Formal verification of the Internet Key Exchange (IKEv2) security protocol

Tristan Ninet

UNIVERSITÉ DE RENNES 1

Thèse de doctorat de

L'UNIVERSITÉ DE RENNES 1

École Doctorale N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Tristan NINET

Formal verification of the Internet Key Exchange (IKEv2) security protocol

Thèse présentée et soutenue à Rennes, le 9 mars 2020
Unité de recherche : Inria

Rapporteurs avant soutenance :

Caroline FONTAINE	Directrice de Recherche, CNRS, LSV / ENS Paris-Saclay
Steve KREMER	Directeur de Recherche, Inria, LORIA Nancy

Composition du Jury :

Président :	Steve KREMER	Directeur de Recherche, Inria, LORIA Nancy
Examinateurs :	Pierre-Etienne MOREAU	Professeur, Université de Lorraine, École des Mines de Nancy
	Thomas GENET	Maître de conférence, Université de Rennes 1
Dir. de thèse :	Stéphanie DELAUNE	Directrice de Recherche, CNRS, IRISA Rennes
Co-dir. de thèse :	Olivier ZENDRA	Chargé de Recherche, Inria Rennes

Invité(s) :

Romaric MAILLARD	Architecte Equipements, Thales SIX GTS France

Acknowledgment

Je tiens à remercier Olivier, pour m'avoir accompagné du (presque) début à la fin, et en particulier pour avoir toujours su trouver les mots pour me rassurer et me remotiver.
J'adresse également toute ma gratitude à Stéphanie, pour avoir accepté de reprendre ma thèse, et sans qui la qualité des travaux accomplis ne serait pas ce qu'elle est.
Je remercie bien sûr Romaric et Frédéric pour m'avoir accueilli au sein de Thalès. Merci à toi Romaric, pour ton investissement et pour toutes ces discussions qui m'ont tant appris. Merci Frédéric, pour m'avoir fait confiance et pour m'avoir laissé toute la liberté de travailler sur ma thèse.
J'adresse également ma reconnaissance aux rapporteurs, Caroline Fontaine et Steve Kremer, pour avoir pris le temps de lire et corriger le manuscrit, et plus largement aux membres du jury, dont Pierre-Etienne Moreau et Thomas Genet, pour l'intérêt que vous avez porté à mes travaux.
Je remercie du fond du cœur toute l'équipe TAMIS. Tous nos moments passés ensemble ont été à la fois passionnants et réconfortants. Tout cela m'a été indispensable.
Je remercie enfin ma famille et mes amis pour leur soutien sans faille, et plus particulièrement Tita, pour avoir traversé la France en train, malgré la montée du COVID-19 en France, afin d'assister à la soutenance de ton petit-fils.

Résumé

Cette thèse se situe au croisement de deux domaines de l'informatique. Le premier domaine est celui de l'architecture réseaux IPsec et son protocole d'échange de clés IKEv2, et le second domaine est la vérification formelle de protocoles de sécurité. Cette thèse vise à appliquer à IKEv2 les dernières avancées en vérification formelle.

Dans ce résumé, nous présentons d'abord le contexte de cette thèse, puis nous introduisons le lecteur aux protocoles de sécurité et à leurs potentielles attaques, puis nous abordons la vérification formelle de protocoles de sécurité, et enfin nous présentons nos contributions.

Contexte

Les protocoles de sécurité sont des protocoles de communication dont l'un des objectifs est de garantir des propriétés de sécurité à des agents. Auparavant, ces protocoles étaient principalement utilisés par les armées afin d'échanger des informations sans que l'ennemi ne puisse les intercepter. Aujourd'hui les protocoles de sécurité sont sortis du seul contexte militaire et sont devenus omniprésents.

Le protocole TLS [27] est probablement l'un des protocoles de sécurité les plus importants. TLS est le protocole utilisé pour sécuriser le protocole HTTP, ce qui est alors indiqué par le préfixe « https:// » dans les URLs. Grâce à TLS, il est notamment possible d'effectuer des achats en ligne, d'accéder à un réseau social, ou encore de consulter son compte bancaire, et ce de manière sécurisée, c'est-à-dire sans qu'un attaquant puisse écouter nos conversations ou apprendre notre numéro de carte bancaire et nos mots de passe.

Un autre exemple de protocole de sécurité utilisé au quotidien est le protocole Wi-Fi [40]. Ce protocole permet d'accéder à un réseau local sans utiliser de câble, en utilisant des ondes électromagnétiques comme moyen de communication. Or par nature, les ondes électromagnétiques peuvent être interceptées par toute machine se situant suffisamment proche de la machine émettrice. Il est donc nécessaire que le protocole Wi-Fi garantisse la confidentialité et l'authenticité des données échangées. Pour cela, le protocole effectue

des opérations cryptographiques telles que le chiffrement.

Nous avons vu que les protocoles de sécurité jouent un rôle essentiel dans la sécurité de nos infrastructures modernes. Pourtant, de nombreuses failles ont été découvertes dans ces protocoles, même des années après leur mise en place. Par example, un attaquant possédant la clé secrète (clé WPA) d'un point d'accès Wi-Fi (le routeur Wi-Fi) peut aujourd'hui déchiffrer les communications de toutes les autres machines qui communiquent avec le routeur. Pour cela l'attaquant écoute les signaux Wi-Fi, intercepte l'échange de clés qui s'effectue à l'établissement d'une connection, puis déduit de cet échange la clé qui sera utilisée pour chiffrer la connection (la « clé de session »). La méthode est décrite dans [89] par exemple. Le fait que l'attaquant puisse déduire aussi aisément la clé de session provient d'une faille logique dans la conception du protocole Wi-Fi.

Dans cette thèse, nous nous concentrons sur le cas des réseaux privés virtuels (ou VPN, pour Virtual Private Network). Un VPN permet à plusieurs machines distantes de communiquer à travers Internet comme s'ils se situaient dans le même réseau local. Les VPNs sont par exemple utilisés par les entreprises pour relier leur site de travail, ou pour permettre à leurs employés d'accéder à leur réseau privé depuis n'importe quel endroit. Les VPNs sont également utilisés par les armées pour sécuriser les connexions entre les sites militaires, ou encore par les citoyens de certains pays pour échapper à la censure.

L'une des architectures (i.e. ensemble de protocoles) utilisées pour créer des VPNs est Internet Protocol Security (IPsec) [76]. Or, préalablement à la mise en place d'un VPN IPsec entre deux machines distantes, il est nécessaire que le deux machines établissent entre elles une clé secrète (i.e. connue seulement des ces deux machines) qui servira à chiffrer et à authentifier leurs communications. Le protocole chargé de générer cette clé secrète pour IPsec est Internet Key Exchange version 2 (IKEv2) [46]. La sécurité d'un VPN repose donc sur la sécurité de IKEv2. Il est alors crucial que la sécurité de IKEv2 soit assurée. Dans cette thèse, nous utilisons des techniques modernes de verification formelle pour tenter de répondre à cet enjeu.

Protocoles et attaques

Regardons maintenant de plus près à quoi ressemble un protocole de sécurité. Ces protocoles sont des programmes qui manipulent des primitives cryptographiques afin de permettre à deux machines distantes de communiquer de manière sécurisée. Les attaques qui ne cassent pas les primitives cryptographiques mais exploitent un assemblage incorrect de

ces dernières sont appelées *attaques logiques*.

Jusque dans les années 1970, la seule primitive existante était le chiffrement symétrique. En chiffrement symétrique, la même clé est utilisée pour le chiffrement et pour le déchiffrement. Utilisée seule, cette méthode de chiffrement présente le problème que chaque machine doit garder en mémoire une clé pour chaque autre machine avec laquelle elle soit communiquer. Cela devient difficile à gérer lorsque le nombre de machines croît.

Les travaux de Diffie et Hellman publiés en 1976 [28] marquent un tournant dans la cryptographie en introduisant la cryptographie asymétrique. Dans cette méthode, chaque machine possède une clé privée et une clé publique. La clé publique est connue de tout le monde, tandis que la clé privée n'est connue que de son propriétaire. Pour communiquer avec une machine, on chiffre notre message avec la clé publique de cette machine, et l'algorithme de chiffrement assure que seul le détenteur de la clé privée associée peut déchiffrer le message. L'avantage de cette méthode est que chaque machine ne doit garder en mémoire que sa clé privée et sa clé publique. Avant chaque communication, les machines communiquent alors leur clé publique à quiconque veut communiquer avec elles.

Notons qu'il existe bien d'autres primitives cryptographiques. Par exemple, la signature numérique permet à une machine d'authentifier un message à l'aide de sa clé privée. La vérification se fait avec la clé publique. Le Message Authentication Code (MAC) remplit la même fonction que la signature numérique, mais la même clé est utilisée pour la génération du MAC et pour sa vérification.

De manière informelle, un protocole est un ensemble de rôles qui effectuent des actions. Les rôles sont appelés « initiateur », « répondeur », « client », « serveur », etc. Les actions sont du type : générer un nombre aléatoire, appliquer une primitive cryptographique, envoyer un message, ou encore recevoir un message. De plus il est courant de diviser les messages en entités logiques appelés *payloads*, et de diviser les payloads en *champs*.

Un protocole de sécurité vise à satisfaire certaines propriétés de sécurité. La propriété la plus courante est la confidentialité d'une donnée. Cette propriété garantit à un agent (exécutant un rôle) que cette donnée n'est pas connue d'un éventuel attaquant. Cependant, même cette propriété à l'apparence simple n'est pas évidente à définir formellement. Par exemple, la confidentialité peut être définie par le fait que l'attaquant ne puisse pas produire la valeur du secret (confidentialité faible), ou par le fait que l'attaquant ne puisse pas distinguer deux valeurs différentes du secret dans deux sessions différentes (confidentialité forte). Dans cette thèse nous ne considérons que la confidentialité faible.

Une autre propriété courante est l'authentification. Cette propriété garantit à un agent

ayant terminé d'exécuter le protocole (appelé l'*acteur*) que l'agent avec lequel il pense avoir effectué le protocole (appelé le *pair*) est bien celui avec lequel il l'a effectué. Cependant, cette formulation de l'authentification présente également de nombreuses ambiguïtés. Par exemple, doit-on exiger que le pair a bien exécuté le bon rôle (e.g. le rôle de répondeur lorsque l'acteur est initiateur) ? De plus, doit-on exiger que tous les payloads des messages reçus par l'acteur aient bien été envoyé par le partenaire, ou seulement certains d'entre eux ? Notons qu'il est difficile d'exiger cela des payloads qui ne sont pas cryptographiquement protégés. Finalement, un message peut effectivement avoir été envoyé par le partenaire, mais lors d'une session passée (cela peut arriver lorsqu'un attaquant rejoue des messages enregistrés auparavant). On pourrait alors exiger qu'à chaque message reçu ne corresponde qu'un seul message envoyé. Pour formaliser cette notion d'authentification, Lowe propose en 1997 une hiérarchie des propriétés d'authentification [51]. Nous nous appuyons sur ce travail dans cette thèse.

La définition d'une propriété de sécurité dépend en outre toujours des capacités données à l'attaquant. En effet, garantir la confidentialité d'une donnée en présence d'un attaquant qui peut simplement écouter le canal de communication n'est pas du tout équivalent à garantir cette propriété en présence d'un attaquant qui peut bloquer des messages et injecter des messages qu'il a lui-même créés. Ainsi plusieurs modèles d'adversaire ont été proposés dans la littérature. Le premier et le plus connu d'entre eux est le modèle Dolev-Yao, introduit en 1981 [30]. En bref, l'adversaire Dolev-Yao peut intercepter n'importe quel message (empêchant ainsi sa réception par son destinataire), apprendre ce qui est en clair ou ce qu'il peut déchiffrer, et enfin construire et envoyer ses propres messages. C'est cet adversaire que nous allons considérer dans cette thèse.

Vérification formelle de protocoles de sécurité

Il est possible de prouver à la main qu'un protocole satisfait une propriété de sécurité. Cependant, les preuves à la main peuvent être sources d'erreurs, et un protocole critique nos infrastructures comme IKEv2 ne peut pas se permettre de telles erreurs. De plus, il existe un grand nombre de protocoles de sécurité. Pour ces raisons, il est plus raisonnable d'utiliser des outils de preuve automatiques.

Il existe deux approches pour la vérification formelle de protocoles. L'approche computationelle [35] représente les messages comme ils sont, c'est-à-dire des vecteurs de bits, et représente l'attaquant par une machine qui peut effectuer quasiment n'importe quelle

opération. A l'inverse, l'approche symbolique représente les messages par des termes qui sont construits par application successives de fonctions sur des variables et des constantes. L'attaquant de l'approche symbolique dispose lui d'un nombre restreint d'opérations cryptographiques qu'il peut appliquer. Les preuves computationelles donnent évidemment des garanties plus fortes que les preuves symboliques, mais elles sont également plus difficiles à obtenir. Dans cette thèse nous nous plaçons dans le modèle symbolique.

Même en considérant des primitives cryptographiques simples, le problème de la vérification symbolique de protocole de sécurité est indécidable en général [31, 21]. Cela signifie qu'il n'existe aucun algorithme dont la terminaison est garantie est qui réponde correctement, soit que le protocole est sûr, soit que le protocole admet une faille. La raison pour cela tient au fait que de nombreux paramètres du problème ne sont pas bornés. Par exemple, il n'y a pas de borne au nombre de sessions que les agents peuvent effectuer, ni au nombre de valeurs aléatoires que l'attaquant peut générer, ni au nombre de fois que l'attaquant peut appliquer une primitive cryptographique pour construire un message. De plus, certaines primitives cryptographiques possèdent des propriétés algébriques qui rendent la vérification encore plus complexe. C'est le cas par exemple de l'exponentiation modulaire, qui est utilisée dans IKEv2.

Pour effectuer une vérification symbolique de protocole, il est d'abord possible d'utiliser un outil de model checking généraliste, même si ceux-ci n'ont pas été spécifiquement conçus pour la vérification de protocoles de sécurité. Un outil de model checking généraliste est un outil permettant de vérifier des propriétés sur des systèmes de processus asynchrones. Par exemple, l'outil Spin [37] permet de vérifier des propriétés sur un système de processus qui communiquent entre eux à travers des canaux. Puisque qu'un protocole correspond bien à cette description, il est possible d'utiliser Spin pour analyser un protocole. Cependant, Spin n'offre nativement aucun moyen de modéliser ni des primitives cryptographiques ni un adversaire. En 2014, [8] propose une méthode de modélisation en Spin qui fait de nombreux progrès pour pallier ce manque. Cependant, la méthode ne propose pas de moyen de modéliser trois primitives cryptographiques utilisées dans IKEv2, qui sont la signature numérique, le MAC et l'exponentiation modulaire. De plus, le modèle d'adversaire de la méthode est très difficile à écrire pour des protocoles de la complexité de IKEv2. Enfin, la méthode ne propose pas de modèle pour les propriétés de sécurité que nous aimerions vérifier sur IKEv2. Nous remédions à ces trois problèmes dans cette thèse.

Il y a cependant une limitation de Spin à laquelle nous ne pouvons pas remédier. Les

modèles Spin sont par nature bornés, ce qui nous force à borner tous les paramètres normalement non bornés de la vérification symbolique de protocoles de sécurité, notamment le nombre de sessions, le nombre de fois que l'attaquant peut appliquer des primitives pour construire un message, et même la mémoire de l'attaquant. Pour cette raison, Spin ne peut fournir que des preuves de sécurité relativement faibles. Pour obtenir des preuves plus fortes, il faut se tourner vers des outils spécialisés dans la vérification de protocoles de sécurité comme ProVerif [9] et Tamarin [55]. Ces deux outils tentent d'analyser les protocoles dans le modèle non borné décrit plus haut. Cependant, comme nous l'avons dit, ce problème est indécidable. Pour tenter de surmonter cette indécidabilité, ProVerif effectue des abstractions qui permettent à l'outil d'obtenir des preuves dans certains cas. Concrètement, étant donné un modèle et un propriété, ProVerif peut soit prouver la propriété, soit trouver une attaque, ou même ne pas conclure. De son côté, Tamarin n'effectue pas d'abstractions mais permet à l'utilisateur d'interagir avec l'outil afin de guider l'outil dans sa preuve. Tamarin a un modèle plus expressif de l'exponentiation modulaire que ProVerif et permet la modélisation de compteurs, ce qui est difficile en ProVerif.

Parce que IKEv2 est un protocole essentiel dans nos infrastructures modernes, il a déjà été analysé par des méthodes de vérification symbolique. En 1999, l'analyse de Meadows [54] découvre deux failles de sécurité dans IKEv1 (l'ancêtre de IKEv2) : la faille de pénultième authentification et l'attaque par réflexion. Ces failles sont des violations de certaines propriétés d'authentification. Cependant, elles ne sont pas considérées comme dangereuses car elles ne remettent pas en cause la confidentialité des échanges IPsec. D'autres analyses se sont alors concentrées sur IKEv2 [70, 48], et constatent que cette version du protocole souffre également de la faille de pénultième authentification. L'analyse la plus complète de IKEv2 à ce jour a été effectuée par Cremers [24], et affirme que IKEv2 est également vulnerable à une attaque par réflexion.

Cependant, deux importantes propriétés d'authentification n'ont pas été vérifiées sur IKEv2. La première est l'*accord non injectif* sur la clé de session, qui énonce que l'acteur et le partenaire sont d'accord sur la valeur de la clé échangée. La deuxième est l'*accord injectif*, qui ajoute à l'accord non injectif la condition que les messages n'ont pas été rejoués par l'attaquant. De plus, le modèle de [24] n'est pas fidèle à IKEv2 car il oublie certains champs importants du protocole. Nous montrons dans cette thèse que ces champs rendent l'attaque par réflexion trouvée par [24] impossible.

Contributions

Dans cette thèse, nous effectuons une analyse poussée du protocole IKEv2 à l'aide de vérification symbolique, et nous déduisons d'une faille logique du protocole un nouveau type d'attaque par déni de service (DoS), à laquelle IKEv2 est vulnérable. Plus précisément :

- Nous étendons la méthode de [8] de modélisation de protocoles en Spin, afin de permettre l'analyse de IKEv2 avec Spin. En particulier, nous proposons un modèle de la signature numérique, du MAC et de l'exponentiation modulaire, nous simplifions le modèle d'adversaire pour le rendre applicable à des protocoles complexes, et nous proposons des modèles pour les propriétés d'authentification de [51].

- Nous appliquons ensuite notre méthode améliorée à l'analyse de IKEv2. Nos résultats montrent que l'attaque par réflexion trouvée par [24] n'est en fait pas possible. Puisque la nature de Spin ne permet d'analyser que des modèles excessivement bornés, nous utilisons les outils ProVerif et Tamarin pour analyser IKEv2 dans le modèle non borné. Nos analyses produisent de nouvelles preuves concernant les garanties d'accord non injectif et d'accord injectif pour IKEv2 dans le modèle non borné.

- Enfin nous montrons que la faille de pénultième authentification, bien que considérée comme bénigne par les analyses précédentes, permet en fait d'effectuer un nouveau type d'attaque par déni de service auquel IKEv2 est vulnérable : l'*Attaque par Déviation*. Cette attaque est plus difficile à détecter que les attaques par déni de service classiques mais est également plus difficile à réaliser. Afin de démontrer concrètement sa faisabilité, nous attaquons avec succès une implémentation open-source populaire de IKEv2. Nous étudions ensuite l'efficacité des contre-mesures classiques aux attaques DoS ainsi qu'une fine configuration des implémentations actuelles de IKEv2 pour prévenir l'attaque. Cependant, nous ne trouvons que des mesures insatisfaisantes. Abordant alors le problème à plus haut niveau, nous proposons deux modifications simples du protocole, et prouvons formellement que chacune d'entre elles empêche l'Attaque par Déviation.

Contents

Chapter 1

Introduction

This thesis stands on a bridge between two topics. One topic is IPsec, whose security is critical to Thales SIX GTS France, the company for which I work. The other topic is formal verification of security protocols, with which my research institute is familiar. Hence this thesis aims to apply formal techniques to the security analysis of IKEv2.

In this Chapter, we introduce the reader to the problem of IKEv2 formal verification. We start by motivating the need for a secure IKEv2 in Section 1.1. In Section 1.2, we provide some theoretical background on security protocols and attacks. In Section 1.3, we broadly sketch the current state-of-the-art in formal verification of security protocols, and in particular of IKEv2. Finally, our contributions are depicted in Section 1.4, and Section 1.5 gives the plan of this thesis.

1.1 Context

Security protocols are communication protocols that aim to guarantee security properties to their participants. Security protocols used to be limited to the military context, where it was crucial to keep strategic information secret when sent over untrusted networks. However, with the development of the Internet, individuals, companies and states now all make heavy use of security protocols.

A major application of security protocols today is the ability to browse web sites in a secure manner. In particular, any individual now has the possibility to check whether a web page belongs to some trusted entity or not. This is achieved using the Transport Layer Security (TLS) protocol [27]. This is of great importance, since it allows e.g. financial transactions to be made entirely online. Online payment has radically changed many industries, such as clothing, food, transport, or entertainment.

Security protocols are also important in the context of broadcast communication media, i.e. communication media to which anyone can listen. This is the case of Wi-Fi for example. Without the use of encryption in the Wi-Fi protocol [40], anyone with some

hacking skills could listen to the traffic of its neighbours.

Many flaws have been discovered in many security protocols, even in protocols that have been deployed for a long time. In fact, even some protocols that have been much studied have been shown to be vulnerable. When these protocols serve important matters of economical, political or military order, attacks can have serious impacts. For example, [90] manages to shop for free on online stores that use external payment platforms such as PayPal or Amazon Payments. This was possible because the online store did not check the gross payed by the client to the payment platform. Obviously, this check should have been enforced by the protocol used to synchronize the online store and the platform payment. Online shopping for free can have disastrous consequences for selling companies. Consider an attacker that exploits the weakness using several client machines, all in a short time (so as to not be detected). The attacker could obtain a considerable amount of goods for free, effectively stealing a lot of money from selling companies.

As an other example, there is a well-known weakness in the current Wi-Fi protocol [40]. An attacker who knows the pre-shared key of a Wi-Fi access point can decrypt the traffic of any station connected to the access point. It suffices that the attacker intercept the handshake at the beginning of the connection, since the session key can be derived from the pre-shared key and from cryptographic values sent in cleartext during the handshake. The process is described in [89] for example. If the connection is already set up, the attacker can even send a deauthentication message to the victim, spoofing the MAC address of the access point, and the victim will perform the handshake again. This is possible because deauthentication messages are not authenticated in the protocol. Overall, the weakness of Wi-Fi effectively increases the vulnerability of people to eavesdropping. Obviously, this can have disastrous consequences in the case of political or industrial espionage.

In this thesis, we will focus on a security protocol related to Virtual Private Networks (VPN). VPNs have been used for decades by companies, governments and people. VPNs allow connecting two or more distant IP entities as if they were in a single Local Area Network (LAN). By "entity" we either mean a network or a single machine. Since the network between the entities may be untrusted, connecting the entities often implies performing some encryption and some address translation on the IP traffic between them. In a company, an entity can among other things denote a traveling employee or a working site. In the military domain, an entity can among other things denote a soldier on the battle field or a military base. More recently, we have observed the rise of commercial VPN services for the greater public. A commercial VPN service owns several servers in different

countries and offers individual consumers to create a VPN with one of its servers. This way, an individual consumer can browse the Internet as if it was physically in the country of the server. Many people use VPN services to spoof their location in order to access services denied to them based on their geographical location. Many people also use them to evade censorship, since VPN services allow them to hide the content of their packets between them and the VPN server. VPNs thus have become a core technology in the modern secure Internet, and it is crucial that VPNs remain secure.

A VPN can be created using multiple technologies, among which the Internet Protocol security (IPsec) architecture [76] is widely used. Moreover, Internet Key Exchange version 2 (IKEv2) [46] is one of the main protocols used to set up IPsec VPNs. IKEv2 aims at guaranteeing mutual authentication of two peers, and at automatically generating the shared secret that will be of the communication's security warrant. Thus a secure VPN relies upon the security of IKEv2, and it is of the utmost importance that IKEv2 be secure. In this thesis, we will use modern verification techniques to study that matter.

1.2 Protocols and attacks

Let us now explain what these security protocols of our daily use typically look like. Security protocols are special programs manipulating cryptographic primitives in order to make two or more machines communicate with each other in a secure manner. These primitives are basic blocks that must then be correctly assembled to obtain a secure protocol. Attacks that do not break cryptography, but exploit an incorrect assemblage of cryptographic primitives are called *logical attacks*.

We present some modern cryptographic primitives in Section 1.2.1. We explain what a security protocol looks like in Section 1.2.2, and show some security properties in Section 1.2.3. Finally, we introduce the concept of adversary model and describe a logical attack in Section 1.2.4.

1.2.1 Cryptography

Before the mid-1970s, security was almost synonymous with symmetric encryption (also called single-key encryption). In symmetric encryption, the same key is used for encryption and decryption. Only the communicating agents should know this key. State-of-the-art symmetric encryption algorithms include AES [81] and RC6 [73]. Symmetric encryp-

tion has two major drawbacks. Firstly, the key needs to be exchanged beforehand in a confidential manner. This yields a chicken-and-egg problem, as encryption would now be needed to exchange the key. Secondly, even when ways to exchange keys in a confidential manner are available, in a system where there are more than two agents, each agent needs to remember one key for each agent with whom it wants to communicate. Hence for n agents we need $\frac{n\times(n-1)}{2}$ keys, i.e. the number of keys is a quadratic function of the number of agents. Such a system quickly becomes unmanageable as the number of agents grows.

A new era of cryptography is started in 1976, when Diffie and Hellman introduce the concept of public-key cryptography [28]. In public-key cryptography, each agent owns a public key and a corresponding private key. A private key is known only to its owner, whereas knowledge of a public key can be accessed by any agent. Accordingly, it is computationally infeasible to deduce the private key from the public key. Public-key cryptography can be used for two different purposes: asymmetric encryption and digital signatures. The first implementation of a public-key cryptosystem was proposed by Rivest, Shamir and Adleman in 1979, and is known as RSA [59].

In asymmetric encryption (also known as public-key encryption), an agent A encrypts a message with the public key of an agent B. The only agent that can decrypt the resulting cyphertext is the owner of the private key of B, which is B. Asymmetric encryption effectively solves the two problems of symmetric encryption mentioned above, as agents only need to remember their own private key, and there is no need to secretly exchange a key prior to encryption. However, asymmetric encryption is much slower than symmetric encryption. Thus it is common to use public-key cryptography to exchange a symmetric key, and then use this key to exchange data. A protocol that aims to set up a shared secret key between two agents is called a key-distribution protocol.

Digital signature, also introduced in [28], works conversely to asymmetric encryption. To produce a digital signature, an agent A uses its private key. Then any agent can check the signature using the public key of A. The public-key cryptosystem should ensure that it is computationally infeasible for an attacker to produce a valid signature for A, without knowing the private key of A. Digital signatures can thus be used for authentication.

Finally, a Message Authentication Code (MAC) also provides authentication, but in the case of a MAC, the same key is used to produce the MAC and to check the MAC. MAC algorithms can be built upon other cryptographic primitives. A hash-based MAC (HMAC) is based on a cryptographic hash function (i.e. a one-way hash function). A Cipher Block Chaining MAC (CBC-MAC) is based on a symmetric encryption algorithm.

1.2.2 Protocols

Informally, a protocol describes the *actions* that each communicating agent should perform. Each agent may execute a *role* of the protocol. Examples of roles are *initiator*, *responder*, *client*, *server*, etc. Common actions include: generating a random number, building a new value by applying some cryptographic primitive to other values, sending a message, receiving a message.

Let us look at some example of security protocol. In [28], Diffie and Hellman propose a key-distribution system that uses a cryptographic primitive called modular exponentiation (exponentiation in finite field). Based upon this key-distribution system, it is common to define the Diffie-Hellman (DH) protocol as in the message flow below. Let m^x denote the modular exponentiation of m by x. g is a public constant called the generator. Let $\{m\}_k^{se}$ denote the symmetric encryption of m by k. In the DH protocol, an agent i generates a random number x_i, and sends g^{x_i} alongside its identity to an agent r. Upon reception of this message, r generates a random number x_r, and sends g^{x_r} alongside its identity to i. Both parties can now compute the value $k = (g^{x_i})^{x_r} = (g^{x_r})^{x_i} = g^{x_i * x_r}$. Finally, agent i symmetrically encrypts a secret s using key k, and sends the result to agent r. As a result, only i and r should know the secret s.

$$
\begin{aligned}
&i \to r : i, g^{x_i} \\
&r \to i : r, g^{x_r} \\
&i \to r : \{s\}_k^{se} \qquad \text{where } k = (g^{x_i})^{x_r} = (g^{x_r})^{x_i}
\end{aligned}
$$

1.2.3 Security properties

Security protocols aim to guarantee security properties to their participants. The most common security property is secrecy. The secrecy of a cryptographic value v states that v is only known to the agents running the protocol. For example, the Diffie-Hellman protocol described in Section 1.2.2 aims to guarantee the secrecy of s. As a more concrete example, when people shop online, they expect their credit card number to be known only to them and the online store. As you can see, this property is often absolutely critical. However, even such a basic property can be defined in several ways. Indeed, when defining secrecy, one must specify which values should remain secret. In addition, one must specify from which point of vue they should remain secret. For example, if the two agents i and r from

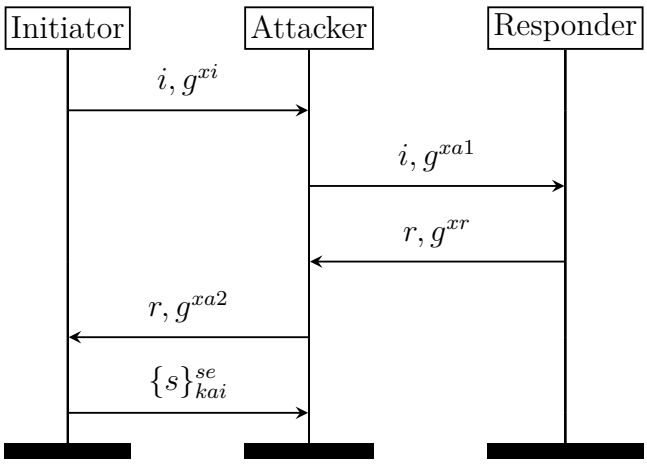
the Diffie-Hellman protocol compute different values for k, and the property "secrecy of k" is considered, then it is not clear if it is the value of k computed by i that should remain secret, or if it is the value of k computed by r, or both. Moreover, "staying secret" is not clear either. For example, it can mean that an attacker is unable to produce the value of the secret (*weak secrecy*), or it can mean that, given two runs of the protocol, an attacker cannot detect that two different values of the secret have been computed (*strong secrecy*).

Another common security property is authentication. The authentication property guarantees to an agent a that the agent with whom a has completed the protocol (let us say, b) was indeed who it pretended to be. For example, the Diffie-Hellman protocol claims to guarantee to i that the agent with whom it now shares key k is indeed r. However, this definition of authentication is not clear either. For example, do we require that agent b ran the protocol in the correct role (e.g. if a was an initiator, do we require that b were a responder)? What does it mean to have run the protocol with a? Does it mean that b has sent the exact same messages as a received, or are some payloads allowed not to match (we can hardly require that unprotected payloads match)? What if messages were indeed sent by b, but during an other past session (this can happen when an attacker replays messages)? To answer these questions and formalize the notion of authentication, Lowe proposed in 1997 a hierarchy of authentication properties [51]. We will rely on this work in this thesis.

1.2.4 Adversary models

The definition of a security property always depends on the capabilities given to the attacker. For example, guaranteeing the secrecy of some cryptographic value in presence of an active attacker that can inject messages into the network is much stronger than guaranteeing the secrecy in presence of an eavesdropper that can only listen to the network. Some protocols also aim to satisfy a security property even if a trusted machine has been compromized, or even if the attacker can learn random numbers (the latter models flawed random number generators). Several adversary models have been proposed in the literature. Two common ones are the Dolev-Yao adversary [30] and the passive adversary. In short, the Dolev-Yao adversary can intercept and drop messages, learn from them (deduce the value of some other cryptographic value), inject self-forged messages, and compromize some machines. The passive adversary is an eavesdropper: it can only see the content of the exchanged messages and learn from it.

While the Diffie-Hellman protocol claims to guarantee the secrecy of k to its participants, there is an attack that falsifies this property. The attack is possible with a Dolev-Yao adversary, but not with a passive adversary. In this attack, depicted below[1], the attacker a intercepts the message sent by the initiator i and drops it. Then the attacker generates its own random number x_{a1}, and sends the pair $(i, g^{x_{a1}})$ to the responder r, thereby impersonating i. The responder r answers with its own message, which the attacker intercepts and drops again. The attacker then generates a second random number x_{a2} and sends $(r, g^{x_{a2}})$ to i, thereby impersonating r. In the end, i and r both believe to share a secret key with each other, when they actually both share a key with the attacker. Let us call *kai* the key shared by the attacker and the initiator. The attacker can then intercept $\{s\}^{se}_{kai}$, decrypt it using key *kai* and learn the value s, which compromises the property (in fact the attacker can even transparently decrypt, encrypt and forward all traffic between agents i and r, without them noticing it). To prevent this attack, Diffie-Hellman based protocols usually add an authentication step, as is the case in IKEv2.



Notice how the above attack does not break any cryptographic algorithm. Instead, the attack exploits a logic flaw in the protocol. *It is the aim of this thesis to check that IKEv2 does not suffer from logic flaws.*

[1]We use the LaTeX MSC package [53] to draw our message sequence graphs.

