UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Antonio Lemos Maia Neto

Authentication of Things: Authentication and Access Control for the Entire IoT Device Life-Cycle

Belo Horizonte
2023

Antonio Lemos Maia Neto

Authentication of Things: Authentication and Access Control for the Entire IoT Device Life-Cycle

Versão Final

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Orientador: Leonardo Barbosa e Oliveira
Coorientador: Ítalo Fernando Scotá Cunha

Belo Horizonte
2023

Maia Neto, Antonio Lemos.
.

M217a AOT: Authentication of Things: authentication and access control for the entire IoT device life-cycle [manuscrito] / Antonio Lemos Maia Neto – 2023.
1 recurso online (208 f. il, color.)

Orientador: Leonardo Barbosa e Oliveira.
Coorientador: Ítalo Fernando Scotá Cunha.

Tese (doutorado) – Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.198 -208.

1. Computação - Teses. 2, Internet das coisas - Teses. 3. Criptografia de dados (Computação) - Teses. 4. Controle de acesso - Teses. I.Oliveira, Leonardo Barbosa e. II. Cunha, Ítalo Fernando Scotá. III.Universidade Federal de Minas Gerais; Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV.Título.

CDU 519.6*43.(043)

Ficha catalográfica elaborada pela bibliotecária do Instituto de Ciências Exatas
UFMG Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

AOT: AUTHENTICATION OF THINGS - AUTHENTICATION AND ACCESS CONTROL FOR THE ENTIRE IOT DEVICE LIFE-CYCLE

ANTONIO LEMOS MAIA NETO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores(a):

Belo Horizonte, 27 de abril de 2022.

Documento assinado eletronicamente por **Hilder Vitor Lima Pereira**, **Usuário Externo**, em 18/05/2022, às 07:45, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcos Antonio Simplicio Junior**, **Usuário Externo**, em 19/05/2022, às 00:54, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Michele Nogueira Lima**, **Professora do Magistério Superior**, em 19/05/2022, às 08:56, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Antonio Alfredo Ferreira Loureiro**, **Professor do Magistério Superior**, em 23/05/2022, às 12:39, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marco Aurelio Amaral Henriques**, **Usuário Externo**, em 30/05/2022, às 12:48, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1402960** e o código CRC **36702519**.

À Marina e Tom.

Acknowledgments

I apologize to English readers, but I must express my gratitude in Portuguese.

Agradeço primeiramente a Deus, pelas bênçãos que derrama diariamente em minha vida, inclusive com a oportunidade de realizar e concluir este trabalho.

Agradeço imensamente à minha esposa Marina, que dividiu grandes momentos de vitória durante todo esse caminhar, mas, especialmente, não arredou o pé e me deu todo carinho, amor e confiança pra seguir adiante nos momentos de provação. Ao meu filho Tom, que era apenas um plano pra depois que este trabalho terminasse. E assim o foi. Mas veja como são as coisas, houve tempo pra lhe agradecer, meu filho. A idéia de que você viria, sempre foi inspiração pra concluir este trabalho. Também agradeço ao Waguinho, a Dona Haydeé, a Lyd, o meu afilhado Cadu, a Bilinha, o Fábio, a Duda e a Maria Alice, que nos apoiaram em todas as etapas dessa jornada.

O meu muito obrigado a meus pais Rui e Toninha, que desconheço generosidade e amor maiores. Muito obrigado pelo tremendo esforço em me dar oportunidade de estudar, de trilhar o meu caminho e por me apoiar em cada decisão que me trouxe até aqui. Agradeço também à minha irmã Lella, meu cunhado e irmão Gabriel e meus afilhados Ana Clara e Bento, família que foi fonte de alegria e de alívio em momentos muito difíceis nessa caminhada. À toda minha família que amo de todo o coração, muito obrigado.

Agradeço especialmente o meu grande amigo, mentor e orientador Leo, por ter me dado tantas oportunidades ao longo de uma também longa caminhada, por ter confiado no meu trabalho, por ter sido sempre tão paciente e insistente. Também agradeço o meu coorientador Ítalo, pela grande parceria, pelos conselhos e por horas e milhares de linhas de artigos revisadas. Não poderia me esquecer de agradecer o professor Antonio Alfredo, por ter sido uma pessoa tão inspiradora e acessível.

Agradeço a todos os membros das bancas de qualificação e de defesa de tese, pela disponibilidade e por tão valiosas contribuições. Um agradecimento especial a todos os funcionários do DCC/UFMG, particularmente à Sonia, Linda, Gilmara e Sheila.

Agradeço as agências de fomento à pesquisa brasileiras, CNPq e CAPES e à LG pelo auxílio financeiro. Enfim, agradeço a todos que, de alguma forma, contribuiram com a conclusão deste trabalho.

