum domain StateInit = {INIT_CTRL | ADD_MODE  | WAIT_EVAL_CSA
    | WAIT_EVAL_KEX_CURVE | WAIT_EVAL_KEX_SCHEME | WAIT_EVAL_KEX_KEY
    | WAIT_EVAL_USER_KEY |  WAIT_NONCE | INSERT_PIN | WAIT_PIN_OR_KEY
    | WAIT_ECDH_PUB_JOIN | WAIT_KEX_REP | WAIT_SEI_KEX_SET_ECHO
    | WAIT_CTRL_AES| OK_C | ERROR_C | TIMEOUT_C}

enum domain StateRec = {INIT_SLV | LEARN_MODE | WAIT_KEX_SET
    | WAIT_EVAL_SET_KEX_KEY | WAIT_EVAL_SET_KEX_SCHEME
    | WAIT_EVAL_SET_KEX_CURVE | WAIT_EVAL_SET_CSA | WAIT_ECDH_PUB_CTRL
    | INSERT_PIN_CSA| WAIT_NONCE_REP_REI | WAIT_KEX_REPORT_ECHO
    | OK_S | ERROR_S | TIMEOUT_S}

controlled controllerState: Initiator –> StateInit
controlled slaveState: Receiver –> StateRec
```

The program of an honest agent's consists of a number of protocol-dependent rules, each modeling the agent's internal computation upon receiving a message, its message sending and state change. The intruder's program has two different set of rules, depending on his/her operation mode (i.e., active or passive) and on the messages he/she intercepts.

Agents communicate with each other through messages. During a protocol run between two honest principals, the communication is direct. In case of an untrusted channel, the communication is intercepted by the intruder that can only understand the information that he/she can decrypt using the knowledge in his/her possession. Therefore, the communication between the two honest actors can be thought as the communication between `Initiator` and `Intruder`, and between `Intruder` and `Receiver`.



```
rule r_controllerRule =
   par
      r_initEnvCtrl[]
      r_kexGet[]
      r_timeoutTa1[]
      r_timeoutTia1[]
      r_evalKexReport[]
      r_timeoutTa2[]
      r_kexSet[]
      r_timeoutTia2[]
      r_insertPin[]
      r_nonceReport[]
      r_evalkexSetEcho[]
      r_failCatchCtrl[]
   endpar
```

```
rule r_slaveRule =
   par
      r_initEnvSlv[]
      r_timeoutTb1[]
      r_kexReport[]
      r_timeoutTb2[]
      r_evalKexSet[]
      r_timeoutTb3[]
      r_sendSlvPubKey[]
      r_timeoutTib1[]
      r_insertPinCsa[]
      r_SPANestablishment[]
      r_evalkexReportEcho[]
      r_failCatchSlave[]
   endpar
```

```
rule r_mitmRule =
   if (mode=ACTIVE) then
      par
         r_kexGetReplay[]
         r_kexReportCraft[]
         r_kexSetCraft[]
         r_saveKexSet[]
         r_sendEslvKey[]
         r_sendEctrlKey[]
         r_nonceGetCraft[]
         r_nonceReportCraft[]
         r_bruteForce[]
         r_SPANCraft[]
         r_kexReportEchoCraft[]
      endpar
   else
      par
         r_kexGetReplay[]
         r_kexReportReplay[]
         r_kexSetReplay[]
         r_sendSlvPubKeyReplay[]
         r_sendCtrlPubKeyReplay[]
         r_nonceGetReplay[]
         r_nonceReportReplay[]
         r_SPANReplay[]
         r_kexReportEchoReplay[]
      endpar
   endif
```

Figure 4.6: ASM models of actors' programs

Fig. 4.6 shows the program of the Z-Wave agents modelling the controller, the slave and the intruder.

Diffie-Hellman key exchange primitives. `CryptoLibrary` also provides a number of security primitives that are often exploited in security protocols.

The listing in Code 4.2 reports, in a symbolic way, the Diffie Hellman key exchange procedure specification. The function `diffieHellman` captures the high-level concept behind the key exchange. A principal who has received an asymmetric public key combines it with his/her asymmetric private key to obtain a symmetric key, which becomes the shared key between the two principals participating in the exchange. Also, an intruder has access to this primitive and may use it for malicious actions. Indeed, the capability of the intruder to generate a shared key allows testing whether or not a protocol is vulnerable to the known Diffie Hellman key exchange weakness (e.g., MITM attack).

```
static diffieHellman:Prod(KnowledgeAsymPubKey,KnowledgeAsymPrivKey)−>KnowledgeSymKey

function diffieHellman($pub in KnowledgeAsymPubKey,$priv in KnowledgeAsymPrivKey)=
 if (($pub = KPUB_MITM_SLV and $priv = KPRIV_CTRL)
 or ($pub = KPUB_CTRL and $priv = KPRIV_MITM_SLV)) then
   KT1
  else
   if (($pub = KPUB_SLV and $priv = KPRIV_MITM_CTRL)
   or ($pub = KPUB_MITM_CTRL and $priv = KPRIV_SLV)) then
    KT2
   else
    if (($pub = KPUB_SLV and $priv = KPRIV_CTRL)
    or ($pub = KPUB_CTRL and $priv = KPRIV_SLV)) then
     KT3
    else
     KT_ERROR
    endif
   endif
  endif
```

Code 4.2: Specification of Diffie-Hellman key exchange procedure

The `diffieHellman` specification reported below is customized for the type of keys exchanged by the principals during the Z-Wave protocol execution.

4.1.4.2 Modeling of security properties

In order to verify the protocol's security goals by means of the `NuSMV` model checker, we defined the schema of the CTL formulas corresponding to the verification of confidentiality, integrity and authentication security properties. They are parameterised with respect to the knowledge function (e.g., `knowsSymKey`, `knowsAsymPrivKey`, etc.) denoted with f_K, and `x`, the piece of information that must be secured. Parameters need to be replaced with the specific values of the protocol under verification.

Confidentiality. In security protocols, encryption is used to guarantee data confidentiality: only actors with correct decryption keys are able to access encrypted data.

This property can be modelled by the following CTL formula:

$$\neg EF(\ \mathtt{f_k(Intruder,\ x)})$$

expressing the condition that *confidentiality* of `x` is assured if *there is not a state in the future in which* `Intruder` *knows* `x`.

Field Integrity. This property requires that a piece of information exchanged between the legitimate actors is not altered by an intruder during a protocol session, i.e., the received payload must correspond to the sent one.

Let `m` be a message. Field *integrity of* `m` is assured by proving that *there is not a state in the future in which an* `Intruder` *altered a field in position* `n` *of message* `m`:

$$\neg EF(\ \texttt{messageField(Initiator,Intruder,n,m)}$$

$$\texttt{!=messageField(Intruder,Receiver,n,m)})$$

Entity Authentication. A principal proves his/her identity by demonstrating to know a secret (not necessarily shared) without explicitly revealing it.

Let `x` be a secret that must be shared by the two actors at the end of the protocol execution. Authentication of entities is assured by proving that *if there is a state in the future in which* `Initiator` *and* `Receiver` *know* `x`, *then* `Intruder` *will never know* `x`.

$$EF(\ \texttt{f}_\texttt{k}\texttt{(Receiver,x) and f}_\texttt{k}\texttt{(Initiator,x)}) \rightarrow$$

$$AG(\ \neg\texttt{(f}_\texttt{k}\texttt{(Intruder,x))})$$

Similar patterns can be used to guarantee further properties. For example, it is possible to express message authentication (also called data origin authentication and not used in our case study) as follows.

Data Origin Authentication. This property specifies that a message has not been modified while in transit and that the receiving party can verify the source of the message. This is commonly obtained by means of the usage of cryptographic primitives (either MAC or signature), therefore it is enforced when the message (or part of it) has been encrypted/signed with a key that is known only by the originator, the only one able to modify it.

Let x be a key known by an honest actor at a given moment of the protocol execution. Authentication of data x is assured by proving that *if there is a state in the future in which* Initiator *knows* x*, then* Intruder *will never know* x.

$$EF(\ \mathtt{f_k(Initiator,x)}) \rightarrow AG(\ \mathtt{\neg(f_k(Intruder,x))})$$

4.1.5 Z-Wave Formal Model

In the following, we exploit the CryptoLibrary described in the previous section to give the full ASM specification of the Z-Wave *joining procedure* that takes place when a new slave device asks the controller to join the network. Verification and validation of the models are described in depth in Section 4.1.9.

In particular, we present:

1. A first model with a *passive attacker* that can read all the messages exchanged between the controller and the slave (wireless protocols are always exposed to attacks since they use a shared channel to communicate).

2. A second model with an *active attacker* that can intercept, delete, modify and inject any message. In this model, the verification process highlighted a previously unknown feasible MITM attack confirmed by Silicon Labs and Z-Wave Alliance, where the attacker can obtain the temporary AES key (and consequently the keys granted to a new device during the joining phase).

3. A third model that implements the solution proposed by the Z-Wave Alliance to fix the vulnerability.

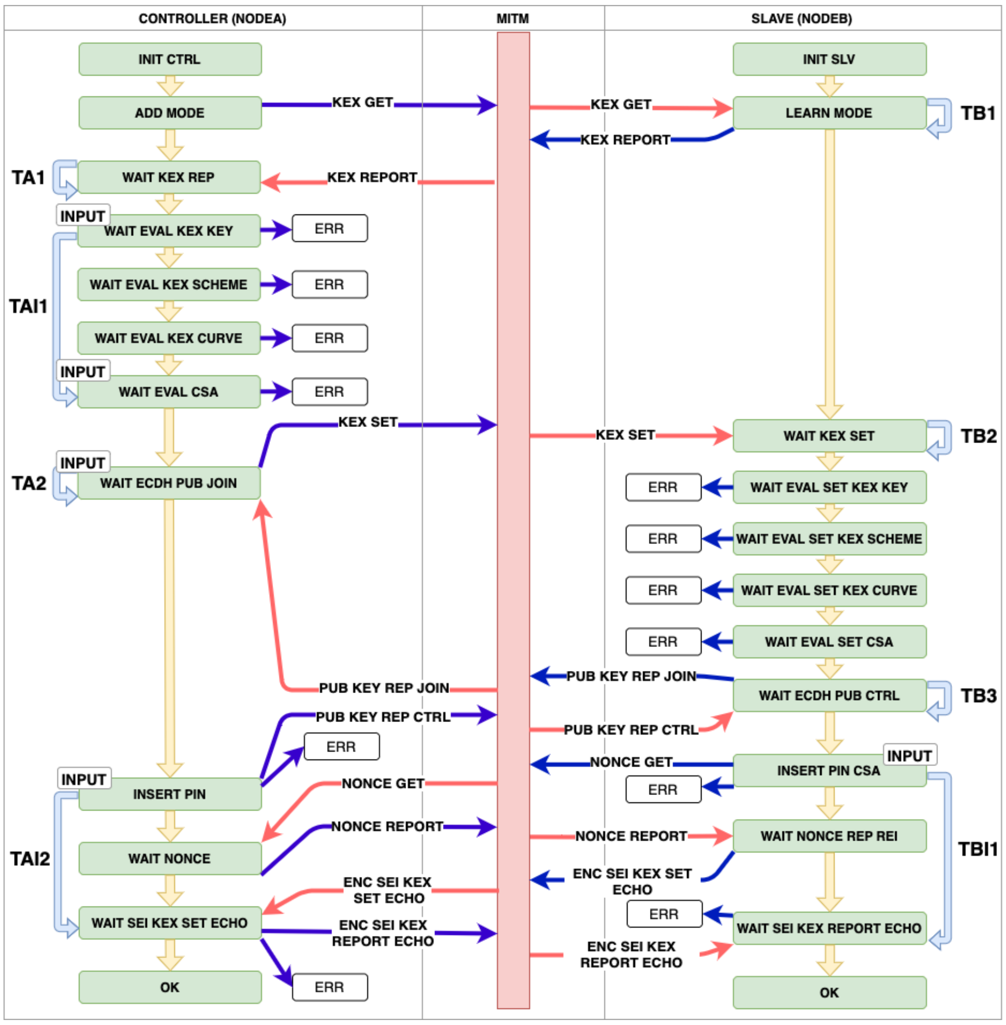


Figure 4.7: ASM model of Z-wave *joining procedure*

4.1.6 ASM model of the joining procedure with a passive attacker

Recall from an excerpt of Code 4.1 that the controller, the slave and the intruder are modelled as three ASM agents with their programs running in parallel. The intruder, running in passive mode, intercepts messages, reads them and forwards them to the intended recipient.

In Fig. 4.7, the sequence diagram of the corresponding ASM model is depicted. There are three swim lanes, one for node *A*, the controller, one for node *B*, the slave, and the central one for the intruder. The internal states of a node are represented by the rectangles coloured in light green. The flow of an agent's behaviour is vertical, whereas the horizontal flow describes the messages exchanged between principals. Blue horizontal arrows pointing to the intruder are the messages sent by honest participants and intercepted by the intruder, while the red ones sent either to the slave or the controller are the intercepted messages forwarded by the intruder to the intended receiver.

The mapping from Z-Wave protocol description to ASMs is rather straightforward: for each message exchange of Fig. 4.3, there is a corresponding *state* in Fig. 4.7, and a *transition rule* is defined to manage the state transition. The INPUT box occurring on some states, e.g., on *INSERT PIN* state, models the request to the external user owning the device to perform an action necessary to proceed to the next state. Mimicking the protocol requirements, every time a control fails (e.g., in case there is a security class mismatch between the requested classes and the granted classes or the wrong ECDH curve is asked), the node sends an error message to the other node and terminates its execution.

The domain `SlaveType` and a monitored function `slave:Slave` $\rightarrow$ `SlaveType` are defined to model the different types of slave devices and the security class a device belongs to. The invocation of the `slave` function during the agent initialisation phase to set up the device type corresponds to the sending of the NIF sent by the slave to the controller before the *joining procedure* starts.

Both in the controller and slave models, most of the transition rules have the same structure: activation is demanded by a rule guard that controls the current state, the message received, and the time left. The core of a rule consists of function updates changing the agent's internal state, setting a timer, and crafting the next messages expected by the Z-Wave protocol, using the primitives described in the `CryptoLibrary`. As an example, consider the `kexReport` rule of the slave program reported in Code 4.3:

- Line 2 sets the intruder as the receiver of the agent communication (`let ($ectrl=nodeE)`). The intruder has a rule that updates his/her knowledge and forwards the message to the intended receiver.

- Lines 3-4 specify the guard, which checks: (i) the internal state of the slave (in this case, the slave must be in `LEARN_MODE` state), (ii) the message received (in this case the message must be part of the negotiation part of the protocol, i.e., `KEX_GET`), and (iii) if timer `TB1` has not expired.
- On Lines 6-8, the slave: (i) updates its internal state to `WAIT_KEX_SET`, (ii) activates timer `TB2`, and (iii) deactivates timer `TB1`.
- On Lines 9-25, the slave sends the `KEX_REPORT` message to the intruder impersonating the controller (line 9) (that will forward it to the legitimate controller by executing its program), and with the expected payload (lines 12-25) built by using the slave knowledge.

Most of the transition rules have the same structure, others are not guarded by a timer, or they execute only controls over the payload of an already received message.

4.1.7 ASM model of the joining procedure with an active attacker

The major issue when modeling an active attacker to possibly get a vulnerability scenario is the choice of which capabilities to assign to the attacker. This task highly depends on the specific protocol – both on the security goals and on the cryptographic primitives used –, and on the use-case scenario. In addition, modeling the most powerful adversary from the very start might downgrade the performance of the verification phase and make the interpretation of results difficult. It is preferable to proceed in an incremental way, by subsequently refining the program of the intruder with new rules expressing the added capabilities.

For the Z-Wave case study, we proceeded in an incremental way defining three different models of the attacker. We started by assessing the possible vulnerabilities of the cryptographic primitives used. Z-Wave uses the Elliptic Curve Diffie-Hellman (ECDH) agreement mechanism to generate a temporary key needed for the encryption of long-term keys. ECDH guarantees confidentiality of the shared key, but it lacks authentication. Thus, it