1.3 Formal verification of security protocols

Proving that a protocol is secure can be made by hand, but there are two reasons why it is better to use automated techniques. Firstly, there are too many protocols, either already in use or on the way to be adopted. Verifying them all by hand is not possible. Secondly, the stakes are too big to rely on manual proofs, which are error-prone. As pointed out in Section 1.1, a flaw in a critical protocol can have serious consequences.

In this Section, we sketch the current knowledge in security protocol formal verification. We start by introducing the concept of symbolic verification in Section 1.3.1. We then emphasize in Section 1.3.2 that symbolic verification of security protocols is undecidable in general. That said, we present some tools that still try to answer the problem in Section 1.3.3. Finally, we summarize past formal analyses of IKEv2.

1.3.1 Symbolic verification in a nutshell

There are two main approaches to formally prove that a protocol satisfies a security property. The *computational approach*, introduced by [35] in 1984, represents cryptographic values as they are: strings of bits. The adversary can peform any probabilistic polynomial-time computation on the values it knows. Different assumptions are made on the strength of cryptographic algorithms used by protocol roles. The adversary then faces a game, where given two scenarii of the protocol, it must guess in which scenario it is. If the adversary cannot make a correct guess with a non-negligible advantage, then the protocol is considered secure. CryptoVerif [11] is an example of tool that works in the computational model.

On the other hand, the *symbolic approach* represents cryptographic values as *terms*. Terms are either simple names, or recursively built upon other terms by applying functions, which represent cryptographic primitives. Cryptography is assumed perfect, meaning that e.g. an attacker can only decrypt a cyphertext if it knows the appropriate decryption key. Therefore, the symbolic approach is only suited to find so-called *logic* attacks, such as the attack described in Section 1.2.4.

Obviously, proofs in the computational model offer stronger guaranties. However, these proofs are often harder to obtain than proofs from the symbolic approach. *In this thesis we focus on the symbolic approach.*

Even in the symbolic approach, there is no unified model. Some models use the multiset rewriting paradigm [14]. In such models, the state of a protocol is represented by a multiset

(i.e. a set where items can appear multiple times) of facts. Facts are statements describing the state of the protocol. The protocol execution is represented by rules, which remove facts from the state and add other facts to the state.

Some other symbolic models represent protocol using process calculi. A process calculus is a language describing processes that run in parallel and send messages to each other. In this thesis, we will use a process calculus called *applied pi calculus* [1]. The pi calculus [56] and spi calculus [2] are similar process calculi.

1.3.2 Symbolic verification of security protocols is difficult

Even when considering simple cryptographic primitives (only pairing and symmetric encryption) and a simple property (secrecy), verification in the symbolic model is undecidable in general [31, 21]. This means that there exists no algorithm that always eventually terminates with a correct answer to this problem.

The reason that symbolic verification is difficult is that there exists many sources of unboundedness in the symbolic model. Firstly, any agent should be able to execute an arbitrary number of sessions, i.e. run the protocol an arbitrary number of times. Furthermore, the attacker should be able to build messages by applying cryptographic primitives any finite number of times. There should be no bound as well to the number of random values (nonces) used by the attacker and roles. Finally, the size of the attacker memory should also be unbounded.

There is at least one parameter that can be bounded: the number of agents. [19] shows that, when all other aspects of the model (such as number of sessions) are unbounded otherwise, if secrecy is satisfied when *two* agents are running the protocol, then it is satisfied when *any number* of agents are running the protocol. Similarly, [19] shows that authentication needs only be proved with three agents.

Some cryptographic primitives possess algebraic properties that increase even more the difficulty of verification. Examples of such primitives are the XOR binary operator and modular exponentiation. As said earlier, modular exponentiation is used in the Diffie-Hellman protocol, which is at the core of IKEv2, hence making formal verification of IKEv2 even more challenging.

Nevertheless, in the next Section we present some verification tools that manage to obtain interesting results. These tools either enforce decidability by bounding the number of sessions, or by accepting to sometimes not be able to conclude.

1.3.3 Verification tools

Let us present some verification tools. We firstly consider the use of general-purpose model checkers for security protocol verification. Then we describe some tools that are specialized in security protocol verification, splitting them into two categories: those that restrict the number of sessions, and those that do not.

General-purpose model checkers

It is possible to use general-purpose model checking tools to analyze security protocols, even if these tools were not specifically tailored to this task.

For example, [58] efficiently analyzes some classic security protocols in the bounded model using the general-purpose model checker Mur/spl phi/.

Another option is to use the Spin model checker [37]. Spin was designed to verify properties on channel systems, i.e. sets of processes communicating together through channels. This fits well to security protocols, but Spin has neither support for cryptographic primitives, nor for an adversary.

This problem is addressed in [8] by Ben Henda, who proposes a modeling method that provides basic support for some cryptographic primitives and an adversary model in Spin. However, the method of [8] lacks models for some cryptographic primitives of IKEv2, namely digital signature, MAC and modular exponentiation. Furthermore, the adversary model of [8] becomes very hard to implement for a complex protocol such as IKEv2. Finally, [8] lacks models for many properties that we would like to verify on IKEv2. In this thesis we provide solutions to these problems.

Verification of security protocols using Spin and the method of [8] suffers from a boundedness problem, which is due to the bounded nature of Spin and cannot be fixed. The bounded nature of Spin forces us to bound the number of sessions, the number of cryptographic operations that the attacker can perform to build a message, and the memory of the attacker. Proofs yielded by Spin are thus much weaker than proofs yielded by tools specialized in security protocol verification.

Decision procedures for bounded session models

Although symbolic verification of security protocols is undecidable in general, it becomes decidable when bounding the number of sessions [74]. Tools implementing decision procedures (i.e. algorithms that are guaranteed to eventually terminate) for bounded session

models include OFMC [7], CL-ATSE [88] and DEEPSEC [15].

Note that bounding the number of sessions allows us to bound the number of nonces that agents and the attacker can use, without loosing further guaranties. However, bounding the number of sessions does not bound the number of cryptographic operations that the attacker can perform to build messages.

Bounding the number of sessions is quite reasonable, as in practice, attacks seldom use more than a few sessions. However, for a higher level of assurance, unbounded session verification is required. Moreover, having decision procedures is not enough: we further need to have implementations of these procedures with reasonable complexity, which is a difficult problem.

Semi-decision procedures for unbounded session models

Some tools do work in the unbounded model but implement semi-decision procedures, i.e. procedures that may correctly conclude that a property is satisfied or falsified, but may also terminate with an inconclusive answer or not terminate at all. Such tools include ProVerif [9] and Tamarin [55].

ProVerif uses a semi-decision procedure in the unbounded model that takes as input a language of the process calculus family which ProVerif translates into a set of Horn clauses. ProVerif is fully automatic and has basic support for modular exponentiation.

Tamarin uses a semi-decision procedure in the unbounded model that takes as input a multiset rewriting system. Compared to ProVerif, Tamarin has better support for modular exponentiation. Furthermore, Tamarin allows modeling counters that can be incremented an unbounded number of times, which ProVerif struggles to support. Like ProVerif, Tamarin can work in a fully automatic fashion. However, Tamarin additionally offers the user to interact with it and guide it through the proof, thereby relying upon the user's knowledge of the analyzed protocol.

Remember that both tools may not terminate or terminate with an unconclusive answer. Hence, one tool is not necessarily better than the other, as for some protocols and properties, ProVerif may provide a conclusive answer while Tamarin may not (even with the user's help), and vice versa.

1.3.4 Formal verification of IKEv2

IKEv1 and IKEv2 are critical protocols, hence they have already been analyzed using formal methods in the past.

In 1999, Meadows analyzes IKEv1 using the NRL protocol analyzer [54]. Two authentication weaknesses are found: the penultimate authentication flaw and a reflection attack. However, the penultimate authentication flaw is considered harmless, as it does not violate the secrecy of the exchanged key.

In 2003, IKEv2 is analyzed using the AVISPA plateform [70] (which includes the OFMC and CL-ATSE tools mentioned earlier). [70] finds that IKEv2 also suffers from the penultimate authentication flaw. Presumably, fixing the flaw comes at a cost, and since the flaw was considered harmless, authors of the IKEv2 RFC [46] chose to ignore it.

The most comprehensive analysis of IKEv2 until now stems from [24] in 2011, where the Scyther tool is used. [24] surpasses previous analyses by analyzing the protocol in the unbounded model, against several adversary models, and by following authentication properties from [51]. [24] finds that IKEv2 also suffers from a reflection attack.

However, two important authentication properties still need to be verified on IKEv2. One property is the *non-injective agreement* on the exchanged key, which essentially states that the initiator and the responder agree on the value of the exchange key. The other property is *injective agreement*, which adds to non-injective agreement that there be a one-one relationship between the sessions of the initiator and the responder, hence preventing replay attacks. Furthermore, the models of [24] are missing some payloads of the protocol, and we will show that these payloads prevent the reflection attack found by [24].

1.4 Contributions

In this thesis, we perform an extensive analysis of the IKEv2 protocol using symbolic verification, and we deduce from a logic flaw of the protocol a new type of DoS attack that works against IKEv2. More precisely:

- We extend the method of [8] for modeling protocols in Spin, thereby fixing some limitations of the method, and making it suitable for the analysis of IKEv2-like protocols. In particular, we extend the method with models for modular exponentiation, MAC and digital signature. We further provide a much simpler adversary

model, as well as a better implementation of secrecy and implementations of the authentication properties from [51]. Our extended method is described in Section 3.3.

- We then use our improved method to analyze IKEv2. Our results confirm the penultimate authentication flaw, but show that the reflection attack mentioned in Section 1.3.4 has no practical existence against IKEv2. The model of [24], which reported the vulnerability, was missing some payloads that actually prevent the attack. Since Spin uses an excessively bounded model, we then use ProVerif and Tamarin to analyze IKEv2 *in the unbounded model*. Our analyses using these three tools provide new proofs regarding non-injective agreement and injective agreement guaranties for IKEv2 in the unbounded model. Our analyses using Spin, ProVerif and Tamarin are respectively described in Sections 3.3.4, 3.4 and 3.5.

- The penultimate authentication flaw was not considered a serious concern because it did not question the secrecy of the shared key generated by IKEv2. However, we show that this weakness allows for a new type of Denial-of-Service (DoS) attack against some security protocols (among which IKEv2), which is harder to detect than existing DoS attacks, but also harder to perform. We call this new DoS attack *the Deviation Attack*. The theory of this attack is described in Section 4.2. To demonstrate the Deviation Attack very concretely, we implement it and attack a popular open-source implementation of IKEv2. This is described in Section 4.3. Finally, we study the use of existing DoS countermeasures and existing configuration options to defeat the attack. However, we only find mitigations or incomplete workarounds. We therefore tackle the problem at a higher level: we propose two possible inexpensive modifications of the protocol, and formally prove that they both prevent the attack. Countermeasures to the Deviation Attack are proposed in Section 4.4.

1.5 Outline

We start this thesis by providing some background about the IKEv2 protocol in Chapter 2. We give an overview of the protocol, present its payloads, its security goals, and its implementations. In Chapter 3, we formally verify the protocol. We present the method of [8] for modeling protocols in Spin, as well as our extension. We then apply our method on IKEv2, and complement the analysis by using ProVerif and Tamarin. Chapter 4 covers

the Deviation Attack. We start with some background about DoS attacks, then present the attack in theory, implement it, and finally seek possible countermeasures. We conclude this thesis in Chapter 5, showing possible future works.

Chapter 2

The IKEv2 protocol

In this Chapter we introduce the IKEv2 protocol. We start by providing an overview of the protocol in Section 2.1, then go into the details of its payloads in Section 2.2. Which security goals a protocol aims to meet is often unclear in specifications, so we try to clarify the security goals of IKEv2 in Section 2.3. Finally we give a short survey of the current implementations of IKEv2 in Section 2.4.

2.1 Overview

In this Section, we describe the IKEv2 protocol and the underlying IPsec architecture. We present the main message types of IKEv2 and their purpose. Finally, we sketch the main security properties that the protocol aims to satisfy.

Internet Key Exchange version 2 (IKEv2) is the authenticated key-exchange protocol used in the Internet Protocol security architecture (IPsec). Its specification is managed by the Internet Engineering Task Force (IETF), and the current RFC is RFC 7296 [46]. The goal of IKEv2 is to allow two parties to dynamically negotiate cryptographic algorithms and keys in order to set up an IPsec communication.

The IPsec architecture provides security at the networking layer. It is defined in RFC 4301 [76]. IPsec defines a framework to establish Virtual Private Networks (VPN). Depending on the underlying security protocol that is used, IPsec provides either integrity/authentication protection (AH protocol) or secrecy and integrity/authentication protection (ESP protocol) of IP packets that are exchanged between two parties. When a party protects an IP packet that the party forged itself, we say that the party acts as an IPsec endoint. Otherwise, we say that the party acts as an IPsec gateway. In the latter case, IPsec protects communication between two subnets. IPsec also natively offers some protection against replay attacks (using this protection is however at the discretion of the receiver of an IPsec protected packet). To protect packets, IPsec parties set up *Security Associations (SA)* between them. A Security Association is a set of security parameters

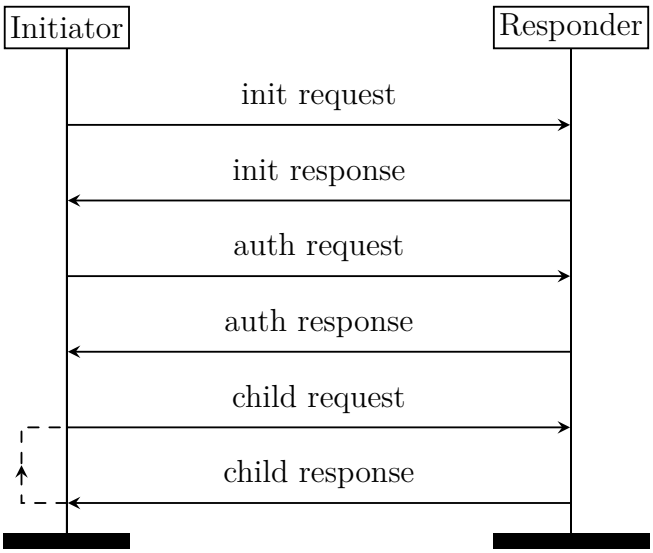


Figure 2.1: An example of IKEv2 session. The init exchange generates a shared secret. The auth exchange authenticates the parties. The child exchange provides freshness by generating a new shared secret.

and keys on which two parties agree and which allows them to communicate in a secure manner in some given direction. Hence an SA is unidirectional, and bidirectional communication requires setting up two SAs (and each party has a local copy of both SAs). However, since SAs are always in pairs, in this thesis we will employ the term "SA" do denote an SA pair, and the terms "unidirectional SA" to denote a single SA.

Setting up an SA requires that the parties share some encryption keys and involves adding entries to their Security Association Database (SAD). The keys can be manually put into the parties' databases, but the maintenance of such a configuration becomes cumbersome as the number of parties grows. To ease management of large VPN setups, it is much more efficient to rely upon dynamic negotiation of cryptographic material as defined by the IKEv2 protocol.

We call an *exchange* the association of a request message and a response message. IKEv2 consists of three main exchanges: init, auth and child.

The init exchange (which is called IKE_SA_INIT in the RFC of IKEv2) performs initial setup of an IKE SA (made of two unidirectional SAs), that will be used to protect subsequent exchanges of the IKEv2 protocol. During this exchange, parties agree upon

cryptographic algorithms that should be used to protect further IKEv2 exchanges and establish some common cryptographic material by running a Diffie-Hellman protocol [28].

The auth exchange (which is called IKE_AUTH in the RFC of IKEv2) authenticates the parties, validates the IKE SA and sets up an initial Child SA (made of two unidirectional SAs). The auth exchange is protected (among other things via encryption) using the key resulting from the Diffie-Hellman protocol run in the init exchange. The authentication can be achieved by using pre-shared key (PSK) or digital signature. It is a countermeasure to the well-known Man-in-the-Middle attack that can be performed against the Diffie-Hellman protocol of the init exchange. This attack is described e.g. at the end of Section 5.1 of [29].

The child exchange (which is called CREATE_CHILD_SA in the RFC of IKEv2) has two different purposes. Firstly, it can be used to rekey an IKE SA, i.e. to replace an old IKE SA with a new one. In this case, its payloads for performing a Diffie-Hellman protocol (the key-exchange payloads) are mandatory. Secondly, it can be used to create a new Child SA, or to rekey an existing one. In this case, the key-exchange payloads are optional. If provided, the two parties compute the keys of the new Child SA using the Diffie-Hellman protocol and these payloads. If not provided, the new keys are simply derived from the IKE SA keys. A substantial benefit of including the key-exchange payloads when creating a Child SA is that compromising the IKE SA keys after the Child SA was created does not compromise the Child SA keys. This property is called Perfect Forward Secrecy (PFS) [36].

The child exchange is protected using the key resulting from the Diffie-Hellman protocol run in the init exchange. While the init and auth exchanges only appear once in an IKE SA lifetime, the child exchange may be performed multiple times (as many times as required to create Child SAs or rekey the mother IKE SA). A Message Sequence Chart (MSC) of IKEv2 is shown on Figure 2.1.

IKEv2 aims to guarantee mostly two security properties. First, that the keying material generated by the init and auth exchanges is secret, i.e. is only known to the two parties involved. Second, that the parties involved are mutually authenticated: each party must prove that it really has the identity it claims to have (in case of public key authentication), or prove that it belongs to some given community (in case of pre-shared key authentication).

2.2 Payloads

In this Section, we describe the IKEv2 protocol in greater details. To simplify the model checking process, we focus on specific parts of IKEv2. These parts constitute protocols on their own, so we call them *subprotocols*. We define subprotocols IKEv2-Sig, IKEv2-PSK and IKEv2-Child. In this Section, we start by presenting the payloads and fields of each subprotocol, keeping only those that we think are relevant to the properties we aim to verify. While doing so, we define the syntax we will use throughout this thesis for denoting these payloads and fields. Finally, we provide justifications to the correctness of the simplifications we make.

2.2.1 IKEv2-Sig

In this Section, we present the IKEv2-Sig subprotocol.

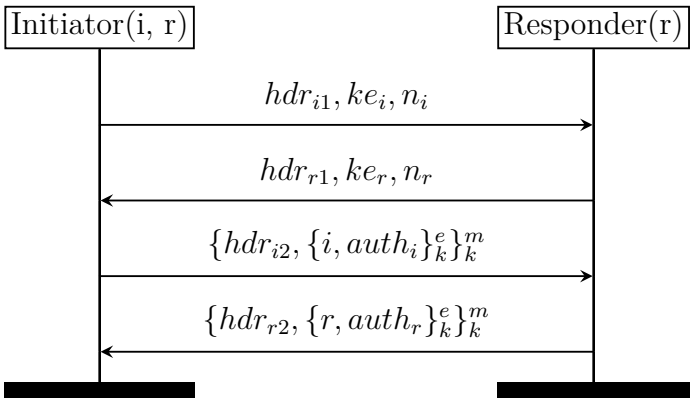
IKEv2-Sig consists of one init exchange and one auth exchange. It uses digital signature authentication. Figure 2.3 shows the MSC of IKEv2-Sig. The initiator starts by generating a DH private part x_i and a nonce n_i. The initiator then computes the DH public part $ke_i = g^{x_i}$ (ke stand for key-exchange payload), where g is a public constant called the DH generator (often $g = 3$). Note that all our subprotocols use the finite field variant of the Diffie-Hellman protocol. IKEv2 supports other variants, such as ECDH [33], but we focus on the finite field variant in this thesis. The initiator then sends ke_i alongside n_i and a header hdr_{i1} in the init request to the responder. We present headers later in this Section. Figure 2.2 sums up our notation.

Upon reception of the init request, the responder generates a DH private part x_r and a nonce n_r, computes $ke_r = g^{x_r}$ and sends n_r and ke_r together with a header hdr_{r1} in the init response to the initiator.

Upon reception of the init response, the initiator computes $k = (ke_r)^{x_i}$. k is the secret shared key resulting from the Diffie-Hellman protocol: on its side, the responder will later compute $(ke_i)^{x_r} = (ke_r)^{x_i} = k$. The initiator then computes a digital signature $auth_i = \{ke_i, n_i, n_r, k, i\}^s_{prk(i)}$ over values ke_i, n_i, n_r, k and i using its private key $prk(i)$. i denotes the identity of the initiator (it can be a Distinguished Name for example). Note that in practice, it is not the shared secret that is included in the signature, but rather *SK_pi* [46], which is derived from the shared secret. However, as we will explain in Section 2.2.4, we model the shared secret and all keys that derive from it by k. Therefore we use k in the signatures of our subprotocols.

Syntax	Semantics
$prk(a)$	Private key of a
$pbk(a)$	Public key of a
$psk(a, b)$	Pre-shared key of a and b
g	Diffie-Hellman generator
hdr_{al}	lth IKEv2 header sent by a
ke_a	Key-exchange payload sent by a
n_a	Nonce payload sent by a
$auth_a$	Authentication payload sent by a
x_a	Diffie-Hellman exponent of a, which is not sent
rf_{al}	lth Response flag field sent by a
if_{al}	lth Initiator flag field sent by a
mid_{al}	lth message ID field sent by a
$\{msg\}_k^e$	Symmetric encryption of msg using key k
$\{msg\}_k^s$	If k is a private key, then this payload is the digital signature of msg using k. If k is a pre-shared key, then this payload is the MAC of msg using k
$\{msg\}_k^m$	msg in cleartext affixed with a MAC of msg computed using key k
q^r	Exponentiation of q by r in a finite cyclic group G of order n (modular exponentiation), where q is an element and generator of G and r is an integer satisfying $1 < r < n$.
$Initiator(i, r)$	Agent i is taking the role of initiator and wants to perform the protocol with agent r
$Responder(r)$	Agent r is taking the role of responder and can perform the protocol with whatever agent correctly authenticates itself and is trusted by r
$Responder(i, r)$	Agent r is taking the role of responder and can only perform the protocol with agent i
k (in IKEv2-Child)	Pre-shared key for IKEv2-Child

Figure 2.2: Our syntax for payloads and fields of IKEv2's subprotocols. In this syntax, a and b denote agents, and $l \in \mathbb{N}^*$. We write hdr_a, rf_a, if_a and mid_a when the subprotocol contains only one exchange.



$ke_i = g^{x_i}$
$ke_r = g^{x_r}$
$keymat = k$
$k = (ke_i)^{x_r} = (ke_r)^{x_i}$

$hdr_{il} = (rf_{il}, if_{il}, mid_{il})$
$hdr_{rl} = (rf_{rl}, if_{rl}, mid_{rl})$
$auth_r = \{ke_r, n_r, n_i, k, r\}_{prk(r)}^s$
$auth_i = \{ke_i, n_i, n_r, k, i\}_{prk(i)}^s$

Figure 2.3: The IKEv2-Sig protocol. IKEv2-Sig is made of an init exchange, which runs a Diffie-Hellman protocol, and an auth exchange, which authenticates the parties to each other. In IKEv2-Sig, authentication is achieved through digital signature.

The initiator now computes $\{i, auth_i\}_k^e$: the symmetric encryption of $(i, auth_i)$ using key k. Finally the initiator sends $\{hdr_{i2}, \{i, auth_i\}_k^e\}_k^m$ in the auth request to the responder, where $\{msg\}_k^m$ denotes msg affixed with a Message Authentication Code (MAC) of msg computed using key k. A MAC aims to guarantee the integrity of the message, i.e. that the message was created by an owner of key k. Examples of algorithms used for computing MACs in IKEv2 are HMAC_MD5, HMAC_SHA1 and DES_MAC, AES_XCBC.

Upon reception of the auth request, the responder computes $k = (ke_i)^{x_r}$. The responder then checks the MAC of the auth request, decrypts the encrypted payload, and verifies the signature $auth_i$. If MAC and signature check are successful, then the responder creates an IKE SA and a Child SA in its SAD. Both SAs use k to protect messages. In each subprotocol, we call *keymat* (i.e. keying material) the key of the created Child SA. Hence in IKEv2-Sig we have $keymat = k$ (note that in practice, *keymat* is derived

from k, see Section 2.2.4). The responder then sends $\{hdr_{r2}, \{r, auth_r\}_k^e\}_k^m$ in the auth response to the initiator. In this message, r denotes the identity of the responder. If MAC or signature check fail, then no SA is created.

Upon reception of the auth response, the initiator checks the MAC of the auth response, decrypts the encrypted payload, and verifies the signature $auth_r$. If MAC and signature check are successful, then the initiator creates an IKE SA and a Child SA in its SAD. Otherwise no SA is created.

Let us explain the content of headers. We denote a header by hdr_{al} (see Figure 2.5). The Response flag rf_{al} of a message is set to 1 when the message is a response. The Initiator flag if_{al} is set to 1 when the message is sent by the initiator of the IKE SA. The message ID mid_{al} is an integer that starts with value 0 (in the init exchange) and is incremented at every new exchange. Its role is to prevent replay attacks, as well as to detect retransmissions and lost messages. IKEv2 needs to detect lost messages because IKEv2 messages are transported over UDP which is not a reliable protocol (UDP datagrams may be lost in the network).

2.2.2 IKEv2-PSK

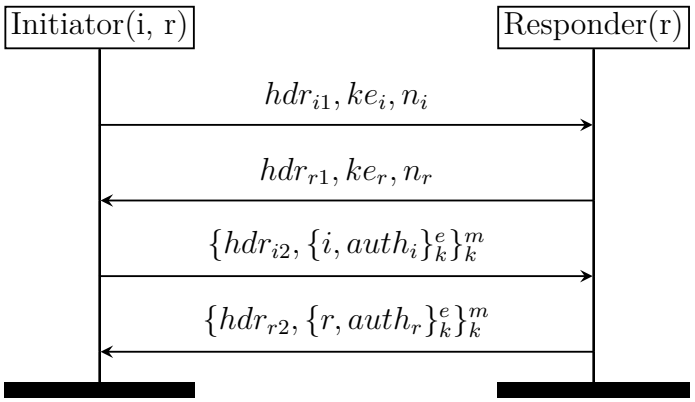
In this Section, we present the IKEv2-PSK subprotocol.

The only difference between IKEv2-PSK and IKEv2-Sig is their mode of authentication: IKEv2-PSK uses a pre-shared key, whereas IKEv2-Sig uses digital signature. Figure 2.4 shows the MSC of IKEv2-PSK. We note $psk(a, b)$ the pre-shared key of two agents a and b. When using pre-shared key authentication mode, instead of computing a digital signature using their private key, parties compute a MAC using the pre-shared key. For simplicity we use the same notation $\{msg\}_k^s$ for the digital signature and MAC using key k (there is no confusion possible with $\{msg\}_k^m$ because $\{msg\}_k^m$ denotes msg in clear *affixed* with a MAC). Thus in IKEv2-PSK, the initiator computes $auth_i = \{ke_i, n_i, n_r, k, i\}_{psk(i,r)}^s$. The responder computes its authentication payload analogously.

2.2.3 IKEv2-Child

In this Section, we present the IKEv2-Child subprotocol.

We define IKEv2-Child as one child exchange that includes key-exchange payloads (hence where a Diffie-Hellman protocol is run). In this thesis, we do not model the child exchange where key-exchange payloads are omitted.



$ke_i = g^{x_i}$
$ke_r = g^{x_r}$
$keymat = k$
$k = (ke_i)^{x_r} = (ke_r)^{x_i}$

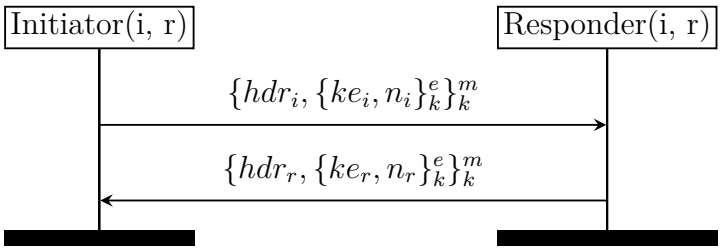
$hdr_{il} = (rf_{il}, if_{il}, mid_{il})$
$hdr_{rl} = (rf_{rl}, if_{rl}, mid_{rl})$
$auth_r = \{ke_r, n_r, n_i, k, r\}_{psk(i,r)}^s$
$auth_i = \{ke_i, n_i, n_r, k, i\}_{psk(i,r)}^s$

Figure 2.4: The IKEv2-PSK protocol. IKEv2-PSK is made of an init exchange, which runs a Diffie-Hellman protocol, and an auth exchange, which authenticates the parties to each other. In IKEv2-PSK, authentication is achieved by using a pre-shared key.

Figure 2.5 shows the MSC of IKEv2-Child. Recall that a child exchange is always protected by an IKE SA. Before a child exchange, its IKE SA was created in an init exchange, authenticated in an auth exchange, and possibly rekeyed many times using previous child exchanges. We call k the key of the IKE SA of a child exchange, to match the notation we used for the init and auth exchanges, but from the perspective of a child exhange, k is nothing but a pre-shared key (that was actually negociated during the mother IKE SA setup).

In IKEv2-Child, the initiator generates a DH private part x_i, a nonce n_i, and computes the DH public part $ke_i = g^{x_i}$. The initiator then encrypts the tuple (ke_i, n_i) using k. Finally, the initiator creates a header hdr_i, protects the tuple $(hdr_i, \{ke_i, n_i\}_k^e)$ using a MAC computed with k, and sends the result in the child request to the responder.

Upon reception of the child request, the responder checks that the MAC is correct



$$hdr_i = (rf_i, if_i, mid_i)$$
$$hdr_r = (rf_r, if_r, mid_r)$$
$$keymat = (ke_i)^{x_r} = (ke_r)^{x_i}$$
$$ke_i = g^{x_i}$$
$$ke_r = g^{x_r}$$

Figure 2.5: The IKEv2-Child protocol. IKEv2-Child runs a Diffie-Hellman protocol, protected by a pre-shared key.

and decrypts the request. The responder then generates a DH private part x_r, a nonce n_r, computes $ke_r = g^{x_r}$, and computes $keymat = (ke_i)^{x_r} = (ke_r)^{x_i} = g^{x_i * x_r}$. The responder can now create a new IKE SA in its memory, replacing the current IKE SA, or create or new Child SA, replacing or not a current Child SA. In practice, the responder knows which action to perform by looking at some other payloads, which we omitted in our IKEv2-Child subprotocol (absence of traffic selector payloads tells the responder to rekey the IKE SA, and presence of the REKEY_SA notification payload tells the responder to rekey the Child SA). However, what parties do at this point does not matter in our models. Finally, the responder encrypts (ke_r, n_r) using k, protects the tuple $(hdr_r, \{ke_i, n_i\}_k^e)$ using a MAC computed with k, and sends the result in the child response to the initiator.

Upon reception of the child response, the initiator checks that the MAC is correct and decrypts the response. The initiator then computes $keymat = (ke_r)^{x_i} = (ke_i)^{x_r} = g^{x_i * x_r}$. The initiator can now create a new IKE SA in its memory, replacing the current IKE SA, or create a new Child SA, replacing or adding up to a current Child SA.

2.2.4 Simplifications compared to the RFC

Because IKEv2 is a complex protocol, we made some simplifications in the definition of our subprotocols. In this Section, we present the different simplifications we made.

Firstly, we did not keep all IKEv2 payloads and fields. For example, we did not include the traffic selector (TS) payloads. TS payloads are used to specify which IP addresses will be allowed to communicate through the resulting Child SA. TS payloads are not used in any cryptographic operation. They neither play a role in key generation nor in authentication. Therefore, we assume that this simplficiation does not remove any attacks (i.e. does not make us overlook some attacks in our analysis).

We did not include Security Association (SA) payloads either. SA payloads are used to specify which cryptographic algorithms will be used in the SA. Our analysis is performed in the symbolic model (see Section 3.1.1), so which specific algorithm is used, e.g. for encryption, is not relevant. Therefore, we assume that this simplification does not remove any attacks.

Other abstractions we make concern the way cryptographic values are computed by parties.

In the IKEv2 RFC, at the end of the IKE_SA_INIT exchange, the two parties compute the *SKEYSEED* value. From this value they then compute the values SK_ai, SK_ar, SK_ei, SK_er, SK_pi, SK_pr and SK_d. SK_ai, SK_ar, SK_ei and SK_er are used to protect messages of the IKE SA. SK_pi and SK_pr are used when computing the AUTH payload. SK_d is used to generate the keying material *KEYMAT* of Child SAs through some derivation function. When a DH exchange is involved in the creation of a new Child SA (CREATE_CHILD_SA exchange with KE payload), the resulting new shared secret is used in the derivation function thus providing the Child SA with the PFS property.

In our model, all values *SKEYSEED*, SK_ai, SK_ar, SK_ei, SK_er, SK_pi, SK_pr and SK_d are represented by a single value k. Therefore k plays the role of a pre-shared key for IKEv2-Child (since SK_d does). Finally, *KEYMAT* is represented by *keymat*.

Because key derivation algorithms of IKEv2 are public, an attacker who learns *SKEYSEED* can also learn all SK_* keys. However, the opposite is not true because key derivation in IKEv2 uses a hash function. In our model, compromising one of these values is equivalent to compromising all other values. By removing the one-way derivation function from our model, we actually grant additional capabilities to the attacker thus enlarging the attack surface compared to real-life IKEv2. Hence this simplification does not remove any attacks.

2.3 Security goals

Protocol specifications try to thorougly prescribe the protocol payloads and how these payloads should be generated and processed. However they tend to be much less clear about what precise security properties the protocol aims to satisfy, and in what context. In fact, the RFC of IKEv2 only speaks of "mutual authentication" and "shared secret information". No definition of authentication is given and nothing is said about the strength of the adversary from which IKEv2 claims to protect the agents running the protocol. In this Section we describe an adversary to which we deem IKEv2 is resiliant, and we give precise definitions of some properties that we deem IKEv2 aims to satisfy.