Resumo

A Internet das Coisas (*Internet of Things* – IoT) pode ser vista como a presença pervasiva de objetos físicos ou "coisas" que, embarcadas com capacidade de computação, armazenamento e comunicação de dados, interagem umas com as outras e com outras entidades computacionais tradicionais, tais como computação móvel em nuvem, para cooperativamente prover serviços do dia-a-dia para usuários em um contexto específico, habilitando os chamados ambientes inteligentes.

Ambientes inteligentes são, de fato, parte de nossas vidas. O número de dispositivos de IoT conectados cresce mais rápido que a população e que o número de usuários na Internet, o que aumenta a necessidade de mecanismos robustos de autenticação e controle de acesso que garantam a segurança em ambientes de tecnologia tão heterogêneos.

Devido a essa alta heterogeneidade de IoT, os esquemas de autenticação tradicionais baseados em Infraestrutura de Chave Pública e certificados digitais são inadequados à maioria dos dispositivos de IoT, que não tem recursos computacionais suficientes para executá-los. A maioria das propostas de autenticação e controle de acesso que tem como alvo dispositivos com recursos limitados, por sua vez, normalmente baseiam seu mecanismo de controle de acesso apenas na autenticação, estratégia que também é conhecida como abordagem tudo ou nada. No entanto, em um ambiente inteligente complexo, os dispositivos IoT geralmente oferecem uma variedade de recursos que, então, exigem um mecanismo de controle de acesso com permissões mais granularizadas. Por outro lado, as propostas que abordam o controle de acesso mais refinado para dispositivos com restrição de recursos em IoT geralmente delegam a decisão de controle de acesso a uma entidade externa confiável, o que cria uma dependência de terceiros na comunicação entre dois dispositivos, que é afetada nos casos de instabilidade ou indisponibilidade no serviço de autorização.

Além de demandar segurança dentro desses ambientes de tecnologia tão diversificados, o paradigma IoT também acena para a interoperabilidade segura e perfeita entre dispositivos que pertençam a diferentes ambientes inteligentes. Por último, há necessidade de soluções de autenticação e controle de acesso que cubram todo o ciclo de vida do dispositivo IoT, ou seja, desde a fabricação do dispositivo até seu descomissionamento.

Neste trabalho, propomos *A*uthentication of Things (AoT) (Autenticação das Coisas), uma solução holística de autenticação e controle de acesso granular para todo o ciclo de vida de dispositivos de IoT. O AoT tem como alvo a natureza altamente heterogênea e interoperável dos ambientes inteligentes de IoT, onde os dispositivos de IoT: (i) operam uns aos outros em um domínio de confiança local, onde as operações exigem permissões de controle de acesso granulares; (ii) não tem qualquer dependência de terceiros durante os processos de autenticação e controle de acesso; (iii) podem operar como dispositivos convidados em um domínio estrangeiro, ou seja, não originalmente seu domínio local de confiança; e (iv) interagem com um servidor remoto em um domínio de confiança do fabricante, que representa a relação de confiança entre os dispositivos e seu fabricante durante seu ciclo de vida.

O AoT tem como alicerces Criptografia Baseada em Identidade (IdentityBased Cryptography – IBC) para distribuir as chaves e autenticar os dispositivos e Criptografia Baseada em Atributos (Attribute-Based Cryptography – ABC) para garantir criptograficamente um esquema de Controle de Acesso Baseado em Atributos (Attribute Based Access Control – ABAC).

Nós projetamos o AoT como uma composição de protocolos e primitivas criptográficas, portanto, nós baseamos a modelagem e a análise de segurança do AoT sobre o paradigma da Composibilidade Universal. Nós avaliamos, para diferentes níveis de segurança, os requisitos computacionais de um protótipo do AoT implementado sobre uma variedade de plataformas, representando uma ampla gama de dispositivos de IoT, desde representantes de smartphones a microcontroladores que podem ser usados em aparelhos de baixo custo. Nossos resultados indicam que o desempenho do AoT varia de acessível em recursos limitados, eficiente em plataformas intermediárias e muito eficiente em dispositivos poderosos.

Palavras-chave: Internet das Coisas, Controle de Acesso, Autenticação, Segurança, Criptografia Baseada em Identidade, Criptografia Baseada em Atributos.

Abstract

The Internet of Things (IoT) can be seen as the pervasive presence of physical objects or "things" that, embedded with computing, storage, and communication capabilities, interact among each other and with other traditional computational entities, such as mobile and cloud computing, to cooperatively provide everyday services for users in a specific context, enabling the so-called smart environments.

Smart environments are, in fact, part of our daily lives. The number of IoT-connected devices grows faster than both population and Internet users, which increases the need for strong authentication and access control mechanisms to guarantee security in such heterogeneous technology environments. Due to this high heterogeneity of IoT, traditional authentication schemes based on Public Key Infrastructure (PKI) and certificates are expensive and most IoT devices cannot afford to run them. Most proposals targeting such resource-constrained devices typically base their access control mechanism solely on authentication, which is also known as the all-or-nothing approach. However, in a complex smart environment, IoT devices often offer a range of resources that require different permissions rights, which, in turn, demands a fine-grained access control mechanism. On the other hand, proposals that address fine-grained access control for resource-constrained devices in IoT usually delegate the access control decision to an external trusted entity, which creates a third-party dependency on device-to-device communication and it is impacted in cases of instability or unavailability on the authorization service.