```
rule r_kexReport =
let ($ectrl=nodeE) in
  if (slaveState(self) = LEARN_MODE  and protocolMessage(  $ectrl , self ) = KEX_GET ) then
    if (not(passed(TB1))) then
      par
          slaveState(self):= WAIT_KEX_SET
          startTimer(TB2):= true
          startTimer(TB1):= false
          protocolMessage(self, $ectrl):= KEX_REP
          if (knowsBitString(self,CSA)) then
            messageField(self, $ectrl,1,KEX_REP):= CSA_1
          else
            messageField(self, $ectrl,1,KEX_REP):= CSA_0
          endif
          ...
          if (knowsBitString(self,UNAUTH_S2)) then
            messageField(self, $ectrl,6,KEX_REP):= UNAUTH_S2_1
          else
            messageField(self, $ectrl,6,KEX_REP):= UNAUTH_S2_0
          endif
          if (knowsBitString(self,S0)) then
            messageField(self, $ectrl,7,KEX_REP):= S0_1
          else
             messageField(self, $ectrl,7,KEX_REP):= S0_0
          endif
    endpar
   endif
  endif
endlet
```

Code 4.3: kexReport rule

is vulnerable to an impersonation attack. To patch this weakness, Z-Wave Alliance added an Out-Of-Band (OOB) authentication method to prevent eavesdropping.

This analysis led us towards the modeling of active attackers with impersonation capabilities. In particular, we defined:

1. A first model, where the attacker can generate a pair of public and private keys to share with the slave (see Code 4.4). The model does not take into account the OOB procedure.

2. A second model adding to the attacker the capability to craft the PIN code usually generated in the OOB procedure. The PIN is obtained by means of a brute force attack implemented in the `r_bruteForce` rule (see Code 4.5 for the updated program).

3. A third model, where the attacker is also able to craft a key using the PIN just obtained, and then sends to the controller the crafted key containing the correct PIN. This new capability is implemented by the `r_sendEslvKey` rule (see Code 4.6). The model is depicted in the sequence diagram of Fig. 4.8, where the yellow *B* flag highlights the brute force attack obtaining the PIN, and the yellow *C* flag identifies the attack on the public key generated by knowing the PIN.

```
rule r_mitmRule =
 if (mode=ACTIVE) then
  par
   r_kexGetReplay[]
   r_kexReportCraft[]
   r_kexSetCraft[]
   r_saveKexSet[]
   r_sendSlvPubKeyReplay[]
   r_sendEctrlKey[]
   r_nonceGetCraft[]
   r_nonceReportCraft[]

   r_SPANReplay[]
   r_kexReportEchoCraft[]
  endpar
 else
 ...
 endif
```

Code 4.4: First Model

```
rule r_mitmRule =
 if (mode=ACTIVE) then
  par
   r_kexGetReplay[]
   r_kexReportCraft[]
   r_kexSetCraft[]
   r_saveKexSet[]
   r_sendSlvPubKeyReplay[]
   r_sendEctrlKey[]
   r_nonceGetCraft[]
   r_nonceReportCraft[]
   r_bruteForce[]
   r_SPANReplay[]
   r_kexReportEchoCraft[]
  endpar
 else
 ...
 endif
```

Code 4.5: Second Model

```
rule r_mitmRule =
 if (mode=ACTIVE) then
  par
   r_kexGetReplay[]
   r_kexReportCraft[]
   r_kexSetCraft[]
   r_saveKexSet[]
   r_sendEslvKey[]
   r_sendEctrlKey[]
   r_nonceGetCraft[]
   r_nonceReportCraft[]
   r_bruteForce[]
   r_SPANCraft[]
   r_kexReportEchoCraft[]
  endpar
 else
 ...
 endif
```

Code 4.6: Third Model

4.1.8 ASM model of the joining procedure with QR code

In order to remove the vulnerability found, Z-Wave Alliance extended the authentication process of the *joining procedure* with a QR code scanning. The use of the PIN is still supported for retro-compatibility. However, new devices are shipped with a QR code containing the device's public key. The *joining procedure* remains unchanged, but instead of inserting the first five digits of the joining device key, the user must scan the QR code on the label applied either on the device or the device box. The public key extracted from the QR code is used: (*i*) to get the missing five digits; (*ii*) to check if the public key extracted matches with the one sent (obfuscated) by the device; (*iii*) to complete the ECDH agreement mechanism. The updated procedure increases the overall security of the protocol since the controller can check the full public key and not only the first five digits.

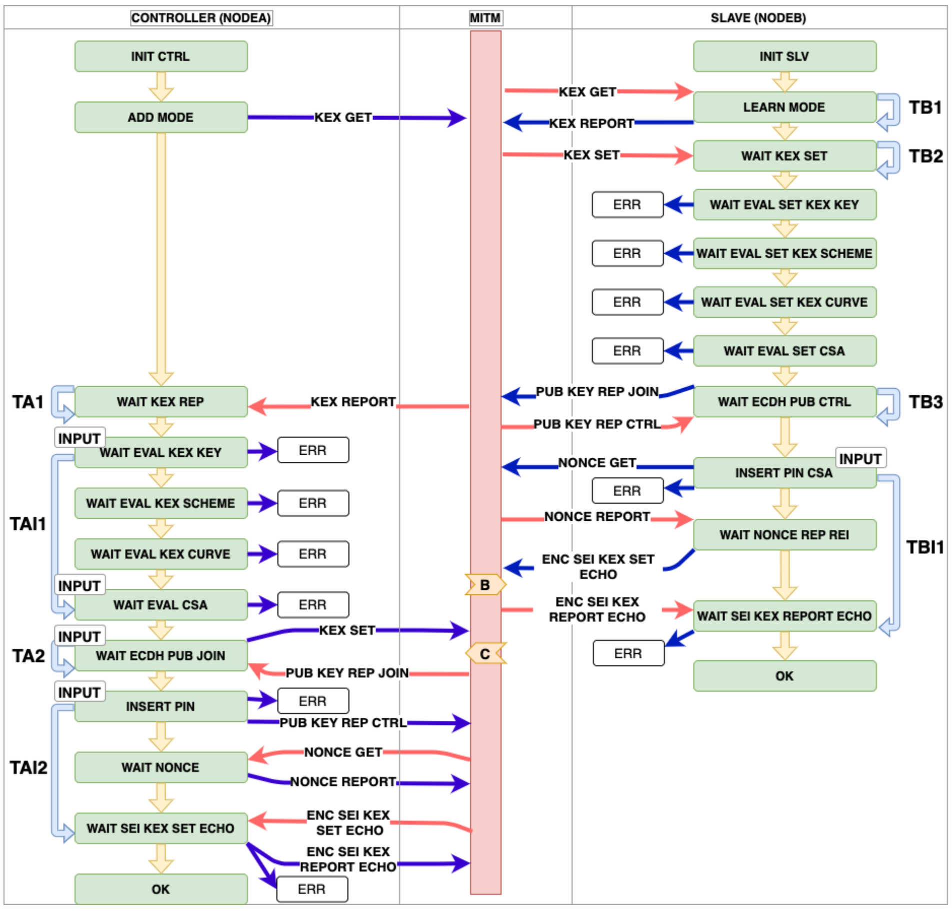


Figure 4.8: Flow chart of the MITM attack

The model can be easily extended to integrate the new functionality by updating the `insertPin` rule of the controller with a new decision branch (the attacker models remain unchanged). The scanning of the QR code is implemented by some new monitored functions: `chosenQrCodeUsage` allows the user to select the default authentication mode (QR code or manual entry); `chosenQrCode` gives the asymmetric public key encoded in the QR code. The monitored values are stored in two controlled functions during the execution (`qrCodeUseDecison` and `qrKey` functions).

The new branch added to the `insertPin` rule of the controller program is reported in Code 4.7 and works as follows:

- Line 2 sets the intruder as the receiver of the agent communication, modelling the intruder intercepting all the traffic.

- Lines 3-5 specify the guard, which checks: (*i*) the internal state of the controller (in this case the controller must be in `WAIT_ECDH_PUB_JOIN` state), (*ii*) the message received (i.e., `PUB_KEY_REP_JOIN`), and (*iii*) if timer `TA2` has not expired.

- On Lines 6-9, the controller:(*i*) updates its internal state to `INSERT_PIN`, (*ii*) activates timer `TAI2` and deactivates timer `TAI2`, and (*iii*) asks the user to continue with the *joining procedure* using the monitored function `ctrlAbort`.

- Lines 12-14 are reached when the controller is in `INSERT_PIN` state. In this case, if the timer has not expired, the controller: (*i*) checks where the user has chosen to insert the PIN (either on the controller or on the joining device in case of Client-Side Authentication), and (*ii*) checks if the user wants to abort the *joining procedure*.

- Line 15 reads the public key received by the controller.

- On Lines 17-19, the controller changes its internal state to `WAIT_NONCE` and sends the message `PUB_KEY_REP_CTRL` containing his public key, as required by the protocol.

- Line 20 starts the branch dedicated to the scanning of the QR code by checking if the slave chose QR code authentication as the default mode.

- On Lines 21-22, the slave public key derived from the QR code and the controller private key are used by ECDH to compute the shared symmetric key.

- Line 24 puts the newly generated symmetric key in the controller's knowledge.

- Line 25 uses function `recomposePubKey` to recompose the obfuscated key and check if it matches the public key extracted from the QR code.

```
1  rule r_insertPin =
2    let ($eslv=nodeE) in
3     if (controllerState(self) = WAIT_ECDH_PUB_JOIN
4     and protocolMessage( $eslv ,self ) = PUB_KEY_REP_JOIN ) then
5       if (not(passed(TA2))) then
6         controllerState(self):= INSERT_PIN
7         startTimer(TAI2):= true
8         startTimer(TA2):= false
9         abortCtrlSaved:= ctrlAbort
10      endif
11    else
12     if (controllerState(self) = INSERT_PIN and  messageField(self, $eslv,1,KEX_SET)=CSA_0
13     and protocolMessage( $eslv ,self ) = PUB_KEY_REP_JOIN and abortCtrlSaved = false) then
14      if (not(passed(TAI2))) then
15       let ($sk_ob =messageField($eslv,self,1,PUB_KEY_REP_JOIN)) in
16        par
17         controllerState(self):= WAIT_NONCE
18         protocolMessage(self, $eslv):= PUB_KEY_REP_CTRL
19         messageField(self , $eslv,1,PUB_KEY_REP_CTRL):= KPUB_CTRL
20         if (qrCodeUseDecison= true) then
21           let ($pubkey_qr=qrKey) in
22             let ($aes_qr=diffieHellman($pubkey_qr,KPRIV_CTRL)) in
23               par
24                 knowsSymKey(self,$aes_qr):= true
25                 if (recomposePubKey(true,$sk_ob)=$pubkey_qr) then
26                   par
27                     knowsBitString(self,PIN_OK):= true
28                     knowsBitString(self,PIN_ERROR):= false
29                   endpar
30                 endif
31               endpar
32             endlet
33           endlet
34         else
35         ...
36      endlet
```

Code 4.7: insertPin rule

- On Lines 27-28, the PIN code is added to the controller's knowledge only if the public key extracted from the QR code matches the obfuscated one.

4.1.9 Z-Wave Formal Validation & Verification

In the following, we describe the analysis process we followed to validate and verify the models we introduced in the previous section. The analysis was performed both on the model with the passive attacker (Sect. 4.1.9.1) and on the model with an active attacker (Sect. 4.1.9.2). In particular:

1. The models of the *joining procedure* were validated by using ASMETAV validator working on Avalla scenarios, assessing model reliability and completeness with respect to the informal protocol description. We expressed one scenario for each requirement in the Z-Wave protocol documentation and checked that the model behaves as expected.

2. To verify the correctness of the Z-Wave protocol despite the presence of the attacker, we expressed a list of security properties as a CTL formula and used the ASMETASMV tool to check if a CTL formula holds by exploiting the NUSMV model checker.

4.1.9.1 Analysis of the model with a passive attacker

Model Validation Avalla language provides special commands to: **set** the values of monitored functions, perform one **step** of simulation, **exec** rules or function updates, **check** that some properties hold. By playing with a combination of these commands, we were able to express significant scenarios and check the correctness of the message flow.

Here, we report and discuss a fragment of a scenario (Code 4.8) where the user of a door lock updated to S2 Security refuses to insert the PIN on the device rather than on the controller side.

- On line 4, we check that the scenario is correctly set at the beginning of the *joining procedure*, i.e., the controller and the slave are in their initial states `INIT_SLV` and `INIT_CTRL`.

- Lines 6-8 configure the scenario, i.e., the type of the controller and the slave (a door lock updated to S2 Security), and the attacker mode (passive).

- The checking on lines 11-13 is done for all the messages exchanged before getting to the point of the protocol in which the user has to insert the PIN on the device: a certain message has to be received when

Code 4.8: Rejected Client-Side Authetication scenario