2.3.1 Adversary model

Let us introduce the two adversary models that we will use in our formal analysis.

[5] translates several adversary models from the literature into their own formalism. In this thesis, we will verify our properties in two of these adversary models: the external Dolev-Yao model (*AdvEXT*) and the internal Dolev-Yao model (*AdvINT*).

AdvEXT is a minimalistic symbolic model. As in any symbolic model, messages exchanged by parties are symbolic terms, and cryptography is assumed to be flawless, i.e. the attacker can only decrypt a message if it possesses the decryption key. In addition, in AdvEXT, the attacker has full control over the network: it can intercept, drop, and learn from all messages that are sent over the network, and can inject its self-forged messages into the network.

As the reader will see in Section 2.3.2, all our properties are of the form: "Whenever an agent has completed the protocol, etc.". We call this agent the *actor*. We call the protocol session the actor has completed the *test session*. Finally, we call the agent with whom the actor thinks it has completed the protocol the *peer*. We borrow these three names from [5].

In AdvINT, the attacker has the same capabilities than in AdvEXT, but is also given the *LKRothers* capability. The LKRothers capability allows the attacker to compromise, before the test session has started, the long-term keys of any agent that is neither the actor nor the peer. AdvINT corresponds to the Dolev-Yao model introduced in [30]. AdvINT also corresponds to the model used in [52] to find a famous attack on the Needham-Schroeder protocol.

2.3.2 Properties

When protocol specifications do tell what security guarantees they aim to provide, their formulation can be quite vague. In particular, the "mutual authentication" mentioned in [46] is an unclear notion. For this reason, Lowe has split authentication into several well-defined properties [51]. Similarly, secrecy has several possible definitions. For example, secrecy may be understood as the unability for the attacker to compute a given value. Secrecy may also be understood as indistinguishability under chosen plaintext attack, which roughly states that, given the ability of obtaining the ciphertexts for arbitrary plaintexts, the attacker should not be able to guess the plaintext of some cyphertext better than random. In this section, we provide an informal definition of the security properties that we will verify on IKEv2 in this thesis.

Secrecy of *keymat* This property states that whenever an agent has completed the protocol, the value that the agent computes for the term *keymat* will never be known (in the sens of adversary knowledge in the symbolic model) to the attacker.

Aliveness This property states that whenever an agent A has completed the protocol, apparently with an agent B, then B has previously been running the protocol.

Weak agreement This property states that whenever an agent A has completed the protocol, apparently with an agent B, then B has previously been running the protocol, apparently with A.

Non-injective agreement This property states that whenever an agent A has completed the protocol, apparently with an agent B, then B has previously been running the protocol, endorsing the correct role, apparently with A, and A and B agree on some terms, e.g. *keymat*.

Injective agreement This property states that whenever an agent A has completed the protocol, apparently with an agent B, then B has previously been running the protocol, endorsing the correct role, apparently with A, and A and B agree on some terms, and if A completes the protocol n times computing the same agreement terms, then B has completed the protocol at least n times computing these same agreement terms.

We call *commit step* the time at which an agent completes the protocol. In IKEv2-Sig and IKEv2-PSK, the peer of the initiator is the agent whose identity is the r payload

it receives in the auth response, whereas the peer of the responder is the agent whose identity is the i payload it receives in the auth request. In IKEv2-Child, the peer is the agent that shares the pre-shared key k with the actor.

Properties may be satisfied only for some role. For example, secrecy may be guaranteed to the initiator but not to the responder. When we do not mention the role endorsed by the actor, we consider our properties satisfied if they are satisfied whatever role the actor is endorsing, i.e. if they are satisfied for both the initiator and the responder.

"B has previously been running the protocol" means that B has at least sent its last message. Obviously, A cannot have any stronger guarantee: if B's last event is a "receive", then the protocol cannot prove to A that this event was triggered. We call *running step* the time at which an agent sends its last message.

Aliveness, weak agreement, non-injective agreement and injective agreement are authentication properties. They were first defined by Lowe in [51]. Note that injective agreement implies non-injective agreement, which in turn implies weak agreement, which in turn implies aliveness. Ideally, we would like IKEv2 to satisfy the strongest property, i.e. injective agreement.

For IKEv2-Sig and IKEv2-PSK, we make parties agree upon the content of *keymat*. For IKEv2-Child, however, we cannot guarantee to the responder that the initiator has computed *keymat* since the initiator has already sent its last message when it computes *keymat*. Therefore, for agreement in IKEv2-Child, we only guarantee to the responder that the initiator has computed *kei*. To the initiator we guarantee that the responder has computed *keymat*, and we make the responder agree on *keymat* at the running step (which the responder can, since at the running step it already received the child request). This allows us to check that in practice, whenever the initiator has completed the protocol and computed some value for *keymat*, then the responder *will* compute the same value for *keymat* if it continues the protocol.

2.4 Implementations

In this Section, we list some of the most popular IKEv2 implementations, be they proprietary or open-source.

The IKEv2 protocol is implemented in a number of proprietary products. Microsoft includes its own implementation in its Windows 10 operating system [43]. ICSA Labs, an independent division of Verizon that provides security certifications to product developers,

has provided IPsec and IKEv2 certifications for companies A10 Networks, F5 Networks Inc. and Fortinet, Inc [39]. Fortinet offers an IPsec/IKEv2 VPN solution in appliances or in cloud [41]. A10 Networks includes its IPsec implementation in e.g. its Thunder CFW firewall product [42]. Cisco offers IPsec/IKEv2 in its AnyConnect software solution [16], which can be integrated in many Cisco appliance products.

20 years ago, IKEv1 has been implemented in open-source software FreeS/WAN [32]. Development stopped in 2004, and led to two forks: Openswan [64] and strongSwan [83]. Both softwares are still maintained and now support IKEv2. Libreswan [49] is another open-source implementation of IKEv1 and IKEv2 that forked from Openswan in 2012 and is still maintained. History of Swan forks can be found at [85]. OpenBSD includes another IKEv2 implementation called iked [63]. The development of the other open-source IKEv2 implementations racoon2 and Linux ipsec-tools has been abandoned [44, 72].

2.5 Concluding remarks

In this Section, we make some closing remarks and mention adversary models and properties on which we have chosen not to focus, but which could constitute interesting further work.

We have chosen to focus on the AdvEXT and AdvINT adversary models. We could have considered other ones. In [5], the authors provide a classification of many adversary models from the literature. All these models can be derived from AdvEXT by granting some additional capabilities to the attacker. For example, the LKRactor capability allows the attacker to learn the long-term keys of the actor at anytime. The LKRafter capability allows the attacker to learn the long-term keys of any agent after the end of the test session. In fact, verifying secrecy in a model where the attacker has LKRafter is equivalent to verifying perfect forward secrecy.

Similarly, there are other security properties that we could have considered in our adversary models. For example, we could have checked the secrecy of other terms than just *keymat*, as well as agreement on other terms than just *keymat*.

Chapter 3

Verifying the specification

In this Chapter, we perform a formal analysis of the IKEv2 specification using three different tools: Spin, ProVerif and Tamarin. We start by giving some background in Section 3.1 on the formal verification of security protocols, and in particular on previous formal analyses of IKEv2. We then evaluate in Section 3.2 the means that are currently available to analyze security protocols with Spin. We observe that the best existing method for modeling a security protocol in Spin is not powerful enough for the needs of a complex protocol such as IKEv2. We tackle this problem in Section 3.3 by proposing several improvements to the method, and apply our improved method on IKEv2. Ultimately in Sections 3.4 and 3.5 we formally analyze IKEv2 using the two tools ProVerif and Tamarin, which unlike Spin are specialized in formal verification of security protocols. We conclude in Section 3.6 by comparing the three methods.

3.1 Background

In this Section we provide some context about formal verification, its application to security protocols, and more specifically to IKEv2. We firstly present in Section 3.1.1 the concept of formal verification and how it can be applied to security protocols. We then survey previous analyses of IKEv2 and their results in Section 3.1.2. Finally in Section 3.1.3 we introduce some formal verification tools that can be used to analyze security protocols.

3.1.1 Model checking security protocols

Let us go over the principles of model checking and see how they can be applied to security protocols.

Formal verification is the act of proving or disproving that a system satisfies some property using a mathematically based technique. Model checking is a formal verification

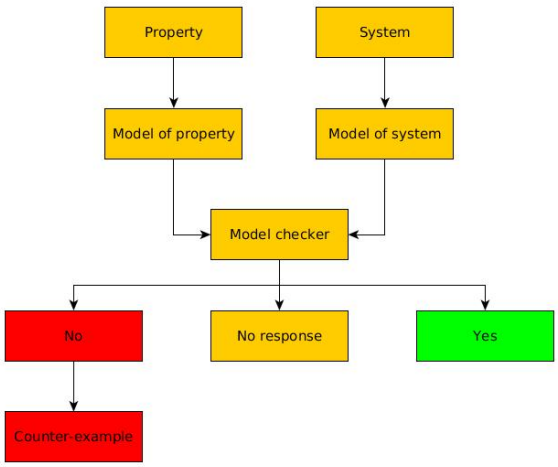


Figure 3.1: The model checking workflow. Either proves that a system satisfies a property, or returns a counter-example, or is unable to answer.

technique in which we represent the system by a model, whose semantics is a transition system, and explore systematically and exhaustively all its states and transitions in order to prove that it satisfies the property. We focus on Linear Temporal Logic (LTL) [50] properties. LTL is built upon boolean logic with the addition of time indicators like *always* (denoted $\Box$) and *eventually* (denoted $\Diamond$). The linear-time property is verified on each and every execution trace of the model. The model is written in a specific modeling language, such as Applied Pi Calculus [1] or Promela [34]. The model checker takes as input the system model, as well as a property over the model state variables, and either returns *Yes*, returns *No*, or does not give any response (e.g. by not terminating). When they return no and when they can, some model checkers also give a counter-example, i.e. an execution trace of the model that contradicts the property. Figure 3.1 sums up the steps of the model checking process. The principles of model checking are explained in great details in [4, 17].

In the adversary models described in Section 2.3.1, the adversary's knowledge and the advancement of some agents in their execution of the protocol can be seen together as constituting a symbolic state. The actions "an agent sends a message", "an agent receives a message", "the adversary builds a message and sends it", etc., can be seen as actions modifying the state. Such a model thus lends itself well to model checking techniques.

Therefore, in our case, the system mentioned earlier is a protocol specification, played in some adversary model (capabilities given to the intruder), and the properties are security properties, like secrecy and authentication. Different techniques can be used to

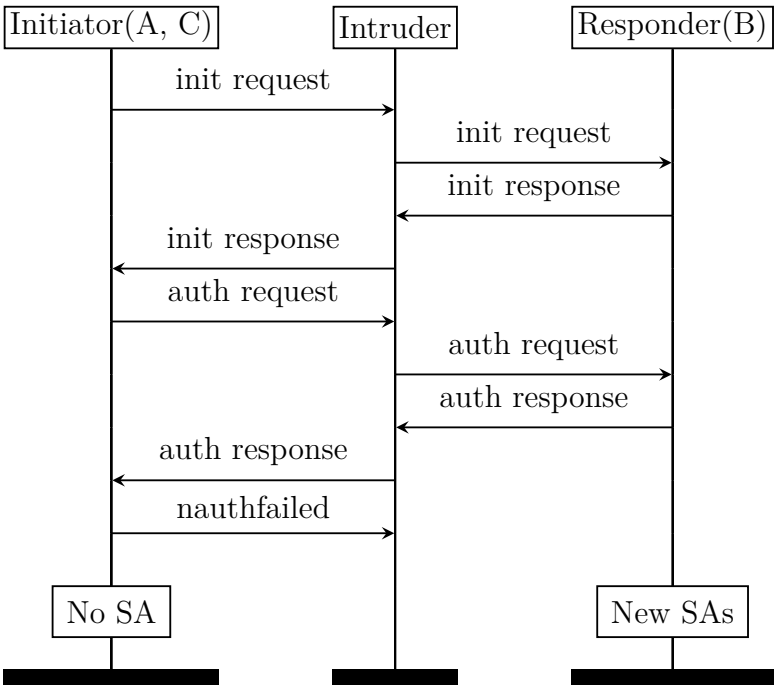


Figure 3.2: The penultimate authentication flaw. Violation of weak agreement for B: B's peer A wanted to talk to C, not B. In this MSC, the intruder does not change the IKEv2 content of the messages it deviates.

represent and explore the model, which model checkers abstract away: Models can be explored in a forward or backward manner. One can allow a finite or infinite number of runs (i.e. protocol executions). States can be represented explicitly or symbolically. Finally, abstractions can be used to trade completeness for efficiency. A thorough state-of-the-art of model checking security protocols is depicted in [6].

3.1.2 Previous analyses

Let us now go over the different existing formal analyses of IKEv2 and summarize their results.

In 1999, Meadows finds two authentication weaknesses in IKEv1 [54], using the NRL protocol analyzer. The first one is a reflection attack, and the second one is called the penultimate authentication flaw.

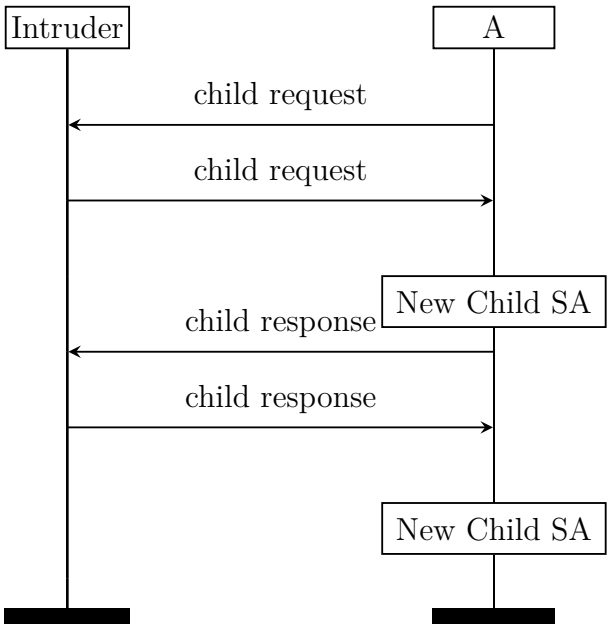


Figure 3.3: The reflection attack. Violation of aliveness for the responder in IKEv2-Child. In this MSC, the intruder does not change the IKEv2 content of the messages it replays.

In 2003, IKEv2 is formally verified in the context of the AVISPA project [70]. The authors verify secrecy and authentication on IKEv2-Sig, IKEv2-PSK, and IKEv2-Child. The AVISPA analysis yields that IKEv2, like IKEv1, suffers from the penultimate authentication flaw. The penultimate authentication flaw is an execution trace of IKEv2 that violates the weak agreement property for the responder in IKEv2-Sig. Figure 3.2 shows its MSC. Recall that weak agreement states that whenever an agent A has completed the protocol, apparently with an agent B, then B has previously been running the protocol, apparently with A. In the penultimate authentication flaw, A starts a session as initiator and wants to talk to an agent C (which may be honest). But the intruder deviates towards B every message sent by A. B sends its responses to A as expected. The parties proceed naturally until A receives the auth response. The AUTH payload is not signed with the private key of C, so A does not set up a Child SA. A then sends an IKEv2 INFORMATIONAL message containing an AUTHENTICATION_FAILED notification payload (which we abbreviate *nauthfailed*). Intruder intercepts it and drops it. In the end, B has set up a Child SA with A, whereas A did not want to set up a Child SA with B.

This is a violation of weak agreement for the responder.

For its part, IKEv2-PSK is not vulnerable is because, in theory, A and B do not share the same PSK as A and C. However, in practice, PSKs are sometimes shared among several machine pairs, such as between all IKEv2 servers and clients of a company's VPN access infrastructure. In those cases, weak agreement is violated as well.

The authors of [70] say that the penultimate authentication flaw cannot be exploited for further purposes. They propose a countermeasure anyway, which they call *key confirmation*. Indeed the penultimate authentication flaw is not a full violation of the intuitive definition of authentication, because there is no actual impersonation and secrecy is still satisfied. However, we show in Section 4 that the penultimate authentication flaw allows a Denial-of-Service attack.

In 2009, Kusters and Truderung use their tool DH-ProVerif to verify IKEv2 [48]. DH-ProVerif is an extension of ProVerif [9] that has a better model of modular exponentiation than ProVerif. Their analysis seems to confirm the results of the AVISPA project, i.e. IKEv2 suffers from the penultimate authentication flaw, but not from any reflection attack.

In 2010, Cremers performs an extensive analysis of IKEv2 [24] using the Scyther tool. He verifies the secrecy, aliveness and weak agreement properties on IKEv2-Sig, IKEv2-PSK and IKEv2-Child in many adversary models. [24] confirms that IKEv2 suffers from the penultimate authentication flaw and, like in the AVISPA project, concludes that this vulnerability is harmless. However, [24] finds that the reflection attack that was noted for IKEv1 is also possible on IKEv2. The reflection attack found by [24] is an execution trace that violates aliveness for both the initiator and the responder in IKEv2-Child. Figure 3.3 shows its MSC. In this attack, an agent A sends a child request to an agent B. The attacker replays this request to A. A commits (i.e. completes the protocol, sets up a Child SA) as a responder with B. A replies with an child response, which the attacker replays to A. A commits as an initiator with B. This results in a violation of aliveness for both the initiator and the responder, because A has commited as both an initiator and a responder with B, but B is not alive (i.e. B has not run the IKEv2-Child protocol). We will show in this thesis that this attack is in fact not possible: it is prevented by the Initiator flag of the IKEv2 header.

3.1.3 Some formal verification tools

Various model checking tools have been developed following various paradigms and for different purposes. In this Section, we introduce three of these tools: Spin, ProVerif and Tamarin, which we use in this thesis to analyze IKEv2.

Spin [37] is an explicit-state model checker. It takes Promela [34] as input language and was designed to check LTL properties on asynchronous process systems. Spin translates processes into finite-state automata (hence the adjective explicit-state), performs an interleaving product on them and searches the resulting state space for a property violation. Since a protocol is an asynchronous process system, and since all the properties we want to verify are safety properties (which are LTL properties), Spin can be used for protocol verification. However, it lacks native support for cryptographic primitives and for an adversary model. To solve this problem, one can use the method introduced in [8]. We describe this method later in Section 3.2.2.

We observe that, even with the method of [8] and our improvements to this method, Spin can only produce proofs in a bounded model. Other tools that are specialized in formal verification of security protocols do not suffer from these limitations. Such tools include ProVerif and Tamarin.

ProVerif [9] performs proofs in the unbounded model, i.e. messages may be of arbitrary size and there may be an arbitrary number of sessions in an execution trace. ProVerif takes as input a description of the protocol to verify in a dialect of the applied pi calculus [1]. The applied pi calculus is itself an extension of the pi calculus [57] with cryptographic primitives. The pi calculus is a process calculus, i.e. a language for describing and analyzing properties of concurrent systems. The protocol to verify is described in the input model by a set of functions symbols (i.e. encryption, pairing, etc.) that can be used to build terms (which represent cryptographic messages), a set of equations over these function symbols (describing e.g. the relationship between encryption and decryption), a description of the message exchanges of the protocol, and finally a description of the properties to verify. The intruder is implicit and corresponds to AdvEXT (see Section 2.3.1). To verify a property, ProVerif translates the protocol model into a set of Horn clauses, and then tries to deduce from the clauses that the property is falsified (i.e. that there is an attack). The translation into Horn clauses is made in such a way that if no attack is found at the Horn clause level, then there is no attack at the applied pi calculus level, and thus we have

proven the property. However, the translation performs some abstractions, so an attack at the Horn clause level does not necessarily mean that there is an attack at the applied pi calculus level. Therefore, if ProVerif finds an attack at the Horn clause level that is not possible at the pi calculus level, then ProVerif answers that the property "cannot be proven". Otherwise, an attack is reported. Finally, ProVerif may not terminate at all.

Since we are verifying IKEv2 in this thesis, we need to model modular exponentiation. ProVerif supports modular exponentiation but models only a subset of the algebraic properties of modular exponentiation. DH-ProVerif [48] is an extension of ProVerif that has a stronger model of modular exponentiation than ProVerif. DH-ProVerif directly takes Horn clauses as input model.

Tamarin [55] is the most recent of the tools presented here. Like ProVerif, Tamarin performs proofs in the unbounded model. Like ProVerif as well, Tamarin takes as input a set of functions symbols, a set of equations over these function symbols, a description of the message exchanges of the protocol, and a set of properties to verify. The state of the protocol is modeled by a multiset of facts. Facts are built using fact symbols applied to terms. Facts make statements about terms. For example, `K(x)` is a fact denoting that the adversary knows term `x`. Rules are multiset rewriting rules, meaning that they replace a multiset of facts from the state with another multiset of facts. Each time a rule applies, it may trigger some action facts. Some timepoint value is associated to each rule application. Lemmas are first-order formulae over action facts and timepoints. Message exchanges are modeled by rules, and properties are modeled by lemmas. Tamarin does not perform any abstractions, but offers the possibility to the user to interact with it in order to guide it during resolution. Furthermore, the user can write intermediary lemmas, which Tamarin will reuse when proving other lemmas. This feature can effectively guide Tamarin towards a proof. A significant advantage of Tamarin over ProVerif is that Tamarin can model protocols with global states, such as protocols with counters.

3.2 Evaluating Spin to analyze security protocols

In this Section, we evaluate the current feasibility of using Spin to analyze security protocols. We start by detailing the workings of Spin. We then describe an existing method that allows verifying security protocols using Spin. Finally, we show that this method has limitations and that we need to overcome some of them in order to analyze more complex

protocols such as IKEv2.

3.2.1 The Spin model checker

Let us look at the mechanics of Spin and its modeling language Promela. In this Section we firstly describe how Spin internally represents a model and sketch the verification algorithm that Spin uses to prove some property on the model. We then provide the syntax of each element of the Promela language and show how they are translated into elements of the internal model of Spin. Finally, we go over some small example of Promela code and analyze it using Spin.

Verification algorithm

The behaviour of Spin is described in [71] and in [4]. Spin internally translates a Promela model into some structure called *global system state*. A global system state contains *processes*, *channels* and global variables.

A channel is an ordered set of *messages*. A message is an ordered set of variables. In the case of IKEv2 verification, we are only interested in *synchronous* channels, i.e. channels that cannot store any message. When a process sends a message over a synchronous channel, another process must receive the message from the channel at the same time.

A process contains a finite set of local variables, a finite set of integers representing local states, an initial state, a current state, and a *transition relation* defined on the local states of the process. A transition relation is a finite set of *transitions*. A transition contains a source state, a target state, an *executability condition*, and an *effect function*. An executability condition is a boolean condition on the global system state. An effect function modifies the global system state.

Initially, the init process is in its initial state, and all channels are empty. Spin then recursively chooses one transition among all *executable* transitions of all processes, applies the effect of the transition to the global system state, and updates the current state of the process of the transition with the target state of the transition.

Transitions describing message sending and receiving on synchronous channels require special treatment by Spin. These transitions must happen at the same time as some matching transition of another process. Therefore, when Spin chooses a send or receive transition, Spin will look for some matching transition and execute both transitions at the same time. If there are no matching transitions, Spin does not execute the transition

and chooses another one.

The goal of Spin is to explore all possible states of the global system, while checking some property. Spin can verify any linear-time property, but we focus on safety properties, as secrecy and authentication are safety properties. To check a safety property in Spin, one places an assertion transition in one of the processes. While exploring all states of the global system, if Spin executes an assertion transition and the assertion fails, then Spin reports a violation of the property. If all states of the global system have been explored, and no assertion has failed, the property has been proved.

Syntax of Promela

We will consider only the minimal subset of Promela that we need in order to model IKEv2. In our syntax description, vertical bars `|` separate choices. Square brackets `[ ... ]` indicate optional parts. Actual square brackets are surrounded by single quotes (e.g. `'[' ... ']'`). A Kleene star `*` denotes zero or more repetitions of some part. Uppercase names denote keywords from the language. These keywords are written lowercase in Promela models. Lowercase names refer to rules from the syntax. We omitted the rules of `name` and `string` for simplicity.

A Promela file is made of several modules, each of them being either a process (rule proctype), an initialization process (rule init), the definition of mtype names, or the declaration of global variables (rule decl_lst).

```
module      : mtype
            | decl_lst
            | proctype
            | init
```

mtype names are defined using the following rule.

```
mtype       : MTYPE = { name [ , name ]* }
```

mtype names can be thought of as C macros. For example, the declaration

```
mtype = { foo, bar, ter };
```

is functionally equivalent to the sequence of macro definitions:

```
#define foo     3
#define bar     2
#define ter     1
```

Variables are declared using the decl_lst rule hereafter. Promela supports the types bit, bool, byte, short, int, mtype and channel. Using square brackets after name in ivar allows declaring a one-dimensional array of size const. expr denotes an expression. Expressions may either be contants, variables, or boolean conditions over other expressions.

```
decl_lst    : one_decl [ ; one_decl ]*

one_decl    : typename  ivar [ , ivar ]*

typename    : BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN

ivar        : name [ '[' const ']' ] [ = expr | = ch_init ]
```

For example, the following code declares a boolean variable called running and initializes it to 0.

```
bool running = 0;
```

The following rule describes a channel that can store at most const messages, and each of the messages must match the type signature of the channel. If const is 0, then the channel is synchronous, i.e. a send statement must always be executed simultaneously with a matching receive statement.

```
ch_init     : '[' const ']' OF { typename [ , typename ]* }
```

In the following example, we declare a variable a of type channel and initialize it to a synchronous channel that can only pass messages of type (bit, mtype, byte).

```
chan a = [0] of {bit, mtype, byte};
```

Processes are defined using the proctype rule hereafter. This rule uses the proctype keyword. A process takes a list of arguments and contains several steps, each step being either a statement or the declaration of variables. Variables declared inside a process are called local.

```
proctype    : PROCTYPE name ( [ decl_lst ] ) { sequence }

sequence    : step ; [ step ; ]*

step        : stmnt
            | decl_lst
```

For example, the following code declares a process Proc. Proc takes two arguments foo and bar of respective types int and mtype, declares a local variable ter, and assigns foo to ter.

```
proctype Proc(int foo, mtype bar)
{
    int ter = foo;
}
```

Transitions mentioned in Section 3.2.1 are inferred by Spin from Promela statements. Statements are defined using the following rule.

```
stmnt       : IF :: sequence [ :: sequence ]* FI
            | DO :: sequence [ :: sequence ]* OD
            | ELSE
            | BREAK
            | GOTO name
            | name : stmnt
            | ATOMIC { sequence }
            | send
            | receive
            | ASSERT expr
            | assign
            | expr
            | RUN name ( [ arg_lst ] )
```

Some statements are classic control flow instructions. An `IF` statement tells Spin to create one transition for each option. Each transition departs from the current state. A `DO` statement does the same, but also creates a transition from each target state towards the current state, so that the `DO` statement is executed over and over until a `BREAK` statement is encountered. The `ELSE` statement is executable if and only if no other statement within the same process is executable in the current state. `ELSE` is intended to be used as the first `step` of an `IF` or `DO` contruct. Outside these constructs, `ELSE` is always executable and has no effect. The `name : stmnt` statement places some label. A `GOTO` statement that uses the same name creates a transition from the `GOTO` to the label. The GOTO transition is always executable and has no effect.

Sometimes one wants to model the fact that processes wait for some process to execute a given sequence of statements. An `ATOMIC` statement tells the semantics system to not

interleave the enclosed statements with statements from other processes. Atomic sequences can significantly reduce the complexity of verification, as less interleaving means less states in the global system.

Sending and receiving over some channel is done using the send and receive rules hereafter.

```
send        : varref ! expr [ , expr ]*

receive     : varref ? recv_arg [ , recv_arg ]*

recv_arg    : varref | EVAL ( varref ) | [ - ] const
```

The `send` and `receive` statements are directly translated by Spin into transitions. Since we are using synchronous channels, the executability condition of a `send` (resp. `receive`) statement is that there is a matching `receive` (resp. `send`) transition departing from the current state of another process. A `send` and a `receive` statement are matching when `varref` are the same (meaning they use the same channel), and when their arguments are matching. When some `receive` argument is `varref`, then any `send` argument is matching. When some `receive` argument is `EVAL ( varref )`, then a `send` argument is matching when its value is equal to the current value of `varref`. When some `receive` argument is `const`, then a `send` argument is matching when its value is equal to `const`. A `send` statement has no effect. A `receive` statement has the effect of assigning values to its `varref` variables. For example, in the following code we send two variables var1 and var2 and one constant cons1 over channel c. We then receive on channel c one message whose second value must equal the current value of var4 and whose third value must equal cons2. Upon reception we assign the first value to var3.

```
c!var1, var2, cons1;
c?var3, eval(var4), cons2;
```

The `assign` statement is always executable and has the effect of changing the value of some variable.

```
assign      : varref = expr
```

The `expr` statement has no effect, but is only executable if `expr` evaluates to true (i.e. does not evaluate to `0` or `FALSE`). For example in the following code, the second statement will never be executed because the first statement is an expr statement that is always false. In other words, Spin will never take the transition from the first statement to the

second statement in the transition system of the process in which these statements are.

```
var && !var;
c!var;
```

The `ASSERT` statement is used for verification. It is translated into an assertion transition, which we mentioned in Section 3.2.1. This statement is always executable. If the boolean condition `expr` evaluates to false, Spin reports an assertion violation. The following code is an example of assertion.

```
assert(var1 && (var2 > cons) || var3 != var4);
```

Upon encountering a `RUN` statement, Spin spawns a new process, assigning given values to the arguments of the process definition. For example:

```
run Initiator(var, cons);
```

Like `proctype`, the following `init` rule desribes some process, but with the particularity that this process is active in the initial global state.

```
init         : INIT { sequence }
```

For convenience, Promela also allows for some macros defined using `#define`, and some functions defined using `inline`. Those really are syntactic sugar. We give the syntax below. Functions are much like macros, except that functions can only be used as statements.

```
INLINE name ( [ arg_lst ] ) { sequence }
#DEFINE name ( [ arg_lst ] ) string
```

Example

As an example, let us consider the following Promela code. We define some constant K and some synchronous channel Comm that can pass messages of type (mtype). We define one process A that sends the constant K on channel Comm. We define one process B that receives some value on channel Comm. Then we check that B has received K. Finally we run the processes in the init process. We run the analysis using command *spin -run mwespin.pml*. Obviously, Spin finds no error, as the assertion is always true.

```
mtype = {K};

chan Comm = [0] of {mtype};
```

```
proctype A()
{
    Comm!K;
}

proctype B()
{
    mtype x;
    Comm?x;
    assert(x == K);
}

init {
    atomic {
        run A();
        run B();
    }
}
```

3.2.2 Modeling security protocols in Promela

A problem we face in IKEv2 Spin modeling is that Promela was not designed to model security protocols, but rather more basic asynchronous process systems. For example, it does not provide a simple way to model encryption. However, [8] proposes a method to model security protocols in Promela. In this Section, we present this method, successively showing how to model the protocol, the adversary and the properties. The method takes as example the following protocol, which we call P. In the following code, $\{msg\}_k^{ae}$ represents asymmetric encryption of msg using public key k, $\{msg\}_k^{se}$ represents symmetric encryption of msg using key k, and $\{msg\}_k^{s}$ represents digital signature of msg using private key k. Protocol P should ensure mutual authentication, as well as secrecy of the Sec constant. Let us call m1 and m2 the two messages of P.

$$(m1)\ i \rightarrow r : \{\{k\}^{s}_{prk(i)}\}^{ae}_{pbk(r)}$$
$$(m2)\ r \rightarrow i : \{Sec\}^{se}_{k}$$

Modeling the protocol

Let us see how the method of [8] models the protocol itself (i.e. the message flow) in Promela.

Following the method, we use three agents: A, B and C. We define a set of mtype names describing (1) all constants of the protocol, (2) a large enough pool of values that variables of the protocol may take, (3) all possible outcome values that functions may take.

For the constants, we add a name for each agent and for constant *Sec*.

Let us create the pool of values. Depending on which agent is running the protocol, variable k may take three different values, so we add names `KA`, `KB`, `KC`.

Let us add names for each possible function outcome. Functions *prk* and *pbk* can both be applied to three different agents. Thus they both have three different outcomes. Hence we add three names for each.