Besides demanding security in these diverse technology environments, the IoT paradigm also beckons safe and seamless interoperability among devices that belong to different smart environments. Last, there is a lack of options for authentication and access control solutions that cover the entire IoT device life-cycle, i.e., from device manufacturing to decommissioning.

In this work, we propose Authentication of Things (AoT), a holistic tailor-made authentication and fine-grained access control solution for the entire IoT device life-cycle. AoT targets the highly heterogeneous and interoperable nature of IoT smart environments, where IoT devices: (i) operate each other in a local domain of trust, where the operations demand fine-grained access control permissions; (ii) do not have any dependency on third parties during the authentication and access control processes; (iii) can operate as guest devices in a foreign domain, i.e., not originally their local domain of trust; and (iv) interact with a remote server in a manufacturer domain of trust, which represents the trust relationship between the devices and their manufacturer during their life-cycle.

In order to accomplish our goals, AoT protocols relies on Identity-Based Cryptography (IBC) to distribute keys and authenticate devices as well as Attribute-Based Cryptography

(ABC) to cryptographically enforce a fine-grained Attribute-Based Access Control (ABAC). We design AoT as a composition of cryptographic protocols and primitives, therefore, we base its modeling and security analysis under the Universal Composability paradigm. We evaluate the requirements, at different security levels, of an AoT prototype implemented on a variety of platforms, representing a wide range of IoT devices, from representative smartphones and microcontrollers that could be used on low-end appliances. Our results indicate AoT performance ranges from affordable on resource-constrained devices, efficient on intermediate devices, and highly efficient on powerful devices.

Keywords: Internet of Things, Access Control, Authentication, Security, Identity-Based Cryptography, Attribute-Based Cryptography.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The Internet of Things (IoT) [6, 7] perhaps represents nowadays the concept that comes closest to Mark Weiser's ubiquitous computing envisioning presented in 1991 [119]. IoT can be seen as the pervasive presence of physical objects or "things" that, embedded with computing, storage, and communication capabilities, interact among each other and with other traditional computational entities, such as mobile and cloud computing, to cooperatively provide everyday services for users in a specific context, enabling the so-called smart environments, e.g., smart houses, offices, manufacturing, and cities.

Smart environments are, in fact, part of our daily lives. The number of IoT-connected devices grows faster than both population and Internet users[1], which increases the need for strong authentication and access control (authorization) mechanisms to guarantee security in such heterogeneous networks [110]. Besides demanding security inside these diverse technology environments, the IoT paradigm also beckons safe and seamless interoperability among devices that belong to different smart environments, e.g., a guest device in a smart house might want access to smart appliances' operations available for visitors, while a foreign citizen needs to consume the digital services a smart city makes available for tourists.

Due to the highly heterogeneous nature of IoT, ranging from smartphones to smart motion sensors in a smart house context, for instance, traditional authentication schemes based on Public Key Infrastructure (PKI) and certificates, which carry significant processing, memory, storage, communication, and management overheads, are deemed unfit for the devices on the lower end of this range, the resource-constrained IoT devices [109]. Several authentication schemes for IoT especially targeting resource-constrained devices have been proposed as a solution to this problem [85, 86, 91, 98, 99, 108]. Albeit authentication differs in concept and purpose from access control, they are indeed closely related security subjects. Notably, most of these proposals on security that target resource-constrained devices typically base their access control mechanism solely on authentication, which is also known as the all-or-nothing approach, i.e., once authenticated, the entity has full access to any resource on the destination [128]. However, in IoT architectures such as smart environments, IoT devices often offer a range of resources that require different permissions rights, e.g., a kitchen smart appliance

[1] `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`

has few operations that are available to be safely executed by the kids in a smart house, which demands a fine-grained access control mechanism.

In fact, there are a variety of works that propose alternatives for fine-grained access control for resource-constrained devices in IoT [30, 35, 39, 56, 76, 105]. The existing approaches usually delegate the access control decision to an external trusted entity, taking such a decision out of the IoT device. On the one hand, the access control processing burden is removed from the resource-constrained device. On the other hand, it creates a third-party dependency on a supposed direct device-to-device operation, which impacts user experience in cases of instability or unavailability on the authorization service.