```
 1 scenario refuseCSA_passive_scenario
 2 load ZWave_join_MITM_validation.asm
 3
 4 check slaveState(nodeB) = INIT_SLV
 5       and controllerState(nodeA) = INIT_CTRL;
 6 set controller(nodeA) := CONTROLLER_S2;
 7 set slave(nodeB) := DOOR_LOCK_UP;
 8 set chosenMode := PASSIVE;
 9 ...
10 step
11 check protocolMessage(nodeE,nodeB) = KEX_GET
12       and controllerState(nodeA) = WAIT_KEX_REP
13       and slaveState(nodeB) = LEARN_MODE;
14 set passed(TB1) := false;
15 set passed(TA1) := false;
16 ...
17 step
18 ...
19 set userCsa(nodeA):=false;
20 step
21 check controllerState(nodeA) =ERROR_C
22       and protocolMessage(nodeA,nodeE) =KEX_FAIL_CANCEL;
```

the controller and the slave are in the correct state according to the definition of the protocol. In this case, the slave in `LEARN_MODE` state receives `KEX_GET` message when the controller is in `WAIT_KEX_REP` state.

- On lines 14-15, the timers `TB1` and `TA1` are set as not expired.
- After some intermediate simulation steps and checks, on lines 19-22, we simulate a user rejecting the request of Client-Side Authentication and check that an error is generated and `KEX_FAIL_CANCEL` message is sent by the controller indicating the failing of the *joining procedure*.

Property verification During the verification phase, we examined a broad spectrum of properties expressed as CTL formulas and verified them by the model checker NuSMV. The result, if the model checker terminates, is a Boolean condition: in case of `True`, the property holds; if it is `False`, it means that the property is violated at least once (and the tool returns the attack trace). In case of a violating path, the trace was rewritten as a scenario in the Avalla language and executed on the model to better understand the origin of the vulnerability.

We derived some of the properties from the patterns presented in Sect. 4.1.4.2. The first two properties on *confidentiality* were proved, meaning that without an active Intruder, the ECDH key exchange is robust:

There exists no state in the future in which the `PASSIVE` *Intruder* `nodeE` *knows the symmetric key* `KT3` (i.e., the key generated by ECDH key exchange using the controller and slave asymmetric keys).

[P1] $\neg EF(\text{mode=PASSIVE} \wedge \text{knowsSymKey(nodeE,KT3)=true})$

There exists no state in the future in which the `PASSIVE` *Intruder* `nodeE` *knows the correct pin code* `PIN_OK`.

[P2] $\neg EF(\text{mode=PASSIVE} \wedge \text{knowsSymKey(nodeE,PIN\_OK)=true})$

We verified the *integrity* property for each field in the exchanged messages. The properties should hold since a passive attacker only re-routes messages.

There exists no state in the future in which a `PASSIVE` *attacker can modify the first field of* `KEX_SET` *message* .

[P3] $\neg EF(\text{mode=PASSIVE} \wedge \text{messageField(nodeA,nodeE,1,KEX\_SET)}$

$\text{!=messageField(nodeE,nodeB,1,KEX\_SET)})$

We conclude our analysis of the passive attacker model of the *joining procedure* with the proof of the following *authentication* property:

There exists a state in the future in which, if the attacker is `PASSIVE`*, the* `slave` *knows its* `public-key` *and the* `controller` *has obtained the correct* `PIN`*, then the attacker will never know the correct* `PIN` .

[P4] $EF(\text{mode=PASSIVE} \wedge \text{knowsAsymPubKey(nodeB,KPUB\_SLV)=true} \wedge$

$\text{knowsBitString(nodeA,PIN\_OK)=true}) \rightarrow AG(\neg(\text{knowsBitString(nodeE,PIN\_OK)=true}))$

The evaluation of the formula is `True` because to recover the correct PIN, the passive attacker must break the encryption using brute force over the full symmetric key.

4.1.9.2 Analysis of the model with an active attacker

Model Validation The modelling of the legitimate actors of the protocol is the same in the case of an active attacker. Thus, there was no need to re-validate the protocol requirements.

Property verification As done in the case of a passive attacker, we derived the CTL formulas of some of the properties from the patterns presented in Section 4.1.4.2. Obviously, during the verification phase, we used the model with the most powerful adversary. As expected, some properties do not hold, meaning that an attacker able to craft the correct PIN code and the associated public key can break the protocol.

The *confidentiality* properties check that the temporary symmetric key K_t is known only by the two honest participants:

There exists no state in the future in which the `ACTIVE` *intruder* `nodeE` *knows the symmetric key* `KT1` *or* `KT2` (i.e., the key generated by ECDH using the asymmetric keys of the slave and the controller).

[P5] $\neg EF\left(\text{mode=ACTIVE} \wedge \text{knowsSymKey(nodeE,KT1)=true}\right)$

[P6] $\neg EF\left(\text{mode=ACTIVE} \wedge \text{knowsSymKey(nodeE,KT2)=true}\right)$

Both *confidentiality* properties are `False`. Intuitively, it means that the attacker is able to take part to the ECDH key exchange with a MITM attack, impersonating the slave with the controller -first formula- and, vice versa, the controller with the slave -second formula-.

Integrity properties check if the attacker is able to modify the messages of the protocol without the legitimate principals noticing it. For example, the following formula does the check for the `KEX_REP` message:

> *There exists no state in the future in which an active attacker can modify the fourth field of the message* `KEX_REP`.

[P7] $\neg EF(\text{mode=ACTIVE} \wedge \text{messageField(nodeB,nodeE,4,KEX\_REP)}$

$\text{!=messageField(nodeE,nodeA,4,KEX\_REP)})$

The *integrity* properties results are `True`, because the active attacker forwards the received message fields from the slave to the controller without altering them, to reduce the probability of triggering some error responses.

We tested the same *authentication property* as for the passive attacker, changing only the intruder's mode to active. The property is expressed as follow:

> *There exists a state in the future in which, if the* `attacker` *is* `ACTIVE`*, the* `slave` *knows its* `public-key` *and the* `controller` *has obtained the correct* `PIN`*, then the attacker will never know the correct* `PIN`.

[P8] $EF(\text{mode=ACTIVE} \wedge \text{knowsAsymPubKey(nodeB,KPUB\_SLV)=true} \wedge$

$\text{knowsBitString(nodeA,PIN\_OK)=true}) \rightarrow AG(\neg(\text{knowsBitString(nodeE,PIN\_OK)=true}))$

The property evaluation returns `False`, showing a partial path of the attack where both the slave and the controller have established a key with the attacker and share the same pin code. We use the slave knowledge of its public key because the pin code is extracted from the five first digits of that key. Thus if the slave knows his/her complete public key, he/she knows the pin code.

By checking the security properties described above, we found that the shared key K_t can be compromised. However, the attack is not successful if the legitimate participants notice it. To this aim, we checked if there is at least one execution trace that leads the controller and the slave to believe that the protocol ended correctly, namely:

There is not a state in the future in which the `controller` *and the* `slave` *are both in the* `OK` *state, that represents the final state they reach only when the protocol runs as expected.*

[P9] $\neg EF(\text{controllerState(nodeA) =OK\_C} \wedge \text{slaveState(nodeB) = OK\_S})$

The property evaluation returns `False`. Thus, we analysed the violating path encoding it in an `Avalla` scenario, as described in the following.

Analysis of the attack trace The failure of the verification of some security properties seems to confirm the feasibility of an MITM attack in the case of an active attacker. In general, a faulty property trace can be encoded in a scenario giving the ability to the protocol designer to visualise and manipulate the trace. The scenario can be later used to test vulnerability patches or it is useful for crafting new verification properties to find further vulnerabilities.

In the specific case of our faulty property, the scenario drawn from the trace shows that the attacker tries to take advantage of the duration of a timer as much as possible, avoiding the timeout (a brute force attack is a time-consuming operation).

Since timers are local to the legitimate actors of the protocol, the attacker cannot know exactly when a timer has expired. To this aim, we give the intruder the capability to estimate the expiration time with a monitored function `nearToEnd` that takes a timer and returns `true` if it is running out. This scenario shows the feasibility of the Man-In-The-Middle (MITM) attack we found in [60]. The vulnerability was confirmed and patched by Silicon Labs and Z-Wave Alliance.

An excerpt of the scenario is reported in Code 4.9. The structure is similar to Code 4.8:

- The scenario involves an alarm device with native S2 Security, a controller supporting the S2 Security and an active attacker (lines 4-8).
- Timer `TA1` has not expired yet (line 9).

Code 4.9: Successful MITM attack scenario