Finally the method defines a name `NULL` whose value is equal to the number of names plus one. `NULL` has two purposes: it is used as the size of the intruder knowledge vector, as described in Section 2.3.1, and also as a constant used for padding, as we will see later in this Section. In our example we thus declare the following names.

```
/* Agents: Alice, Bob, Charlie */
mtype = { A, B, C };

/* Private keys */
mtype = { PRKA, PRKB, PRKC };

/* Public keys */
mtype = { PBKA, PBKB, PBKC };

/* Key K */
mtype = { KA, KB, KC };

/* Some secret */
mtype = { Sec };

/* Number of names plus one */
mtype = { NULL };
```

In the method of [8], the network is modeled by a single synchronous channel. The

size of messages that can pass in the channel is set to the maximum of the sizes of all messages of the protocol. To model encryption, i.e. to model an agent sending $\{x\}_k^e$, [8] writes `Chan! k, x`. Therefore the size of a message is defined as the number of payloads it contains, plus the number of times an encryption appears in the message. In our example, the biggest message of the protocol contains one payload and two encryptions. Thus we use a channel of message size 3, which we declare as follows.

```
chan Comm = [0] of {mtype, mtype, mtype};
```

[8] further uses macros to model functions. For example, the following inline function takes a name type and an agent as arguments and assigns to var the name of the given type and of the given agent. This way, the statement `Setvar(A, PRKTYPE, var)` assigns PRKA to var.

```
inline Setvar(id, type, var)
{

    if

    ::  ( type == PRKTYPE && id == A ) -> var = PRKA
    ::  ( type == PRKTYPE && id == B ) -> var = PRKB
    ::  ( type == PRKTYPE && id == C ) -> var = PRKC

    ::  ( type == PBKTYPE && id == A ) -> var = PBKA
    ::  ( type == PBKTYPE && id == B ) -> var = PBKB
    ::  ( type == PBKTYPE && id == C ) -> var = PBKC

    fi

}
```

Each role is modeled by a process taking agent names as input. These names represent the identity of the agent running the role and the intended peer of the agent (when there is one). In our example, we thus define the initiator as follows. The `atomic` is used to remove unnecessary interleavings, effectively reducing verification complexity. The NULL constant is used as padding in the receive statement, because messages passing over a channel must always be of correct size.

```
proctype Initiator(mtype i, r)
```

```
{

    mtype pbkr;
    mtype prki;
    mtype k;
    mtype sec;

    atomic {

        Setvar(r, PBKTYPE, pbkr);
        Setvar(i, PRKTYPE, prki);
        Setvar(i, KTYPE, k);

        Comm!pbkr, prki, k;

    };

    atomic {

        Comm?eval(k), sec, NULL;

    };

}
```

Finally, processes are instantiated. We instantiate two sessions: agent A taking the role of initiator and non-deterministically intending to speak with B or C, and agent B taking the role of responder. Non-determinism is achieved using the `if` construct.

```
init {
        atomic {
                if
                :: run Initiator(A, B);
                :: run Initiator(A, C);
                fi;

        run Responder(B);
        }
```

```
}
```

Modeling the adversary

Recall that Spin does not natively offer any model for a Dolev-Yao adversary. Let us see how the method of [8] manages to create such a model.

[8] proposes an adversary model similar to the following one. We explain this code throughout this Section.

```
bool Knows[NULL + 1];
bool Keys[NULL + 1];

proctype Adversary()
{

    /*
     * p1 = payload 1
     * pp1 = previous payload 1
     */

    mtype p1 = NULL, p2 = NULL, p3 = NULL;
    mtype pp1 = NULL, pp2 = NULL, pp3 = NULL;

    do

    /* Receive from the channel */
    ::  Comm?p1, p2, p3;
        atomic {

            Addtoknowledge(p1, p2, p3);

            if
            ::  skip
            ::  pp1 = p1; pp2 = p2; pp3 = p3;
            fi;

            if
```

```
            ::  skip
            ::  Comm!p1, p2, p3
            fi

        }

    /* Forge a message and send it */
    ::  Randmessage(p1, p2, p3) ->
        atomic {
            Isvalidmessage(p1, p2, p3, pp1, pp2, pp3) ->
            Comm!p1, p2, p3;
        }

    od

}
```

The intruder is modeled by a process (just like the initiator and responder roles), that repeatedly and non-deterministically chooses between receiving from the channel, or forging a new message and sending it. Repeated non-deterministic choice is achieved using the `do` construct.

The intruder's knowledge is modeled by two boolean vectors called `Knows` and `Keys` indexed by all names. When `name` is not a key and `Knows[name]` is set to 1, it means that the intruder knows `name`. When `name` is a key and `Knows[name]` is set to 1, it means that the intruder knows the inverse key of `name` [1]. When `name` is a key and `Keys[name]` is set to 1, it means that the intruder knows `name`.

[8] further uses some local variables (pp1, pp2, pp3) (pp stands for previous payload) in the intruder process so that the intruder can additionally store exactly one full message. This represents the ability of the attacker to store some encrypted payload for later decryption in case it currently cannot decrypt the payload.

When the intruder chooses to receive a message, we store the message in variables (p1, p2, p3). The intruder updates its knowledge using a function called `Addtoknowledge`. The intruder can then choose (or not) to update his memory by storing the message into (pp1,

[1] In the context of asymmetric encryption, we call inverse key of the private key the public key, and vice versa. In the context of symmetric encryption, we call inverse key of the encryption key the encryption key itself.

pp2, pp3), thus forgetting any previously stored one. Finally, the intruder can choose to forward the message or do nothing. If the intruder does nothing, then this represents the dropping of a message.

To create a new message, the intruder calls a function called `Randmessage` that non-deterministically builds a message from all names we globally declared. The intruder then uses a macro called `Isvalidmessage` that checks the consistency of the generated message with the intruder knowledge.

In the init process, we set the intruder's initial knowledge and run the intruder process. The intruder should know all public constants: the agents, the padding constant NULL, and the public keys. Note that, following our definition of the Knows and Keys vectors, when giving a key to the intruder, we need to set the key to 1 in Keys, and its inverse key to 1 in Knows. We also give the long-term keys of C (PRKC in our example) to the intruder knowledge, thereby implementing our AdvINT adversary model from Section 2.3.1. Therefore, we also give to the intruder all names that we created for C (KC in our example).

```
init {

    atomic {

        Knows[A] = 1;
        Knows[B] = 1;
        Knows[C] = 1;
        Knows[NULL] = 1;

        Knows[PRKA] = 1;
        Keys[PBKA] = 1;

        Knows[PRKB] = 1;
        Keys[PBKB] = 1;

        Knows[PRKC] = 1;
        Keys[PBKC] = 1;

        Keys[PRKC] = 1;
        Knows[PBKC] = 1;
```

```
        Keys[KC] = 1;
        Knows[KC] = 1;

        run Adversary();

        if
        ::  run Initiator(A, B);
        ::  run Initiator(A, C);
        fi;

        run Responder(A, B);

    }

}
```

Modeling the properties

Let us see how the method of [8] represents security properties in Promela.

Each security property is expressed as a boolean condition over state variables. The boolean condition is called *invariant*. If the invariant becomes false during execution, then the corresponding property is violated. Each invariant is checked at each exection step via a dedicated "monitoring" process such as in the code below. At each execution step, if the invariant is not true (`!Invsecrecy` denotes the negation of `Invsecrecy`), the monitoring process asserts the invariant and therefore reports a violation. This way of checking an invariant was measured to be the most effective by [75].

```
proctype Monitorsecrecy()
{
        atomic
        {
                do
                :: !Invsecrecy -> assert(Invsecrecy)
                od
        }
}
```

Values of state variables are set during protocol execution. The goal of these variables is to keep track of: (a) where each agent is in its protocol execution, (b) to whom the agent believes it is talking.

State variables for authentication properties are named after the following grammar.

```
{ini,res}{running,commit}ab!
```

`inicommitab` (resp. `rescommitab`) means that A (resp. B) has completed the protocol as an initiator (resp. responder), apparently with B (resp. A). Similarly, `inirunningab` (resp. `resrunningab`) means that A (resp. B) has been running the protocol (in the sense defined in Section 2.3.2) as an initiator (resp. responder), apparently with B (resp. A).

State variables are set using four functions called `Inirunning`, `Inicommit`, `Resrunning` and `Rescommit`. As an example, `Inirunning` sets the `inirunningab` variable to 1 only in a session between A and B. Inirunning is written in the following way. The other three functions analogously set their variables.

```
inline Inirunning(i, r)
{
        if
        :: i == A && r == B -> inirunningab = 1
        :: else -> skip
        fi
}
```

We call the aforementioned functions progress functions, as they indicate the progress of the sessions in which we are interested (e.g. sessions between A and B). `Inirunning` is called in the Initiator process at the running step. Recall that the running step is right before the last send statement in a role (see Section 2.3.2). `Inicommit` is called in the Initiator process at the commit step, which is always the last statement of a role. Therefore, in the initiator process of our protocol example, progress functions are placed in the following way. The responder progress functions are called in analogous places.

```
proctype Initiator(mtype i, r)
{

    mtype pbkr;
    mtype prki;
    mtype k;
    mtype sec;
```

```
    atomic {

        Setvar(r, PBKTYPE, pbkr);
        Setvar(i, PRKTYPE, prki);
        Setvar(i, KTYPE, k);
        Inirunning(i, r);

        Comm!pbkr, prki, k;

    };

    atomic {

        Comm?eval(k), sec, NULL;

        Inicommit(i, r);

    };

}
```

[8] implements secrecy and some form of authentication. [8] implements authentication as in the code below (backslashes are required to tell Spin not to end the macro). His implementation of authentication can be seen as weak agreement with the additional condition that the peer has the correct role.

```
# define Invauth ( \
    (!inicommitab || resrunningab) && \
    (!rescommitab || inirunningab) )
```

The method of [8] implements secrecy as in the following code. This implementation states that the intruder should never have the constant Sec in its knowledge, which was the intended goal. However, when the goal of a protocol is the secrecy of some variable local to each role, then secrecy needs to be written differently. This is the case of key-exchange protocols, such as IKEv2, so we will provide another implementation of secrecy in Section 3.3.3.

```
# define Invsecrecy ( !Knows(Sec) )
```

Finally, we run the monitoring processes in the init process.

```
init {

    atomic {

        Knows[A] = 1;
        Knows[B] = 1;
        Knows[C] = 1;
        Knows[NULL] = 1;

        Knows[PBKA] = 1;
        Knows[PBKB] = 1;
        Knows[PBKC] = 1;

        Knows[PRKC] = 1;
        Knows[KC] = 1;

        run Monitorsecrecy();
        run Monitorauth();

        run Adversary();

        if
        ::  run Initiator(A, B);
        ::  run Initiator(A, C);
        fi;

        run Responder(A, B);

    }

}
```

3.2.3 Limitations of the existing method

We observe that the method of [8] suffers from several limitations, some of them to overcome in order to analyze IKEv2. Some others are due to Spin boundedness and cannot be defeated. In this Section we depict both types of limitations.

Firstly, proofs provided by Spin are rather weak because the boundedness of the tool forces us to bound our model in every aspect. The method we will present in Section 3.3 does not remove this limitation. We bound our model in the following ways.

- We need to bound the number of sessions per execution trace. In our example of Section 3.2.2, the init process reveals that an execution trace contains at most two sessions (one initiator and one resonder).

- We need to bound the number of agents. We bound it to three in our example: A, B and C.

- The size of messages that the adversary can send is also bounded. For example, if a role receives $\{x\}_k^e$ in a normal execution trace, then the adversary cannot send $\{\{x\}_k^e\}_k^e$, as this would require a larger channel.

- We bound the memory of the intruder. In our example, we give a memory of exactly one message. Therefore we miss attacks where the intruder needs to remember more than one message. For example, suppose the intruder receives $\{x1\}_k^e$, then $\{x2\}_k^e$, and then k. If the intruder stores $\{x2\}_k^e$ in its one message memory, it will not be able to learn $x1$ upon reception of k.

Secondly, there is a class of protocols that do enter the scope of protocols targeted by [8], but that is in fact impossible to model using the method of [8]. Those are the protocols containing at least two messages that are modeled by the same number of payloads in the method, and that contain a pair (k, m) and encrypted payload such as $\{m\}_k^{se}$, at the same place in the Promela message. Recall that $\{m\}_k^{se}$ denotes symmetric encryption of m using key k. For example, a protocol containing messages $\{m\}_k^{se}$ and (k, m) cannot be modeled by the method of [8]. Indeed, sending $\{m\}_k^{se}$ and sending (k, m) are modeled the same way in role processes: in both cases the pair `(k,m)` is sent to the network (`Comm!k,m`). The problem arises when the intruder intercepts the message and tries to learn from it. If the message models $\{m\}_k^{se}$, then the intruder should not be able to learn m if it does not know the key k. However, if the message models (k, m), then the intruder should

always be able to learn m. Unfortunately, the intruder has no way to know what the message `(k, m)` represents. If the message containing $\{m\}_k^{se}$ and the message containing (k, m) had different number of payloads, the intruder could check the number of payloads of the message it received, but we assumed that it was not the case. To overcome this difficulty, one could add a payload indicating the type of message. In the protocol above, there would be two message types: type 1 (`t1`) for $\{m\}_k^{se}$, and type 2 (`t2`) for (k, m). The role processes would thus send messages `(t1, k, m)` and `(t2, k, m)`. The intruder then only needs to look at the message type payload to know if `(k, m)` represents $\{m\}_k^{se}$ or (k, m). Since IKEv2 does belong to the scope of the method of [8] (because there are no two messages of the same size that have a pair and an encrypted payload at the same place), there was no need to apply the above correction.

Thirdly, the existing method provides ways to model some cryptographic primitives, but lacks ways to model other one. In particular, IKEv2 requires models for digital signature with appendix, affixed MAC, and modular exponentiation (for the Diffie-Hellman protocol). [45] describes two different classes of digital signature schemes. A digital signature with message recovery is a digital signature whose verification yields the signed message. A digital signature with appendix is a digital signature from which no one can recover the signed message (this property is ensured by the use of a cryptographic hash function). In his method, [8] models digital signature with message recovery. In IKEv2, RSA [59] and Digital Signature Standard (DSS) [82] authentication methods are both digital signatures schemes with appendix: it should not be possible for anyone to retrieve the signed content from the signature. To sum up, facing encryption, the adversary can retrieve the content if and only if it possesses the decryption key. Facing digital signature with appendix, the adversary can never retrieve the content. Facing affixed MAC, the adversary can always retrieve the content. We propose models for affixed MAC and digital signature with appendix in Section 3.3.1, and a model for modular exponentiation in Section 3.3.1.

Fourthly, writing the `Isvalidmessage` function, which is used to check the consistency of messages generated by the `Randmessage` function (see Section 3.2.2), is extremely cumbersome and error-prone. We propose a much simpler model of intruder message creation in Section 3.3.2.

Finally, as explained in Section 3.2.2, the secrecy implementation proposed by [8] is only adapted to the secrecy of constants. However, our definition of secrecy (see Section 2.3.2) states that the value of the local variable k computed by each party should remain

secret. Thus we need a way to model the secrecy of variables, i.e. of whatever value a variable has. In addition, [8] only implements one form of authentication, which is rather weak and does not follow the hierarchy of [51]. We extend [8] to consider our security properties from Section 3.3.3.

3.3 Improving Spin modeling to analyze security protocols

In this Section, we extend the method of [8] in order to fit the adversary model described in Section 2.3.1 and the properties described in Section 2.3.2. We classify our extensions into three categories: those that extend protocol modeling, those that extend adversary modeling, and those that extend properties modeling. We then apply our extended method the IKEv2 protocol. Finally, we summarize the limitations that, despite our modeling method, remain in the use of Spin for analyzing security protocols.

3.3.1 Improving protocol modeling

Let us go over our two extensions of protocol modeling: (1) we provide a model for the Diffie-Hellman protocol, and (2) we provide models for affixed MAC and digital signature with appendix.

Modeling the Diffie-Hellman protocol

Firstly, we propose some way to model the Diffie-Hellman protocol. The Diffie-Hellman protocol can be written as such:

$$i \to r : g^{x_i}$$
$$r \to i : g^{x_r}$$

Where i and r are agents, g is a constant and x_i and x_r are nonces. At the end of the protocol, i and r both compute the value $g^{x_i * x_r} = (g^{x_i})^{x_r} = (g^{x_r})^{x_i}$.

To model the Diffie-Hellman protocol in Promela, we add name `G` representing g. We add names `XA`, `XB` and `XC` representing values x_a, x_b and x_c. Each agent executes the

protocol at most once in an execution trace of our model, so each agent will need at most one value of type `X` to send.

We add names `KEA`, `KEB` and `KEC` representing values g^{x_a}, g^{x_b} and g^{x_c}. We add names `KAA`, `KAB`, `KAC`, `KBB`, `KBC` and `KCC` representing values $g^{x_a * x_a}$, $g^{x_a * x_b}$, $g^{x_a * x_c}$, $g^{x_b * x_b}$, $g^{x_b * x_c}$ and $g^{x_c * x_c}$.

We create a function called `Dhexp`, which represents modular exponentiation by mapping each couple of names (ke, x) of type (`KETYPE`, `XTYPE`) to a name k of type `KTYPE`. This is done in the following way. Note that this allows to model $(g^{x_i})^{x_r} = (g^{x_r})^{x_i}$ but not e.g. $((g^{x_a})^{x_b})^{x_c} = ((g^{x_a})^{x_c})^{x_b}$.

```
inline Dhexp(x, ke, k)
{
        if
        :: x == XA && ke == KEA -> k = KAA
        :: x == XA && ke == KEB -> k = KAB
        :: x == XB && ke == KEA -> k = KAB
        :: x == XA && ke == KEC -> k = KAC
        :: x == XC && ke == KEA -> k = KAC
        :: x == XB && ke == KEB -> k = KBB
        :: x == XB && ke == KEC -> k = KBC
        :: x == XC && ke == KEB -> k = KBC
        fi
}
```

Finally, modeling the Diffie-Hellman protocol requires adding a deduction step in function `Addtoknowledge` (we introduce this function in Section 3.2.2). Indeed, when the intruder learns a new name, we now need to check whether it can deduce some key by modular exponentiation, i.e. we need to allow the adversary to retrieve $g^{x_a * x_b}$ from g^{x_a} and x_b. The deduction step is made of one deduction attempt for each key of type k. The following code, taken from our improved `Addtoknowledge` tries to deduce key `KAA`. In this code, if the intruder knows `XA` and `KEA`, then it knows the key `KAA` (`Keys[KAA] = 1`), and it knows the inverse key of the inverse key of `KAA`, i.e. it knows the inverse key of `KAA` (because `KAA` is symmetric), i.e. `Knows[KAA] = 1`. Otherwise, the intruder moves on and tries to deduce another key of type k.

```
if
::  Knows[XA] && Knows[KEA] ->
```

```
    Knows[KAA] = 1; Keys[KAA] = 1;
::  else -> skip
fi;
```

Modeling affixed MAC and digital signature with appendix

To analyze IKEv2, we further need to model digital signature with appendix and affixed MAC. To do so, we naturally extend the encryption modeling method of [8]. To model digital signature with appendix $\{x\}^s_k$, we use: `k, x`, like for encryption. However when receiving this message, in `Addtoknowledge`, the intruder can never learn `x`, even if it possesses k^{-1} (the inverse key of k). To model affixed MAC $\{x\}^m_k$, we also use: `k, x`. However, upon reception of this message, the intruder can always learn `x`, even if it does not possess k^{-1}.

3.3.2 Improving adversary modeling

To model intruder message creation, [8] uses a `RandMessage` function that non-deterministically generates a message using names, and an `Isvalidmessage` function that checks consistency of the generated message with intruder knowledge. Unfortunately, the `Isvalidmessage` can be very complex to write, especially for a complex protocol such as IKEv2. In this Section, we propose a simpler way to model message creation by the intruder. For comparison, we apply this improvement to our protocol example of Section 3.2.2. As a reminder, the protocol is the following one.

$$(m1)\ i \rightarrow r : \{\{k\}^{ae}_{prk(i)}\}^{ae}_{pbk(r)}$$
$$(m2)\ r \rightarrow i : \{Sec\}^{se}_{k}$$

[8] checks consistency with intruder knowledge in his `Isvalidmessage` function, and performs type checking in role processes. As a consequence, most of the time the `Randmessage` function of [8] yields messages that either do not pass the `Isvalidmessage` test or do not pass the type checking done in role processes. In order to avoid state explosion, we propose to directly check type and consistency with intruder knowledge in `Randmessage`. To do so, our `Randmessage` function directly chooses for each payload a value among names whose type fits the payload. When using this method, there is no need for some `Isvalidmessage` macro since `Randmessage` always generates a valid message. Our method

greatly limits path explosion and allows our analysis to terminate.

In fact, because our `Randmessage` function performs type checking, it becomes message-specific. We thus create one `Randmessage` function for each message of the protocol. In our example, we call them `Randm1message` and `Randm2message`. We provide the commented code of `Randm1message` below, and explain it afterwards.

```
/*
 * Randm1message builds a new message by assigning values to p1, p2
 * and p3. Randm1message takes saved values pp1, pp2 and pp3 as
 * arguments so that it can inject some protected payloads of the
 * saved message into the new message (p1, p2, p3).
 */

inline Randm1message(p1, p2, p3, pp1, pp2, pp3)
{

    /*
     * If the saved message is an m1, then the Intruder can inject
     * the outer encrypted payload of the saved message.
     */

    if
    ::  pp3 != NULL ->

        /*
         * This branch can only be taken if the saved message is
         * an m1. Now inject outer encrypted payload.
         */

        p1 = pp1;
        p2 = pp2;
        p3 = pp3;

        /*
         * All payloads have been assigned a value, so go to end
         * of function.
         */
```

```
    goto allset;

::  skip

    /*
     * This branch can always be taken: the intruder can
     * choose to build the outer encrypted payload itself.
     */

fi;

/*
 * The intruder has chosen to build the outer encrypted payload
 * itself. Now it chooses a public key for p1.
 */

if
::  Keys[PBKA] -> p1 = PBKA
::  Keys[PBKB] -> p1 = PBKB
::  Keys[PBKC] -> p1 = PBKC
fi;

/*
 * If the saved message is an m1 and the intruder knows how to
 * decrypt the outer encrypted payload of the saved message,
 * it can inject the inner encrypted payload of the saved
 * message.
 */

if
::  pp3 != NULL && Knows[pp1] ->

    /*
     * This branch can only be taken if the saved message is
     * an m1 and the intruder knows how to decrypt the inner
```

```
        * encrypted payload of the saved message. Now inject
        * inner encrypted payload.
        */

       p2 = pp2;
       p3 = pp3;

       /*
        * All payloads have been assigned a value, so go to end
        * of function.
        */

       goto allset;

::     skip

       /*
        * This branch can always be taken: the intruder can
        * choose to build the inner encrypted payload itself.
        */

fi;

/*
 * The intruder has chosen to build the inner encrypted
 * payload itself. Now it chooses a public key for p1.
 */

if
::  Keys[PRKA] -> p2 = PRKA
::  Keys[PRKB] -> p2 = PRKB
::  Keys[PRKC] -> p2 = PRKC
fi;

if
::  Keys[KA] -> p3 = KA
```

```
    ::  Keys[KB] -> p3 = KB
    ::  Keys[KC] -> p3 = KC
    fi;

allset:

    /* End of function. Message (p1, p2, p3) has been created. */

        skip;

}
```

Notice how this function takes the saved message (pp1, pp2, pp3) as argument. This allows the intruder to build a message using protected payloads of the saved message that it could not have created itself. In our example, if the saved message is an m1, then this message is of the form `pbk, prk, k`. Since there are two encryptions in this message, let us call `pbk, prk, k` the outer encrypted payload, and `prk, k` the inner encrypted payload.

If the saved message is an m1 (condition `pp3 != NULL`) then the intruder can simply replay the whole message (outer encrypted payload). In this case, we set payloads p1, p2 and p3 to pp1, pp2 and pp3, and go to the end of message creation (`goto allset`). If the saved message is not an m1 or if the intruder chooses (recall that choices are non-deterministic in Spin) not to replay the message, then the intruder can try to build its own outer encrypted message. It thus chooses one key that it knows (e.g. `Keys[PBKA]`) among keys of the correct type, and assigns it to p1.

The intruder then faces a similar choice for the inner encrypted payload. Only this time, if it wants to inject the saved inner encrypted payload, it needs to know the inverse key of pp1, because the saved inner encrypted payload is encrypted by pp1. Finally, if the intruder chooses to build the whole message itself, then it must choose a name of type `k` assign it to p3.

3.3.3 Improving properties modeling

In this Section we explain how we model our security properties in Promela. We successively show how we implement our secrecy, aliveness, weak agreement and non-injective agreement invariants. Finally, we present our progress functions, which set the variables used in property invariants.

Note that it is not relevant to check injective agreement in Spin. Indeed, injective agreement adds to non-injective agreement the condition that if A completes the protocol n times computing the same agreement terms, then B has completed the protocol at least n times computing these same agreement terms. Since our Promela model allows only one session per role in a trace, injective agreement is equivalent to non-injective agreement in our model.

Modeling secrecy

We model the secrecy of constants as was done in Section 3.2.2:

```
# define Invsecrecy ( !Knows(Sec) )
```

However, we often want to model the secrecy of variables. For example, in key-exchange protocols (which IKEv2 is), we want to model the secrecy of the key resulting from the key-exchange. To do so, we use two variables `sta` (for secrecy term of A) and `stb` of type mtype. We set these variables to the secrets computed by A and B using progress function `Secret`, which we present in Section 3.3.3.

```
mtype sta = NULL;
mtype stb = NULL;
```

Finally, we define our secrecy invariant as follows: if A has computed a secrecy term (`sta != NULL`), then the intruder does not know this term (`!Knows[sta]`), and similarly for B.

```
# define Invsecrecy ( \
        ( sta == NULL || !Knows[sta] ) \
    &&  ( stb == NULL || !Knows[stb] ) )
```

Modeling aliveness

To implement authentication properties, we use the same variables as in Section 3.2.2, plus some new ones. In Section 3.2.2 we had variables `inirunningab`, `resrunningab`, `inicommitab` and `inicommitab`. Recall that e.g. `inirunningab` is true if and only if A has been running the protocol, apparently with B. In our method we add `inirunningba`, `resrunningba`, `inicommitba` and `inicommitba`. In fact this is not really necessary with our current method, as our init function always runs A as initiator and B as responder. However, we still add these four variables because when CPU resources, memory resources,

and the complexity of the analyzed protocol allow it, it is possible to add sessions in the init process, and thus yield stronger proofs. For example, one can add the `Responder(A)` session.

To implement aliveness, we further need two new boolean variables, namely `arunning` and `brunning`. Our progress functions of Section 3.3.3 set `arunning` (resp. `brunning`) to true if and only if A (resp. B) has been running the protocol (whoever the peer was). The first line of our aliveness invariant (see code below) thus becomes: if A has completed the protocol as an initiator, apparently with B (inicommitab), then B has been running the protocol (brunning). The other lines cover the cases where the actor is B and/or where the actor is a responder.

```
# define Invaliveness ( \
    (!inicommitab || brunning) \
&&  (!rescommitab || arunning) \
&&  (!inicommitba || arunning) \
&&  (!rescommitba || brunning) )
```

Modeling weak agreement

Our weak agreement invariant resembles the authentication invariant of [8], except that we do not require that the peer endorse the correct role (correct role endorsement is checked by non-injective agreement). The first line of our weak agreement invariant (see code below) thus becomes: if A has completed the protocol as an initiator, apparently with B (`inicommitab`), then B has been running the protocol, apparently with A, be it as responder (`resrunningab`), or as initiator (`inirunningba`). The other lines cover the cases where the actor is B and/or where the actor is a responder.

```
# define Invweakagree ( \
    (!inicommitab || resrunningab || inirunningba) \
&&  (!rescommitab || inirunningab || resrunningba) \
&&  (!inicommitba || resrunningba || inirunningab) \
&&  (!rescommitba || inirunningba || resrunningab) )
```

Modeling non-injective agreement

To implement non-injective agreement, we need to store the names on which each agent agrees when "running" and when "committing". We therefore define variables `atra`, `atrb`,

`atca` and `atcb`, where e.g. `arta` means "Agreement Term of A at Running step". Our progress functions of Section 3.3.3 set these variables during protocol execution.

The first line of our non-injective agreement invariant (see code below) then becomes: if A has completed the protocol as an initiator, apparently with B (`inicommitab`), then B has been running the protocol as a responder, apparently with A (`resrunningab`), and both parties have computed the same agreement term (`atca != NULL && atca == atrb`). The other lines cover the cases where the actor is B and/or where the actor is a responder.

```
# define Invniagree ( \
    (!inicommitab || resrunningab && atca != NULL && atca == atrb) \
&&  (!rescommitab || inirunningab && atcb != NULL && atcb == atra) \
&&  (!inicommitba || resrunningba && atcb != NULL && atcb == atra) \
&&  (!rescommitba || inirunningba && atca != NULL && atca == atrb) )
```

Progress functions

We have introduced new state variables. We now need to write our progress functions to make them set these variables during protocol execution.

We use a new progress function called `Secret` (see code below), which is called at the end of role processes (at the same time as commit progress functions), and sets the values of sta and stb to the secrets computed by A and B.

```
inline Secret(agent, t)
{

    if
    ::  agent == A -> sta = t
    ::  agent == B -> stb = t
    ::  else -> skip
    fi;

}
```

We now need to rewrite functions `Inirunning`, `Resrunning`, `Inicommit` and `Rescommit`, so that, in addition to setting variables `{ini,res}{running,commit}ab`, they also set variables `{a,b}running` and variables `at{r,c}{a,b}`. Below implementation of `Inirunning` does this intuitively.

```
inline Inirunning(i, r, t)
{

    if
    ::  i == A -> arunning = 1; atra = t
    ::  i == B -> brunning = 1; atrb = t
    ::  else -> skip
    fi;

    if
    ::  i == A && r == B -> inirunningab = 1
    ::  i == B && r == A -> inirunningba = 1
    ::  else -> skip
    fi

}
```

3.3.4 Application to IKEv2

In this Section, we use our improved modeling method to analyze IKEv2 using Spin. We start by applying the method to model the protocol and the adversary, showing only protocol-specific modeling choices. Our property model described in Section 3.3.3 is entirely not protocol-specific, so we keep it as is. We then reveal our analysis results and draw conclusions on the security of IKEv2. Finally, we discuss limitations that, despite our modeling effort, remain when using Spin to verify security protocols.

Modeling the protocol

Let us apply our improved method to the modeling of our three IKEv2 subprotocols defined in Section 2.2. There are three protocol-specific aspects in protocol modeling: name declaration, channel declaration and role definition. We take IKEv2-Sig as example.

We start by creating the names, as we did in Section 3.2.2. We create names for our three agents and one name for the g constant. The three header flags may each take two different values (0 and 1), so we create six names for them. Each agent will need at most one payload of type x and one payload of type n, and one private key, so we create three names of each type. We create names for the result of modular exponentation applied

to the three (G, x) couples, which makes three payloads of type ke. We create names for the result of modular exponentation applied to the six (ke, x) payloads, which makes six payloads of type k.

```
/* Agents: Alice, Bob, Charlie */
mtype = { A, B, C };

/* Constant g */
mtype = { G };

/* Possible values for the Response flag */
mtype = { FR0, FR1 }

/* Possible values for the Initiator flag */
mtype = { FI0, FI1 }

/* Possible values for the message id */
mtype = { MID0, MID1, MID2 }

/* Private Keys */
mtype = { PRKA, PRKB, PRKC };

/* DH exponents */
mtype = { XA, XB, XC };

/* Nonce payloads */
mtype = { NA, NB, NC };

/* Key-exchange payloads */
mtype = { KEA, KEB, KEC };

/* Master keys */
mtype = { KAA, KAB, KAC, KBB, KBC, KCC };

/* Number of possible names that the adversary may learn */
mtype = { NULL };
```

We now count the maximum message size and create a channel that accepts messages

of that size. The biggest messages of IKEv2-Sig are the auth request and response. The auth request of IKEv2-Sig is:

$$\{rf_{i2}, if_{i2}, mid_{i2}, \{i, \{ke_i, n_i, n_r, k, i\}^s_{prk(i)}\}^e_k\}^m_k$$

We thus need one payload for MAC protection using key k, three payloads for the three header flags, one payload for encryption using key k, one payload for the identity (i or r), one payload for signature using a private key, and five payloads for the content of the signature. In the code below, we thus declare a channel that accepts messages of size 12. For IKEv2-PSK, the size is 12 as well. For IKEv2-Child, the size is 5.

```
chan Comm = [0] of {mtype, mtype, mtype, mtype, mtype, mtype, mtype,
   mtype, mtype, mtype, mtype, mtype};
```

Finally, we define the roles. For example, the code below, located in the Initiator process of IKEv2-Sig, models the initiator sending the auth request. The initiator has identity i and communicates with agent r. Let us assume that $i = A$ and $r = B$. The initiator has already set the value of `xi`, `ni` and `kei` according to its identity. Thus `xi` holds `XA`, `ni` holds `NA` and `kei` holds `KEA`. The initiator received `ker` and `nr` in the init response. Now the initiator uses function `Dhexp` to set key `k` to the name representing the modular exponentiation of `key` by `xi`. In our example, `k` now holds `KAB`. Then the initiator sets the value of his private key `prki` ro `PRKA`. Now the initiator is right before the sending of its last message. According to our definition of the running step in Section 2.3.2, it thus calls `Inirunning`, which sets `arunning` to 1, `inirunningab` to 1 and `atra` to `KAB`. Finally, the initiator sends the auth request, correctly setting the header flags to $(rf_{i2}, if_{i2}, mid_{i2}) = (0, 1, 1)$.

```
atomic {
    Dhexp(x, ker, k);
    Setvar(i, PRKTYPE, prki);
    Inirunning(i, r, k);

    /* Auth request */
    Comm!k, FR0, FI1, MID1, k, i, prki, k, i, kei, ni, nr;
};
```

Modeling the adversary

Let us apply our adversary modeling method described in Section 3.3.2 to IKEv2. There are two protocol-specific aspects in adversary modeling: the `Randmessage` and `Addtoknowledge` functions. Our `Randmessage` functions follow quite intuitively Section 3.3.2, so we will not detail it in this Section. However, we explain our implementation of `Addtoknowledge`, as this function must now correctly process the cryptographic primitives of IKEv2.