Although the enormous attention authentication and access control for IoT has received from the research community, there is scant literature covering the entire IoT device life-cycle [124], i.e., from the beginning-of-life, when the device is manufactured then deployed in a smart environment, passing through the middle-of-life, when it communicates not only with other devices in its smart context but also with its manufacturer to potentially be updated, and, last, the end-of-life, when it is disposed [69]. The majority of the authentication and access control proposals for IoT on the literature, even without being explicit, approach the middle-of-life of IoT devices, i.e, when the device is cooperatively providing its operations in its smart environment domain. However, neither a trusted relationship with the manufacturer nor a secure decommissioning process for end life are usually contemplated. The lack of the former has a high probability of leaving IoT devices out to date, potentially vulnerable to security threats, which, in turn, might lead to devastating consequences since it puts all devices in the smart environment connected to them at risk [1]. The absence of the latter, on the other hand, gives adversaries access to consumers' personal information and to the cryptographic material from their trusted environments, which might be used to abuse the trusted relationships, also becoming a potential threat to the other devices in the smart environment [57].

1.1 Goal

In this work, we aim at designing, developing, and evaluating a holistic authentication and fine-grained access control solution for IoT. Our solution, *Authentication of Things* (AoT), provides authentication and access control to all stages in an IoT device's life-cycle (Figure 1.1), in particular: pre-deployment, ordering, deployment, functioning, and retirement. AoT targets the highly heterogeneous and interoperable nature of IoT smart environments, where IoT devices: (i) operate each other in what we call a local domain of trust, where the operations demand fine-grained access control permissions; (ii) do not have any dependency on third parties during the authentication and access control processes; (iii) can operate as guest devices in a foreign domain, i.e., not originally their local domain of trust; and (iv) interact with a remote server in a manufacturer domain of trust, which represents the trust relationship between the devices and their manufacturer during their life-cycle.

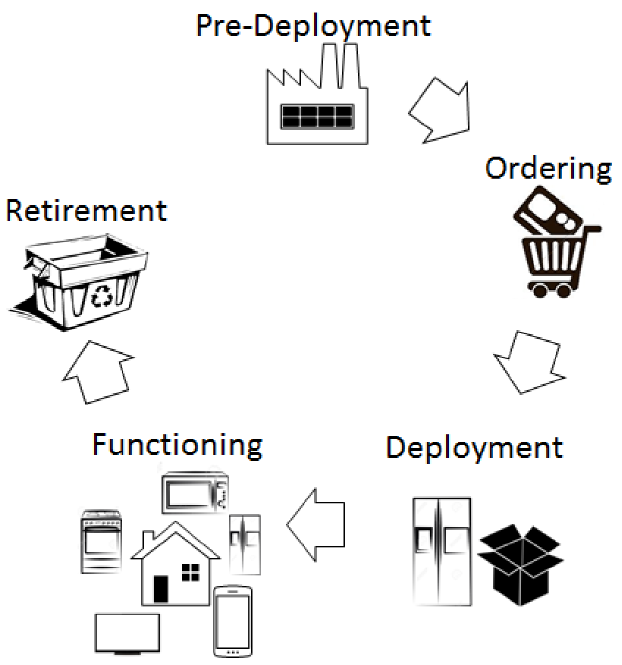


Figure 1.1: Entire IoT device life-cycle.

1.2 Approach

In order to accomplish our goals, AoT protocols relies on Identity-Based Cryptography (e.g., [19, 31, 102, 106]) to distribute keys and authenticate devices as well as Attribute-Based Cryptography (e.g., [13, 44]) to cryptographically enforce a fine-grained Attribute-Based Access Control [50, 125]. We chose these cryptosystems because they are certificate-free and thus do not impose certificate-related overheads on devices. Our *key insight* is to tackle the well-known key escrow problem [28] of identity-based schemes designing a two-domain architecture composed of a manufacturer domain and a local domain. These domains manage manufacturer-to-device and local device-to-device trust relationships, respectively.

We design AoT as a composition of cryptographic protocols and primitives, therefore, we base its modeling and security analysis under the Universal Composability paradigm [23, 49, 61]. In this context, we extend a specific functionality [62] to support a set of cryptographic primitives which, in turn, supports the analysis of the identity-based authenticated key agreement protocols categorized into the same family of protocols proposed by [81] under the Universal Composability paradigm.

We implement an AoT prototype, at different security levels, on different platforms, varying computational resources, representing a wide range of IoT devices. We use a recently launched Android mobile phone, a Google Pixel 6, as a representative of smartphones that could be used in a smart environment supported by AoT. Other powerful entities in a smart environment are represented by a Raspberry Pi3, a low-cost programmable computer. We represent intermediate smart devices with Raspberry Pi1. Last, as our representative of microcontrollers that could be used on low-end appliances supporting AoT, we use an Arduino Due. We use our prototype to quantify CPU, memory, storage, and communication overheads imposed by AoT protocols. Our results indicate AoT performance ranges from affordable on resource-constrained devices like the Arduino Due to efficient on intermediate devices like the Raspberry Pi1, and negligible on powerful devices like the Raspberry Pi3 and on smartphones like the Google Pixel 6.

1.3 Contributions

The majors contributions of our work are summarized as follows.