```
1  scenario validation_protocol_mitm
2  load zwaveProtocol_join_MITM_validation.asm
3
4  set chosenMode:=ACTIVE;
5  check slaveState(nodeB) = INIT_SLV
6        and controllerState(nodeA) = INIT_CTRL;
7  set controller(nodeA) := CONTROLLER_S2;
8  set slave(nodeB) := ALARM_NAT;
9  set passed(TA1) := false;
10 ...
11 step
12 ...
13 set nearToEnd(TA1) := true;
14 step
15 check protocolMessage(nodeE,nodeB) = PUB_KEY_REP_CTRL
16       and slaveState(nodeB) = WAIT_ECDH_PUB_CTRL
17       and controllerState(nodeA) = WAIT_KEX_REP;
18 ...
19 step
20 ...
21 check protocolMessage(nodeA,nodeE)=EC_KEX_REPORT_ECHO
22       and slaveState(nodeB) = OK_S
23       and protocolMessage(nodeE,nodeB)=EC_KEX_REPORT_ECHO
24       and controllerState(nodeA) = OK_C;
```

- The attacker uses `nearToEnd` to delay as much as possible the sending of the message `KEX_REPORT`, i.e., just before timer `TA1` expires (lines 13-17, as depicted also in Fig 4.8).

- Function `nearToEnd` is used also to delay just before `TA2` expires the execution of the brute force attack (lines 21-24).

4.1.10 Discussion and Threat to validity

We here provide some evaluating discussions of our approach. Applying the modelling primitives to the specification of a real case study has demonstrated their adequacy and expressiveness. The model presented here is more concise than that in [60], and performs better for the verification, allowing us a more complete model validation and verification.

More specifically, the CTL properties presented in this paper have been verified using an AMD Ryzen 2600 processor with 16 GB of RAM. The results

	Confidentiality		**Integrity**	**Authentication**	
Property	P1	P2	P3	P4	Avg.
Time (s)	0,06	107,86	23,17	96,04	56,78
BDD Allocated	106127	919310	427160	464471	479269
Outcome	true	true	true	true	

Table 4.1: Time and BDD nodes required for the verification of a passive attacker

of property verification are shown in Tables 4.1 , 4.2 and 4.3 in terms of time and space complexity, i.e., time required for the execution and nodes of the binary decision diagram (BDD) allocated in memory. Table 4.1 shows the results obtained by analysing the CTL properties of a passive attacker, and Table 4.2 that of an active attacker. We also include the results of some reachability properties that performed worst in time. This is due to the fact that the verification of these specific properties requires reachability of the model terminal states, thus the model checker must search deeper into the BDD to guarantee that the property holds.

	Confidentiality		**Integrity**	**Authentication**	
Property	P5	P6	P7	P8	Avg.
Time (s)	53,88	53,37	62,35	85,08	63,67
BDD Allocated	379759	94198	381814	499791	338890
Outcome	false	false	true	false	

Table 4.2: Time and BDD nodes required for the verification of an active attacker

	Reachability		
Property	P9	P10	Avg.
Time (s)	214,01	143,33	178,67
BDD Allocated	154573	829309	491941
Outcome	false	true	

Table 4.3: Time and BDD nodes required for the verification of the reachability properties

The results confirmed us the utility of the ASMs as formal method, mainly related to their possibility to specify a model in a symbolic way and at any level of abstraction, their understandability as pseudo-code over abstract data, and the simplicity of refining modes by adding details along the modeling process.

The refinement process was very useful not so much for modeling the protocol itself (as communication between honest principles), as for modeling the intruder capabilities. This refining process helped a lot for the verification phase, either in terms of its scalability, and in terms of understanding the results and establishing the minimal intruder's abilities to exploit the protocol vulnerability.

Our results show the importance of exploiting formal methods during the design process of a security protocol in order to find vulnerabilities since the early stages of protocol development. Already in [60], the formal verification of the Z-Wave joining procedure revealed a vulnerability later confirmed by the Z-Wave Alliance. The formalisation of the protocol helped to increase the overall security of the *joining procedure*; indeed, the Alliance has introduced a new way of authenticating the joining device using a QR code, as explained in Sect. 4.1.8.

On the refined model in Section 4.1.8, we have proved the same property checked for the model under active attacker. More precisely, we verified the property:

There is not a state in the future in which the Controller and the Slave are both in the OK state, the QR code is used for the authentication, and the key encoded in the QR code is the public key of the Slave.

[P10]
$$\neg EF\big(\text{controllerState(nodeA) =OK\_C} \wedge$$
$$\text{slaveState(nodeB) = OK\_S} \wedge \text{qrKey=KPUB\_SLV} \wedge \text{qrCodeUseDecison=true}\big)$$

As expected the property returns `True`, proving that it is no longer possible to exploit the vulnerability.

However, the approach has some limitations. They are mainly due to the state explosion of the model checker, caused by the possible high cardinality of the knowledge domain. Although the usage of high level mathematical functions helps in a symbolic representation of a given protocol, their mapping to a sequence of model checker's variables may compromise the performance of the verification because the number of variables to manage might be very high.

To overcome this verification limits, we are working towards the definition of a domain-specific language (DSL) for IoT protocols by exploiting the meta-modeling approach of DSL construction. Such DSL should work as a sort of *"lingua franca"* between protocol requirements and back-end verification tools. The obtained protocol program can be mapped into different verification tools by exploiting model-to-model transformations. For example, the mapping to ASM models goes through the mapping of the DSL constructs to the `CryptoLibrary` functions and domains. Such transformations will allow for the use of verification tools having verification techniques different from model checking, for example, clauses resolution in ProVerif [24] or rewriting rules for constraint solving in TAMARIN [65].

4.1.11 Related Work

After the seminal paper on BAN logic [36] introducing one of the first formalisms designed to reason about protocols, many techniques and tools have been proposed for verifying protocols. Unfortunately, formal methods are still not widely adopted by industry, mainly due to their mathematical base, which discourages many designers or engineers [42]. For this reason, there are only a few mature tools for automatic verification [65, 24] and many of the protocols verified so far have been a case study for a specific tool.

Specification and verification of IoT protocols is a recent topic in the context of formal methods, and small literature exists in this area. A comprehensive review of formal methods for IoT protocols is given in [51]. Z-Wave has small literature since it has been proprietary for a long time. The security of the Z-Wave protocol has been mainly investigated and tested in practice by means of test beds in real environments. In [82], Z-Wave is analysed with respect to common attacks to IoT systems. In [16], the frame forwarding and topology management aspects of a previous version of the Z-Wave routing protocol are reverse-engineered. Then, a security analysis is also performed on the network, and the possibility of modifying the topology and routes by an outsider is found. In [56], the authors propose three different attack vectors that, if combined, can cause critical damage (e.g., DoS).

To the best of our knowledge, apart our previous work [60], the only contribution on formal methods applied to Z-Wave protocol is [71], where the possibility of flaws generated by wrong configurations in Z-Wave scenes is evaluated by means of model checking. In this paper, we extend and enhance the results we obtained in our previous works on security protocol analysis [59, 35].

We extend and give a more complete set of templates to formalise, in the Abstract State Machine formal method [30, 31], common behavioural patterns in security protocols and a set of security properties. These templates were presented in [35] to facilitate the protocol formal verification by providing built-in mathematical elements (functions, domains, transition rules, etc.) to be customised according to the specific protocol to be verified. The improvements regard the specification of protocol principals' knowledge and how it is communicated and shared during the protocol execution, e.g., we model the message structure in a way that allows us to obtain better performance and scalability in the verification phase. Moreover, two different models of intruder, a passive attacker that simply eavesdrops on the messages in the communication channel and an active attacker able to intercept and craft messages, are provided for the validation and verification. The improvements with respect to the results in [60] mainly concern the evaluation of the effectiveness of our modelling and verification primitives on a real and complex case study. Here, following a model-based approach, the improved templates are used to model the protocol. The resulting specification is more concise, so the verification scales better with the new modelling choices. Moreover, the model also includes the specification of the solution implemented by the Z-Wave Alliance to avoid the vulnerability found in [60].

The idea of pattern reusing is not new but has not been investigated a lot in the context of security protocol design and verification. In [22], a systematic way to design secure-by-construction cryptographic protocols, where the proof process reuses smaller protocol parts previously proven to be correct and secure. In this case the approach is based on the B notation. The work in [21] describes a method for implementing and analyse a specific class of security protocols (i.e., classical key distribution protocols) in SPIN. In particular, the authors focus on modelling a generic intruder model working with all the protocols within the class. The work in [70] presents a model-driven approach to design security-critical systems based on cryptographic protocols and to prove application-specific security properties. A smart card application is analysed. UML is used as front-end modelling notation, whereas ASMs and a theorem prover are used as back-end formalisms for property verification; the underlying idea is similar to ours.

Among the other IoT protocols, the mostly analysed are Zigbee, LoRaWAN, Bluetooth, Narrowband IoT, 6LoWPAN, and 5G. The Zigbee protocol is analyzed in [66] by using AVISPA and Casper, where the existence of a known security flaw is discovered. AVISPA (Automated Validation of Internet Security Protocols and Applications) [12], in combination with the graphical tool SPAN, allows a sort of protocol runtime verification where Message Sequence Charts are built during the protocol execution and are used to check if the execution performs consistently with the expected message exchange [85].

Two versions of LoRaWAN are compared in [45] using models specified with SCYTHER: for LoRaWAN v1.0, they find some flaws already known in the literature, while they do not find any evident vulnerability in v1.1. Some of the Bluetooth protocol versions have been verified with formal methods. Version v5.0-Part I has been checked by Sun and Sun [80]; they perform a formal analysis of the secure, simple pairing (SSP) that constitutes a considerable part of the security in home automation scenarios. Four different models are used depending on the device's capability, but they all resist passive eavesdropping and MITM attacks.

Two schemas for the 6LoWPAN protocol have been verified. In [76], the authors propose a secure Proxy Mobile IPv6 (MPIPv6) that prevents some attacks, including replay attacks, man-in-the-middle attacks, privileged insider attacks and Sybil attacks. The attacks and the schema are executed by using AVISPA and a Java simulation. The work in [75] presents an extension schema of 6LoWPAN, which grants Confidentiality, Integrity and Authentication for a group of resource-constrained 6LoWPAN devices. In [19], a formal analysis of the 5G authenticated key exchange (AKA) protocol is provided. The 5G AKA is formalised with TAMARIN, and the verification shows possible privacy flaws. A provable fix for the vulnerability found is then proposed.

4.2 Multi-tool Verification Approach

4.2.1 Overview

Developed progressively since 2016, WPA3 has been designed to enhance wireless security from the current WPA2 standard. Yet, despite this claim, it has not been impervious to security vulnerabilities [84]. This paper delves into the critical analysis of WPA3's security framework using a formal verification approach, able to fully capture the WPA3 Simultaneous Authentication of Equals protocol and show to what extent it really *is secure*.

A notable observation in the field of protocol verification is its predominant concentration on the message exchange. While this focus has yielded robust verification methodologies[20, 10, 26], it inadvertently overlooks potential vulnerabilities residing in the behaviour of single devices [33]. This oversight becomes particularly critical when protocols are deployed in real-world scenarios, where lower-level vulnerabilities can lead to significant security breaches.

In response to this challenge, our research proposes a unified and integrated approach to protocol verification that encompasses both these aspects, which, for brevity, we call them respectively *Communication level* and *Device level*. Our methodology exploits different formalisms and verification techniques at the two levels (*stateless algebraic system* at the Communication level and *stateful transition system* at the Device level), thus leveraging the strengths of each approach to achieve verification goals of each level. This holistic methodology enables a more comprehensive and effective security verification process. By integrating verification across multiple levels, our approach not only identifies but also helps mitigate vulnerabilities that might otherwise go undetected in traditional, single-level verification processes.

Our findings using this approach are twofold: i) the two modelling efforts intuitively capture different aspects of the protocol, and each finds a subset of errors in the standard - some of which link to serious known and new vulnerabilities; and, interestingly ii) combining these two modelling approaches, we gained new insights into the nature of the vulnerabilities, greatly aiding our understanding and ability to patch the standard - which is a desirable result for future such investigations. In this work, we showcase the strength of this methodology by discovering a series of attacks found through our formal analysis at both layers, alongside a series of errors in the specification.

Our models are validated against existing attacks on WPA3 [84]; as we are able to replicate known previous attacks, we gain confidence that our model captures significant parts of real-world implementations. Through this analysis, we found other attacks and several errors, which are described in Appendix A; for brevity, the rest of the thesis will focus on a subset of these to highlight the advantage of multi-level analysis. We found that some of the errata were due to misalignments between the Communication and the Device level. Rather than having a correct specification at the original level, they were patched at a different level. Whilst a common misconception may be that being a newer standard and more recent might mean better security, we find that a greater numerical number does not always imply better security guarantees; and indeed, often, unfortunately, several security safeguards are discarded in newer standards, leading to repetitions of attacks as we find in this work. These vulnerabilities and errors have been acknowledged by the IEEE standards and are in the process of being rectified [the latest standard is coming soon].

Our *contributions* are the following: i) a unified and integrated approach to protocol verification that encompasses both Communication and Device levels; ii) a detailed security analysis, applying this approach to the WPA3-SAE; and iii) results of security analysis, showcasing over 20 new errors and security flaws, as well as our suggested patches verified to fix the problems.

4.2.2 Related Work

The unification of multiple tools for the verification of protocols has come under substantial attention recently [14, 72], this is not altogether new as it was one of the core principles of AVISPA [10] (although all based on AnB logic), and later on, the earliest work discussing this for modern verifiers was in 2019, where authors combined the verification of a protocol using both TAMARIN and PROVERIF [13]. This was not shocking, as protocols formally verified in one tool have been found vulnerable by another [49].

Recent work unifying protocol verification [14] presents a methodology for the modular verification of protocol implementations. The methodology leverages verification logics and tools, supporting a wide range of implementations and programming languages. The effectiveness of this approach is demonstrated through the verification of memory safety and security of various protocol implementations. Similarly, another work extends the security guarantees typically associated with protocol designs to their actual implementations [72]. This is achieved by instrumenting common cryptographic libraries and network interfaces with a runtime monitor. By focusing on runtime verification, the paper addresses the dynamic aspects of protocol security, ensuring that the protocols behave as expected not just in theory but also in real-world operational environments. Early work on verification of TLS1.2 [23], the authors here identify that the state machine is an important part of the implementation and perform some verification on it. The verification process includes type checking the state machine of the TLS protocol. Type checking is a method of verifying that the program adheres to the specified types, which in this context, relate to the correct sequence of operations and data handling in the protocol. This ensures that the state transitions in the TLS protocol adhere to the defined security specifications. This is a very important first step; however, this is never linked to the upper layers of the protocol, and no further verification beyond type checking is performed. We argue these are not distinct steps, and indeed the verification of the state machine properties can lead to better analysis of the network protocol and vice versa.

4.2.3 WPA3 Simultaneous Exchange of Equals Key Exchange

WPA3 security is ensured using the Simultaneous Exchange of Equals (SAE) protocol, which is introduced to replace the Pre Shared Key Exchange (PSK) used in WPA2, which was known to be vulnerable to attacks [83]. SAE provides authentication at the Link layer referring to the TCP/ IP stack. It was originally introduced in 2016 as part of IEEE 802.11 - IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems [53], with claims to resolve the previous vulnerabilities affecting PSK and WPA2. We focus our security analysis on the latest version protocol, following the specifications from IEEE 802.11w [54] when conducting our assessment.

4.2.3.1 The SAE Key Exchange (Communication level)

The SAE key exchange is a two-party protocol that takes two rounds of messages. Each round is symmetric so that we do not have a notion of an Initiator and Responder or of a Supplicant and Authenticator. Each side may initiate the protocol simultaneously such that each side views itself as the *initiator* for a particular run of the protocol. This design is necessary to address the unique nature of mesh basic service set [54]. Several variants of SAE are specified; nonetheless, we only focus on the variant of SAE adopted in the WPA3 protocol, especially the variant that operates in finite field cryptography.

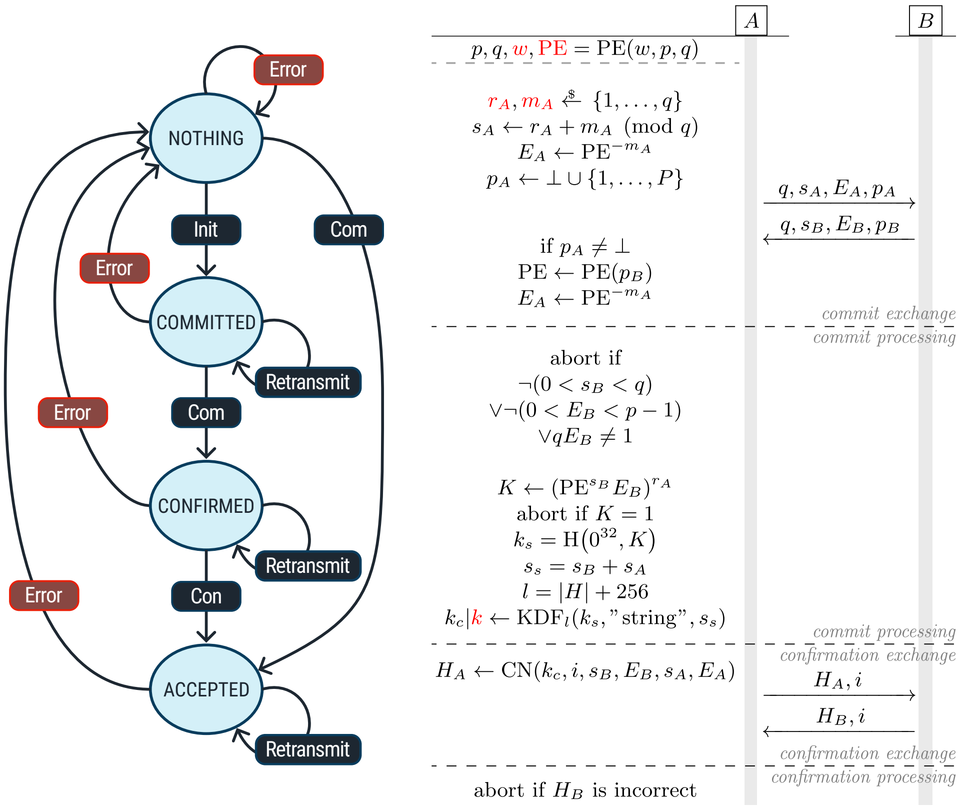
The SAE protocol operates in $\mathbb{G}$, a common and known subgroup of $\mathbb{Z}_p^\star$ of multiplicative order q, where q is a Sophie Germain prime, i.e., $p = 2q+1$, where the discrete logarithm problem is assumed to be hard. The protocol also uses H representing a hash-based message authentication code (HMAC). Two remote parties, Alice and Bob, share a common secret password from which they apply a transformation to calculate a corresponding *password element*, PE. This secret element can be derived as described by the Key Derivation Function KDF described in the standard [54]. The (last) confirmation phase also defines a confirmation function CN, also described in the standard. Both KDF and CN are calls to the hash function H. The SAE protocol runs in two rounds: the *commit exchange* and the *confirmation exchange*, as illustrated in Fig. 4.9 (*right*) for the peer A, which communicates with a symmetric peer B.

4.2.3.2 The SAE Message Handling (Device level)

Message handling in WPA3 is described in §12.4.8 of the standard, in terms of a state machine, as interactions between three entities: 1. Station Management Entity (SME), 2. Parent Process (PP), and 3. Protocol Instances (PI).

SME is a component responsible for managing various aspects of a wireless station or device. It is primarily concerned with managing the physical and medium access control (MAC) layers of the 802.11 protocol. SME provides an interface for higher-layer protocols to interact

Figure 4.9: SAE protocol in WPA3 [54] at both Device level (*left*) and Communication level (*right*). A and B share the secret password w and computed PE in private; $\mathrm{PE} \in \mathbb{G}$ and $\mathbb{G}$ is a subgroup of $\mathbb{Z}_p^\star$ of order q; i is a counter. B is symmetric to A and thus omitted.


Error
NOTHING
Init
Com
Error
COMMITTED
Retransmit
Error
Com
CONFIRMED
Retransmit
Error
Con
ACCEPTED
Retransmit
A
B
$p, q, w, \mathrm{PE} = \mathrm{PE}(w, p, q)$
$r_A, m_A \xleftarrow{\$} \{1, \ldots, q\}$
$s_A \leftarrow r_A + m_A \pmod q$
$E_A \leftarrow \mathrm{PE}^{-m_A}$
$p_A \leftarrow \perp \cup \{1, \ldots, P\}$
q, s_A, E_A, p_A
q, s_B, E_B, p_B
if $p_A \neq \perp$
$\mathrm{PE} \leftarrow \mathrm{PE}(p_B)$
$E_A \leftarrow \mathrm{PE}^{-m_A}$
commit exchange
commit processing
abort if
$\neg(0 < s_B < q)$
$\vee\neg(0 < E_B < p - 1)$
$\vee q E_B \neq 1$
$K \leftarrow (\mathrm{PE}^{s_B} E_B)^{r_A}$
abort if $K = 1$
$k_s = \mathrm{H}(0^{32}, K)$
$s_s = s_B + s_A$
$l = |H| + 256$
$k_c|k \leftarrow \mathrm{KDF}_l(k_s, \text{"string"}, s_s)$
commit processing
confirmation exchange
$H_A \leftarrow \mathrm{CN}(k_c, i, s_B, E_B, s_A, E_A)$
H_A, i
H_B, i
confirmation exchange
confirmation processing
abort if H_B is incorrect

with the lower layers and handles tasks related to the wireless station's configuration, operation, and maintenance. However, this entity is not strictly described in the standard, which specifies:

> Some of the functions of the SME are specified in this standard.
>
> §6.1, pg. 314

Moreover, the description is fragmented and given incrementally in different parts of the documentation. This easily causes misinterpretation and misunderstanding.

PP is in charge of managing the database of the protocol instances (PIs). It performs a number of tasks, such as allocating and deallocating instances, and keeping track of their respective states. Additionally, it is responsible for routing incoming messages from the environment to the correct PI (allocating a new instance when needed). To accomplish these operations, it sends a suitable event to drive the state change of the PI. It also keeps the database updated based on the events it receives from both the SME and the PIs.

PIs behave according to a state machine. State changes are triggered by messages and events received from the PP, but transitions fire upon the analysis of message content. Transitions execution is accompanied by actions that the PI executes before entering a state. Such actions consist of messages for the peer or generation of *error* or *completion* output events to be sent to the PP for subsequent deallocation of the instance.

Fig. 4.9 (*left*) depicts a simplified version of a PI's state machine consisting of four states and having transitions labelled with the relevant events sent by the PP and able to trigger a state transition (note that we omit information regarding the received messages that might prevent transition firing, and actions performed by the PI when a transition fires). A PI is in the state Nothing initially, as a new instance, and finally, as a terminal state before being deallocated in case of error. Depending on their allocation, new PI immediately transitions out of the Nothing state to either Committed or Confirmed. Protocol instances that transition into the Nothing state shall immediately and irretrievably be deleted[6]. A PI enters state Committed when it receives the event Init from the PP and has sent an SAE *commit* message to the peer. In this state, the PI waits for the Com event from the PP and moves to state Confirmed having sent an SAE *confirm* when it receives the correct SAE *commit* message from the peer. The PI remains in this state if a Retransmission[7] event happens. When an unmanageable error occurs, a Del event is sent to the PP, and the PI moves to state Nothing to be deallocated. In the state Confirmed, the PI operates as in state Committed in case of events Del and Retransmission, and moves to state Accepted on the event Con from the PP and by receiving a peer's correct *commit* message. In the (final) Accepted state, events Retransmission and Del, if happen, are dealt

[6]This relevant information is reported only in section 12.4.8.2.2 of [54]

[7]This event groups several wrong (but manageable) error cases that can happen, e.g., wrong order of the received messages, wrong received group, etc.

with as in the previous states; otherwise, the PI has successfully concluded the process.

4.2.4 Methodology

The unified methodology, at the Communication and Device level that we propose, concretely involves:

- **Dual-Level Modeling**. Specify models that separately represent the Communication and Device levels.
- **Level-Specific Properties Analysis**. Exploit different formalisms and verification paradigms at the two levels: a *stateless modelling system* at the Communication level and a *stateful modelling system* at the Device level. This dual verification approach leverages the strengths of each methodology to cover a comprehensive range of aspects and verification goals of each level:
  - At the Communication level, we use π-calculus as specification formalism, Horn clauses resolution as a verification mechanism, and PROVERIF [27] as a tool. For this, we analyse properties common to key exchange verification processes, with a special focus on Forward Secrecy claims.
  - At the Device level, we use state-based transition systems as specification formalism (specifically, the Abstract State Machines [31]), model checking of CTL temporal properties [17] as verification mechanism, and ASMETA [15] as supporting tools. At this level, we concentrate on the operational correctness of message handling, synchronisation mechanisms, queue implementations, and state reachability within the protocol.

This approach allows for a much deeper study into the security of a protocol, not only allowing for greater insights into how well the protocol functions (thus allowing for better implementations).

4.2.5 Formal Models of WPA3-SAE

As per phase one of our methodology, we first modelled the Communication and Device levels of the protocol.

4.2.5.1 Models at the Communication level in the π-calculus

As can be easily seen, the π-calculus code inside the boxes in Fig. 4.10 is the part modelling the protocol scheme depicted in Fig. 4.9.

Figure 4.10: The vanilla description of the SAE protocol.