Let us take IKEv2-Sig as example again. init and auth messages look like below.

```
init: RF0 , IF1  , MID0 , kei  , ni , NULL , NULL , NULL , NULL
          , NULL , NULL , NULL

auth: k   , FR0  , FI1  , MID1 , k  , i    , prki , k    , i
          , kei  , ni   , nr
```

The code below shows some part of our `Addtoknowledge` function in IKEv2-Sig.

```
inline Addtoknowledge(p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11,
   p12)
{

    /*
     * From an init message, we learn p5 and p6. Nothing to learn
     * from the decryption of an auth message.
     */

        if
        ::  p12 == NULL -> Knows[p5] = 1; Knows[p6] = 1;
        ::  else -> skip
        fi;

    /*
     * With its new knowledge, the adversary may obtain some keys
     * through DH exponentiation.
     */

    if
    ::  Knows[XA] && Knows[KEA] ->
        Knows[KAA] = 1; Keys[KAA] = 1;
```

```
    ::  else -> skip
    fi;

    ...

}
```

In this function, if the received message (p1, ..., p12) is an init message (`p12 == NULL`), then the intruder learns every payload of the message, because an init message is not protected by encryption or digital signature with appendix. However, the first three payloads are public constants, so the intruder already knows them. The intruder thus only learns paylaods p5 and p6. If the message is an auth message, all payloads that are not public constants are protected by a digital signature with appendix. There is thus no way for the intruder to learn these payloads. Therefore the intruder learns nothing from an auth message using `Addtoknowledge`. Note however that as explained in Section 3.2.2, in the next step of the adversary process, the intruder may save the whole message itself in variables (pp1, ..., pp3). it will then be able to inject parts of the saved message in `Randmessage` functions.

Finally, as prescribed by Section 3.3.1, we add to the `Addtoknowledge` function the attempts of the intruder to deduce a new key of type k by modular exponentation.

Note as well that we implement AdvEXT in addition to AdvINT. AdvEXT is obtained by simply removing the lines of the init process that give the long-term keys of C to the intruder knowledge.

Results on the security of IKEv2

We now apply Spin to our model of IKEv2. In this Section we present and discuss our analysis results.

Our Promela code and the exact Spin commands we used are available at `https://gitlab.inria.fr/tninet/spin`. We present our analysis results in Table 3.4. Allowing only two sessions in a trace, very few amount of time and memory was necessary to perform the verifications. For example, on a 2.6 GHz CPU, only 3s and 128 MB of memory were necessary to prove aliveness on IKEv2-Sig in AdvINT. Note that we used Depth-First Search (DFS) mode and bitstate hashing optimisation.

Our analysis yields two notable results. Firstly, it proves that IKEv2-PSK and IKE-Child satisfy non-injective agreement in our adversary model. Secondly, it refutes the

Property	Adv. model	IKEv2-Sig	IKEv2-PSK	IKEv2-Child
Secrecy	AdvEXT	✓	✓	✓
	AdvINT	✓	✓	✓
Aliveness	AdvEXT	✓	✓	✓
	AdvINT	✓	✓	✓
Weak agreement	AdvEXT	✗	✓	✓
	AdvINT	✗	✓	✓
Non-injective agreement	AdvEXT	✗	✓	✓
	AdvINT	✗	✓	✓

Figure 3.4: Analysis of IKEv2 using Spin. We write ✓ when a subprotocol guarantees some property to some role in some adversary model, and ✗ when it does not. A subprotocol satisfies a property if and only if the property is satisfied for both the initiator and the responder. Our analysis refutes the reflection attack that was found by previous analyses, and proves that IKEv2-PSK and IKE-Child satisfy non-injective agreement in our adversary model.

reflection attack that was found by previous analyses.

[24] claims that IKEv2-Child is vulnerable to a reflection attack against the initiator. We described this attack in Section 3.1.2. In this attack, the intruder replays an agent's child request to itself. The agent commits as a responder and replies with a child response, which the intruder replays again to the agent. The agent then commits as an initiator. Since the agent believes it was talking to some other agent, when it was actually talking to itself, the attack violates aliveness both for the initiator and the responder. However, [24] does not include the Initiator flag of the IKEv2 header in his model. The role of this flag is explained in Section 2.2.1. By adding the Initiator flag, our analysis shows that IKEv2-Child satisfies aliveness, weak agreement and non-injective agreement. Indeed, because of this flag, during a reflection attack, the initiator will notice that the request it receives comes from the original initiator (which is itself). it will thus refuse to answer it. Furthermore, the flag is integrity-protected: the RFC of IKEv2 says (and we found through experiment that it was the case in the strongSwan implementation) that the "Integrity Checksum Data" field of the encrypted payload is "the cryptographic checksum of the entire message starting with the Fixed IKE header through the Pad Length". The

intruder thus cannot successfully change the Initiator flag without knowing key `k`. Since secrecy of `k` is satisfied, the reflection attack is not possible.

[24] already pointed out the obvious defense against the reflection attack: "breaking the symmetry of the messages, e.g., by including distinct constants and checking their presence". We have shown that this defense is already in place in the protocol. Furthermore, the defense involves more than simply including distinct constants: these constants need to be integrity protected.

Our analysis also confirms that IKEv2-Sig does not satisfy weak agreement. This vulnerability is called the penultimate authentication flaw and was already found in previous analyses. We presented this vulnerability in Section 3.1.2. The penultimate authentication flaw is not a full violation of the intuitive definition of authentication, because there is no actual impersonation and secrecy is still satisfied. However, we show in Chapter 4 that the penultimate authentication flaw makes it possible to perform a Denial-of-Service attack against IKEv2.

We have removed some of the limitations of the method of [8]. However, limitations due to the boundedness of Spin remain. Recall that we bound the number of sessions, the number of agents, the memory of the adversary and the size of messages. More precisely, our Promela model only allows up to two sessions in an execution trace, and a maximum number of three agents running these sessions. Furthermore, the intruder knowledge is bounded to one message and the complexity of messages that the intruder can send over the network is limited. In fact, because of these restrictions, we will show in Section 3.4.3 that we have missed attacks that require more than two sessions. Hence in the next Section we use another tool for the analysis of IKEv2.

3.4 Model checking IKEv2 using ProVerif

As we pointed out in Section 3.3.4, Spin can only verify security protocols in a model where the number of sessions, the adversary knowledge and the size of messages are all bounded. In this Section, we analyze IKEv2 using the ProVerif model checker, which does not suffer from these limitations. We firstly present the main features of ProVerif, then explain how we wrote our ProVerif model of IKEv2, and finally present our results regarding the security of IKEv2.

3.4.1 ProVerif in a nutshell

Let us go over the main elements of the applied pi calculus and see how ProVerif interprets them.

In ProVerif, the user represents the protocol to verify using the following syntax.

```
P, Q  :=  0
        | P | Q
        | !P
        | in(c, x:T); P
        | out(c, M); P
        | new a: T; P
        | let x: T = D in P else Q
        | if M then P else Q
```

The process 0 represents the null process, which does nothing. The process `P | Q` denotes the processes P and Q running in parallel. The process `!P`, called replication of P, represents an infinite number of copies of P running in parallel. The process `in(c, x); P` receives some term over channel c and then runs the process P, but with x replaced by the received term. The process `out(c, M); P` sends the term u over channel c and runs the process P. The process `new a: T; P`, called restriction, creates a new name a of type T and then runs P. Restrictions allow modeling fresh values. The process `let x: T = D in P else Q` assigns to x, variable of type T, the result of some expression evaluation. If evaluation of D succeeds, then the process runs P with x bound to the result of the evaluation of D. Otherwise, the process runs Q. The failure of an expression evaluation represents for example an attempt to decrypt a message with some key that is different from the key with which it was encrypted. The process `if M then P else Q` models a conditional choice. If expression M evaluates to true, then the process runs P, otherwise the process runs Q.

To model cryptographic primitives, ProVerif offers the user to define some constructors. Constructors are declared using keyword `fun`. They may be used by processes to build new terms. For example, to model the protocol example from Section 3.2.2, we define the constructors prk, pbk and aenc, which resp. model the private key of an agent, the public key of an agent, and asymmetric encryption. The keyword `[private]` forbids the intruder from using the prk function.

```
fun prk(bitstring): bitstring [private].
fun pbk(bitstring): bitstring.
```

```
fun aenc(bitstring, bitstring): bitstring.
```

ProVerif then offers two ways to assign meaning to constructors, namely destructors and equations. Destructors and their relation to constructors are defined using keyword `reduc`. Destructors may be used in expressions, which processes evaluate in `let` constructs. For example, asymmetric encryption schemes can be modeled by the following code.

```
reduc forall x: bitstring, y: bitstring; adec(aenc(x, pbk(y)),
   prk(y)) = x.
```

Some cryptographic primitives cannot be modeled by destructors. This is the case of modular exponentiation. For these primitives, one should use equations. Equations define axioms on constructors. They are defined using keyword `equation`. For example, modular exponentiation can be modeled in the following way. We declare some constant G, which represents the Diffie-Hellman generator, some constructor exp, which represents modular exponentation, and some equation on exp, which models some axiom of modular exponentiation. Note that many algebraic properties of modular exponentiation are not modeled by this axiom. Therefore this model of modular exponentiation misses some attacks. However, we cannot do better with ProVerif, as ProVerif only supports some specific types of equations, which either have the finite variant property or satisfy some linearity constraints [20].

```
const G: bitstring.
fun exp(bitstring, bitstring): bitstring.
equation forall x: bitstring, y: bitstring;
    exp(exp(G, x), y) = exp(exp(G, y), x).
```

For example, let us model the protocol example from Section 3.2.2. This protocol was:

$$(m1)\ i \to r : \{\{k\}^{s}_{prk(i)}\}^{ae}_{pbk(r)}$$
$$(m2)\ r \to i : \{Sec\}^{se}_{k}$$

In the above protocol, the signature is a digital signature with message recovery (see Section 3.3.1), i.e. verifying the signature yields the signed message. Quite intuitively, we use the following code to model signature with message recovery.

```
fun signrecov(bitstring, bitstring): bitstring.
reduc forall x: bitstring, y: bitstring; getmess(signrecov(x,
   prk(y)), pbk(y)) = x.
```

The above protocol can be modeled in ProVerif by the following process.

```
new k: bitstring;
out(c, aenc(signrecov(k, prk(A)), pbk(B)));
in(c, m2: bitstring);
0 |
in(c, m1: bitstring);
let s: bitstring = adec(m1, prk(B)) in
let k: bitstring = getmess(s, pbk(A)) in
out(c, senc(Sec, k));
0
```

The user further queries some properties, using the `query` construct. We will use two types of properties: reachability properties and correspondance assertions. Secrecy is a reachability property. To check secrecy of some term M, we add the following query in our code. In Proverif, a process preserves the secrecy of term M if for all the adversary processes, there is no trace in which M is sent over a public channel.

```
query attacker(M).
```

Authentication, however, requires using correspondence assertions. To use correspondence assertions, the user first places some `events` in the code of its processes. To do so, it employs the additional construct `event(M); P`. Semantically, this construct is simply rewritten to `P`. However, the user may use events in its correspondence assertion queries. We show some example of correspondence assertion below. In this query, we check that for all a, whenever event `commit(a)` is executed, then event `running(a)` has been executed before. This form of query is perfectly adapted to verify non-injective authentication properties.

```
query a: bitstring; event(commit(a)) ==> event(running(a)).
```

To verify injective authentication properties, ProVerif provides the keyword `inj-event`, such as in the example below. In this example, we check that for all a, whenever event `commit(a)` is executed, let n be the number of times `commit(a)` has been executed in the execution trace, then event `running(a)` has been executed at least n times. The semantics of inj-event follows the definition of injectivity given by [51].

```
query a: bitstring; inj-event(commit(a)) ==> inj-event(running(a)).
```

To perform the verification, ProVerif translates the user-written process into a set of Horn clauses, to which ProVerif adds Horn clauses modeling the adversary. The adversary

can be configured to be an AdvEXT. ProVerif then tries to prove the properties on the set of Horn clauses. The adversary Horn clauses and the rules that translate a ProVerif process into Horn clauses have been carefully chosen, so that if a property is true on the set of Horn clauses, then the property is true on the ProVerif model. However if ProVerif finds an attack on the set of Horn clauses, then the attack may not be possible on the ProVerif model. The reason lies in the translation rules, which perform some abstractions over the semantics of ProVerif. In this case, ProVerif tries to reproduce the attack on the ProVerif model. If the attack is not possible, ProVerif says that the property "cannot be proven". Otherwise, we have found an attack.

3.4.2 Modeling IKEv2 in ProVerif

Let us now model the IKEv2 protocol in ProVerif. In this Section we will successively model primitives, processes and properties.

Primitives

We need to model the following cryptographic primitives: symmetric encryption, digital signature (with appendix), MAC, affixed MAC, and modular exponentiation.

For symmetric encryption, we need a constructor `enc` and a destructor `dec`. They are defined as follows.

```
fun enc(bitstring, bitstring): bitstring.
reduc forall x: bitstring, y: bitstring; dec(enc(x, y), y) = x.
```

We need three constructors to model digital signature: `prk`, `pbk` and `sign`. For each agent a, `prk(a)` represents the private key of a, and `pbk(a)` represents the public key of a. For each pair of terms (m, k), `sign(m, k)` represents a signature computed over `m` by using key `k`. We then need the destructor `checksign` to check signatures.

```
fun prk(bitstring): bitstring [private].
fun pbk(bitstring): bitstring.
fun sign(bitstring, bitstring): bitstring.
reduc forall m: bitstring, a: bitstring; checksign(sign(m, prk(a)),
   pbk(a), m) = true.
```

MAC is similar to digital signature, except that the same key is used for computing and checking the MAC.

```
fun mac(bitstring, bitstring): bitstring.
reduc forall m: bitstring, k: bitstring; checkmac(mac(m, k), k, m) =
   true.
```

Affixed MAC computes a MAC of some message using some key and appends the MAC to the message. In the following model, both `getmess` and `checkrmac` retrieve the protected message, but `checkrmac` also verifies the MAC. Typically, `checkrmac` is used in role processes, and `getmess` is used by the attacker.

```
fun rmac(bitstring, bitstring): bitstring.
reduc forall m: bitstring, k: bitstring; getmess(rmac(m, k)) = m.
reduc forall m: bitstring, k: bitstring; checkrmac(rmac(m, k), k) =
   m.
```

Modeling modular exponentiation in ProVerif requires using an equation. We use the model that we explained in Section 3.4.1.

```
const G: bitstring.
fun exp(bitstring, bitstring): bitstring.
equation forall x: bitstring, y: bitstring;
    exp(exp(G, x), y) = exp(exp(G, y), x).
```

Finally, we need pre-shared keys in our models: once for authentication in IKEv2-PSK, and once for symmetric encryption and MAC in IKEv2-Child (`k` acts as a pre-shared key in this subprotocol). However, we model pre-shared keys slightly differently in both subprotocols. In IKEv2-PSK, each pair of agents shares one pre-shared key. We thus define a constructor `psk`, and use an equation to model the fact that `psk(a, b)` and `psk(b, a)` both denote the PSK between a and b.

```
fun psk(bitstring, bitstring): bitstring [private].
equation forall x: bitstring, y: bitstring; psk(x, y) = psk(y, x).
```

In IKEv2-Child, each pair of agents (a, b) shares two pre-shared keys at the beginning of the subprotocol: one modeling the key resulting from an IKEv2-Sig or IKEv2-PSK session where a was the initiator, and one modeling the key resulting from an IKEv2-Sig or IKEv2-PSK session where b was the initiator. Instead of adding a third argument to psk, denoting who is is the original initiator, we remove the equation. Thus for IKEv2-Child, `psk(a, b)` denotes the PSK between a and b where a is the original initiator, and `psk(b, a)` denotes the PSK between a and b where b is the original initiator.

Processes

Now we need to write our processes, and thereby model the two roles of IKEv2.

For each subprotocol, we define one initiator and one responder process. Every process takes as argument the agent executing the process. However, depending on the subprotocol, processes may take additional arguments. In IKEv2-Sig and IKEv2-PSK, the initiator process also takes as argument the agent to whom it wants to talk. In IKEv2-Child, both our processes take this additional argument. We adopt the convention that Initiator(i, r) is executed by i, Responder(r) is executed by r, and Responder(i, r) is executed by r.

Furthermore in IKEv2-Child, depending on whether it is the original initiator or not, an agent must set the Initiator flag accordingly. Therefore, both processes in IKEv2-Child also take as argument the boolean `origini`, which is true only if the initiator of the session is the original initiator. Therefore in the responder process, `origini` is true only if the responder is the original responder. This is just a convention: we could have defined `origini` the other way around.

Similarly to Section 3.3.4, let us explain some portion of the initiator process in IKEv2-Sig. In the following code, the initiator process is run by i and the apparent peer is r. The initiator has already generated `xi` and `ni`, computed `kei`, and has just received `ker` and `nr` from the init response. Now the initiator computes `k` using modular exponentiation, computes the signature `authi`, and creates the unMACed auth request. The initiator sets the Response flag to false (which represents 0), and the Initiator flag to true (which represents 1). The initiator then triggers its running event, and finally outputs the (MACed) auth request on the channel.

```
let Initiator(i: bitstring, r: bitstring) =

    ...

    let k = exp(ker, xi) in

    (* Auth request *)
    let authi = sign((k, i, kei, ni, nr), prk(i)) in
    let unmacedauthrequest = (false, true, enc((i, authi), k)) in

    event Inirunning(i, r, k);
```

```
out(Chan, rmac(unmacedauthrequest, k));

...

.
```

In order to have a strong model, in each subprotocol, we must instantiate all possible sessions. We make both A and B execute both roles. We do not need to make C execute any role because C is compromised, thus the adversary can already perform all actions C would perform in a role. Then for processes that take as argument the agent to whom they want to talk, we must also cover every possibility, so we make them possibly talk to both other agents. However, we do not make agents initiate sessions with themselves. In fact, we also make the responder refuse any session completion with a peer agent that is itself. In other words, we totally exclude self-communication. We do so because we assume that a reasonable IKEv2 implementation would not accept or initiate self-communication. Finally, we instantiate each of these sessions an arbitrary number of times, using the `!P` replication construct. In IKEv2-Sig and IKEv2-PSK, we thus instantiate role processes in the following way (only for non-injective properties).

```
!Initiator(A, B) | !Initiator(A, C) |
!Initiator(B, A) | !Initiator(B, C) |
!Responder(A)    | !Responder(B)
```

For IKEv2-Child, the agent executing some process may be the original initiator or not. This doubles the possibilities. As a result, for IKEv2-Child, we instantiate role processes in the following way (only for non-injective properties).

```
!Initiator(A, B, true) | !Initiator(A, B, false) |
!Initiator(A, C, true) | !Initiator(A, C, false) |
!Initiator(B, A, true) | !Initiator(B, A, false) |
!Initiator(B, C, true) | !Initiator(B, C, false) |
!Responder(A, B, true) | !Responder(A, B, false) |
!Responder(C, B, true) | !Responder(C, B, false) |
!Responder(B, A, true) | !Responder(B, A, false) |
!Responder(C, A, true) | !Responder(C, A, false)
```

We do not include the message ID field in our ProVerif models. For IKEv2-Sig and IKEv2-PSK, we make the reasonable assumption that this field has no impact on whether these subprotocols satisfy our properties or not. Indeed, in these subprotocols, the message

ID field only acts as a constant, breaking the symmetry of the protocol. However, this is already taken care of by the Response flag.

The message ID field only becomes relevant in IKEv2-Child. In this subprotocol, this field acts as a counter that may be incremented an arbitrary number of times. Each role uses a different message ID in each new IKEv2-Child session. Including the message ID in our model of IKEv2-Child would thus require that ProVerif increments this field each time a new session of the subprotocol is run. To do so, ProVerif would need to keep track of the counter's value between sessions, which is hard in ProVerif. We thus do not include the message ID field in IKEv2-Child, and expect to find some replay attacks that are not possible in practice. In fact we do find these attacks (see Section 3.4.3).

ProVerif natively offers an AdvEXT attacker through option "set attacker = active". However, the LKRothers capability needs to be manually modeled. To do so, we reveal the long-term keys of agent C at the beginning of the main process. Now we must ensure that the peer is not compromised. Our solution is to allow agents to commit only if their apparent peer is not C.

Security properties

We now need to model our properties from Section 2.3.2.

We check properties for agent A. Since A and B have symmetric roles, proving the property for A proves the property for any honest agent.

To verify secrecy in ProVerif, we define two constants SecretIni and SecretRes. At the end of the Initiator (resp. Responder) process, if the actor is A and the apparent peer is not C, we output enc(SecretIni, keymati) (resp. enc(SecrectRes, keymatr)). Obviously, if secrecy of SecretIni (resp. SecretRes) is satisfied, then secrecy of keymati (resp. keymatr) is satisfied. We thus define secrecy as the secrecy of SecretIni and SecretRes.

```
query attacker(SecretIni); attacker(SecretRes).
```

To verify authentication properties in ProVerif, we define the events Inirunning, Inicommit, Resrunning and Rescommit. We place these events at specific points in the initiator and responder processes, just like we did in Spin: the Inirunning (resp. Resrunning) event is placed in the initiator (resp. responder) process, just before the initiator (resp. responder) sends its last message in the subprotocol. The Inicommit (resp. Rescommit) event is placed at the end of the initiator (resp. responder) process.

For example, we implement aliveness for the initiator in the following way.

```
query b: bitstring, c: bitstring;
event(Inicommit(A, b))
==> event(Inirunning(b, c)) || event(Resrunning(c, b)).
```

ProVerif natively offers a way to write injective properties through keyword "inj-event". According to Section 3.2 of [10], ProVerif strictly follows Lowe's definition of injectivity, which we gave in Section 2.3.2. We thus model injective agreement on keymat for the initiator in the following way.

```
query b: bitstring, k: bitstring;
inj-event(Inicommit(A, b, k)) ==> inj-event(Resrunning(A, b, k)).
```

At first, we observed that ProVerif yielded an error for injective agreement. The error was of the form:

> Error: the event Resrunning is present several times in the same branch of a test. Injective agreement cannot be proved.

To bypass this error, the trick is to make a file dedicated to the verification of injective agreement for the initiator (resp. for the responder). In this file, we create a copy of the initiator process, named Initiatorwithoutinicommit, where we do not trigger Inicommit. We also create a copy of the responder process, named Responderwithoutresrunning, where we do not trigger Resrunning. Because we focus on proving injective agreement for agent A, we do not need to trigger Inicommit when the agent executing the initiator process is not A. Similarly, we do not need to trigger Resrunning when the agent executing the process is not B (because B is the only acceptable peer for A). Therefore, when verifying injective agreement for the initiator, we instantiate role processes in the main process like below. The method is similar for verifying injective agreement for the responder.

```
!Initiator(A, B) |
!Initiatorwithoutinicommit(A, C) |
!Initiatorwithoutinicommit(B, A) |
!Initiatorwithoutinicommit(B, C) |
!Responderwithoutresrunning(A) |
!Responder(B)
```

3.4.3 Analysis results

Let us examine the outcome of our analysis of IKEv2 using ProVerif.

Property	Role	IKEv2-Sig	IKEv2-PSK	IKEv2-Child
Secrecy	Initiator	✓	✓	✓
	Responder	✓	✓	✓
Aliveness	Initiator	✓	✓	✓
	Responder	✓	✓	✓
Weak agreement	Initiator	✓	✓	✓
	Responder	✗	✓	✓
Non-injective agreement	Initiator	✓	✓	✗
	Responder	✗	✓	✓
Injective agreement	Initiator	✓	✓	✗
	Responder	✗	✓	✗

Figure 3.5: Analysis of IKEv2 using ProVerif. We write ✓ when a subprotocol guarantees some property to some role, and ✗ when it does not. Our analysis confirms the penultimate authentication flaw and finds two false attacks against IKEv2-Child.

Our models are available at `https://gitlab.inria.fr/tninet/proverif`. The analysis took only a few seconds and little memory on our 2.6 GHz CPU. Figure 3.5 shows our analysis results. Firstly, our analysis confirms the penultimate authentication flaw. We presented this attack in Section 3.1.2. Secondly, we find two attacks against IKEv2-Child. However, we note that these attacks are not possible in practice, because the message ID field prevents them. We find these attacks because we could not include the message ID field in our models, due to the limitations of ProVerif. These attacks are interesting anyway, as they point out from which attacks the message ID field protects IKEv2.

Figure 3.6 shows the attack that violates non-injective agreement (and therefore injective agreement as well) for the initiator in IKEv2-Child. In this attack, `i` sends a first request, which the intruder intercepts and drops. `i` then sends a second request. The responder `r` receives it, commits and replies. `i` receives the response and commits some key that it computes using the exponent of its first request. Since `r` committed a key computed using the key-exchange payload of `i`'s second request, `i` and `r` have committed different values. This violates non-injective agreement, as this property states that both parties

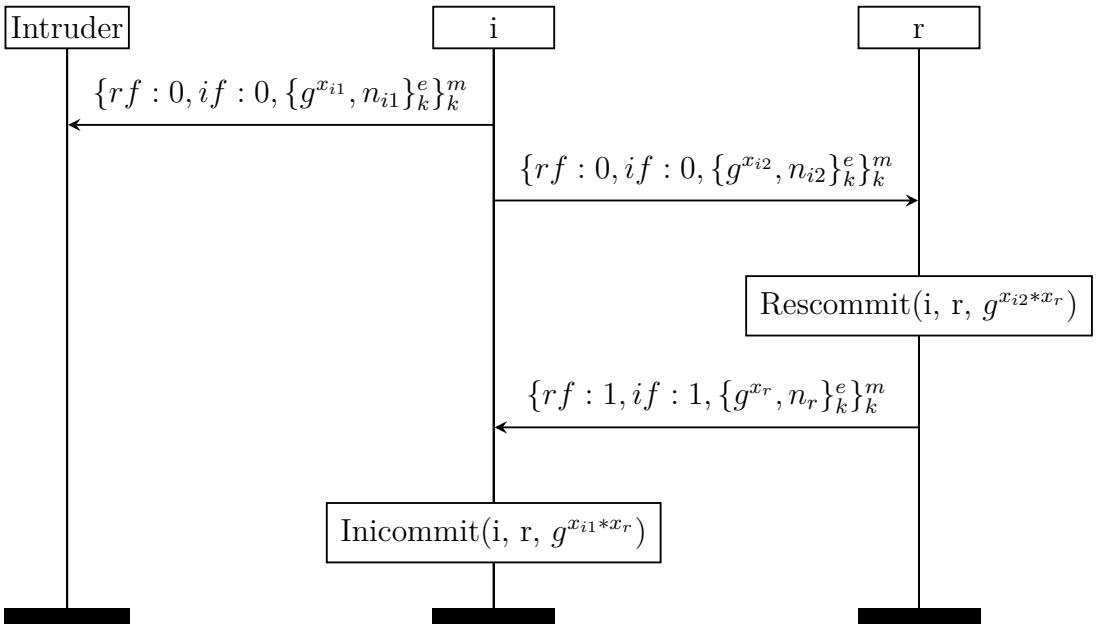


Figure 3.6: Violation of non-injective agreement for the initiator in IKEv2-Child

commit the same values. In this attack, the initiator associated some response with the wrong request. This is not possible in practice. Indeed, the message ID of each request is set to the message of the previous request, plus one, and the responder replies to a request by using the same message ID in its response. Therefore in practice, the initiator cannot associate a response with the wrong request, and this attack is not possible.

Figure 3.7 shows the attack that violates injective agreement for the responder in IKEv2-Child. In this attack, the initiator `i` sends a request to `r`. The intruder saves this request in memory. `r` commits and replies to the request. The intruder replays the saved request to `r`. `r` commits the same value again and replies. There are two identical Rescommit events and exactly one corresponding Inirunning event. This violates injective agreement. Clearly, this attack is not possible in practice: it is prevented by message IDs. In practice, the responder refuses the replayed request, as the message ID of this request is strictly lower than "the next [message ID] it expects to see in a request from the other end" [46]. However, there might be other attacks.

For secrecy, aliveness, weak agreement and non-injective agreement, our ProVerif analysis provides much stronger security guarantees than our Spin analysis. Indeed, ProVerif

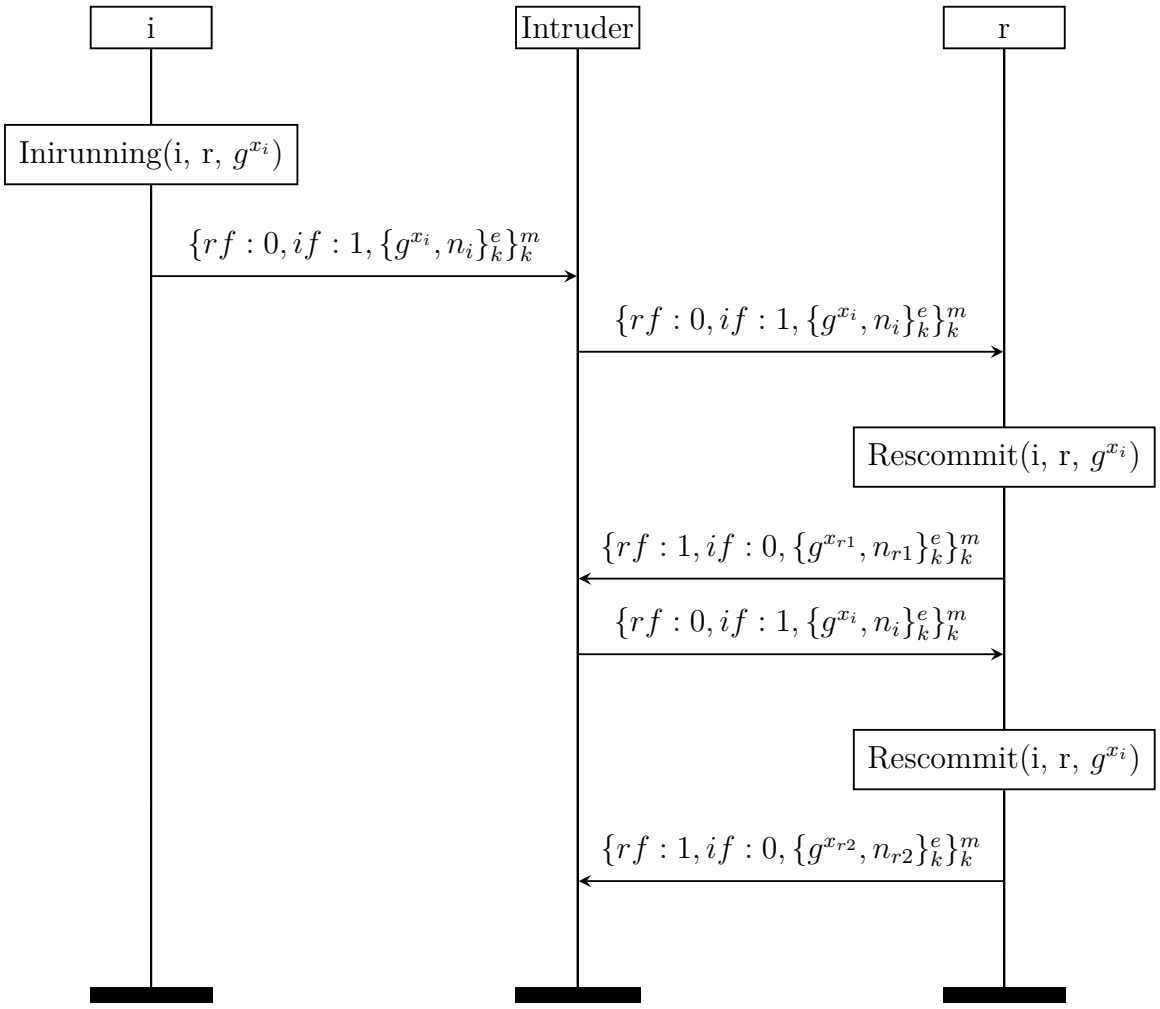


Figure 3.7: Violation of injective agreement for the responder in IKEv2-Child

does not suffer from many limitations of Spin. Using ProVerif, we have proved our properties in a model that allows for an unbounded number of sessions. Moreover, as explained in Section 1.3.2, [19] shows that having proved our properties with 3 agents in the unbounded model actually proves them for an unbounded number of agents. Finally, in ProVerif the attacker has an unbounded memory and can build messages of arbitrary sizes. On all these aspects of the model, Spin does not support unboundedness.

Our ProVerif analysis provides the first proof that IKEv2-PSK guaranties injective agreement to both the initiator and the responder, and that IKEv2-Sig guaranties injective

agreement to the initiator.

3.5 Model checking IKEv2 using Tamarin

We have seen in Section 3.4.1 that while ProVerif surpasses Spin by its unbounded model, ProVerif still suffers from a rather weak model of modular exponentation and does not support counters. In this Section we analyze IKEv2 using the Tamarin tool, which is also capable of verification in an unbounded model, but allows a better model of modular exponentation. We start by depicting the main features of Tamarin and its language. Then we write a Tamarin model of IKEv2. Finally we present our analysis results and draw conclusions on the security of IKEv2.

3.5.1 Tamarin in a nutshell

Let us go over the main aspects of Tamarin.

In Tamarin, like in ProVerif, cryptographic messages are represented by terms. Terms are built using recursive application of functions on names and variables. Names and variables are typed. In Tamarin there are two types: public and fresh. `$x` denotes a variable of type public, `~x` denotes a variable of type fresh, `x` denotes a variable of any type, and `'x'` denotes the public name 'x', i.e. a constant. The user is not allowed to use fresh constants.