- An authentication and access control solution that cover all the stages in an IoT device life-cycle, i.e., from device manufacturing to decommissioning.
- A fine-grained Attribute-Based Access Control mechanism cryptographically enforced by Attribute-Based Cryptography.
- An extension of a functionality [62] in the Universal Composability framework which ends up supporting the analysis of identity-based authenticated key agreement protocols [81].

We also have the following minor contributions:

- A two-domain architecture that allows separated device-to-manufacturer and device-to-device trust relationships during the IoT device life-cycle.
- A protocol for device ownership reassignment and a protocol for authentication and access control between devices from different domains of trust.
- Device-to-device authentication and access control processes' decision without any delegation to third parties.

- Modeling and security analysis of AoT protocols under the Universal Composability paradigm.
- Experimental evaluation of AoT protocols based on real prototypes and quantification of various performance metrics on diverse hardware at different security levels.
- To the best of our knowledge, we provide the first ABS experimental evaluation results for powerful, intermediate, and resource-constrained devices.

1.4 Scientific Production & Awards

- Neto, A. L. M., Souza, A.L., Cunha, I., Nogueira, M., Nunes, I. O., Cotta, L., Loureiro, A. A. F., Aranha, D. F., Oliveira, L. B. (2016). AdC: um Mecanismo de Controle de Acesso para o Ciclo de Vida das Coisas Inteligentes. *Brazilian Symposium on Information and Computer System Security (SBSeg).*
- Neto, A. L. M., Souza, A.L., Cunha, I., Nogueira, M., Nunes, I. O., Cotta, L., Loureiro, A. A. F., Aranha, D. F., Oliveira, L. B. (2016). AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle. *ACM Conference on Embedded Network Sensor Systems (SenSys).*
- Neto, A. L. M., Pereira, Y. L., Souza, A. L., Cunha, I., Oliveira, L. B. (2018). Attribute-based authentication and access control for IoT home devices. *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN).*
- Oliveira, L. B. E., Loureiro, A. A. F., Neto, A. L. M. (2019). U.S. Patent No. 10,523,437. *Washington, DC: U.S. Patent and Trademark Office. United States Patent Application Publication US 2017/0214529 A1.*
- Neto, A. L. M., Oliveira, L. B. (2019). AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle. *Google Latin America Research Awards.*
- Neto, A. M., Richardson, S., Horowitz, M., Oliveira, L. (2019). Aceleração de Assinaturas baseadas em Atributos para Internet das Coisas. *Brazilian Symposium on Information and Computer System Security (SBSeg).* Honorable Mention in the Best Paper Award.
- Neto, A. L. M., Cunha, I., Oliveira, L. B. (2021). Uma Extensão de Framework de Análise de Protocolos de Composibilidade Universal para Acordo de Chaves com Aut-

enticação Baseado em Identidade. *Brazilian Symposium on Information and Computer System Security (SBSeg).*

- Neto, A. L. M., Cunha, I., Oliveira, L. B. (2021). Short Talk - Análise Modular de Segurança de Protocolos Fundamentados em Criptografia Baseada em Identidade. *Digital Identity Management Workshhop - Brazilian Symposium on Information and Computer System Security (SBSeg).*
- Neto, A. L. M., Cunha, I., Oliveira, L. B. (2022). Authentication of Things: Authentication and Access Control for the Entire IoT Device Life-Cycle. *Best Doctoral Thesis Award in the Thesis and Dissertation Competition of the Brazilian Symposium on Information and Computational Systems Security (SBSeg).*

1.5 Organization

The remainder of this work is structured as follows. Chapter 2 introduces the fundamentals of AoT's design. We discuss related work in Chapter 3. Chapter 4 presents AoT protocols and the assumptions under which it has been designed. In Chapter 5 we perform the security analysis of AoT protocols. Chapter 6 describes the development of AoT prototype, detailing the implementation and optimization of cryptographic primitives as well as a demo comprising all stages of an IoT device life-cycle in a smart house. Chapter 7 presents AoT prototype resource requirements. Chapter 8 summarizes our work and gives directions for future work.

Chapter 2

Background

In this chapter, we cover the fundamentals of AoT, namely Authentication in Section 2.1, Attribute-Based Access Control (ABAC) in Section 2.2, Identity-Based Cryptography (IBC) in Section 2.3, Attribute-Based Cryptography (ABC) in Section 2.4, and Pairing-Based Cryptography (PBC) in Section 2.5.

2.1 Authentication

Authentication [75] is allegedly the most important security property in IoT [107]. One of authentication's major goals is to prevent illegitimate nodes from taking part in network activities. This can protect most network operations, unless legitimate nodes have been compromised.

Broadly speaking, authentication comprises two properties. The first is *source authentication* [109], which guarantees a receiver that the message indeed originated from the claimed sender. The second is *data authentication* [109], which prevents integrity violation, i.e., it prevents that a message is altered while in transit between the sender and the receiver, and ensures that the received message is "fresh", i.e., not being replayed.