```
$P_L \leftarrow$  get $t_p(=L, =R, =pw, \mathrm{PE})$ in
     $\nu r_L.\nu m_L.$
     let $s_L = r_L + m_L$ in
     let $E_L = \mathrm{PE}^{-m_L}$ in
     let $c_L = (s_L, E_L)$ in
     out$(c, c_L)$;in$(c, c_R)$;
     let $s_R, E_R = c_R$ in ;
     let $K = (\mathrm{PE}^{s_R} E_R)^{r_L}$ in
     let $k_s = \mathrm{H}(0^{32}, K)$ in
     let $s_s = s_L + s_R$ in
     let $k_c = kcf(k_s, \text{'SAE'}, s_s)$ in
     let $k = pmk(k_s, \text{'SAE'}, s_s)$ in
     $\nu i_L$.  // send-confirm
     let $H_L = \mathrm{CN}(k_c, i, c_L, c_R)$ in
     out$(c, H_L, i_L)$;in$(c, H_R, i_R)$;
     out$(c, \mathrm{enc}(k, m))$;
```

We highlight the symmetric nature of the protocol, letting both processes write to the channel before reading from it.

The π-calculus code outside of the box serves to model two aspects: first, to implement the usage of the same password element PE; and second, to model the usage of the key k after the key exchange for secrecy properties.

The two parties. We call one the Leftmost, L, and the other the Rightmost, R, for the convenience of naming in our model; however, we stress that given its symmetric structure, the protocol can be initiated by any device.

The pre-shared password. A table t_p of passwords is filled with all password elements PE that would be calculated by the participants before engaging the protocol, i.e., $\mathrm{PE} \in \mathbb{G}$ is the secret group generator for L and R implemented with finite field cryptography.

The main process. Due to its symmetric nature, any peer can initiate the protocol, so the implementation could just be a single self-composed process that will be used by both parties. Unfortunately, such a naive, direct model of the SAE protocol would easily produce *false* attacks where an initiator would speak to itself. To avoid this unwanted behaviour, a solution is to explicitly support the session between the two parties. However, the session denoted as sID, is not private information and is known to the attacker. To model that, we simply push it to the insecure channel c, i.e., $\mathsf{out}(c, sID)$. Before calling the processes P_L and P_R, we allow an external source to establish which party runs P_L and which runs P_R, i.e., $\mathsf{in}(c,(L,R))$; doing this, we capture the ability of any honest party to engage with the protocol with either algorithm, whose structure is anyway mirrored. This choice requires us to make sure that the two parties would not engage in the protocol if they do not share the password element. For this reason, we also added an environment process P_P, which is in charge of inserting shared password elements into a table that will be accessed by P_L and P_R but cannot be accessed by the attacker.

Finally, the main process P that the tool checks has the following structure:

$$P \leftarrow P_P \mid ((\nu sID.\ \mathsf{out}(c,sID); \mathsf{in}(c,(L,R));\ (P_L(sID) \mid P_R(sID)))) \mid !P_A.$$

4.2.5.2 Models at the Device level in ASMETA

The ASMETA framework allowed us to model the state machines of the PP and the PIs as ASMs. They are an extension of Finite State Machines where unstructured control states are replaced by mathematical *algebras* and state transitions are expressed in terms of *transition rules*. As an example in Fig. 4.11, we show the ASM rule of the PI model[8], which causes the transition from state Nothing to state Committed upon receiving the event Init. By changing the state, the PI sends the *commit* message and stats the timer `T0`.

Writing such transition rules from the documentation was straightforward since the ASMs are a state-based formal method and have the same computation paradigm of a state machine (used to describe the PI's behaviour in the standard). However, many inconsistencies and incompleteness were found

[8]All the PIs execute the same machine and each PI identifies *self* as itself.

in the official documentation. The complete final model has been obtained by a model refinement process. Building scenarios to validate the requirements of the standard was useful in finding inconsistencies and errors in our model. Our first model incorporated all three agents (SME, PP and PI) and this model focused on understanding the agents' interaction and learning whether the rule sets defined for each agent meet the requirements. Upon the initial phase of model validation, when we were confident enough of its correctness and completeness, we verified our model by using AsmetaSMV. For verification purposes, the SME model was not considered. since this agent does not consume events produced by the other agents, the contribution of this machine is irrelevant for verification purposes. We modelled only the reception of SME events by the PP.

Figure 4.11: Step from Nothing to Committed.