A Tamarin file, or *security protocol theory*, begins with the declaration of function symbols and their arity, i.e. a signature. For example, we can define the signature below. In this signature, `pair` (which can also be written `<_, _>`), `fst` and `snd` represent pairing and projection. `senc` and `sdec` represent symmetric encryption and decryption. Symbol `^` represents DH exponentiation. Symbols `inv`, `*` and `1` represent inversion, multiplication, and the unit in the group of exponents.

```
functions: pair/2, fst/1, snd/1, senc/2, sdec/2, ^/2, inv/1, */2, 1/0
```

We must then define some equational theory to assign meaning to our functions. For example, we can use the following equational theory. Axioms 1 and 2 model pairing, axiom 3 models symmetric encryption, and axioms 4 to 10 model DH exponentiation.

```
0 equations:
1     fst(<x.1, x.2>)           = x.1,
2     snd(<x.1, x.2>)           = x.2,
3     sdec(senc(x.1, x.2), x.2) = x.1,
```

```
4       (x^y)^z                 = x^(y*z),
5       x^1                     = x,
6       x*y                     = y*x,
7       (x*y)*z                 = x*(y*z),
8       x*1                     = x,
9       x*inv(x)                = 1
```

Tamarin provides some built-in message theories that define some function symbols and equations to model specific cryptographic primitives. For example, the functions and equations that we defined above are all included in the `pairing`, `symmetric-encryption` and `diffie-hellman` built-in message theories. Furthermore, the `pairing` message theory is automatically included by Tamarin. Therefore, we could have simply included the missing message theories using the following syntax.

```
builtins: symmetric-encryption, diffie-hellman
```

Terms only represent messages. To model a protocol, we need to make statements about these messages. Statements are called facts in Tamarin. Facts are of the form `F(t1, ..., tn)`, where `F` is a fact symbol and each `ti` is a term. The execution of a security protocol is modeled by a transition system, where each state is a multiset of facts, and each transition is labeled by a multiset of facts. The single initial state is an empty multiset. Transitions are defined using rules. Rules are of the form `l --[a]-> r`, where l, a and r are sets of facts. Facts are implicitely defined in rules. Facts in l are the premises, facts in a are the action facts, and facts in r are the conclusions. Action facts specify transition labels. For a rule to be applicable, all facts of l must be in the current state. There are two types of facts: *linear* fact and *persistent* facts. The latter are prefixed by the symbol "`!`". Application of a rule removes linear facts of l from the state, and adds facts of r to the state. Persistent facts are never removed from the state.

Tamarin uses some built-in rules and facts, which we describe hereafter.

The `fresh` rule is the only rule that produces `Fr` facts and Tamarin considers only runs with unique instances of this rule. Linear fact `Fr(x)` represents the generation of the fresh term `~x`.

```
rule fresh:
[] --[]-> Fr(~x)
```

The `send` rule allows the adversary to send any term it knows over the network. Persistent fact `K(x)` means that the adversary knows term `x`. Linear fact `In(x)` means

that `x` is sent over the network by the adversary. When the adversary does so, the action fact K(x) is recorded, which will be used to express adversary knowledge in properties.

```
rule send:
    [ !K( x ) ] --[ K( x ) ]-> [ In( x ) ]
```

The `recv` rule allows the adversary to learn any term that is sent over the network. Linear fact `Out(x)` means that `x` is sent over the network by some protocol session.

```
rule recv:
    [ Out( x ) ] --> [ !K( x ) ]
```

The `pub` and `freshadv` rules create new fresh and public names known to the adversary. The adversary can then use the names created by the `pub` and `freshadv` rules to run the protocol itself, as required by the LKRothers capability, which we need for our adversary model (see Section 2.3.1). Note that a premise `F($x)` will only match the fact `F` applied to a public name. Similarly, `~x` will only match a fresh name, `x` will match any name, and `'x'` will match the public name 'x'.

```
rule pub:
    [ ] --[ !K( $x ) ]-> [ !K( $x ) ]

rule freshadv:
    [ Fr( ~x ) ] --[ !K( ~x ) ]-> [ !K( ~x ) ]
```

Tamarin adds rules allowing the adversary to build new terms using the (non-private) functions and equations. For example, the rule `pair` allows the adversary to use the pair function.

```
rule pair:
    [ !K( x ), !K( y ) ]
  --[ !K( <x, y> ) ]->
    [ !K( <x, y> ) ]
```

The user then defines protocol rules, i.e. rules describing how each role of the protocol behaves. For example, the following rule models a role generating a fresh value and sending it over the network.

```
rule foo:
    [ Fr(~ni) ]
  -->
    [ Out(~ni) ]
```

Finally, the user specifies security properties using *lemmas*. In lemmas, security properties are expressed using trace formulas. Trace formulas are first-order formulas over trace atoms. These formulas use the functions of the model. Trace atoms are of the form $\bot$, `t1 = t2`, `i1 < i2`, `i1 = i2`, or `f @ i`, where t1 and t2 are terms, i, i1 and i2 are integers representing timepoints, and f is a fact. Trace atom `f @ i` means that fact f is in the state at timepoint i. For example, the following lemma states that if any state of the transition system contains the fact Secret(t), then there must be no state of the same trace that contains the fact K(t). This lemma is a simple implementation of secrecy.

```
lemma secrecy:
    "All t #i.
        Secret(t) @i
        ==> not (Ex #j. K(t) @j)
    "
```

Tamarin further allows the user to state some axioms that must be satisfied by all traces considered. Such axioms are called restrictions. For example, the following restriction states that if a trace contains action fact `Eq(x, y)` at some timepoint i, then x and y must be equal at this timepoint.

```
restriction Equality:
    "All x y #i. Eq(x, y) @i ==> x = y"
```

When given a security protocol theory, Tamarin will try to prove the lemmas. During resolution, Tamarin iteratively faces the choice of what *goal* it will try to solve. A goal is a fact for which Tamarin has no rule producing it yet. The tools then offers two modes: automatic and interactive. In automatic mode, Tamarin chooses the next goal automatically using some heuristic. Tamarin offers several built-in heuristics, and even allows the user to write its own heuristic. To do so, the user writes some python program, which Tamarin will execute each time there is a choice of goal to make. In interactive mode, the user chooses the next goal to solve on a graphical interface. Interactive mode is the way Tamarin deals with the undecidability problem of protocol verification.

3.5.2 Modeling IKEv2 in Tamarin

In this Section we model IKEv2 in Tamarin. To do so, we successively write a message theory to model cryptographic operations, protocol rules to model the protocol message flow, and lemmas to implement our properties from Section 2.3.2.

Message theory

Let us present the message theories (function symbols and equations) that we use for our subprotocols.

Following the description of IKEv2 provided in Section 2.2, we need to model symmetric encryption, Diffie-Hellman exponentiation, digital signature, affixed MAC, and MAC. Note that we could model affixed MAC using MAC and pairing, but directly using a theory for affixed MAC greatly simplifies the code.

We use the `symmetric-encryption` and `diffie-hellman` built-in message theories to model symmetric encryption and Diffie-Hellman exponentiation. We already provided the function symbols and equations of these theories in Section 3.5.1.

We use the `signing` built-in message theory to model digital signature. This theory represents signature by function symbol `sign`, which takes as arguments a message and a private key. The theory then defines function symbol `pk` which, given a private key, returns the corresponding public key. Finally, the theory defines functions symbols `verify` and `true`, such that `verify` returns true when given a signed message, the original message and the public key corresponding to the private key that was used to sign the message.

```
functions: sign/2, verify/3, pk/1, true/0
equations: verify(sign(m, sk), m, pk(sk)) = true
```

To model affixed MAC, we define function symbol `revealmac`, which takes as arguments a message and a key. We then define function symbol `getmacmess`, which, given the result of `revealmac`, returns the message.

```
functions: revealmac/2, getmacmess/1
equations: getmacmess(revealmac(m, k)) = m
```

To model MAC, we simply define a function symbol mac/2 that takes as argument a message and a key. We define no corresponding equation, so that no one can retrieve the original message.

Rules

Let us present the rules we write to model our subprotocols.

The signature and MAC theories provide models for signature and MAC algorithms, but they do not provide the corresponding keys. To do so, we define the rule `Generatekeypair` as below. This rule may be instantiated for each agent a. The rule uses some fresh name

`~prk` and declares it as private key for a. For the signing theories to work, the rule declares `pk(~prk)` as public key for a. A public key should be known to the adversary, so the rule outputs `pk(~prk)`.

```
rule Generatekeypair:
    [ Fr(~prk) ]
  --[ Generatekeypair($a, ~prk) ]->
    [ !Pbk($a, pk(~prk))
    , !Prk($a, ~prk)
    , Out(pk(~prk))
    ]
```

We further need to model pre-shared keys. We need them for authentication in IKEv2-PSK, and for MAC in IKEv2-Child (recall that k acts as a pre-shared key in this subprotocol). The `Generatepsk` rule of IKEv2-PSK below uses some fresh value `~psk` and declares it at pre-shared key of a and b. Since the order of arguments in facts matter, and since there is no reason to distinguish the pre-shared key of a and b and the pre-shared key of b and a, the `Generatepsk` rule adds both `!Psk($a, $b, ~psk)` and `!Psk($b, $a, ~psk)` to the state. Note that this rule may be executed an unbounded number of times, so that a and b share an unbounded number of pre-shared keys.

```
rule Generatepsk:
    [ Fr(~psk) ]
  --[ Generatepsk($a, $b, ~psk) ]->
    [ !Psk($a, $b, ~psk), !Psk($b, $a, ~psk) ]
```

However in IKEv2-Child, there is a reason to distinguish the pre-shared key of a and b and the pre-shared key of b and a. Indeed in reality, each pre-shared of IKEv2-Child comes from a phase 1 session, where either a was initiator or b was initiator. Knowing which agent was initiator in phase 1 matters when we set the Initiator flag in the protocol rules of IKEv2-Child. Therefore we model the pre-shared key of an IKEv2-Child session where a is the phase 1 initiator by the fact `!Psk($a, $b, ~psk)`, and vice-versa. The `Generatepsk` rule of IKEv2-Child thus adds only `!Psk($a, $b, ~psk)` to the state.

```
rule Generatepsk:
    [ Fr(~psk) ]
  --[ Generatepsk($a, $b, ~psk) ]->
    [ !Psk($a, $b, ~psk) ]
```

In the code below, we provide an example of protocol rule we write, namely the `Initiator1` rule of IKEv2-Sig. In this rule, the initiator requests some long-term private key `~prki`, thereby requiring that rule `Generatekeypair` appear just before in execution traces. The rule also chooses two fresh names `~xi` and `~ni` and builds the key-exchange term `kei` as specified by the IKEv2 protocol. Next the rule outputs the init request. The term `'rf0'` in the request is a public constant modeling a Response flag set to 0. Finally, the rule saves the state of the initiator session in the fact `Initiator1`. The state of the initiator contains the identity of the agent executing the session, the private key of this agent, and the names that it will need in subsequent rules (`Initiator2` and `Initiator3`).

```
rule Initiator1:
    let kei = 'g'^~xi
    in
    [ !Prk($i, ~prki)
    , Fr(~xi)
    , Fr(~ni)
    ]
  -->
    [ Out(<'rf0', kei, ~ni>)
    , Initiator1($i, ~prki, ~xi, kei, ~ni)
    ]
```

In IKEv2-Child, we must distinguish protocol rules where the agent is original initiator from protocol rules where the agent is original responder. In ProVerif, we achieved that by adding an boolean argument `origini` to processes. In Tamarin, there are no boolean variables. We thus duplicate each protocol rule of IKEv2-Child. For example, the rule `Initiator1` is split into `Initiator1origini` (where the initiator is the original initiator) and `Initiator1origres` (where the initiator is the original responder).

When a rule receives some signature, we could use a premise of the form `In(sign(m, ~prki))`. However, the agent executing the rule is not supposed to know the key pki. The correct way to model signature verification is to use the premise `In(signedmess)` and to trigger the action fact `Eq(verify(authr, m, pbkr), true)`. The equality restriction `Eq`, which we presented in Section 3.5.1, ensures that both its arguments are equal, and thus that signature verification succeeds.

Lemmas

Let us present the lemmas we write to model our security goals from Section 2.3.

Firstly we must implement the LKRothers capability. In other words, we must allow the adversary to learn the long-term keys of agents that are neither the actor nor the peer. In IKEv2-Sig we use the `Revealltk` rule below. This rule can reveal (by outputting it on the network) the private key of any agent. We record the reveal in some action fact `Revealltk`. Then we use this action fact in our lemmas to exclude traces where the the peer's key has been revealed. In fact, we do not exclude traces where the actor's key has been revealed. In other words, our models also gives the LKRactor capability of [5] to the adversary, which yields even stronger proofs.

```
rule Revealltk:
    [ !Prk($a, ~prk) ] --[ Revealltk($a) ]-> [ Out(~prk) ]
```

In IKEv2-PSK, since long-term keys are pre-shared keys, any long-term key reveal affects two agents. Thus in our IKEv2-Child `Revealltk` rule below, the action fact `Revealltk` takes two arguments. Since the pre-shared key of a and b models the same value as the pre-shared key of b and a, and since the order of arguments matter in facts, we must record both `Revealltk($a, $b)` and `Revealltk($b, $a)`.

```
rule Revealltk:
    [ !Psk($a, $b, ~psk) ]
  --[ Revealltk($a, $b), Revealltk($b, $a) ]->
    [ Out(~psk) ]
```

In IKEv2-Child, the pre-shared key of a and b models a value that is different from the pre-shared key of b and a. We thus use `Revealltk($a, $b)` to model the reveal of the pre-shared key used by a session where a is the original initiator. Therefore in the rule below, we trigger only the `Revealltk($a, $b)` action fact.

```
rule Revealltk:
    [ !Psk($a, $b, ~psk) ] --[ Revealltk($a, $b) ]-> [ Out(~psk) ]
```

We provide our implementation of secrecy for IKEv2-PSK and IKEv2-Child in the code below. This lemma simply states that if the `Secret(a, b, t)` action fact has been triggered, meaning that a has computed the secret term t and believes that it has completed the protocol with agent b, then the adversary does not know t, or else it must be that a pre-shared key between a and b has been revealed.

```
lemma secrecy:
    "All a b t #i.
        Secret(a, b, t) @i
        ==> not (Ex #j. K(t) @j) | Ex #r. Revealltk(a, b) @r
    "
```

In the code below, we implement secrecy for IKEv2-Sig in a slightly different way. Because long-term keys are not shared in IKEv2-Sig, here the reveal trace atom states that it must be that the private key of peer b has been revealed.

```
lemma secrecy:
    "All a b t #i.
        Secret(a, b, t) @i
        ==> not (Ex #j. K(t) @j) | Ex #r. Revealltk(b) @r
    "
```

To implement aliveness, weak and non-injective agreement, we proceed similarly to what we did in Spin and ProVerif. We use the facts Inirunning, Resrunning, Inicommit and Rescommit to record the progress of each session. These fact always take as arguments the initiator of the session, the responder of the session, and the term over which the actor agrees. As an example, we provide our implementation of aliveness for IKEv2-Sig below.

```
lemma alivenessini:
   "All a b t #i.
        Inicommit(a, b, t) @i
        ==> (Ex a2 t2 #j. Resrunning(a2, b, t2) @j)
            | Ex #k. Revealltk(b) @k
    "
```

In ProVerif, injectivity was built-in through keyword “inj-event”. In Tamarin, injectivity needs to be implemented. Recall that, as said in Section 2.3.2, injective agreement for the initiator means that for all agents a, b and for all terms t, if a trace contains n times the fact `Inicommit(a, b, t)`, then the trace contains at least n times the fact `Resrunning(a, b, t)`. This property is hard to implement in Tamarin, so we write a slightly different version. Our property states that if a trace contains the fact `Inicommit(a, b, t)`, then the trace contains the fact `Resrunning(a, b, t)` and `Inicommit(a, b, t)` appears only once in the trace. This is implemented in the code below. Our property implies the property of Section 2.3.2 (the opposite is not true), so proving our property does prove the property of Section 2.3.2.

```
lemma injagreeini [heuristic=o]:
   "All a b t #i.
        Inicommit(a, b, t) @i
        ==> (
            Ex #j.
                Resrunning(a, b, t) @j
                & j < i
                & (All #k. Inicommit(a, b, t) @k ==> #i = #k)
            )
            | Ex #l. Revealltk(a, b) @l
    "
```

Note that the inequality "j < i" is necessary. Removing it causes a memory exhaustion during the analysis.

Note in addition that we had to specify our own heuristic (as explained at the end of Section 3.5.1) to make Tamarin able to complete the proof of any injective agreement property. Tamarin offers to do so via the `heuristic=o` option. Using any heuristic built-in to Tamarin caused the analysis to not terminate. In our oracle, we give priority to goals of the form "`!KU(~prk)`", "`!KU(~x)`", and "`!KU('g'^(~xi*~xr))`" (`KU(k)` is roughly equivalent to `K(k)`, i.e. "the adversary knows k"). These goals have in common that they should not figure in a normal execution trace, so most of the times, solving them leads to a contradiction. When Tamarin finds a contradiction, Tamarin moves on to another goal, and the analysis terminates faster. In fact with our oracle we always obtain termination in less than 20s.

3.5.3 Analysis results

Figure 3.8 shows the results of our analysis of IKEv2 using Tamarin. Our models are available at `https://gitlab.inria.fr/tninet/proverif`. The analysis took only a few seconds and little memory on our 2.6 GHz CPU. We obtain similar results to those of our analysis with ProVerif. The attacks we find using Tamarin are the same that we find using ProVerif.

In both cases, security properties are proved in models with unbounded number of sessions and unbounded message size. Moreover in both cases, we did not model IKEv2 message IDs. Note that Tamarin does support counters, but we leave the rather complex work of modeling message IDs as a future work with Tamarin, as explained in Section

Property	Role	IKEv2-Sig	IKEv2-PSK	IKEv2-Child
Secrecy	Initiator	✓	✓	✓
	Responder	✓	✓	✓
Aliveness	Initiator	✓	✓	✓
	Responder	✓	✓	✓
Weak agreement	Initiator	✓	✓	✓
	Responder	✗	✓	✓
Non-injective agreement	Initiator	✓	✓	✗
	Responder	✗	✓	✓
Injective agreement	Initiator	✓	✓	✗
	Responder	✗	✓	✗

Figure 3.8: Analysis of IKEv2 using Tamarin. We write ✓ when a subprotocol guarantees some property to some role, and ✗ when it does not.

3.6.2.

However, our proofs with Tamarin and ProVerif are different on four aspects:

- Tamarin's model of Diffie-Hellman exponentiation is stronger than ProVerif's one.
- Our Tamarin model allows each pair of agents to have an unbounded number of pre-shared keys.
- We gave the LKRactor capability to our Tamarin adversary.
- As said in Section 3.5.2, our implementation of injective agreement in Tamarin is not faithful to Lowe's definition.

3.6 Concluding remarks

We have performed analyses of IKEv2 using three different tools: Spin, ProVerif and Tamarin. In this Section, we compare the three tools by gathering the differences we have come across in our work, and then depict the main challenges that remain in the topic of IKEv2 formal verification.

3.6.1 Tool comparison

Let us go over the differences between the three tools.

Spin is quite different from ProVerif and Tamarin. Most importantly, Spin forces us to bound the number of sessions, the size of messages, and the intruder memory. These three characteristics make Spin provide significantly weaker security proofs than ProVerif and Tamarin. Furthermore, even if e.g. ProVerif and Tamarin do not terminate, Spin may not be the most appropriate alternative, as there exists other tools that are specialized in bounded model checking of security protocols. For example, the tool SATMC [3] bounds the number of sessions, but not the size of messages or the adversary knowledge.

Hereafter are some other differences between the tools.

- The Diffie-Hellman model is more powerful in Tamarin than in ProVerif.
- To deal with the undecidability problem of security protocol verification, Spin uses a bounded model, ProVerif uses abstractions, and Tamarin interacts with the user.
- Tamarin is the most user-friendly of these tools, as it provides a well designed GUI.
- In ProVerif, we can only model a finite number of pre-shared keys per pair of agents. We do not face this problem in Tamarin.
- In Tamarin it is hard to strictly follow Lowe's definition of injectivity. ProVerif natively supports this type of property.

3.6.2 Further research

We now identify research challenges that remain in the topic of IKEv2 formal verification.

Further research must focus on using the Tamarin tool when analyzing IKEv2, as its model is more powerful. In particular, Tamarin offers a way to model counters through the `multiset` builtin message theory. This feature should be used to include the message ID field in IKEv2-Child.

We split IKEv2 into three subprotocols. However, in practice IKEv2-Child is always run after an IKEv2-Sig or IKEv2-PSK session. It could be interesting to merge all three subprotocols into one single Tamarin file. The model would be more faithful to reality.

Finally, stronger adversary models could be implemented. For example, we could implement the adversary models from [5], giving to the attacker the capabilities LKRafter,

SKR, RNR and SR. This would allow us to verify e.g. perfect forward secrecy. These properties have already been checked on IKEv2 using the Scyther tool in [24], but their models did not include the IF, RF and message ID flags, and their model of Diffie-Hellman exponentiation is weaker than that of Tamarin.

Chapter 4

The Deviation Attack

We have seen in Chapter 3 that IKEv2-Sig satisfies only a weak form of authentication. More precisely, IKEv2-Sig does not satisfy the weak agreement property. In this chapter we design and implement a Denial-of-Service (DoS) attack, namely the Deviation Attack, that exploits this vulnerability. The Deviation Attack has different requirements than classic DoS attacks and has lower bandwidth consumption, which makes it stealthier.

We firstly provide in Section 4.1 some background on DoS attacks, and in particular on those against IKEv2. We then present in Section 4.2 the Deviation Attack as a generic attack against any IKEv2 implementation and possibly against other similar protocols. In Section 4.3, we implement the attack against a well-known open-source implementation of IKEv2, and experimentally validate some important characteristics of the attack. In Section 4.4, we tackle the problem of finding countermeasures to the Deviation Attack. We end this Chapter in Section 4.5 with some concluding remarks.

4.1 Background

In this Section we introduce the problem of DoS attacks in general and in the case of IKEv2.

4.1.1 Denial-of-Service attacks

Let us briefly sketch the DoS problem by defining what a DoS is and pointing out some characteristics that make a DoS attack strong.

A DoS attack is an attack that makes a given ressource (e.g. server application) unavailable to some or all of its clients during a finite or infinite duration. An important characteristic of DoS attacks is the strength of the attacker. Obviously, if an attacker has total control over the server, or can drop all packets between the clients and the server, the attacker can easily cause a DoS. However, if an attacker can achieve Denial-of-Service by

simply sending self-made messages from a single machine to the server, then the resource definitely needs some more protection. Another important characteristic of DoS attacks is how much bandwidth they consume. Indeed, the higher the bandwidth consumption, the easier it is for Intrusion Detection/Prevention Systems (IDS/IPS) to catch the attack.

4.1.2 Denial-of-Service attacks against IKEv2

In this Section we present the main DoS attacks that are currently possible against IKEv2, as well as their mitigations. Note that this topic is also well covered in [62].

The most obvious DoS attack against IKEv2 consists in sending a high throughput of init requests to some victim. Upon processing of each init request, the victim (playing the role of responder) will generate a Diffie-Hellman (DH) private part x_r and bundle it with the received DH public part ke_i and some other values in a *half-open Security Association (half-open SA)*. Each half-open SA gets stored in a half-open SA database. Eventually, the victim will run out of memory.

As a first protection, IKEv2 mandates that implementations *remove half-open SAs after a short time*.

As a second protection from such an attack, IKEv2 mandates that the victim *limits the amount of concurrent half-open SAs from a particular address or prefix*. The attacker could overcome this protection by using spoofed source addresses. However, the victim can in turn use the stateless cookie mechanism [46]. If a responder uses the cookie mechanism, when the responder detects a large number of half-open SAs, it responds to each init request (that does not contain a cookie) with an IKEv2 INFORMATIONAL message containing a cookie. The cookie is a keyed hash of the request. The initiator then sends the same request again with the cookie added to it, and the responder verifies that the cookie and the request match. This means that the attacker needs to keep in memory the init requests it sends, making it more costly for an attacker to fill the half-open SA database of a gateway. But more importantly, it forces the initiator to receive the cookie at the IP address that it used as source IP address of the corresponding init request. This effectively prevents an attack with spoofed source addresses. Note that cookies are enabled by default in strongSwan, one of the most popular open-source IKEv2 implementations.

As a third protection, the victim can use the *puzzle mechanism* introduced in [62]. Puzzles are an improvement of the cookie mechanism. A responder who uses puzzles asks the initiators to solve some challenge before the responder sets up a half-open SA. An initiator now needs to remember its request (a puzzle contains a cookie), and to solve a

puzzle before sending its request again. Puzzles increase the cost in CPU power of setting a half-open SA in a victim. Doing so, puzzles effectively set an upper bound to the number of half-open SAs each attacking machine can create in the victim over a given period. Therefore if the CPUs of the attacking machines are not powerful enough, the attacker may need to use even more machines to achieve sending the required throughput of init requests for causing a DoS.

A more sophisticated DoS attack against IKEv2 (also described in [62]) consists in sending a high throughput of init requests, and for each response, sending a dummy auth request as well. A dummy auth request is made of a valid IKE header, and an encrypted payload with a valid payload header but junk data as its content. This will cause the victim to perform key derivation (note that in order to be more resilient to DoS attacks, key derivation is only performed now, and not upon reception of the init request) and to check the MAC of the encrypted payload. These operations are CPU-wise heavy. This time the goal of the attack is to exhaust the CPU of the victim. The three protections presented above (limits the amount of concurrent half-open SAs from a particular address or prefix, removing half-open SAs after a short time, and the puzzle mechanism) remain efficient against this attack as well. Someone using this attack would likely have to send a high throughput of init and auth requests, using a high number of real machines (a botnet).

The Deviation Attack, which we introduce in this chapter, consumes less bandwidth than classic DoS attacks. Furthermore, the Deviation attack has different requirements. Where classic DoS requires a botnet, the Deviation Attack requires some control over the network.

4.2 The Deviation Attack in theory

In this Section, we present the Deviation Attack as an implementation-agnostic and sometimes protocol-agnostic attack. We start by giving an overview of the attack in Section 4.2.1. We then present the context in which the attack may take place in Section 4.2.2, give a necessary and sufficient list of requirements for the attack in Section 4.2.3, and describe the attack flow in Section 4.2.4. We next evaluate the strength of the attack by comparing it to classic DoS attacks in Section 4.2.5. We end the Section in Section 4.2.6 with some remarks.

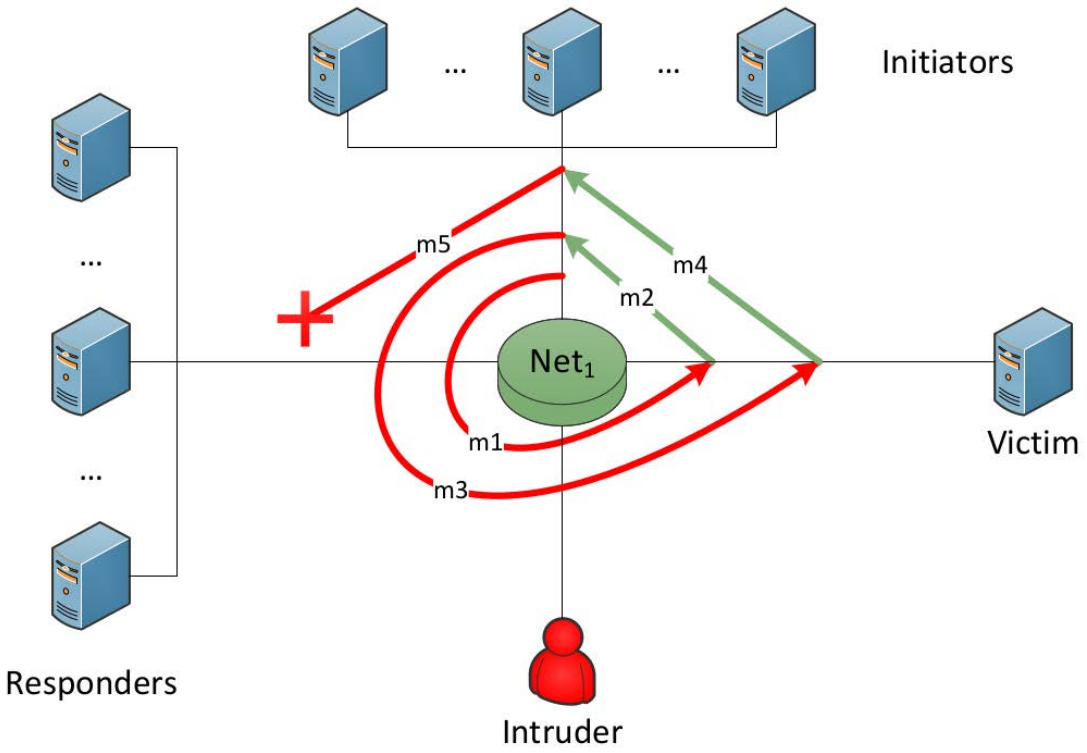


Figure 4.1: Scenario of the Deviation Attack. Intruder deviates all init and auth requests and drops all authfailed notifications. Red arrows indicate deviated or dropped messages, whereas green arrows indicate messages untouched by the attacker. $m1$, $m2$, $m3$, $m4$ and $m5$ respectively denote an init request, an init response, an auth request, an auth response and an authfailed notification.

4.2.1 Characteristics

Let us depict the main characteristics of the Deviation Attack. We provide a schematic view of the attack on Figure 4.1. We will explain this Figure more in detail in Sections 4.2.2, 4.2.3 and 4.2.4.

As explained in Section 3.1.2, IKEv2 only satisfies a weak form of authentication. More precisely, when the responder sets up a Child SA with a peer, there is no guarantee that the peer wanted to set up a connection with the responder. Indeed, an attacker can simply intercept an init request coming from an initiator and send it to a different peer than originally intended by the initiator. This other peer (the victim), has no way of noticing that it is not the intended partner of the initiator. The victim thus responds with an init response to the initiator. The initiator then sends an auth request, which the intruder deviates again towards the victim. Under the condition that the victim trusts

the initiator, the victim will set up a Child SA with initiator, who did not want to set up a Child SA with the victim.

The situation described above seems harmless because the key of the Child SA remains secret to the intruder. However, if the intruder finds many requests to deviate, then many Child SAs will be set up in the victim, which will eventually exhaust the victim's memory. We call such an attack a Deviation Attack, as it uses deviation of legitimate messages in order to exhaust the victim's memory. Notice that Deviation Attacks are not specific to IKEv2, but rather to any stateful protocol that suffers from the same authentication weakness.

The main difference with classic DoS attacks described in Section 4.1.2 is that the intruder leverages requests from legitimate IKEv2 parties. Doing so, it sets full IKE and Child SAs in the victim, instead of simple half-open SAs. This is of importance for two reasons. Firstly, full SAs take more place in memory. Secondly, full SAs stay longer in memory. As a consequence, the intruder can run the attack at a slower pace, and thus consume much less bandwidth. This makes the attack harder to detect than classic DoS attacks.

Furthermore, requirements for a Deviation Attack are simply not the same as for a classic DoS attack. Where an intruder performing a classic DoS attack needs to send a high throughput of forged requests from a high number of real machines (see Section 4.1.2), an intruder performing a Deviation Attack needs to control the link between initiators and their original recipients, and find enough requests to deviate.

4.2.2 Context

In this Section we present the context in which the Deviation Attack may take place and give some preliminary notations.

For more readability, we use the term *init request* as an abbreviation for IKE_SA_INIT request, the term *init response* as an abbreviation for IKE_SA_INIT response, the term *auth request* as an abbreviation for IKE_AUTH request and the term *auth response* as an abbreviation for IKE_AUTH response. We use the term *authfailed notification* as an abbreviation for INFORMATIONAL message containing an AUTHENTICATION_FAILED notification payload (see [46]).

We assume the existence of N_{ini} IKEv2 parties called *Initiators*, N_{res} IKEv2 parties called *Responders*, and one IKEv2 party called *Victim*. Each party may either be an IKEv2 endpoint (also called host) or a gateway. All parties are connected by an IP network Net,

which may be untrusted. Figure 4.1 presents the attack scenario.

We call *connection* the total set of SAs created by an init and an auth exchange: If everything goes well, *connection* denotes one IKE SA (made of two unidirectional SAs) and one Child SA (made of two unidirectional SAs). However, if authentication succeeds but e.g. traffic selector (TS) negotiation fails, then *connection* only denotes one IKE SA alone. Recall that traffic selectors are payloads of the auth exchange that are used to specify which IP addresses will be allowed to communicate through the resulting Child SA (see Section 2.2.4).

Let L be the memory load of Victim, i.e the amount of Victim's memory that is consumed by the IKEv2 application and by connections with machines that do not belong to the Initiators defined above. We assume for simplicity that L is constant over time. We also assume that Victim's memory is statically limited, i.e. there is no way for Victim to obtain more memory capacity during the attack. Let C be the memory capacity that was allocated to IKEv2 in Victim. We assume that C was carefully chosen and that we have $C > L$.

We say that a connection in a party's memory is *unintended* when the connection was not taken into account when memory was allocated to the party, and when the size of that memory was decided. Let m be the amount of memory needed to store all data related to one connection. We say that a connection in a party's memory is *stale from the beginning*, when the party has received no IKEv2 message for it since it was installed. After some time and some unanswered rekeying requests or keep-alive requests, the party removes the connection. Let S be the amount of time a connection stale from the beginning stays in Victim's memory. For simplicity, we assume that S is constant.

Let *Intruder* be a machine, connected to Net. Let $t = 0$ be the beginning of the attack and D be the attack duration. We assume for simplicity that the Initiators send init requests to Responders at a total constant rate σ between $t = 0$ and $t = D$. We express σ in number of IKE messages per second (and not in packets per second, in case fragmentation occurs in the network). We also assume that, at $t = 0$, no Initiator party has any SA established with Victim.