In the world of cryptography, authentication can be achieved through the use of symmetric or asymmetric (public-key) cryptosystems [109]. More precisely, through the use of Message Authentication Codes (MACs) [109] and Digital Signatures [109], respectively. (It is worth noting only the latter provides nonrepudiation, i.e., prevent a device from denying previous commitments or actions.)

AoT leverages both MACs and Digital Signatures for authentication in its suite of protocols.

2.2 Attribute-Based Access Control

Access Control [14] regulates who or what (e.g., a user) can perform an operation (e.g., read or write) on an object (e.g., a file).

Traditional access control models are inconvenient for IoT [125]. For instance, Mandatory Access Control, Discretionary Access Control, and Role-Based Access Control are user centric and do not consider factors like resource information, relationship between the user (the requester) and the resource provider, and dynamic information (e.g., current time and user identifier). Moreover, in large scale networks like IoT deployments, it might be unmanageable for one to keep a list of who is granted access to what.

ABAC [125], on the other hand, simplifies access control by replacing discretionary permissions with policies based on attributes. The model grants rights to users based on attributes like resource characteristics and contextual information. The idea is based on the observation that, in a given organization, permissions are often assigned to the attributes of users rather than to their identity. Attributes, in turn, are assigned to users according to their responsibilities or qualifications. In ABAC, the possession of an attribute can be easily altered without modifying the underlying access structure; and new permissions can be conveniently granted to attributes as new objects or operations are incorporated into the system.

AoT adopts ABAC as its fine-grained access control model for IoT.

2.3 Identity-Based Cryptography

The notion of IBC dates back to Shamir's original work [106], but it has only become practical with the advent of PBC [18, 54, 82, 102]. The main advantage of IBC is that it does not require the expensive explicit public-key authentication (like traditional PKI does). In these systems, users may have a meaningful public-key rather than a random string of bits; and those keys can thus be derived from the user's public information [28, 113]. Note this information intrinsically binds a user to its public-key, which makes other means of accomplishing this binding, e.g., digital certificates, unnecessary [28, 113]. Finally, IBC also allows secure communication between users of different IBC domains [81].

IBC, however, is not a panacea and it turns out that private-keys in IBC are not generated by their respective owners but rather by Private Key Generators (PKGs), i.e., those keys are not actually private. So, a PKG could, if it wanted to, impersonate any user in the system [28, 113]. This is the well-known key escrow problem of identity-based systems; and the main challenge for the wide adoption of IBC [28, 113].

2.4 Attribute-Based Cryptography

ABC [13, 44], also known as *Fuzzy* IBC [101], is an extension of the idea used in IBC. As such, ABC also suffers from the key escrow problem. Compared to IBC, ABC focuses on groups of users rather than solely on users' identities. The cryptosystem relies on a subset of user attributes to control private-key ownership.

Broadly speaking, there are two classes of ABC schemes, namely Key-Policy ABC (KP) (e.g., [44]) and Ciphertext-Policy ABC (CP) (e.g., [13]). In the former, the policy is attached to private keys, and attributes annotate messages. In the latter, messages carry the policy and users possess a key for each of their respective attributes. There is a close fit between KP and applications that deliver digital content like cable TV [44]. In KP, however, the sender of a message has no control over who or what will be able to access the contents of his messages. CP, on the other hand, does allow this control.

In the context of signature schemes, CP is commonly referred to as *Policy-Endorsing Attribute-Based Signature* [117]. Here, users are assigned a set of attributes and the corresponding private keys. A user's ability to perform operations over a message (e.g., sign a message) depends on the attribute set associated with the user as well as the policy associated with the message. To be concrete, messages carry a boolean expression of attributes called the signing policy or the *predicate* of the message. To verify a signature correctly, a user must sign the message using the keys associated with any subset of attributes that satisfies the predicate.

It is worth stressing the synergy between ABC and ABAC (Section 2.2). On the one hand, ABC is similar to ABAC in that they both are based on user attributes. On the other hand, ABC and ABAC are also complementary, since they may be combined so the former cryptographically enforces the later.

We chose CP to be the underlying cryptographic construction of AoT's access control mechanism. Precisely, AoT employs ABC signatures (ABS for short) during its most frequent operation and maps ABAC policies onto ABC predicates.

2.5 Pairing-Based Cryptography

PBC [19, 53, 102] has paved the way for the design of original cryptographic schemes and made existing cryptographic protocols both more efficient and convenient. It has also shed some light on many long-standing open problems allowing quite a few of them to be solved elegantly. Identity-Based Encryption (IBE) [19] is most likely the main evidence of this, as IBE has enabled complete IBC schemes. (But note that there are other pairing-free ways of performing IBE today [20, 31].)

The bilinear pairing is the major cryptographic primitive in PBC. Pairings, for short, were first used in the context of cryptanalysis [82], but their pioneering use in cryptosystems is due the works of Sakai, Ohgishi, and Kasahara [102], Boneh and Franklin [19], and Joux [53]. AoT relies heavily on PBC. For instance, AoT makes use of pairings to distribute keys and to implement ABS. In the following, we present an overview of the mathematical foundation behind pairings.