```
rule r_Nothing_State_INIT=
if (state(self)=NOTHING) then
  if (PI_Event(self)=INIT) then
    par
      message_to_peer_COMMIT(self):=true
      start_Timer(self,TIMER0):=true
      state(self):=COMMITTED
      ...
    endpar
  endif
endif
```

Simulation and verification of the two ASMs allowed us to obtain essential insight into execution paths and discover some attacks. Since security and authentication properties have been handled by verification at the Communication level, where the messages' content is thoroughly checked, during our analysis of the agents' machines, we focused on verifying *safety* (bad configurations do not happen), *reachability* (desired configuration are reachable) and *deadlock-free* (it is always possible to exit from given configurations) properties. These properties are expressed by the CTL formula $AG(\phi)$, meaning that the condition ϕ must always be globally true. Additional CTL formulas have been developed to assess corner cases. These formulas include the evaluation of *edge safety* properties that check for agents' failure to receive events when an agent is running low on RAM or receiving events from multiple agents simultaneously.

4.2.6 Findings/Results

As discussed in Section 4.2.4, we created models to capture the specification at two different levels - the Communication level, which is analysed through the lens of a Dolev-Yao attacker [43], and the Device level, which investigates the safety and completeness of the machine states of the agents involved in the protocol.

As expected, both modelling efforts could capture different aspects of the WPA3-SAE protocol, detailed in Section 4.2.6.1, as well as some common aspects, detailed in Section 4.2.6.2. Interestingly, combining insights from both verification tasks allowed us to refine one or the other model to better understand the nature of the vulnerabilities we could find and reproduce. Consequently, findings in the verification in one level suggested ways to patch issues in the other level; for example, from the partial correctness at the Device level, we could suggest how to patch the state machine modelling the Communication level, something greatly useful for applying the most appropriate patch to a vulnerability. We detail these *combined* findings in Section 4.2.6.3.

Table 4.4 reports the classes of checked properties and the verification results. In the following, only a relevant sample of these properties is presented.

Table 4.4: Summary of results on formal verification of security properties in ProVerif (left) and ASMETA (right).

	Communication level						*Device level*			
	CO	**SA**	**WA**	**SK**	**SPE**	**PFS**	**SF**	**DL**	**ES**	**RC**
IEEE 802.11:2020	◒	○	○	●	●	●	○	○	○	○
patched models	●	◓	◓	●	●	●	●	●	●	●

Formal verification results of our models on IEEE 802.11 and our patches.
Legend. Correctness (CO), Strong/Weak authentication (SA)/(WA), Key secrecy (SK), Password element secrecy (SPE), Perfect Forward Secrecy (PFS); Safety (SF), Deadlock (DL), Edge Safety (ES), Reachability (RC). **Outcomes**: (●) - verified, (◒) - partly verified, (○) - attacks found, (◓) - no attacks found, yet cannot be proved.

4.2.6.1 Disjoint findings: Replay Attack and Deadlock

4.2.6.1.1 Replay attack

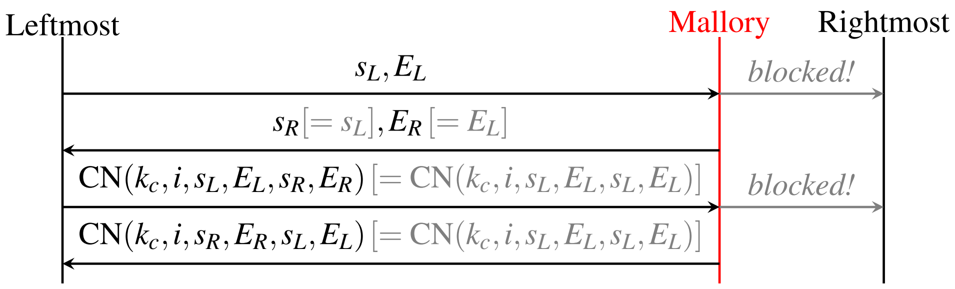
The verification of the design of WPA3-SAE in the latest standard [54, §12.4] highlighted a vulnerability in the logical flow that led to breaking its authentication in our model.

On the basis of the vanilla code illustrated in Fig. 4.10, authentication of L in R (mirroring the process L) can be captured through the artificial injection of two events, e_i in P_L and e_f in P_R. Informally, e_i signifies the belief of L of having started an authentication process with R, and e_f the belief of R of having terminated an authentication process that must have been started by L, i.e., R believes that it is communicating to a genuine L. The standard way to capture authentication in the symbolic model is through verifying a correspondence of events in the traces of executions. In particular, if for all the traces t of the symbolic execution of P (defined in Section 4.2.5.1), the presence of e_f in t is *always* after a *single* presence of e_i in t, then we have verified authentication[9]. This is called a correspondence of events and can be formally described as

$$\forall t \in T.\ e_f \in t \Rightarrow !\exists e_i \in t.$$

The formal tool ProVerif [26] is able to reconstruct the flow of an attack as a counterexample of the property described above. By inspecting the reconstruction, we noticed that it is *simply* carried through blocking commit and confirm messages from L to R, when L initiates the protocol, and finally reflecting back to L its own messages. Fig. 4.12 shows the mathematical operations for which a simple replay of messages is (mathematically) acceptable.

Figure 4.12: Replay attack at the design level specification of the WPA3-SAE protocol in IEEE 802.11:2020, the variant that uses finite field cryptography.


Leftmost
Mallory
Rightmost
s_L, E_L
blocked!
$s_R [= s_L], E_R [= E_L]$
$\mathrm{CN}(k_c, i, s_L, E_L, s_R, E_R)\ [= \mathrm{CN}(k_c, i, s_L, E_L, s_L, E_L)]$
blocked!
$\mathrm{CN}(k_c, i, s_R, E_R, s_L, E_L)\ [= \mathrm{CN}(k_c, i, s_L, E_L, s_L, E_L)]$

The patch to this specific attack would be that of discarding messages with the same s_L, E_L or both. In our model, we can add the guard if $c_L \neq c_R$ then just after line 7 of the model of P_L described in Fig. 4.10 (and likewise for

[9]For mutual authentication, we need to inject other two analogous events inverting the processes.

P_R which is omitted). That guard would allow the rest of the model to run and would stop in the case of a replay, i.e., $c_L = c_R$. With our patch, the same verification in ProVerif cannot find any attack on authentication any more.

An interesting note is that the specification of the state machine [54, §12.4.8.6.4] ignores replayed commit messages, de facto evidencing awareness about this attack. However, by deviating from the protocol specification, the state machine compromises authentication. It becomes evident that our comprehensive verification approach across multiple levels plays a crucial role in ensuring alignment with the intended protocol behaviour, thereby reducing the risk of insecure and buggy implementations.

4.2.6.1.2 Deadlock

Requirements in the WPA3 SAE standard include safety properties stated in natural language that can be translated into CTL (safety) properties and checked on the ASM model to ensure its robustness and correctness. E.g., the specification states that:

> For any given peer identity, there shall be only one protocol instance in the Committed or the Confirmed states.
>
> §12.4.8.6.1

This is expressed by the following CTL formula stating that globally (AG) does not exist a state where two PIs (pi_1 and pi_2) have the same *mac* (i.e., the same identity) but incompatible *state* according to the specifications.

$$
\begin{aligned}
AG\Big(\neg(\mathrm{mac}(pi_1) = \mathrm{mac}(pi_2) \wedge \\
&\big(\quad (\mathrm{state}(pi_1) = \mathsf{Committed} \wedge \mathrm{state}(pi_2) = \mathsf{Confirmed}) \\
&\vee\ (\mathrm{state}(pi_1) = \mathsf{Confirmed} \wedge \mathrm{state}(pi_2) = \mathsf{Committed}) \\
&\vee\ (\mathrm{state}(pi_1) = \mathsf{Confirmed} \wedge \mathrm{state}(pi_2) = \mathsf{Confirmed}) \\
&\vee\ (\mathrm{state}(pi_1) = \mathsf{Committed} \wedge \mathrm{state}(pi_2) = \mathsf{Committed})\ ))\Big)
\end{aligned}
$$

Since this property requires checking the internal states of multiple PIs, it was verified against the PP model since PP has information on the current states of the PIs. The result of the property detects no attacks.

However, the standard documentation does not mention other relevant properties, such as the absence of deadlock, necessary to guarantee the completeness of error-handling cases. During the analysis of the absence of deadlock in the Committed state, we discovered, through the following failing property, which states that does not exist a configuration where the PI (*pi*) is in *state* Committed and an error occurs (*fail* holds),

$$AG(\neg(\text{state}(pi) = \mathsf{Committed} \land fail))$$

that the standard does not handle the error in case of receiving a wrong commit message. This not-handled error causes the PI's state machine to deadlock, with no chance for the PI to be deallocated by the PP.

The standard specifies that a timer `T0` could have mitigated the deadlock by sending a Del event when it expired. However, the timer is deactivated when the PI is in the Committed state, and it checks the content of the Commit message, preventing the attack mitigation.

The deadlock is even more severe due to the safety property mentioned above because the attack can be easily executed by a malicious agent. Indeed, as a result of this deadlock, the PP is unable to create any new instances with the same MAC address as the one associated with the deadlocked PI. As a result, this may lead to a Denial of Service (DoS) attack on the affected peer, making it unable to connect to the network. The attacker could also exploit this vulnerability to cause a peer to run out of memory, as it is unable to deallocate all instances that ended up in the aforementioned error state. The last attack exhibits a similar pattern to the one described by [84], but it exploits a different vulnerability. The attack can be easily solved by adding, in the state machine, two transitions from the Committed state to the Nothing state. These transitions must involve sending a Del event to the PP to signal it to deallocate the PI.

4.2.6.2 Common findings: Correctness violation and Stall on bad password identifiers

Both the analyses of the two levels could model common issues with a novel feature that has been introduced in the revision 2020 of IEEE 802.11 [54]: the possibility of using multiple passwords. This change is meant to enhance security by resisting dictionary attacks, providing (selective) forward secrecy, allowing user flexibility, preventing credential sharing, facilitating secure connections for IoT devices, adapting to evolving threats, and improving overall usability. To enact this, an SAE entity can *require* its peer to use a specific password by sending a password identifier:

> If the peer's SAE Commit message contains a password identifier, the value of that identifier shall be used in the construction of the password element for this exchange.
>
> §12.4.5.4

And failure if no password maps from that password identifier. We could model the specification of correctness (from slightly different angles) in all the analysis levels, Communication and Device levels. In both, we found that correctness was violated: on one hand, the formal verification at the protocol design level showed a case where peers would use a different password and, thus, cannot complete the protocol; on the other hand, the formal verification of the safety property on the state machine showed that an unrecognised password identifier would lead to a stall where, again, the protocol could not complete.

4.2.6.2.1 Correctness violation

We implemented the exchange of password identifiers and their specified behaviour in ProVerif and found that this new feature breaks the security property of correctness in some unhandled cases. On the basis of the code illustrated in Fig. 4.10, we modified line 6 of P_L to $\mathsf{out}(c,(c_L,p_L));\mathsf{in}(c,(c_R,p_R))$, where p_L and p_R are the requested password identifiers. An analogous change is made for P_R. We have three cases: if $p_R = \perp$, a password identifier is not requested; if $p_R \neq \perp$ but p_R is not valid, then we abort; otherwise, if p_R is a valid identifier, the password element PE' depending on p_R is used accordingly. This behaviour is captured by the following code:

```
if $p_R \neq \perp$ then
    get $t_p(=L,=R,=p_R,\mathrm{PE}')$ in          fails if $p_R$ not found
    let $E_L = \mathrm{PE}'^{-m_L}$ in              recalculate password element
    out$(c,(c_L,p_L))$;                             re-commit
```

where a missing entry for an invalid p_R in t_p for L and R would automatically fail. Then, correctness is captured as a reachability property of an event e_{CORR} at the end of the protocol that includes the exchanged key k. The two processes P_L and P_R would write their entity names, the session and the exchanged key, each in their own table, t_L and t_R respectively, that cannot be read by the adversary but it is shared among processes. Then we instantiate the additional process P_A with an event that e_{CORR} that collects information from both tables. Formally, we first inject the table writing operation

$$\text{insert } t_L(L,R,s,k) \qquad \text{and} \qquad \text{insert } t_R(R,L,s,k)$$

just after line 7 of the model of P_L described in Fig. 4.10. Then, for all sessions s, keys k, we require that the reasoning core is able to show a trace t where the event e_{CORR} is recorded and is such that two honest participants agree on their identities, the password, and the pairwise master key k.

$$\forall\, s,k.\ \exists\, t.\ e_{\text{CORR}}(A,B,s,k,A,B,s,k) \in t.$$

where A and B are (the only) honest parties, s is the (common) session ID, and k is the pairwise master key. If such an event is reached, i.e., e_{CORR} is found in t, then there exists a run of the protocol in which the two parties have authenticated each other and they have correctly exchanged the same session key. This formalism requires us to hardcode all different cases independently to make a comprehensive analysis, so if some cases but not all are verified, we quickly say that correctness is *partly verified*.

ProVerif is unable to reach e_{CORR}, thus violating the security property of correctness, in the case when both P_L and P_R require the usage of a specific password through a valid but different password identifier. In such a case, the password element PE will be different between the two peers, and they will not be able to verify the peer confirmation messages. The situation can be remarkably relevant in meshes, where the possibility of two peers initiating simultaneously can be frequent, and about which WPA3-SAE is particularly focussed:

> SAE shall be implemented on all mesh STAs to facilitate and promote interoperability.
>
> §12.4.1

In practice, meshes treat all peers uniformly in the network, yet certain peers might possess multiple passwords tied to distinct profiles that they can seamlessly switch between. Again, we notice a deviation of the state machine to the protocol specification, as two peers starting the protocol with two different password identifiers would fail the authentication. Even if this *seems* just a patch in the wrong place, it is not and that is why: first of all it is coincidental, as there is no justification for such a deviation, but more importantly both peers would act legitimate and simply see the authentication failing, so they would just try again. In other words, there is no mechanism to prevent both peers from attempting the (failed) authentication repeatedly. Additionally, as the map of password identifiers is *not bijective*, they could have mapped to the same password allowing the protocol to end correctly.

We can patch correctness requesting that, upon reception of the password identifier, a peer would *not* reply with a different one. In the case when both peers initiate and request for a different password identifier that is valid, then

- the peer with numerically lesser identity (MAC) shall accept the requested password identifier and regenerate the peer-commit-element accordingly, and

- the peer with numerically greater identity (MAC) shall ignore the requested password identifier.

With this patch, the problematic case can be formally verified for correctness. It is important to notice that this solution may not be ideal, depending on the reason behind clients sharing multiple passwords, but generally speaking it is pointless to deny both connections. This aspect of the semantics of multiple passwords goes beyond fixing correctness at the design level and requires further analysis that is not captured by the formalisation efforts in our work.

Finally, we stress that our model derives from an intuitive interpretation of missing specifications on how to handle password identifiers at the design level. In Section 4.2.6.3, we will examine how the outlined specifications, supported by formal verification, address the missing specifications.

4.2.6.2.2 Stall on bad password identifiers

In the documentation of the standard, the description of the multiple branches of the state machine capturing the WPA3-SAE protocol behaviour is given in natural language and leaves out important details. This can result in ambiguity and interpretation, leading to serious vulnerabilities or attacks during implementation. For instance, when we modelled the PI state machine with ASMETA, we discovered an attack that was caused by unclear instructions on how the principal PI should handle an error message. The vulnerability lurks in the protocol requirement describing a case of failure:

> If so, and there is no password associated with that identifier, *BadID* will be set, and the protocol instance will construct and transmit an authentication frame with StatusCode set to *UNKNOWN_PASSWORD_IDENTIFIER*
>
> §12.4.8.6.3

However, this requirement infers the following deadlock-free property:

$$AG(\neg(\mathsf{state}(pi) = \mathsf{Nothing} \wedge \mathsf{event}(pi) \neq \mathsf{Del}))$$

indeed, in case of a failure, PIs (pi) (in the state Nothing) should always request to be deallocated by the PP by sending the termination *event* Del. Failing, the property returns a trace where the PI remains stuck in the Nothing state. As shown by the requirement, the Del event is missing, and this leads the PI to become unresponsive, and the PP can no longer remove it.

This attack, in turn, leads to the violation of additional safety properties in the PP model. More precisely, the standard in a note states:

> NOTE—A protocol instance in Nothing state will never receive an SAE Confirm message due to the state machine behaviour of the parent process
>
> §12.4.8.6.3

However, by analysing the behaviour of the PI state machine when it is stalled, we discovered that the PP can send the `CON` event to the PI that is waiting in the Nothing state (this is because the PP only checks the presence of a PI with the correct MAC but does not check the PI state). The PI does not handle the `CON` in the Nothing state, and this could further aggravate the situation as unhandled severe exceptions could be triggered.

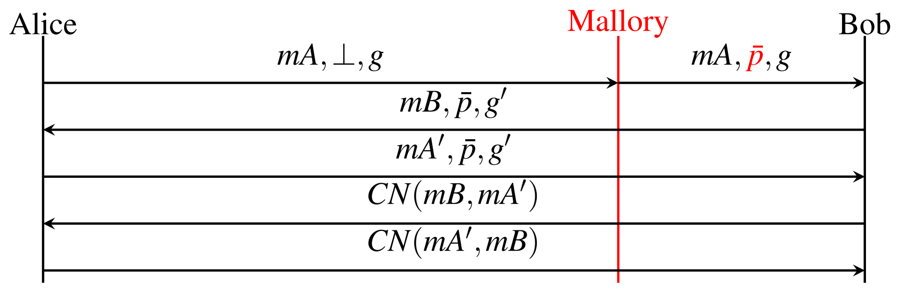
To ensure that the standard is safeguarded against possible future regressions, it would be best to include two *patches*: one for PP and one for PI. The *first patch* should require PP to not only check if the MAC associated with the one in the confirm message is present in its database but also verify the PI's state. The *second patch* (which, at the current state of the specification, patches both) would involve sending the Del event to PP, in addition to the frame with StatusCode set to `UNKNOWN_PASSWORD_IDENTIFIER` when the PI receives a message containing a password identifier not present in the table.

4.2.6.3 Unified results: Secure specification for password identifiers

At the Communication level, two peers are not forbidden to choose different password identifiers, but when the two password identifiers are different, the protocol will fail without exchanging a common key (correctness violation). Surprisingly, the analysis of the state machine highlighted that two peers are *not* allowed to choose different password identifiers (failure), contradicting the protocol specification. We discovered that the only states where a renewal of the commit message could have happened was the Commited. The main case of interest turned out to be the agreement on groups, as, if two peers initiated the protocol with different but supported groups, then one group would be selected by both on the basis of the natural ordering of their MAC address: in detail, the one with the lowest MAC has to re-commit and confirm, and the other has to ignore and wait for another commit message.

We leveraged this result back to our model at the Communication level to include group agreement according to the specification, as well as our *patch* to handle password identifiers, see Sec. 4.2.6.2. From the Communication perspective, we end up finding a *false* attack that seems to downgrade password identifiers. The Communication level model confirms the existence of

Figure 4.13: Attack on password identifier WPA3-SAE protocol in IEEE 802.11:2020 at the Communication level, that reveals to be a *false* attack when we analyse the Device level.



this attack, allowing two peers to start with different groups and password identifiers. However, the state machine actually implements a check on this, disallowing the start of the third message in Fig. 4.13.

This demonstrates that the flaw handled at the Device level (without a justification) is covering for a vulnerability affecting the Communication level, supporting the high number of misalignments in our Errata, see Appendix A. This is to exemplify that our unified approach allowed our formal verification to enjoy a deeper security analysis than traditional single-model verification. We argue that the correct design of the protocol (which is what our patch accounts for) should supersede the patching of the incorrect design at a different level. Additionally, the deeper analysis here sketched also suggests that two other important points have to be added to the Communication level specifications: i) despite both peers being allowed to start the protocol, when one receives a message before having sent its first message, it should consider itself a Receiver and act accordingly; and ii) a Receiver should always accept a supported group, never offer a different group, and terminate the protocol in a failure state, if an unsupported group is offered according to the expected rejection list of groups. *Our verification shows that with such specifications, the above attack will not occur.*

5 Conclusions and Future Work

Our ultimate goal is for all security protocol designers to be capable of producing formal proof of correctness. We view the user-friendly GUI, aided by the KANT language and the multi-level verification that combines the results of multiple back-ends, as a substantial step towards achieving this vision. In particular, we demonstrated the goodness of the analysis approach that combines the device level with the communication level by evaluating it on real-world security protocols with critical dimensions that could affect the lives of millions of people if not correctly designed. In the following, we highlighted the benefits of the APROVER framework.

The APROVER framework is designed to make it easier for people who are not familiar with formal methods to create their security protocols. Users can build their protocols from scratch by dragging and dropping elements to build the messages composing them. The framework comes with a graphical user interface that includes basic checks on the parameters used in cryptographic primitives, which helps to prevent major errors. For more experienced users, the framework includes the KANT language, a domain-specific language that allows for both syntactic and semantic validation queries. This semantic-level validation helps to identify and eliminate known vulnerabilities even before back-ends verify the protocol. By using this approach, users can have greater confidence in the protocol requirements and reduce the time it takes to verify the protocol. Once the protocol is modelled, it can be verified by back-ends. The APROVER framework provides both classical tools used in the literature to verify security protocols at the communication level and verification at the device level via the ASMETA framework and a library developed specifically for it. The library developed for ASMETA makes it possible to perform verification covering both the communication and device levels. The library for ASMETA is tested to formalise the Z-Wave protocol, proving capable of good performance (as shown in Table 4.2, 4.1, 4.3) in verification time and allowing us to discover a severe vulnerability in the protocol, confirmed by Silicon Labs.

However, a single verification method with its formalism constitutes a single view of the protocol to be analysed by capturing only part of the details expressed in the requirements of a real-world security protocol. To overcome this limitation, we have developed an approach to unifying results that takes full advantage of the expressive capabilities of the individual tools. The approach tested combines the use of PROVERIF and ASMETA (it can also be extended to TAMARIN) to test the communication and device levels, respectively. The results were compared separately, looking for similarities and creating a feedback loop in which the results of one tool provided additional details to be added to the other verification tool. The effectiveness of the approach has enabled us to discover 20 errata on the WPA3-SAE protocol (summarised in Table 4.4), which the IEEE 802.11 standard committee has confirmed, and fixes are in the process of being integrated.

In the future, the APROVER framework will be improved with a focus on fully automated verification. The aim is to enhance the user experience by providing feedback via metadata collected during the modelling phase. Additionally, the goal is to generate code for implementing the security protocol based on the verified requirements. We plan to conduct further usability tests to target a broader audience and make the framework more accessible to those without a background in cybersecurity. These tests will ensure the tool is user-friendly and suitable for a broader range of users. Furthermore, we intend to continue testing the framework with various security protocols, even though we have already demonstrated its use on complex protocols such as WPA3-SAE and Z-Wave. This will enable us to expand the support for cryptographic primitives and implement new security property patterns.

Bibliography

[1] M. Abadi and R. Needham. “Prudent engineering practice for cryptographic protocols”. In: *IEEE Transactions on Software Engineering* 22.1 (1996), pp. 6–15. DOI: 10.1109/32.481513.

[2] Martin Abadi and Cedric Fournet. “Mobile Values, New Names, and Secure Communication”. In: *Sigplan Notices - SIGPLAN*. Vol. 36. Mar. 2001. ISBN: 1-58113-336-7. DOI: 10.1145/360204.360213.

[3] Roberto M. Amadio and Witold Charatonik. “On Name Generation and Set-Based Analysis in the Dolev-Yao Model”. In: *CONCUR 2002 — Concurrency Theory*. Ed. by Luboš Brim et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 499–514. ISBN: 978-3-540-45694-0.

[4] Paolo Arcaini, Angelo Gargantini and Elvinia Riccobene. “AsmetaSMV: a way to link high-level ASM models to low-level”. In: Nov. 2010, pp. 61–74. ISBN: 978-3-642-11810-4. DOI: 10.1007/978-3-642-11811-1_6.

[5] Paolo Arcaini, Angelo Gargantini and Elvinia Riccobene. “Automatic Review of Abstract State Machines by Meta-Property Verification”. In: (Jan. 2010).

[6] Paolo Arcaini, Angelo Gargantini and Elvinia Riccobene. “CoMA: Conformance Monitoring of Java Programs by Abstract State Machines”. In: *Runtime Verification*. Ed. by Sarfraz Khurshid and Koushik Sen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 223–238. ISBN: 978-3-642-29860-8.

[7] Paolo Arcaini, Angelo Gargantini and Elvinia Riccobene. “Rigorous development process of a safety-critical system: from ASM models to Java code”. In: *Int. J. Softw. Tools Technol. Transf.* 19.2 (2017), pp. 247–269. DOI: 10.1007/s10009-015-0394-x.

[8] Paolo Arcaini et al. “A model-driven process for engineering a toolset for a formal method”. In: *Software: Practice and Experience* 41.2 (2011). Publisher: John Wiley & Sons, Ltd., pp. 155–166. DOI: 10.1002/spe.1019.

[9] Alessandro Armando, Roberto Carbone and Luca Compagna. “SATMC: A SAT-Based Model Checker for Security-Critical Systems”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 31–45. ISBN: 978-3-642-54862-8.

[10] Alessandro Armando et al. “Avispa: automated validation of internet security protocols and applications”. In: *ERCIM News* 64 (January 2006).

[11] Alessandro Armando et al. “The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures”. In: Mar. 2012, pp. 267–282. ISBN: 978-3-642-28755-8. DOI: 10.1007/978-3-642-28756-5_19.

[12] Alessandro Armando et al. “The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications”. In: *Proc. of Int. Conference on Computer Aided Verification*. Springer-Verlag, 2005, pp. 281–285. DOI: 10.1007/11513988_27.

[13] Luca Arnaboldi and Roberto Metere. “Poster: Towards a data centric approach for the design and verification of cryptographic protocols”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 2585–2587.

[14] Linard Arquint et al. “A generic methodology for the modular verification of security protocol implementations”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 1377–1391.