We say that Intruder *deviates* a message towards a machine, when Intruder intercepts the message on Net, changes the message's destination IP address to the machine's IP address, and sends the message back on Net. When Intruder intercepts a message on Net, the original recipient does not receive it. This is thus different from "eavesdropping".

Let $\Phi(\sigma, t)$ be the predicate: "Victim is in Denial-of-Service at time t when using a

rate σ of init requests".

4.2.3 Requirements

In this Section, we make a necessary and sufficient list of conditions that must be met for the Deviation Attack to be possible.

Req1 Intruder must have the ability to intercept every IP packet sent by an Initiator party to a Responder party. After interception, Intruder must be capable of either dropping the message, or deviating it towards Victim. These abilities are a subset of Dolev-Yao [30] abilities.

*Req*1 requires that the attacker control some router between the initiators and the responders, so that it can intercept and drop messages. Only high-potential attackers (such as governments) may have such abilities.

Req2 Authentication of the Initiators to Victim must succeed. This requirement may be satisfied even though the init requests are not originally destined to Victim. This comes from the fact that Initiators do not specify to whom they want to talk in their requests. The fact that *Req*2 is sometimes satisfied constitutes a violation of the weak agreement property, as defined in Section 2.3.2. If IKEv2 had satisfied weak agreement, the Deviation Attack would not have been possible.

Authentication methods supported by IKEv2 include pre-shared key, digital signature, and Extensible Authentication Protocol (EAP) methods. In this thesis we focus pre-shared key and digital signature. If the Initiators authenticate themselves using signature mode, then the attack requires that all Initiators be trusted by Victim, i.e. that Victim authorizes their identity in its IKEv2 configuration. If the Initiators authenticate themselves using pre-shared key mode, then the attack requires that Victim be part of the pre-shared key community to which the Initiators and Responders belong.

*Req*2 is often satisfied inside large companies or governments. Indeed, machines belonging to the same entity often trust (i.e. accept the authentication of) all other machines of the entity.

Req3 There must be a sufficient throughput of init requests going from the Initiators towards the Responders, for the intruder to deviate. The attack also requires that each of these init requests come from different Initiators (with different identities). Otherwise

Victim will simply replace current connections with new ones. In fact replacing current connections with new ones is not mandated by the RFC, but strongSwan behaves this way by default.

*Req*3 depends on the size of the entity to which the machines belong (see *Req*2) and on the activeness of the Initiators. *Req*3 is likely to be satisfied for large companies or governments.

Req4 All init requests sent by Initiators contain at least one SA proposal (see Section 2.2.4) that is acceptable to Victim. *Req*4 is often satisfied, because if Victim trusts Initiator (*Req*2), then Victim and Initiator are likely to have been configured with some common cryptographic algorithms.

4.2.4 Attack flow

In this Section, we thoroughly describe how the Intruder may proceed to perform the Deviation Attack. The message flow is shown on Figure 4.1. On this Figure, $m1$, $m2$, $m3$, $m4$ and $m5$ respectively denote an init request, an init response, an auth request, an auth response and an authfailed notification.

Let us assume that *Req*1, *Req*2, *Req*3 and *Req*4 are satisfied. The attack proceeds as follows. Intruder intercepts all init requests sent by Initiators to Responders. Let us consider the implications of one specific init request sent by some Initiator to some Responder.

Thanks to *Req*1, Intruder deviates the request towards Victim. In response, because *Req*4 is satisfied, Victim sends an init response to Initiator. Initiator receives it, and sends an auth request to Responder. Intruder deviates the auth request to Victim.

On reception of the auth request, authentication of Initiator to Victim succeeds thanks to *Req*2. However, traffic selector negotiation may fail. If traffic selector negotiation fails, then only one IKE SA (made of two unidirectional SAs) is stored. If traffic selector negotiation does not fail, then one IKE SA and one Child SA (made of two unidirectional SAs) are stored. According to our definition of *connection*, at this point Victim has installed one connection with Initiator. Victim then sends an auth response to Initiator.

On reception of the auth response, Initiator fails the authentication step, since it intended to speak with Responder, not with Victim. Initiator thus sends an authfailed notification to Responder. Thanks to *Req*1, Intruder intercepts the notification and drops it. As a result, an unintended connection was added to Victim's memory.

In this thesis, we only explore memory exhaustion as a possible cause of Denial-of-Service. Since Victim's memory is statically limited and thanks to $Req3$, there exists a throughput σ such that, at some time during the attack, Victim is in Denial-of-Service by memory exhaustion ($\Phi(\sigma, t)$). More formally, we have shown that if $Req1$, $Req2$, $Req3$ and $Req4$ are satisfied, then:

$$\exists \sigma \mid \exists t \in [0, D] \mid \Phi(\sigma, t)$$

DoS ends when both the attack stops *and* connections are removed. If the attack stops, but connections are not removed (because $t < S$), then the memory of Victim remains exhausted and does not allow for new connections: DoS continues. If connections are being removed (because $t > S$), but the attack continues, then nothing changes (recall that we assumed that σ and S are constant in time) and DoS continues.

4.2.5 Attack strength

In this Section we evaluate the strength of the Deviation Attack by comparing its requirements and bandwidth to those of classic DoS attacks. To do so, we start by expressing the minimum deviation throughput allowing a DoS, as well as the time it takes for the Deviation Attack to generate a DoS. We then present the Dead Peer Detection (DPD): a mechanism sometimes used in IKEv2 implementations that impacts the computation of DoS times and minimum throughputs. Finally, computation of DoS times and minimum throughputs allows us to assess the strength of the attack in terms of requirements and bandwidth consumption.

Minimum throughput and DoS time

In this Section, we express the minimum deviation throughput that can trigger a DoS, as well as the moment at which the DoS will occur, in function of the parameters of the problem.

We define Σ_{mem} as the minimum throughput triggering a memory exhaustion.

Definition 4.2.1.

$$\Sigma_{mem} = min(\{\sigma \mid \exists t \in [0, D] \mid \Phi(\sigma, t)\})$$

When $\sigma \geq \Sigma_{mem}$, we define $T^s_{mem}(\sigma)$ as the moment at which DoS starts, and $T^e_{mem}(\sigma)$ as the moment at which DoS ends.

Definition 4.2.2. Let $\sigma \geq \Sigma_{mem}$. We define:

$$T^s_{mem}(\sigma) = min(\{t \in [0, D] \mid \Phi(\sigma, t)\})$$
$$T^e_{mem}(\sigma) = max(\{t \in [0, D] \mid \Phi(\sigma, t)\})$$

Theorem 4.2.1. *Recall that S denotes the (constant) amount of time a connection stale from the beginning stays in Victim's memory. We state that:*

$$\Sigma_{mem} = \frac{C - L}{m \times min(D, S)}$$

Furthermore, when $\sigma \geq \Sigma_{mem}$, memory exhaustion starts and ends at:

$$T^s_{mem}(\sigma) = \frac{C - L}{m\sigma}$$
$$T^e_{mem}(\sigma) = max(D, S)$$

Proof. Summing up implications of all init requests sent by Initiators to Responders, at time t, Victim has installed $\sigma \times t$ connections in its memory. However, all these connections are stale from the beginning, so they are removed after S seconds. Therefore at time t, the number of unintended connections that are present in Victim's memory is $\sigma \times min(t, S)$. Victim thus suffers from memory exhaustion at time t if and only if:

$$\sigma \times min(t, S) > \frac{C - L}{m}$$

Since the attack last D seconds, the attack leads to a memory exhaustion if and only if:

$$\sigma \times min(D, S) > \frac{C - L}{m}$$

Which yields our expression of Σ_{mem}.

Now we assume that $\sigma \geq \Sigma_{mem}$. Memory exhaustion will start as soon as:

$$\sigma \times t > \frac{C - L}{m}$$

Which yields our expression of T^s_{mem}.

As said at the end of Section 4.2.4, memory exhaustion lasts until both the attack

stops and connections are removed, i.e. until $T^e_{mem}(\sigma) = max(D, S)$.

□

We confront the above theorem with the experiment in Section 4.3. To do so we implement the Deviation Attack and measure in different experimental setups the moment at which DoS starts.

Dead Peer Detection

Let us introduce the DPD mechanism, which impacts the calculation of DoS times and minimum throughputs.

The RFC of IKEv2 [46] recommends the use of Dead Peer Detection [38] (DPD):

> If no cryptographically protected messages have been received on an IKE SA or any of its Child SAs recently, the system needs to perform a liveness check in order to prevent sending messages to a dead peer.

When DPD is in use, whenever a party sees that no traffic has recently been received on an IKE SA or any of its Child SAs, then it may send a keep-alive request to the SA's peer. If the peer does not respond, after several retransmissions, the party may remove the IKE SA and all its Child SAs from its memory. In the context of the Deviation Attack, DPD reduces S, the average time a connection stale from the beginning stays in memory. Since Σ_{mem} is inversely proportional to S, DPD makes it harder to achieve a memory exhaustion using the Deviation Attack.

However, reducing S too much can create an overload on the network, so S cannot be too low. Therefore, DPD only mitigates memory exhaustion. We provide some numerical application in Section 4.2.5. Note that strongSwan and Libreswan slightly overlook the recommendation of the RFC since in both implementations, DPD is disabled by default.

Comparing requirements with classic DoS attacks

Let us compare the requirements of the Deviation Attack with those of classic DoS attacks.

Let us consider a victim with a memory of 2 GB, and an attacker aiming to exhaust the memory of the victim only by sending a high throughput of self-forged init requests (this is one of the classic DoS attacks described in Section 4.1.2). In strongSwan by default half-open SAs are removed after 30 s. A half-open SA is about 1 kB large [62]. Hence, using this attack, an attacker would need to send:

$$\frac{2 \times 10^9}{10^3 \times 30} = 67\,000 \text{ init requests per second}$$

Moreover, as mentioned in Section 4.1.2, IKEv2 mandates that Victim limit the amount of concurrent half-open SAs from a particular address or prefix. This limit is set to 5 by default in strongSwan (it is the option `charon.block_threshold` of *strongswan.conf*). Recall that the attacker can not use source IP spoofing because of cookies. Therefore, the attacker should generate the init requests from at least 13 000 different real machines. In addition, as explained in Section 4.1.2, if the victim uses puzzles, the attacker may need to use even more machines to achieve sending the required throughput of init requests for causing a DoS.

Where classic DoS attacks set up half-open SAs in Victim's memory, the Deviation attack sets up full IKE SAs (as well as Child SAs if traffic selectors are acceptable to Victim, see Section 4.2.2). We measured that a full connection (one IKE SA + one Child SA) is 23 kB in strongSwan. In strongSwan, when DPD is enabled, connections with a dead peer (such as in the Deviation Attack) are removed after a default duration of 195 s (default DPD delay + default total retransmission timeout). It does not make sense to go much lower, and some implementations might want to set this timeout higher to avoid network congestion. Using these values for our 2 GB, we find that an attacker performing the Deviation Attack would need to find 446 init requests to deviate per second.

When DPD is disabled in strongSwan, connections with a dead peer are removed at the time of rekeying, i.e. by default after 1 h (option `lifetime` of *ipsec.conf*). If DPD is disabled in our example, an attacker performing the Deviation Attack would thus need to find only 24 init requests to deviate each second (18 times less than if DPD is enabled).

We see that a classic DoS attack requires sending at least 67 000 init requests per second using a botnet of at least 13 000 different machines. Moreover, each machine of the botnet must be capable of remembering its requests and answering to Victim's cookie response. Note that botnets of this size exist. One of the world's largest botnets, srizbi [86], was estimated to be around 450 000 compromised machines [78]. On the other hand, the Deviation Attack has two strong requirements: the attacker must have deviation capabilities ($Req1$), and must find a sufficient throughput of init requests to deviate ($Req3$). The deviation capabilities are the most difficult to have. As said in Section 4.2.3, only very high-potential attackers may have these abilities.

Comparing bandwidth consumption with classic DoS attacks

Let us compare the bandwidth consumption of the Deviation Attack with that of classic DoS attacks. We will show that the Deviation Attack uses much less bandwidth and is therefore much harder to detect.

Let us calculate the bandwidth consumed by the two types of attack in the example of Section 4.2.5. Considering a size of approximately 1 kB for an init request, the classic DoS attack uses $67\,\mathrm{MB\,s^{-1}}$ of bandwidth. If the victim uses DPD, the Deviation Attack consumes a bandwidth of $446\,\mathrm{kB\,s^{-1}}$, which is 150 times lower than classic DoS attacks. If the victim does not use DPD, then the Deviation Attack consumes a bandwidth of $24\,\mathrm{kB\,s^{-1}}$, which is 2800 times lower than classic DoS attacks.

The direct consequence of low bandwidth consumption is that the Deviation Attack is much harder to counteract using intrusion detection/prevention systems. In fact other DoS attacks sharing the stealthiness of the Deviation Attack have already been discovered against other protocols. The slowloris attack [79], for example, maintains open many TCP connections in a victim by sending many HTTP requests and HTTP keep-alives to the victim. Eventually the thread pool of the victim is exhausted. DoS attack that requires very low amount of bandwidth is called a slow DoS attack (SDA) [12]. Similarly to the Deviation Attack, SDAs usually target a listening daemon on a host by exploiting some application layer vulnerability.

4.2.6 Remarks

In this Section we make some remarks about the attack. Firstly, we answer the legitimate question of why the Initiator of our scenario does not refuse the init response based on the source IP address of the packet. Secondly, we point out a possible way for Intruder to find a sufficient throughput of init requests to deviate in order to exhaust the memory of Victim.

Why Initiator does not refuse the init response

In the context of a Deviation Attack, Initiator sees that the source IP address of the init response is not the destination address of the init request that was sent. At first glance, this observation could be considered as an indication that something wrong is happening. However, the IKEv2 RFC specifically says that “Incoming IKE packets are mapped to an

IKE SA only using the packet's SPI, not using (for example) the source IP address of the packet" [46]. This is why Initiator does not refuse the init response.

A way to obtain enough requests to deviate

For the attack to work, there needs to be a high enough rate of init requests that are sent from the Initiators to the Responders, in a short enough duration. If this situation never arises, Intruder may have a workaround: it can drop all messages coming from the Initiators and going to the Responders, for a given time. After some unanswered IKEv2 keep-alive requests (if Dead Peer Detection is activated, see Section 4.2.5) or some unanswered child requests, the Initiators may consider their connections with the Responders as broken and may send new init requests for each broken connection. This solution works for example if the connections are configured to be automatically set back up when broken (option `closeaction=restart` of *ipsec.conf* in strongSwan), or so as to be automatically set up when an outbound IP packet arrives (option `auto=route` of *ipsec.conf* in strongSwan).

4.3 The Deviation Attack in practice

In this Section, to concretely demonstrate the Deviation Attack implications, we attack the IKEv2 implementation strongSwan [83] version 5.1.2. Our experiment code is available at `https://gitlab.inria.fr/tninet/demo`. Our experiment allows us to experimentally verify our expression of T^s_{mem} that we give in theorem 4.2.1.

In this Section, we successively present strongSwan, our experiment setup, the method we use to validate our expression of T^s_{mem} and the results we obtain. Finally we discuss some limitations of our experiment.

4.3.1 strongSwan

In this Section, we provide an overview of the strongSwan software and observe that strongSwan does not strictly follow the IKEv2 RFC on some aspects (this will matter in our experiment).

We chose to attack because it is currently one of the most popular open-source IKEv2 implementations. strongSwan runs on modern Linux kernels, Android, FreeBSD, OS X, iOS and Windows [83]. strongSwan is available as package in many Linux distributions,

such as Arch Linux, Debian, openSUSE, Gentoo, Fedora, Red Hat Enterprise Linux and CentOS via EPEL [84].

We observe that strongSwan differs from IKEv2's RFC on one point. When the auth request IDr payload does not correspond to any of the responder's identities, strongSwan notifies that no matching peer configuration has been found and cancels the IKE SA establishment. In the RFC, it is said the following: "If the IDr proposed by the initiator is not acceptable to the responder, the responder might use some other IDr to finish the exchange". In other words, if IDr does not correspond to one of its identities, the responder might install the initial Child SA anyway, and use some other IDr to finish the exchange. This IDr may then be refused by the initiator, who then closes the connection.

The auth request IDr payload is optional, both in the RFC and in strongSwan. The behaviour adopted by strongSwan and described above thus implies the following for the Deviation Attack: When the Initiator machines run strongSwan and are configured so as not to send an IDr payload, the attack works. However, when they run strongSwan and are configured so as to send an IDr payload, the attack does not work, since IDr would be equal to the identity of a Responder, and not to the identity of Victim.

Not sending IDr payload in an auth request is not an uncommon configuration, since it allows to hide the responder's identity to an active attacker. According to [69], IKEv2 was designed so as to hide both identities from a passive attacker and the responder's identity from an active attacker as well. However, IKEv2 was not designed to hide the initiator's identity from an active attacker. To learn this identity, an active attacker can intercept the init request, drop it, and send its own init response by spoofing the responder's IP address. The initiator will then send an auth request encrypted by the key it shares with the attacker. The attacker intercepts the auth request, decrypts it, and learns the initiator's identity. Now assume that we send an IDr method in the auth request. Using the same attack, the intruder is now able to learn the identity of the responder, and even to prove that the initiator intended to speak to the responder. In other words, the responder's identity is no longer hidden from an active attacker. This may be a problem if sensitive information can be found in the ID payload. This is often the case, as people often use Distinguished Names (DN), where the country, institution name and email address are given.

4.3.2 Setup

In this Section, we present the setup of our experiment. We start by presenting the setup globally, and then focus on the configuration of each machine that is part of the setup.

Global setup

To reproduce the attack, we create 2 Linux Virtual Machines (VM) representing Victim and Intruder, and N_{ini}^{demo} VMs representing the N_{ini} Initiators, where N_{ini}^{demo} is a configurable parameter. We do not instantiate the Responders, since we only need their IP addresses; we do not really need the machines. In addition, we create a VM called *Probe*, whose purpose is to detect DoS situations.

All machines are connected through the same local (virtualized) network (representing Net), and they are the only ones connected to it. Using a local network allows us to easily reproduce the deviation of packets by Intruder, as explained below. Moreover it ensures more control over networking propagation times and over the daemon's resource loads.

We create a Certificate Authority that we call *CA*. We generate for Probe, Victim and each Initiator a certificate signed by CA, and its associated private key. We place in Probe, Victim and each Initiator their respective certificates and private keys, along with CA's certificate.

Implementing the Initiators

We decided, for practical reasons, not to create the N_{ini} Initiators of the generic scenario, but instead to create only N_{ini}^{demo} Initiators, with each of them sending $\frac{N_{ini}}{N_{ini}^{demo}}$ init requests to the Responders. However, by default in strongSwan, a party requires an IKE ID to be unique among the IKE SAs it manages. That is, when that party receives an auth request with an IDi payload that is equal to the peer ID of an existing IKE SA in its SAD, it will delete the old IKE SA and replace it with the new one. Because of this in our experiment, the Deviation Attack would add only N_{ini}^{demo} unintended connections in Victim. To stay faithful to the generic scenario, we tell Victim not to delete the old IKE SA in this situation, and instead, to set up the new one alongside. We achieve this by setting below option of *ipsec.conf* in Victim. This way, the Deviation Attack adds up to N_{ini} unintended connections in Victim. We stress that setting that option in our demonstration is equivalent to not setting that option in the generic scenario. Hence our demonstration keeps its demonstrative power in that regard.

```
uniqueids = never
```

Furthermore, we make the Initiators intending to talk to only one Responder. This makes it easier to implement Intruder because that way Intruder needs to spoof only one IP address (see below to understand why Intruder performs spoofing). However in strongSwan, when a party needs to set up a new Child SA with a peer, and already has an IKE SA set up with it, the party will reuse this IKE SA and send a child request. This would not be faithful to the generic scenario. We thus make Initiator send an init request every time it wants to set up a Child SA. We achieve this by setting the following option of *strongswan.conf* in the Initiator VMs. Similarly to above, setting that option in our demonstration is equivalent to not setting that option in the generic scenario. Hence our demonstration keeps its demonstrative power in that regard.

```
charon.reuse_ikesa = no
```

For strongSwan, we set the `rightid` option of *ipsec.conf* in the Initiators like below. The "%" sign forces strongSwan not to send IDr in the auth request. As explained in Section 4.3.1, the auth request IDr payload is optional in the RFC of IKEv2, and strongSwan processes this payload differently than mandated by the RFC. Hence, not sending IDr in the auth request is required in order to run the protocol in a strictly RFC-compliant way.

```
rightid="%CN=responder"
```

The Initiators try to establish connections with Responder at a configurable rate σ. We stop the attack after some configurable time D.

Implementing Intruder

To reproduce the deviation of packets by Intruder, we use an Address Resolution Protocol (ARP) cache poisoning attack [22]. In this attack, Intruder sends ARP replies to all Initiators, binding its MAC address to Responder's IP. This way, all packets sent by Initiators to Responder are intercepted by Intruder. To perform this attack, we use the *arpspoof* tool [80] in Intruder. We then use Linux *iptables* command to redirect the traffic towards Victim and to drop the AUTHENTICATION_FAILED notification. Of course this method is only possible because we use a local network. In reality, when Net is not a local network, deviation has to be made using other ways. For example, deviation could be achieved by taking control of a router between the Initiators and the Responders.

Implementing Probe

The Probe VM tries to set up a new IPsec connection every two seconds (configurable). For each attempt, after five seconds (configurable), Probe checks if the attempt has succeeded (by looking at whether this particular connection is up) and reports the result on standard output. If two consecutive attempts fail (configurable), we consider that Probe has been denied a service.

Implementing Victim

We use Linux Control groups (Cgroups) to allocate a (configurable) memory of exactly C to Victim for strongSwan. Note that in our setup, traffic selector payloads (see Section 2.2.4) sent by the Initiators are not valid propositions for Victim. Thus a connection will only consist of one childless IKE SA (containing two unidirectional SAs).

We observed in our experiment that when there is no memory left for strongSwan, the Linux Out-Of-Memory killer (OOM killer) of Victim kills the strongSwan process. This leads to the loss of all installed SAs. This is undesirable behaviour: setting back up all SAs might take some time, meanwhile suspending the protected IP flow that used Child SAs (which is a DoS as well). To observe the DoS in the form of a memory exhaustion in our experiment, we disable the Out-Of-Memory killer of the kernel and of strongSwan's control group in Victim. However, we stress that, as we just pointed out, DoS happens even if OOM killers are enabled (but in an other form). Thus OOM killers are not a countermeasure.

We do not want Probe to affect Victim's memory consumption. We thus make Probe's connections in Victim ephemeral, i.e. Probe's connections are removed after a short time from Victim's memory. To do so, we use the following options for strongSwan in Victim:

```
# In ipsec.conf
conn probe
    inactivity=5s

# In strongswan.conf
charon.inactivity_close_ike=yes
```

4.3.3 Method

Let us present the method we use to experimentally verify our expression of T^s_{mem}.

Recall that L is the load of Victim, i.e the amount of Victim's memory that is consumed by the IKEv2 application (and by connections with machines that are not Probe or Initiator machines, but there are none of them in our experiment). C is the memory capacity that was allocated to IKEv2 in Victim. m is the amount of memory needed to store all data related to one connection. S is the average time a connection stale from the beginning (see Section 4.2.2) stays in Victim's memory. Finally, σ is the rate at which the Initiators send init requests to Responders.

To experimentally verify our expression of T^s_{mem}, we measure L and m for strongSwan in the context of our setup (we explain how we make these measures below). We then verify that L and m are constants (i.e. that they do not depend on C or σ), and measure T^s_{mem} ($measured\ T^s_{mem}$) when tuple (C, σ) varies.

To measure L we simply measure the amount of memory used by strongSwan when there are no connections installed. To measure m we fill Victim's memory with a high number of connections at some given throughput, and divide the memory increase by the number of connections. There are several ways to measure the amount of memory used by a process. To be consistent with how we limit memory available to strongSwan (using cgroups), we take the value stored in strongSwan's cgroup file *memory.usage_in_bytes*.

We also measured σ during the experiment ($measured\ \sigma$) and observed that it was different from the σ we configured ($configured\ \sigma$). This is most probably due to virtualization. For the experimental verification of theorem 4.2.1 not to be affected by the fact that $configured\ \sigma$ and $measured\ \sigma$ are different, we use $measured\ \sigma$ to calculate the T^s_{mem} value predicted by theorem 4.2.1 ($Expected\ T^s_{mem}$).

For consistency we use a warm up run. Furthermore for each tuple (C, σ) we perform the measure of T^s_{mem} (resp. σ) 10 times. Finally, if the measures are consistent (i.e. if standard means are small) we take the average of T^s_{mem}'s (resp. σ's) measures as our value of $measured\ T^s_{mem}$ (resp. $measured\ \sigma$).

4.3.4 Results

We now provide and explain our experimental results, and produce some screenshots of the experiment.

Our measures of L and m confirm that L and m do not depend on C or σ. We obtain $L \approx 1\,\text{MB}$ and $m \approx 18.805\,\text{kB}$.

Figures 4.3, 4.4, and 4.5 respectively show time evolution of measured σ, memory used by strongSwan in Victim, and connection success indicator in an experiment where

C (in MB)	50	50	50	50	200	200	200	200
Configured σ (in init requests/s)	1	5	10	30	1	5	10	30
Measured σ (in init requests/s)	1.0	4.8	9.5	25.6	1.0	4.9	9.5	24.1
Expected T^s_{mem} (in s)	2605	541	274	101	10582	2159	1111	438
Measured T^s_{mem} (in s)	2617	547	280	103	10771	2202	1124	445
Relative error in %	0.5	1.1	2.1	2.4	1.8	2.0	1.1	1.6

Figure 4.2: Predicting and measuring DoS start time T^s_{mem} during some Deviation Attacks against strongSwan.

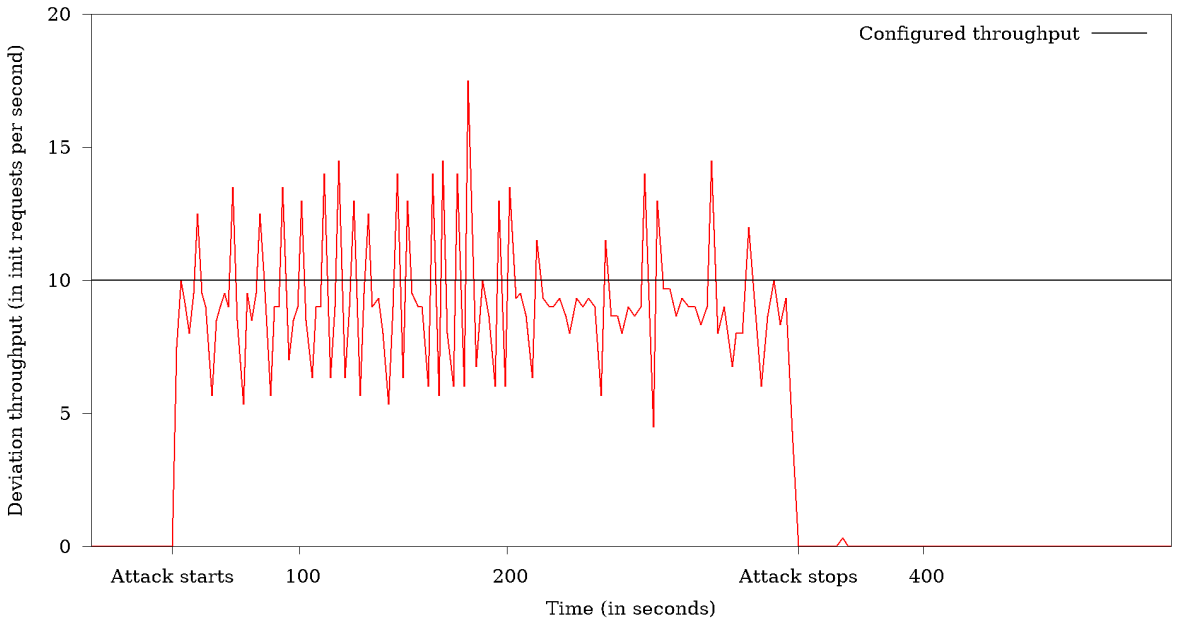


Figure 4.3: Evolution of *measured* σ through time in an experiment where $C = 50\,\text{MB}$ and *configured* σ = *10 init requests/s*. Deviation starts at $t \approx 30\,\text{s}$. Then *Measured* σ fluctuates around *configured* σ until memory exhaustion happens at $t \approx 300\,\text{s}$. Memory exhaustion stops the deviation because Initiators stop sending messages to Victim when they see no responses to their requests.

$C = 50\,\text{MB}$ and *configured* σ = *10 init requests/s*. The connection success indicator is set to 1 if the attempt of Probe to set up a connection with Victim 5 seconds ago has succeeded (see Section 4.3.2), and is set to 0 otherwise. This value thus indicates if there is a DoS or not. We see that as init requests are deviated, memory of Victim increases, showing that the Deviation Attack manages to set up connections in Victim. Theoretical memory consumption is computed using *configured* σ. Measured memory consumption is

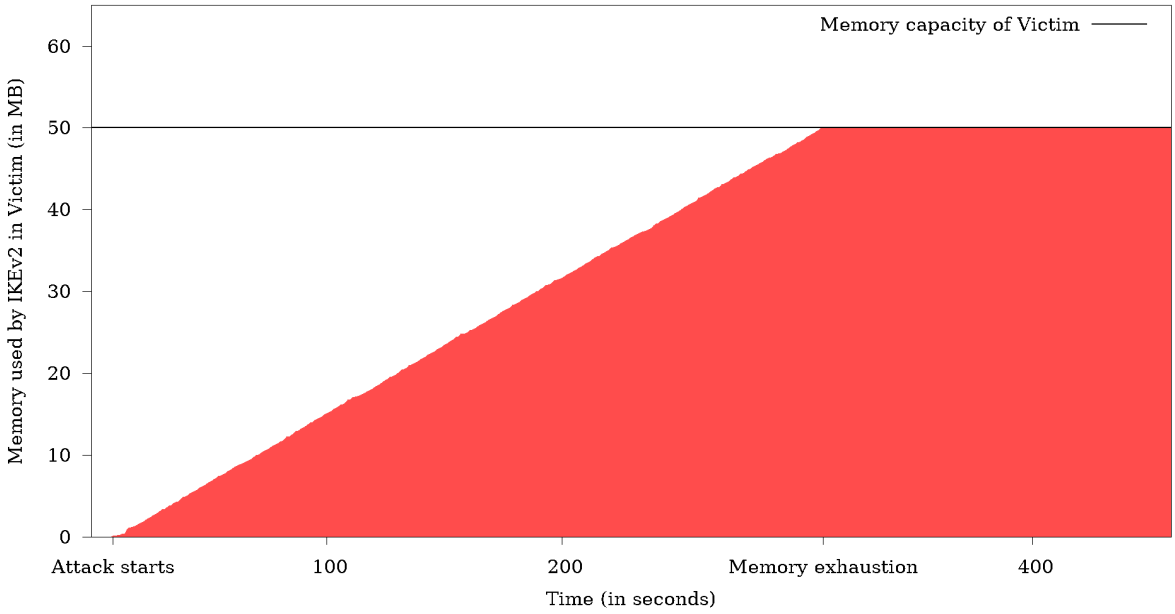


Figure 4.4: Evolution of the memory used by IKEv2 (strongSwan) in Victim through time in an experiment where $C = 50\,\text{MB}$ and *configured* $\sigma = 10$ *init requests/s*. Deviation starts at $t \approx 20\,\text{s}$. The memory used by strongSwan raises linearly until it reaches $C = 50\,\text{MB}$, causing memory exhaustion at $t \approx 300\,\text{s}$. Theoretical value is computed using *configured* σ. Measured memory consumption is lower than theoretical memory consumption because *measured* σ is lower than *configured* σ.

lower than theoretical memory consumption because *measured* σ is lower than *configured* σ. When memory of Victim reaches C, DoS begins.

Figure 4.2 shows the result of our measures of T^s_{mem}. For each tuple (C, σ), we obtain consistent measures (low standard deviations) of T^s_{mem} and σ. Our measures are close to the values predicted by theorem 4.2.1: we obtain an average relative error of 1.3% and a maximum relative error of 2.4%.

4.3.5 Discussion

We made some simplifications in our setup compared to the theoretical attack. For example, we did not instantiate all Initiators, which forced us to tweak the configuration of Victim (by using the `uniqueids` option, see Section 4.3.2). We could not instantiate many Initiators because we used Virtualbox VMs to represent them. To make the experiment more realistic, we could have instead used docker containers, as they are way lighter than Virtualbox VMs.