2.5.1 Elliptic Curves

Let $\mathbb{F}_{q^k}$ denote the field of integers modulo q^k. An elliptic curve E over $\mathbb{F}_{q^k}$ is defined by an equation of the form $y^2 = x^3 + ax + b$, with $a, b \in \mathbb{F}_{q^k}$. A pair (x, y), where $x, y \in \mathbb{F}_{q^k}$, is a point on E curve if (x, y) satisfies the curve's equation. An additional point at infinity, denoted $\mathcal{O}$, is also said to be on the curve. The set of all pairs of solutions together with the point at infinity is denoted $\mathbb{E}(\mathbb{F}_{q^k})$, $\mathbb{E}$ for short. There exists well-known method for adding two elliptic curve points P_1, $P_2 \in \mathbb{E}$ producing a third point on the elliptic curve $P_3 = P_1 + P_2 \in \mathbb{E}$. The scalar multiplication of a point $P \in \mathbb{E}$ is defined as the repeated addition to the point $P \in \mathbb{E}$ to itself, e.g, $s \cdot P = \overbrace{P + ... + P}^{s \text{ times}} \in \mathbb{E}$ if $s > 0$ and $s \cdot P = \mathcal{O}$, if $s = 0$. With this addition rule, the set of points $\mathbb{E}$ forms an abelian group with point $\mathcal{O}$ as the identity element. The number n of points in $\mathbb{E}$ is called the order of the curve over the field $\mathbb{F}_{q^k}$. The order r of a point $P \in \mathbb{E}$ is the least $r > 0$ such that $r \cdot P = \mathcal{O}$. The order r always divides the order n of the curve. The set of r-torsion points on $\mathbb{E}$ is the set $\mathbb{E}[r] = \{P \in \mathbb{E} \mid r \cdot P = \mathcal{O}\}$. The set of points generated by P is a subgroup of $\mathbb{E}[r]$, consequently, a subgroup of $\mathbb{E}[n]$. A subgroup $\mathbb{G}$ of an elliptic curve $\mathbb{E}(\mathbb{F}_{q^k})$ has embedded degree k if its order r divides $q^k - 1$ for the smallest possible k.

2.5.2 Bilinear Pairings

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be additively-written with identity $\mathcal{O}$ and generators G_1 and G_2, respectively, and $\mathbb{G}_T$ is a multiplicatively-written group with identity 1 [40] such that they all have the same order., i.e., $\mid \mathbb{G}_1 \mid = \mid \mathbb{G}_2 \mid = \mid \mathbb{G}_T \mid$.

A *bilinear pairing* is a map $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is said to be admissible if it satisfies the following properties:

- Bilinear: $\hat{e}(aP, bQ) = \hat{e}(P, Q)^{ab}$ $\forall P \in \mathbb{G}_1, Q \in \mathbb{G}_2$ and $\forall, a, b \in \mathbb{Z}^*_{|\mathbb{G}_1|}$.
- Non-degenerate: $\hat{e}(G_1, G_2) \neq 1$.
- Computable: there is a polynomially bounded algorithm to compute $\hat{e}(aP, bQ)$ $\forall P \in \mathbb{G}_1, Q \in \mathbb{G}_2$.

The choice of the groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, and the pairing $\hat{e}$ are part of the set up of the identity- and attribute-based cryptosystems' parameters. During the key generation phase, the cryptosystem KGC define users' public and private keys with respect to such parameters.

In this work, we restrict to elliptic curves. The groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups of points on the elliptic curve over the field $\mathbb{F}_q$ and $\mathbb{G}_T$ is a subgroup of the multiplicative group of a finite field related to $\mathbb{F}_q$, e.g., one of its extensions [41].

When $\mathbb{G}_1 = \mathbb{G}_2$, the pairing is said to be symmetric. In this case, the elliptic curve E is supersingular defined over $\mathbb{F}_q$ with embedded degree $k > 1$. The group $\mathbb{G}_1$ is always a subgroup of $\mathbb{E}(\mathbb{F}_q)$, and there exists a "distortion map" ψ which maps $\mathbb{G}_1$ into $\mathbb{E}(\mathbb{F}_q)$ and the symmetric pairing of P, $Q \in \mathbb{G}_1$ is obtained by computing $\hat{e}(P, \psi(Q))$. The actual goal of ψ is to guarantee the points P and $\psi(Q)$ are always linearly independent.

When $\mathbb{G}_1 \neq \mathbb{G}_2$, the pairing is said to be asymmetric. In this case, a common instantiation is to choose $\mathbb{G}_1$ as a subgroup of points of the elliptic curve $\mathbb{E}(\mathbb{F}_q)$, $\mathbb{G}_2$ as a subgroup of points of the elliptic curve $\mathbb{E}(\mathbb{F}_{q^k})$ and $\mathbb{G}_T$ as the multiplicative subgroup of the finite extension $\mathbb{F}_{q^k}$ [40].