[15] *ASMETA (ASM mETAmodeling) toolset*. URL: https://asmeta.github.io/.

[16] Christopher W. Badenhop et al. “The Z-Wave routing protocol and its security implications”. In: *Computers & Security* 68 (2017), pp. 112–129. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2017.04.004.

[17] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[18] David Basin, Cas Cremers and Catherine Meadows. "Model Checking Security Protocols". In: *Handbook of Model Checking*. May 2018, pp. 727–762. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8_22.

[19] David Basin et al. "A Formal Analysis of 5G Authentication". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. event-place: Toronto, Canada. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1383–1396. DOI: 10.1145/3243734.3243846.

[20] David Basin et al. "Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols". In: *IEEE Security & Privacy* 20.3 (2022). Publisher: IEEE, pp. 22–31.

[21] Noomene Ben Henda. "Generic and Efficient Attacker Models in SPIN". In: *Proc. of Int. SPIN Symposium on Model Checking of Software*. Association for Computing Machinery, 2014, pp. 77–86. DOI: 10.1145/2632362.2632378.

[22] Nazim Benaissa and Dominique Méry. "Cryptographic Protocols Analysis in Event B". In: *Perspectives of Systems Informatics*. Springer Berlin Heidelberg, 2010, pp. 282–293. ISBN: 978-3-642-11486-1. DOI: 10.1007/978-3-642-11486-1_24.

[23] Karthikeyan Bhargavan et al. "Implementing TLS with verified cryptographic security". In: *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 445–459.

[24] B. Blanchet. "An efficient cryptographic protocol verifier based on Prolog rules". In: *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.* 2001, pp. 82–96. DOI: 10.1109/CSFW.2001.930138.

[25] Bruno Blanchet. "Security Protocol Verification: Symbolic and Computational Models". In: *Principles of Security and Trust*. Ed. by Pierpaolo Degano and Joshua D. Guttman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–29. DOI: 10.1007/978-3-642-28641-4_2.

[26] Bruno Blanchet, Vincent Cheval and Véronique Cortier. "ProVerif with lemmas, induction, fast subsumption, and much more". In: *IEEE Symposium on Security and Privacy (S&P'22)*. San Francisco, CA: IEEE Computer Society, May 2022, pp. 205–222.

[27] Bruno Blanchet et al. *ProVerif 2.04: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. 2021. URL: https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf.

[28] Yohan Boichut et al. "Using Animation to Improve Formal Specifications of Security Protocols". In: *The 2nd National Conference on Security in Network Architectures and Information Systems*. France, 2007. URL: https://hal.science/hal-00468718.

[29] Silvia Bonfanti, Angelo Gargantini and Atif Mashkoor. "Design and validation of a C++ code generator from Abstract State Machines specifications". In: *J. Softw. Evol. Process.* 32.2 (2020). DOI: 10.1002/smr.2205.

[30] Egon Börger and Alexander Raschke. *Modeling Companion for Software Practitioners*. Springer, 2018. DOI: 10.1007/978-3-662-56641-1.

[31] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003. DOI: 10.1007/978-3-642-18216-7.

[32] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3-642-07716-1.

[33] Chiara Braghin, Mario Lilli and Elvinia Riccobene. "A model-based approach for vulnerability analysis of IoT security protocols: The Z-Wave case study". In: *Comput. Secur.* 127 (2023), p. 103037. DOI: 10.1016/J.COSE.2022.103037. URL: https://doi.org/10.1016/j.cose.2022.103037.

[34] Chiara Braghin, Mario Lilli and Elvinia Riccobene. "Kant: A Domain-Specific Language for Modeling Security Protocols". In: *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering, MODELSWARD 2024, Rome, Italy, February 21-23, 2024*. SCITEPRESS, 2024.

[35] Chiara Braghin, Mario Lilli and Elvinia Riccobene. "Towards ASM-Based Automated Formal Verification of Security Protocols". In: *Rigorous State-Based Methods*. Ed. by Alexander Raschke and Dominique Méry. Springer International Publishing, 2021, pp. 17–33. DOI: 10.1007/978-3-030-77543-8_2.

[36] Michael Burrows, Martin Abadi and Roger Needham. “A Logic of Authentication”. In: *ACM Trans. Comput. Syst.* 8.1 (Feb. 1990). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 18–36. DOI: `10.1145/77648.77649`.

[37] Alessandro Carioni et al. “A Scenario-Based Validation Language for ASMs”. In: Sept. 2008, pp. 71–84. ISBN: 978-3-540-87602-1. DOI: `10.1007/978-3-540-87603-8_7`.

[38] Vincent Cheval et al. *Sapic+: protocol verifiers of the world, unite!* Published: Cryptology ePrint Archive, Paper 2022/741. 2022. URL: `https://eprint.iacr.org/2022/741`.

[39] Thierry Coquand and Gérard Huet. “The calculus of constructions”. In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(88)90005-3`. URL: `https://www.sciencedirect.com/science/article/pii/0890540188900053`.

[40] Cas Cremers et al. “Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 470–485. DOI: `10.1109/SP.2016.35`.

[41] Cjf Cas Cremers. “Scyther : semantics and verification of security protocols”. In: 2006. URL: `https://api.semanticscholar.org/CorpusID:221206725`.

[42] Jennifer A. Davis et al. “Study on the Barriers to the Industrial Adoption of Formal Methods”. In: *Formal Methods for Industrial Critical Systems*. Ed. by Charles Pecheur and Michael Dierkes. Springer Berlin Heidelberg, 2013, pp. 63–77. DOI: `10.1007/978-3-642-41010-9_5`.

[43] D. Dolev and A. Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: `10.1109/TIT.1983.1056650`.

[44] Nancy Durgin, Patrick Lincoln and John Mitchell. “Multiset Rewriting and the Complexity of Bounded Security Protocols”. In: *Journal of Computer Security* 12 (Feb. 2004), pp. 247–311. DOI: `10.3233/JCS-2004-12203`.

[45] Mohamed Eldefrawy et al. “Formal security analysis of LoRaWAN”. In: *Computer Networks* 148 (2019), pp. 328–339. DOI: `10.1016/j.comnet.2018.11.017`.

[46] B. Fouladi and S. Ghanoun. “Security evaluation of the Z-Wave wireless protocol”. In: *Proceedings of Black Hat USA*. 2013, pp. 1–2.

[47] Angelo Gargantini, Elvinia Riccobene and Salvatore Rinzivillo. “Using Spin to Generate Tests from ASM Specifications”. In: *Abstract State Machines 2003*. Ed. by Egon Börger, Angelo Gargantini and Elvinia Riccobene. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 263–277. ISBN: 978-3-540-36498-6.

[48] Angelo Gargantini, Elvinia Riccobene and Patrizia Scandurra. “A Metamodel-based Language and a Simulation Engine for Abstract State Machines”. In: *J. UCS* 14 (Jan. 2008), pp. 1949–1983.

[49] Feng Hao et al. “Analyzing and patching speke in iso/iec”. In: *IEEE Transactions on Information Forensics and Security* 13.11 (2018). Publisher: IEEE, pp. 2844–2855.

[50] Robert Heinrich et al. “Integration and Orchestration of Analysis Tools”. In: *Composing Model-Based Analysis Tools*. 2021, pp. 71–95. DOI: `10.1007/978-3-030-81915-6_5`. URL: `https://doi.org/10.1007/978-3-030-81915-6%5C_5`.

[51] Katharina Hofer-Schmitz and Branka Stojanović. “Towards formal verification of IoT protocols: A Review”. In: *Computer Networks* 174 (2020), pp. 107–233. DOI: `10.1016/j.comnet.2020.107233`.

[52] Philip K. Hooper. “The undecidability of the Turing machine immortality problem”. In: *Journal of Symbolic Logic* 31.2 (1966). Publisher: Cambridge University Press, pp. 219–234. DOI: `10.2307/2269811`.

[53] “IEEE Standard for Information technology–Telecommunications and information exchange between systems - Local and metropolitan area networks–Specific requirements -”. In: *IEEE Std 802.11ah-2016 (Amendment to IEEE Std 802.11-2016, as amended by IEEE Std 802.11ai-2016)* (2017), pp. 1–594. DOI: `10.1109/IEEESTD.2017.7920364`.

[54] “IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks–Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”. In: *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)* (2021), pp. 1–4379. DOI: `10.1109/IEEESTD.2021.9363693`.

[55] Florent Jacquemard, Michaël Rusinowitch and Laurent Vigneron. “Compiling and Verifying Security Protocols”. In: *Logic for Programming and Automated Reasoning*. 2000, pp. 131–160. ISBN: 978-3-540-44404-6.

[56] Kyounggon Kim et al. “What’s your protocol: Vulnerabilities and security threats related to Z-Wave protocol”. In: *Pervasive and Mobile Computing* 66 (July 2020). Publisher: Elsevier. ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2020.101211.

[57] Nadim Kobeissi, Georgio Nicolas and Mukesh Tiwari. “Verifpal: Cryptographic Protocol Analysis for the Real World”. In: *Progress in Cryptology – INDOCRYPT 2020*. Springer International Publishing, 2020, pp. 151–202. ISBN: 978-3-030-65277-7.

[58] Steve Kremer and Robert Künnemann. “Automated Analysis of Security Protocols with Global State”. In: *Journal of Computer Security* 24 (Aug. 2016), pp. 1–34. DOI: 10.3233/JCS-160556.

[59] Mario Lilli. “Formal verification of Z-Wave protocol security properties”. Master’s Thesis. Italy: Università degli Studi di Milano, 2020.

[60] Mario Lilli, Chiara Braghin and Elvinia Riccobene. “Formal Proof of a Vulnerability in Z-Wave IoT Protocol”. In: *Proc. of the 18th Int. Conference on Security and Cryptography, SECRYPT 2021, July 6-8, 2021*. Ed. by Sabrina De Capitani di Vimercati and Pierangela Samarati. SCITEPRESS, 2021, pp. 198–209. DOI: 10.5220/0010553301980209.

[61] Mario Lilli et al. *Unified Security Analysis of WPA3 SAE Wireless Authentication*.

[62] G. Lowe. “A hierarchy of authentication specifications”. In: *Proceedings 10th Computer Security Foundations Workshop*. 1997, pp. 31–43. DOI: 10.1109/CSFW.1997.596782.

[63] Anthony J. Masys. *Security by Design: Innovative Perspectives on Complex Problems*. Springer International Publishing, 2018. ISBN: 978-3-319-78021-4.

[64] Ueli Maurer, Pierre Schmid and Omnisec Ag. “A Calculus for Security Bootstrapping in Distributed Systems”. In: *Journal of Computer Security* 4 (Aug. 2001). DOI: 10.3233/JCS-1996-4104.

[65] Simon Meier et al. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48.

[66] Anthony Patrick Melaragno et al. “Securing the ZigBee Protocol in the Smart Grid”. In: *Computer* 45.4 (2012), pp. 92–94. DOI: 10.1109/MC.2012.146.

[67] Robin Milner, Joachim Parrow and David Walker. “A calculus of mobile processes, I”. In: *Information and Computation* 100.1 (1992), pp. 1–40. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90008-4. URL: https://www.sciencedirect.com/science/article/pii/0890540192900084.

[68] Robin Milner, Joachim Parrow and David Walker. “A calculus of mobile processes, II”. In: *Information and Computation* 100.1 (1992), pp. 41–77. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90009-5. URL: https://www.sciencedirect.com/science/article/pii/0890540192900095.

[69] Sebastian Mödersheim. “Algebraic Properties in Alice and Bob Notation”. In: *Int. Conf. on Availability, Reliability and Security*. 2009, pp. 433–440. DOI: 10.1109/ARES.2009.95.

[70] Nina Moebius, Kurt Stenzel and Wolfgang Reif. “Generating formal specifications for security-critical applications - A model-driven approach”. In: *2009 ICSE Workshop on Software Engineering for Secure Systems*. 2009, pp. 68–74. DOI: 10.1109/IWSESS.2009.5068461.

[71] M. Mohsin et al. “IoTRiskAnalyzer: A Probabilistic Model Checking Based Framework for Formal Risk Analytics of the Internet of Things”. In: *IEEE Access* 5 (2017), pp. 5494–5505. DOI: 10.1109/ACCESS.2017.2696031.

[72] Kevin Morio et al. “Modular black-box runtime verification of security protocols”. In: *PLAS 2020* (2020).

[73] A. N.A.DurginP.D.LincolnJ.C.Mitchell and ScedrovComputer. “Undecidability of bounded security protocols”. In: 1999. URL: https://api.semanticscholar.org/CorpusID:18473344.

[74] Taewoo Nam. “Understanding the gap between perceived threats to and preparedness for cybersecurity”. In: *Technology in Society* 58 (2019), p. 101122. ISSN: 0160-791X. DOI: https://doi.org/10.1016/j.techsoc.2019.03.005. URL: https://www.sciencedirect.com/science/article/pii/S0160791X18301179.

[75] Y. Qiu and M. Ma. “Secure Group Mobility Support for 6LoWPAN Networks”. In: *IEEE Internet of Things Journal* 5.2 (2018), pp. 1131–1141. DOI: 10.1109/JIOT.2018.2805696.

[76] Yue Qiu and Maode Ma. “A PMIPv6-Based Secured Mobility Scheme for 6LoWPAN”. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. 2016, pp. 1–6. DOI: 10.1109/GLOCOM.2016.7841534.

[77] Ahmad-Reza Sadeghi, Christian Wachsmann and Michael Waidner. “Security and privacy challenges in industrial Internet of Things”. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2747942.

[78] Patrizia Scandurra et al. “A Concrete Syntax Derived From the Abstract State Machine Metamodel”. In: *Proceedings of the 12th International Workshop on Abstract State Machines, ASM 2005, March 8-11, 2005, Paris, France*. 2005, pp. 345–368. URL: http://www.univ-paris12.fr/lacl/dima/asm05/scandurra.ps.gz.

[79] Patrizia Scandurra et al. “Functional requirements validation by transforming use case models into Abstract State Machines”. In: *Proceedings of the ACM Symposium on Applied Computing* (Mar. 2012). DOI: 10.1145/2245276.2231942.

[80] Da-Zhi Sun and Li Sun. “On Secure Simple Pairing in Bluetooth Standard v5.0-Part I: Authenticated Link Key Security and Its Home Automation and Entertainment Applications”. In: *Sensors (Basel, Switzerland)* 19.5 (7th Mar. 2019). Publisher: MDPI, p. 1158. DOI: 10.3390/s19051158.

[81] Jonathan Tournier et al. “A survey of IoT protocols and their security issues through the lens of a generic IoT stack”. In: *Internet of Things* 16 (2021), p. 100264. DOI: 10.1016/j.iot.2020.100264.

[82] Ishaq Unwala, Zafar Taqvi and Jiang Lu. “IoT Security: ZWave and Thread”. In: *2018 IEEE Green Technologies Conference (GreenTech)*. 2018, pp. 176–182. DOI: 10.1109/GreenTech.2018.00040.

[83] Mathy Vanhoef and Frank Piessens. “Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2”. In: *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[84] Mathy Vanhoef and Eyal Ronen. “Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd”. In: *IEEE Symposium on Security & Privacy (SP)*. IEEE, 2020.

[85] Luca Viganò. “Automated Security Protocol Analysis With the AVISPA Tool”. In: *Electronic Notes in Theoretical Computer Science* 155 (2006), pp. 61–86. DOI: 10.1016/j.entcs.2005.11.052.

[86] Z-Wave Alliance. *Z-Wave and Z-Wave Long Range Network Layer Specification*. 2022. URL: https://sdomembers.z-wavealliance.org/document/dl/897.

[87] Z-Wave Alliance. *Z-Wave Protocol*. 2022. URL: https://www.z-wave.com/.

[88] Z-Wave Alliance. *Z-Wave Transport-Encapsulation Command Class Specification*. 2021. URL: https://sdomembers.z-wavealliance.org/document/dl/652.

A Errata Summary

The full Errata document is currently with IEEE and is being discussed at the WG in Panama on January 14th. A high level summary of our findings can be found below. Full details will be published for the final version.

The errata documents for IEEE Std 802.11:2020 contain twenty-four specific corrections and patches to the discovered vulnerabilities and errors. These include clarifications and modifications addressing issues such as password identifier handling in Mesh networks, management of rejected group lists, procedures when peers start with different groups, and roles in the SAE protocol. Other significant errata concern the use of Anti-Clogging Tokens, incomplete KDF formulas, misleading section titles, participant roles, the use of indicators, and event handling in the protocol's state machine. Each erratum details the original text, proposed corrections, and rationale, aiming to resolve ambiguities and potential vulnerabilities in the standard. These corrections are critical for the standard's accuracy and integrity. Below is a high level summary of all the proposed corrections:

1. **Password Identifier**: Addresses operations for Mesh connections and different password identifiers.

2. **List of Rejected Groups**: Clarifies handling of group element lists.

3. **Start with Different Groups**: Proposes random delays and group selection.

4. **Principal Role**: Removes 'SAE Initiator' for role clarity.

5. **Anti-Clogging Token**: Amends text for Open variable threshold cases.

6. **Incomplete KDF Formula**: Updates pseudocode for formula changes.
7. **Misleading Titles**: Reorganizes titles for accuracy.
8. **Conflation of Participants**: Clarifies the SME participant's role.
9. **Use of Indicators**: Discusses indicators' intended use or removal.
10. **Misleading SME Events**: Clarifies SME-related event descriptions.
11. **Non-existent Event**: Corrects Sync event naming.
12. **Declared but Never Used Event (Fail)**: Addresses 'Fail' event redundancy.
13. **Missing COM Event**: Clarifies COM event handling in state machine.
14. **Missing Del Event and State Transition**: Details post-authentication frame actions.
15. **Silent Deletion of Messages**: Proposes explicit error case handling.
16. **Algorithm Identifier Confusion**: Seeks clarity on 'algorithm identifier.'
17. **Rejection Frame Terminology**: Aims for naming consistency.
18. **Deterministic PWE**: Corrects procedure description.
19. **Probability of PT Value**: Modifies algorithm for PT=1 cases.
20. **Replay Attack Vulnerability**: Suggests protocol revisions for attack prevention.
21. **Different Password Identifiers/Groups**: Enhances handling logic.
22. **Missing Status Code**: Improves error message processing.

23. **Algorithm Identifier Issue**: Clarifies algorithm identifiers in frames.

24. **Del Event and State Transition**: Addresses error handling in the state machine.