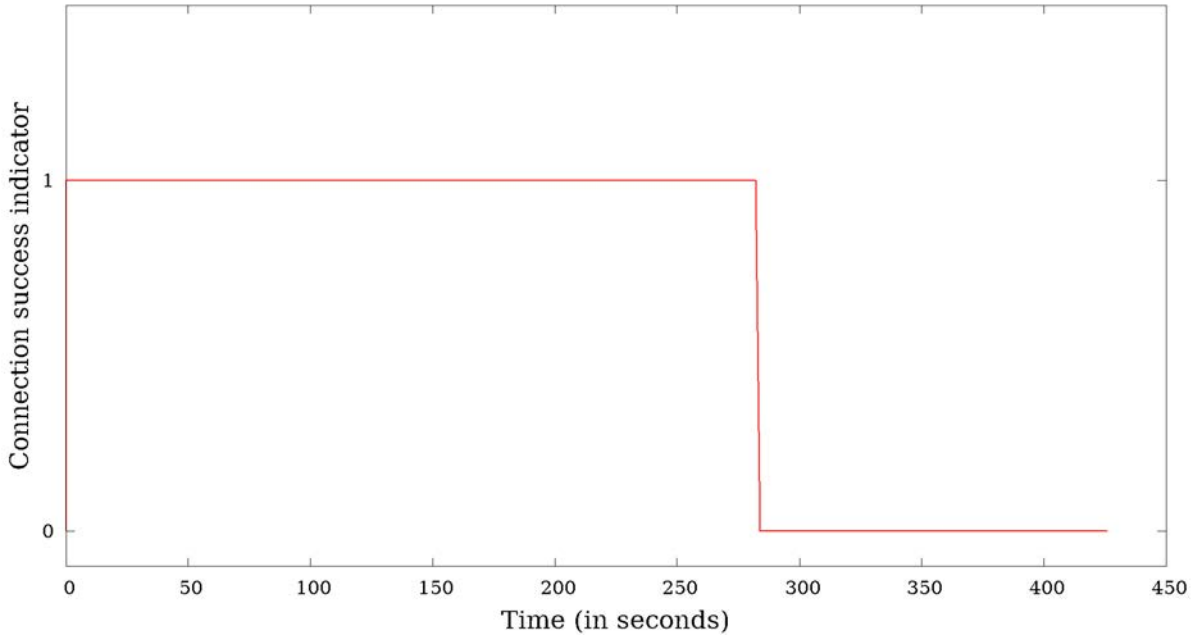


Figure 4.5: Evolution of the connection success indicator through time in an experiment where $C = 50\,\text{MB}$ and *configured* $\sigma = 10$ *init requests/s*. Connection success indicator is set to 1 when there is no DoS and set to 0 otherwise.

4.4 Countermeasures

In this Section, we explore countermeasures against the Deviation Attack. We start by considering existing defense mechanisms against DoS attacks, but find that they are absolutely pointless against the Deviation Attack. We then try to use a well-chosen set of parameters in an IKEv2 implementation, but we find either mitigations, or full countermeasures but that suffer from significant drawbacks. Hence we finally propose two inexpensive modifications of the protocol, and formally verify that these modifications prevent the Deviation Attack.

4.4.1 Existing defense mechanisms against DoS attacks

Two defense mechanisms against DoS attacks are already in place in IKEv2 (at least in IKEv2 specifications): cookies and puzzles. We introduced cookies and puzzles in Section 4.1.2. We show hereafter that these mechanisms are of no use against the Deviation Attack.

In the Deviation Attack, cookies will be handled by the Initiators and not by Intruder. Activating cookies thus has absolutely no effect on Intruder's memory requirements. It

does not increase the cost of the attack in terms of memory. Note that the cookie mechanism also increases the time between the reception of an init request by Victim and the filling of its memory with the new SA. But this increases neither the throughput of packets the attacker needs to deviate, nor the duration needed for the Deviation Attack to succeed.

Similarly, it is the Initiators who need to solve puzzles, not the Intruder. So puzzles, like cookies, are of no use against the Deviation Attack.

4.4.2 Using a well-chosen set of parameters

In this Section, we try to prevent the Deviation Attack by simply using a well-chosen set of parameters in an IKEv2 implementation. We note that Dead Peer Detection can be a small mitigation, and consider two countermeasures, which deter the attack but suffer from significant drawbacks: PSK authentication, and giving enough resources to Victim.

Using the default configuration of strongSwan, DPD multiplies by 18 (see Section 4.2.5) the number of init requests that Intruder needs to deviate. Therefore DPD is only a mitigation to the Deviation Attack.

As we pointed out in Section 4.2.4, the Deviation Attack is not possible when PSKs are used for authentication, provided that Victim is not part of the Initiator-Responder PSK community. Therefore using PSK is a countermeasure. However, PSKs and certificates do not fulfil the exact same needs. Hence, when digital signature is used, another countermeasure must be considered.

One solution to the memory exhaustion is to give enough memory capacity to Victim to handle as many connections as there can be. Let N_t be the number of peers that Victim trusts. Recall that m is the amount of memory used to store one connection. Victim would then need to be given a memory of at least $N_t \times m$. We measured $m = 23kB$ for strongSwan. If e.g. $N_t = 10^6$ (in an Internet of Things (IOT) context for example), 23GB of memory would be required in each potential victim to prevent the attack. This countermeasure is efficient, but may not be acceptable e.g. for embedded gateways.

4.4.3 Improving the protocol specification

Attempts to protect implementations of the current protocol are either not sufficient to deter the Deviation Attack, or present significant drawbacks. For this reason, we propose

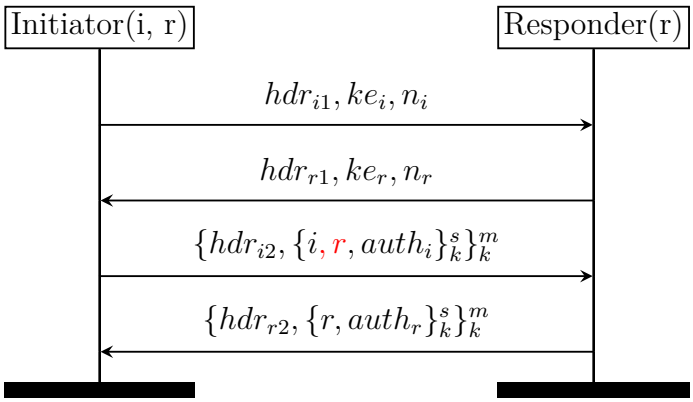


Figure 4.6: The IKEv2-Sig-IDr protocol. We make IDr (r in the Figure above) payload mandatory in the auth request and modify its processing by Responder.

two modifications of the protocol specification that remove its vulnerability to the Deviation Attack.

Using IDr payload

The first fix we propose uses the auth request IDr payload.

A way to fix the protocol is to make the IDr payload mandatory in auth request, and to modify its processing by the responder. The appropriate behaviour would be not to install a Child SA when IDr is not *acceptable* (defined below), but instead, to cancel the establishment of the IKE SA, by sending an AUTHENTICATION_FAILED notification to the Initiator. The message sequence chart of Figure 4.6 represents IKEv2-Sig with this IDr fix.

Let us define the term *acceptable* we used above. The IPsec and IKEv2 specifications call Peer Authorization Database (PAD) the database of an IKEv2 party that contains the peer IDs allowed to set up connections with that party. We propose that the specifications mandate a new field in the PAD. We call this field the *local ID* field. For consistency, we should also rename the existing *ID field* of the PAD, described in RFC 4301, Section 4.4.3, to *remote ID field*. The *local ID* field would be a non-empty list of IDs. Each ID would be in the same format as the current *ID field* (it can use wildcards, for example). When an auth request arrives, a PAD lookup is done. A PAD entry would match the request

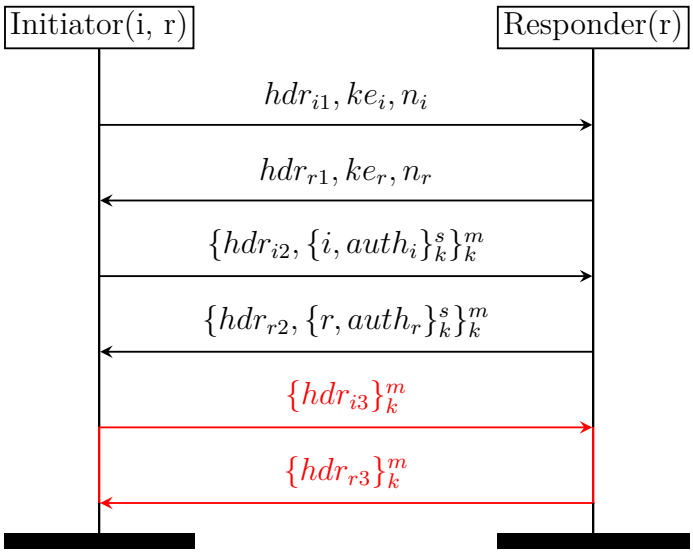


Figure 4.7: The IKEv2-Sig-conf protocol. We add the key confirmation exchange. Connections are installed only after this exchange.

when both the *remote ID field* and the *local ID field* match respectively the IDi and IDr payloads.

Note that, as we explain in Section 4.3.4, strongSwan already implemented the local ID field. It corresponds to the $leftid$ attribute of the ipsec.conf configuration file.

Sending an IDr payload in the auth request, however, has the important drawback of revealing the responder's identity to active attackers, as we explained in Section 4.3.1.

Adding key confirmation

Let us present a fix of the protocol that uses a third exchange called *key confirmation*.

We propose a way to fix IKEv2-Sig that does not require disclosure of the responder ID. We add a third exchange, called *key confirmation* and written KEY_CONF. The key confirmation exchange is made of a keep-alive request and a keep-alive response. Keep-alive messages are used in the context of Dead Peer Detection, as described in Section 4.2.5. A keep-alive message is an INFORMATIONAL message whose encrypted payload has an empty "Encrypted IKE Payloads" field (see [46]). Hence a MAC is computed, but no encryption is performed. The MSC of Figure 4.7 models IKEv2-Sig with key

confirmation in our formalism from Section 2.2.

We require that the responder install its connection only after having received a valid KEY_CONF request. However, the initiator may install its connection right after having received a valid auth response. Key confirmation has already been proposed in [70] as a countermeasure to the penultimate authentication flaw. Since the Deviation Attack exploits the latter, key confirmation is a countermeasure to the Deviation Attack.

Note that the KEY_CONF response is required for reliability purposes. In IKEv2, all messages, except for error messages, exist in pairs. This is because IKEv2 is carried over UDP, so the only way to be sure that a request has been received is to wait for a response message and to set up retransmissions in case it does not arrive (until a timeout). If the initiator does not receive the KEY_CONF response, then it will assume that its KEY_CONF request has been lost in the network, and will retransmit its KEY_CONF request a given number of times. The KEY_CONF response is only a network acknowledgment of the KEY_CONF request and plays no role in security.

Adding an exchange to the protocol can be seen as a increase of its cost, as it will take more time to establish a connection. But a KEY_CONF message requires only very little amount of time and computational power for its generation and processing, because there are no asymmetric cryptography or key derivation operations involved. Thus it is an efficient solution to prevent the Deviation Attack.

Verification

Let us now verify using model checking that our two modifications of the protocol do remove the penultimate authentication flaw.

To do so, we apply the modifications to our Tamarin models. We apply the countermeasures to IKEv2-PSK as well to verify that IKEv2-PSK does not loose any guarantee. We define four new subprotocols:

IKEv2-PSK-IDr consists of one init exchange and one auth exchange. It uses PSK authentication and the IDr countermeasure.

IKEv2-Sig-IDr consists of one init exchange and one auth exchange. It uses Signature authentication and the IDr countermeasure.

IKEv2-PSK-Conf consists of one init exchange, one auth exchange and one keyconf exchange. It uses PSK authentication and key confirmation.

IKEv2-Sig-Conf consists of one init exchange, one auth exchange and one keyconf exchange. It uses Signature authentication and key confirmation.

The modified Tamarin models can be found with our other Tamarin models at [68]. We find that our fixed versions of IKEv2-Sig and IKEv2-PSK guarantee all our properties in all our adversary models to both our roles: the current security properties of IKEv2 are preserved and the protocol does gains the stronger authentication properties needed to deter the Deviation Attack.

4.5 Concluding remarks

In this chapter, we have taken advantage of an authentication weakness of IKEv2 to create a new DoS attack against the protocol. In this attack, the intruder leverages legitimate requests to achieve a DoS using a lower bandwidth. This idea of leveraging legitimate requests is not protocol-dependent. In fact, it could be applied to any protocol that:

- is stateful, so that resources are consumed upon protocol completion,
- performs authentication, otherwise our attack is not better than an attacker that creates messages itself, and
- does not satisfy the weak agreement property, which states that if a party completes the protocol, then its partner did intend to complete the protocol with this party.

Examples of stateful authentication protocols are TLS, SSH. Further work is required to verify the weak agreement property on these protocols.

Further work should also check if other IKEv2 implementations are vulnerable to the Deviation Attack. If a software correctly implements IKEv2, then the software is likely to be vulnerable. However, we have seen that implementations do not always strictly follow the RFC. For example, strongSwan processes the r payload of the auth request differently than mandated, which makes strongSwan not vulnerable in some cases. Thus it would be interesting to try the attack on other IKEv2 implementations, such as Libreswan [49] and Openswan [64].

Finally, further work should try to use the Deviation Attack to cause CPU exhaustion. Indeed, when Victim's memory capacity is high, or even dynamically allocated, memory exhaustion is not possible but CPU exhaustion might be. To cause CPU exhaustion, further work should try a high value for C and higher values for σ.

Chapter 5

Conclusion

In this Chapter, we conclude this thesis. We summarize our contributions in Section 5.1, and draw perspectives for future work in Section 5.2.

5.1 Contributions

In this thesis, we have performed an extensive analysis of IKEv2 using symbolic verification, and deduced from a logic flaw of the protocol a new type of DoS attack that works against IKEv2. More precisely:

- We have extended the method of [8] for modeling security protocols in Spin (Section 3.3). We added to the method support for modular exponentiation, digital signature and MAC. We greatly simplified the adversary model, and added support for classic security properties (secrecy of a variable and Lowe's authentication properties).

- We have analyzed IKEv2 using our Spin method, ProVerif and Tamarin (Sections 3.3.4, 3.4 and 3.5). Our analysis showed that the reflection attack found against IKEv2-Child by [24] is not possible, due to the Initiator and Response flags of IKEv2. Moreover, we obtained new results on the security of IKEv2 *in the unbounded model*. We found that:

  - IKEv2-Sig satisfies injective agreement for the initiator, that
  - IKEv2-PSK satisfies injective agreement for both roles, and that
  - IKEv2-Child satisfies weak agreement for both roles and non-injective agreement for the responder.

  Finally, we performed a comparison between Spin, ProVerif and Tamarin (Section 3.6.1), whereby we emphasized that Spin fails to provide strong security guaranties due to its bounded nature.

- We have designed a new type of Denial-of-Service attack, which works against IKEv2: *the Deviation Attack* (Section 4.2). The Deviation Attack is a slow DoS attack that is harder to perform than classic DoS attacks, but that is also harder to detect. To concretely demonstrate the attack, we successfully implemented it (Section 4.3) against strongSwan, a popular open-source implementation of IKEv2. The source code to reproduce the Deviation Attack is available at `https://gitlab.inria.fr/tninet/demo`. We discussed the efficiency of the available means to protect implementations of the current protocol (Sections 4.4.1 and 4.4.2). However, none of them were complete solutions. Worse, the cookie and puzzle mechanisms that were introduced in IKEv2 to counter DoS attacks are completely ineffective against the Deviation Attack. We thus tackled the problem at a higher level: we proposed two possible inexpensive modifications of the protocol, and formally prove that they both prevent the attack (Section 4.4.3).

These works led to two publications in which I took part:

- [60] describes our extension of the method of [8] and its application to IKEv2.
- [61] introduces the Deviation Attack.

Hopefully, our work results in a better understanding of the security guaranties of IKEv2, and a better grasp upon the practicalities of the weak agreement property. In addition, our countermeasures to the Deviation Attack should improve the security of IKEv2, whose role is critical to some modern infrastructures.

5.2 Future work

Let us look at possible future work, going from immediate extensions of our work to broader research directions.

Firstly, symbolic verification of IKEv2 could be continued, following the pointers that we gave in Section 3.6.2. To sum these pointers up:

- In order to check if IKEv2-Child satisfies non-injective agreement for the initiator and injective agreement for both roles, we could model the counters of IKEv2 in Tamarin. Since our current model of IKEv2-Child does not include the counters and because we found (false) attacks on IKEv2-Child that we know are prevented

by counters in the RFC of IKEv2 and in its implementations, we currently have no proof that IKEv2-Child satisfies these properties. Including the counters in our model would certainly allow obtaining these proofs (or finding real attacks).

- We could model the protocol as a whole, instead of divided into subprotocols. IKEv2 is not a set of three protocols, as we modeled it in this thesis, but rather one protocol made of one IKEv2-Sig or IKEv2-PSK session followed by unboundedly many IKEv2-Child sessions, where all IKEv2-Child sessions use the same long-term key (the shared key generated by the IKEv2-Sig or IKEv2-PSK session). Our proofs would be much stronger if we managed to model IKEv2 as one whole protocol. Note that this can only be achieved if we have first modeled the counters, because having several IKEv2-Child sessions using the same long-term keys and without counters would yield trivial replay attacks.
- We could analyze IKEv2 in other adversary models, such as all those from [5]. Compared to the adversary models we have used in this thesis, these adversary models grant additional capabilities to the adversary. Knowing if IKEv2 is resilient to such adversaries would allow users of the protocol to know precisely which security guaranties they can expect from IKEv2.

Secondly, we have seen in Section 4.5 that even though the Deviation Attack is theoretically possible against any implementation of IKEv2, some implementations may not follow the specification faithfully enough and thus may not be vulnerable in practice. Hence, future work could try the Deviation Attack on other popular IKEv2 implementations than strongSwan for more exhaustiveness.

Thirdly, as we have seen in Section 4.5, this thesis outlines *the importance of the weak agreement property for authentication protocols*. Its violation does not necessarily imply a violation of secrecy, but we have shown that it can allow other attacks. In particular, when the protocol sets up some connection in the parties' memories, it can lead to a DoS attack. It could be interesting to verify weak agreement for TLS, SSH, and other stateful authentication protocols.

Fourthly, although we have analyzed the ability of the specification to meet its security goals, *this does not eliminate implementation-level flaws*. Buffer overflows are a well-known example of such flaw. They are described e.g. under the vulnerability identifier 787 in the Common Weakness Enumeration (CWE) [18]. In a buffer overflow, given specific

entries, a program writes beyond the intended bounds of a buffer, possibly overwriting important values such as the return pointer (in the case of a stack-based buffer overflow). In the case of communication protocols, entries are usually packets. If an attacker manages to overwrite the return pointer with its own value, then the attacker may be able to effectively redirect the execution flow to some shellcode or to the C library, thereby executing arbitrary code. As a consequence, further work must be performed to detect these flaws on current and future IKEv2 implementations.

Classical ways to detect implementation-level flaws include proper development flow and code review. More recently, the industry has adopted automated techniques such as fuzzing. Fuzzing executes a program with some input, reports any bug, changes the input (e.g. by flipping a bit), executes the program with the new input, and so on. Well-known fuzzing tools include [91] and [87]. Fuzzing cannot prove the absence of flaws in a program because there are generally too many possible inputs.

To prove the absence of implementation-level flaws, one could use formal software verification techniques. There exists many such techniques, which are implemented in tools such as CBMC [13], Angr [77], Astree [23], Klee [47], Frama-C [25] and VCC [26]. For example, CBMC takes some C code as input, inserts generic assertions that check for vulnerabilities such as buffer overflows, unwinds loop constructs up to a given bound, and inserts unwinding assertions. At runtime, an unwinding assertion placed in a loop is true if and only if execution has not unwinded the loop more than the given bound. CBMC then transforms the resulting code into a bit-vector equation whose satisfiablity is equivalent to the truth of all assertions at runtime, and gives the bit-vector equation to a Satisfiablity Modulo Theory (SMT) solver. Unfortunately, formally verifying a software, like formally verifying a security protocol specification, is a difficult task. Hence tools like CBMC do not yet scale to large programs. Static analysis, as implemented in Astree and Frama-C Value, has had slightly more success in industry.

Before tackling any future work, it is nonetheless worth keeping in mind that great caution is necessary when claiming that an IKEv2 implementation has been formally proved to be secure. Indeed, the model written when verifying the specification may be flawed, the tool used to verify the specification may be flawed, the source code may not strictly follow the specification, the source code may suffer from implementation-level flaws (as mentioned above), compilation may have introduced flaws as well, and finally, there may be hardware-related vulnerabilities (e.g. side-channel attacks).

Bibliography

[1] M. Abadi, B. Blanchet, and C. Fournet. "The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication". In: *Journal of ACM* (2017).

[2] M. Abadi and A. D. Gordon. "A Calculus for Cryptographic Protocols: The Spi Calculus". In: *Information and Computation* (1999).

[3] A. Armando and L. Compagna. "SATMC: A SAT-Based Model Checker for Security Protocols". In: *Logics in Artificial Intelligence*. 2004.

[4] C. Baier and J.-P. Katoen. *Principles of model checking*. 2008.

[5] D. Basin and C. Cremers. "Modeling and analyzing security in the presence of compromising adversaries". In: *European Symposium on Research in Computer Security*. 2010.

[6] D. Basin, C. Cremers, and C. Meadows. "Model checking security protocols". In: *Handbook of Model Checking* (2015). URL: `http://www-oldurls.inf.ethz.ch/personal/basin/pubs/security-modelchecking.pdf`.

[7] D. Basin, S. Mödersheim, and L. Viganò. "OFMC: A symbolic model checker for security protocols". In: *International Journal of Information Security* (2005).

[8] N. Ben Henda. "Generic and efficient attacker models in spin". In: *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. 2014.

[9] B. Blanchet. "An efficient cryptographic protocol verifier based on prolog rules". In: *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.* 2001.

[10] B. Blanchet. "Automatic Verification of Correspondences for Security Protocols". In: *J. Comput. Secur.* (2009).

[11] B. Blanchet. "CryptoVerif: Computationally sound mechanized prover for cryptographic protocols". In: *Dagstuhl seminar Formal Protocol Verification Applied*. 2007.

[12] E. Cambiaso, G. Papaleo, and M. Aiello. "SlowDroid: Turning a Smartphone into a Mobile Attack Vector". In: *2014 International Conference on Future Internet of Things and Cloud*. 2014.

[13] *CBMC: Bounded Model Checking for Software*. URL: https://www.cprover.org/cbmc/.

[14] I. Cervesato et al. "A meta-notation for protocol analysis". In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. 1999.

[15] V. Cheval, S. Kremer, and I. Rakotonirina. "DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[16] *Cisco AnyConnect Secure Mobility Client*. URL: https://www.cisco.com/c/en/us/products/security/anyconnect-secure-mobility-client/index.html.

[17] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. 1999.

[18] *Common Weakness Enumeration (CWE)*. URL: https://cwe.mitre.org/index.html.

[19] H. Comon-Lundh and V. Cortier. "Security Properties: Two Agents Are Sufficient". In: *Programming Languages and Systems*. 2003.

[20] H. Comon-Lundh and S. Delaune. "The Finite Variant Property: How to Get Rid of Some Algebraic Properties". In: *Term Rewriting and Applications*. 2005.

[21] H. Comon and V. Cortier. "Tree automata with one memory set constraints and cryptographic protocols". In: *Theoretical Computer Science* (2005).

[22] W. contributors. *ARP spoofing*. 2018. URL: https://en.wikipedia.org/wiki/ARP_spoofing.

[23] P. Cousot et al. "The ASTREÉ Analyzer". In: *Programming Languages and Systems*. 2005.

[24] C. Cremers. "Key exchange in IPsec revisited: Formal analysis of IKEv1 and IKEv2". In: *European Symposium on Research in Computer Security*. 2011.

[25] P. Cuoq et al. "Frama-C". In: *Software Engineering and Formal Methods*. 2012.

[26] M. Dahlweid et al. "VCC: Contract-based modular verification of concurrent C". In: *2009 31st International Conference on Software Engineering - Companion Volume*. 2009.

[27] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt.

[28] W. Diffie and M. Hellman. “New directions in cryptography”. In: *IEEE transactions on Information Theory* (1976).

[29] W. Diffie, P. C. Van Oorschot, and M. J. Wiener. “Authentication and authenticated key exchanges”. In: *Designs, Codes and Cryptography* (1992).

[30] D. Dolev and A. Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* (1983).

[31] S. Even and O. Goldreich. “On the security of multi-party ping-pong protocols”. In: *24th Annual Symposium on Foundations of Computer Science (SFCS 1983)*.

[32] *FreeS/WAN*. URL: `https://www.freeswan.org/`.

[33] D. Fu and J. Solinas. *Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2*. RFC 5903. 2010. URL: `http://www.rfc-editor.org/rfc/rfc5903.txt`.

[34] R. Gerth. *Concise Promela Reference*. 1997. URL: `http://spinroot.com/spin/Man/Quick.html`.

[35] S. Goldwasser and S. Micali. “Probabilistic encryption”. In: *Journal of Computer and System Sciences* (1984).

[36] C. G. Günther. “An Identity-Based Key-Exchange Protocol”. In: *Advances in Cryptology — EUROCRYPT ’89*. 1990.

[37] G. J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on Software Engineering* (1997).

[38] G. Huang, S. Beaulieu, and D. Rochefort. *A Traffic-Based Method of Detecting Dead Internet Key Exchange (IKE) Peers*. RFC 3706. 2004. URL: `http://www.rfc-editor.org/rfc/rfc3706.txt`.

[39] *ICSA Labs Certified Products*. URL: `https://www.icsalabs.com/products?tid[]=4218`.

[40] “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 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016).

[41] *IPsec at Fortinet*. URL: https://www.fortinet.com/products/vpn.html#models-specs.

[42] *IPsec in A10 Networks Thunder Convergent Firewall (CFW)*. URL: https://www.a10networks.com/products/firewall-vpn-secure-web-gateway#tab1.

[43] *IPsec in Microsoft*. URL: https://docs.microsoft.com/en-us/windows/security/identity-protection/vpn/vpn-connection-type.

[44] *IPsec tools*. URL: http://ipsec-tools.sourceforge.net/.

[45] J. Katz et al. *Handbook of applied cryptography*. 1996.

[46] C. Kaufman et al. *Internet key exchange protocol version 2 (IKEv2)*. RFC 7296. 2014. URL: http://www.rfc-editor.org/rfc/rfc8019.txt.

[47] *KLEE LLVM Execution Engine*. URL: https://klee.github.io/.

[48] R. Küsters and T. Truderung. "Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation". In: *22nd IEEE Computer Security Foundations Symposium*. 2009.

[49] *Libreswan*. URL: https://libreswan.org.

[50] *Linear Temporal Logic reference in Spin*. 2017. URL: http://spinroot.com/spin/Man/ltl.html.

[51] G. Lowe. "A hierarchy of authentication specifications". In: *Proceedings 10th Computer Security Foundations Workshop*. 1997.

[52] G. Lowe. "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 1996.

[53] S. Mauw and V. Bos. "Drawing Message Sequence Charts with LaTeX". In: *TUGBoat* (2001).

[54] C. Meadows. "Analysis of the Internet Key Exchange protocol using the NRL Protocol Analyzer". In: *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. 1999.

[55] S. Meier et al. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification*. 2013.

[56] R. Milner. *Communicating and mobile systems: the pi calculus*. 1999.

[57] R. Milner, J. Parrow, and D. Walker. “A calculus of mobile processes, I”. In: *Information and Computation* (1992).

[58] J. C. Mitchell, M. Mitchell, and U. Stern. “Automated analysis of cryptographic protocols using Mur/spl phi/”. In: *IEEE Symposium on Security and Privacy*. 1997.

[59] K. Moriarty et al. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. 2016. URL: http://www.rfc-editor.org/rfc/rfc8017.txt.

[60] T. Ninet et al. “Model Checking the IKEv2 Protocol Using Spin”. In: *17th International Conference on Privacy, Security and Trust (PST)*. 2019.

[61] T. Ninet et al. “The Deviation Attack: A Novel Denial-of-Service Attack Against IKEv2”. In: *18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2019.

[62] Y. Nir and V. Smyslov. *Protecting Internet Key Exchange Protocol Version 2 (IKEv2) Implementations from Distributed Denial-of-Service Attacks*. RFC 8019. 2016. URL: http://www.rfc-editor.org/rfc/rfc8019.txt.

[63] *OpenBSD IKEd*. URL: http://man.openbsd.org/OpenBSD-current/man8/iked.8.

[64] *Openswan*. URL: https://www.openswan.org/.

[65] *Our demonstration of the Deviation Attack*. URL: https://gitlab.inria.fr/tninet/demo.

[66] *Our Promela model of IKEv2*. URL: https://gitlab.inria.fr/tninet/spin.

[67] *Our ProVerif model of IKEv2*. URL: https://gitlab.inria.fr/tninet/proverif.

[68] *Our Tamarin model of IKEv2*. URL: https://gitlab.inria.fr/tninet/tamarin.

[69] R. Perlman. “Understanding IKEv2: Tutorial, and rationale for decisions”. In: *RFC Editor* (2003).

[70] A. Project. *Deliverable D6.2: Specification of the Problems in the High-Level Specification Language*. Tech. rep. 2003. URL: http://www.avispa-project.org/.

[71] *Promela semantics*. URL: http://spinroot.com/spin/Man/Intro.html.

[72] *RACOON 2*. URL: http://www.racoon2.wide.ad.jp/w/?News.

[73] R. L. Rivest et al. “The RC6 Block Cipher”. In: *First Advanced Encryption Standard (AES) Conference*. 1998.

[74] M. Rusinowitch and M. Turuani. “Protocol insecurity with a finite number of sessions and composed keys is NP-complete”. In: *Theoretical Computer Science* (2003).

[75] T. C. Ruys. “Low-fat recipes for SPIN”. In: *International SPIN Workshop on Model Checking of Software*. 2000.

[76] K. S. and S. K. *Security Architecture for the Internet Protocol*. RFC 4301. 2005. URL: https://www.rfc-editor.org/rfc/rfc4301.txt.

[77] Y. Shoshitaishvili et al. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy*. 2016.

[78] *Size of the Srizbi botnet*. URL: http://news.bbc.co.uk/2/hi/technology/7749835.stm.

[79] *SlowLoris*. URL: https://github.com/gkbrk/slowloris.

[80] D. Song. *ARPspoof software*. URL: https://linux.die.net/man/8/arpspoof.

[81] N. I. of Standards and Technology. *Advanced Encryption Standard (AES)*. Standard. 2001.

[82] N. I. of Standards and Technology. *Digital Signature Standard (DSS)*. Standard. 2013.

[83] *StrongSwan*. URL: https://www.strongswan.org/.

[84] *StrongSwan distribution packages*. URL: https://wiki.strongswan.org/projects/strongswan/wiki/InstallationDocumentation#Distribution-packages.

[85] *Swan softwares history*. URL: https://libreswan.org/wiki/History.

[86] *The Srizbi botnet*. URL: https://www.symantec.com/security-center/writeup/2007-062007-0946-99.

[87] *The ZZUF fuzzer*. URL: http://caca.zoy.org/wiki/zzuf.

[88] M. Turuani. “The CL-Atse Protocol Analyser”. In: *Term Rewriting and Applications*. 2006.

[89] *Tutorial to crack WPA*. URL: https://www.aircrack-ng.org/doku.php?id=cracking_wpa.

[90] R. Wang et al. “How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores”. In: *2011 IEEE Symposium on Security and Privacy*. 2011.

[91] M. Zalewski. *American fuzzy lop*. URL: http://lcamtuf.coredump.cx/afl/.

UNIVERSITÉ DE RENNES 1

Titre : Vérification formelle du protocole d'échange de clé IKEv2

Mots-clés : Protocole cryptographique, Vérification formelle, IKEv2, Déni de service

Résumé : Dans cette thèse, nous analysons le protocole IKEv2 à l'aide de trois outils de vérification formelle : Spin, ProVerif et Tamarin. Pour effectuer l'analyse avec Spin, nous étendons une méthode existante de modélisation. En particulier, nous proposons un modèle de la signature numérique, du MAC et de l'exponentiation modulaire, nous simplifions le modèle d'adversaire pour le rendre applicable à des protocoles complexes, et nous proposons des modèles de propriétés d'authentification. Nos analyses montrent que l'attaque par réflexion, une attaque trouvée par une précédente analyse, n'existe pas. De plus, nos analyses avec ProVerif et Tamarin produisent de nouvelles preuves concernant les garanties d'accord non injectif et d'accord injectif pour IKEv2 dans le modèle non borné.

Nous montrons ensuite que la faille de pénultième authentification, une vulnérabilité considérée comme bénigne par les analyses précédentes, permet en fait d'effectuer un nouveau type d'attaque par déni de service auquel IKEv2 est vulnérable : l'*Attaque par Déviation*. Afin de démontrer concrètement sa faisabilité, nous attaquons avec succès une implémentation open-source populaire de IKEv2. Les contre-mesures classiques aux attaques DoS ne permettent pas d'éviter cette attaque. Nous proposons alors deux modifications simples du protocole, et prouvons formellement que chacune d'entre elles empêche l'Attaque par Déviation.

Title : Formal verification of the Internet Key Exchange (IKEv2) security protocol

Keywords: Cryptographic protocol, Formal verification, IKEv2, Denial-of-Service

Abstract: In this thesis, we analyze the IKEv2 protocol specification using three formal verification tools: Spin, ProVerif and Tamarin. To perform the analysis with Spin, we extend and improve an existing modeling method with a simpler adversary model and a model for common cryptographic primitives and Lowe's authentication properties. As a result we show that the reflection attack, an attack found by a previous analysis, is actually not applicable. Moreover, our analysis using ProVerif and Tamarin provides new results regarding non-injective agreement and injective agreement guaranties of IKEv2 in the unbounded model.

We then show that the penultimate authentication flaw, a vulnerability that was considered harmless by previous analyses, actually allows for a new type of Denial-of-Service attack, which works against IKEv2: *the Deviation Attack*. To concretely demonstrate the attack, we successfully implement it against a popular open-source implementation of IKEv2. Finally, we study the use of existing DoS countermeasures and existing configuration options to defeat the attack, but we only find mitigations or incomplete workarounds. We therefore tackle the problem at a higher level: we propose two possible inexpensive modifications of the protocol, and formally prove that they both prevent the attack.