As pointed in [41], asymmetric pairings provide good performance and flexibility for high security levels. Therefore, this is our adoption in AoT.

2.5.3 Security Assumption

There are in the literature many different security assumptions used to build pairing-based cryptography protocols, e.g., Bilinear Inverse Diffie-Hellman Problem (BIDH) [126], the q-Strong Diffie-Hellman (q-SDH) [17], etc. We describe here the Bilinear Decisional Diffie-Hellman (BDDH) [11, 18], a security assumption that we use in Chapter 5.5.

BDH Parameter Generator. Let $\mathcal{B}(1^\eta)$ be a *BDH Parameter Generator* [18] algorithm, which takes as input a security parameter η and returns two additively-written groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with order r, identity $\mathcal{O}$ and generators G_1 and G_2, respectively, a group $\mathbb{G}_T$ with order r, a multiplicatively-written group with identity 1, and a admissible bilinear map $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.

BDDH Assumption. The BDDH assumption holds for $\mathcal{B}$ if for every polynomial time algorithm A (in η) the BDDH advantage of A, defined as

$$\mathsf{Adv}_{A,\mathcal{B}}^{BDDH}(1^\eta) = \left|\Pr\left[\mathsf{Exp}_{A,\mathcal{B}}^{BDDH-0}(1^\eta) = 1\right] - \Pr\left[\mathsf{Exp}_{A,\mathcal{B}}^{BDDH-1}(1^\eta) = 1\right]\right|$$

is negligible as a function in η, where the experiments $\mathsf{Exp}_{A,\mathcal{B}}^{BDDH-0}$ and $\mathsf{Exp}_{A,\mathcal{B}}^{BDDH-1}$ are defined as follows. In the experiments $\mathsf{x} \xleftarrow{R} \mathbb{Z}_q^*$ describes x an element chosen uniformly at random from $\mathbb{Z}_q^*$.

```
Function $\mathsf{Exp}_{A,\mathcal{B}}^{BDDH-0}(1^\eta)$:
    $(\mathbb{G}_1, \mathbb{G}_2, G_1, G_2, \mathbb{G}_T, q, \hat{e}) := \mathcal{B}(1^\eta)$
    $\mathsf{a} \xleftarrow{R} \mathbb{Z}_q^*$ , $\mathsf{b} \xleftarrow{R} \mathbb{Z}_q^*$ , $\mathsf{c} \xleftarrow{R} \mathbb{Z}_q^*$, $\mathsf{z} \xleftarrow{R} \mathbb{Z}_q^*$
    return $A(\eta, (G_1, G_2, q, \hat{e}), \mathsf{a} \cdot G_1, \mathsf{a} \cdot G_2, \mathsf{b} \cdot G_1, \mathsf{c} \cdot G_1, \mathsf{c} \cdot G_2, \hat{e}(G_1, G_2)^{\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c}})$
end
```

```
Function $\mathsf{Exp}_{A,\mathcal{B}}^{BDDH-1}(1^\eta)$:
    $(\mathbb{G}_1, \mathbb{G}_2, G_1, G_2, \mathbb{G}_T, q, \hat{e}) := \mathcal{B}(1^\eta)$
    $\mathsf{a} \xleftarrow{R} \mathbb{Z}_q^*$ , $\mathsf{b} \xleftarrow{R} \mathbb{Z}_q^*$ , $\mathsf{c} \xleftarrow{R} \mathbb{Z}_q^*$, $\mathsf{z} \xleftarrow{R} \mathbb{Z}_q^*$
    return $A(\eta, (G_1, G_2, q, \hat{e}), \mathsf{a} \cdot G_1, \mathsf{a} \cdot G_2, \mathsf{b} \cdot G_1, \mathsf{c} \cdot G_1, \mathsf{c} \cdot G_2, \hat{e}(G_1, G_2)^{\mathsf{z}})$;
end
```

Chapter 3

Related Work

In what follows, we briefly describe previous work on security for constrained devices (Section3.1) and Elliptic Curve Cryptography (ECC) and PBC implementation on resource-constrained devices (Section3.2). Then, in Section3.3, we describe authentication and access control schemes for IoT.

3.1 Security for constrained devices

Before the advent of IoT, much work has been proposed for securing Mobile Ad hoc NETworks (MANETS) and sensor networks in general. The studies for MANETs (e.g., [27, 51, 115, 127, 129]) are not applicable to IoT because they assumed Personal Digital Assistants (PDAs), which have more computational resources than many IoT devices. Traditional PKC-based solutions are such an example. The works tailored to sensor networks (e.g., [22, 37, 47, 73, 79, 92, 94, 96, 111, 118, 130]) are usually ill-suited to IoT, too. This is because they often make assumptions that do not apply to IoT and thus the proposed solutions cannot be applied as-is. For instance, while sensor devices often run the same application and are owned by a single entity in sensor networks, devices in IoT may execute different applications and report to more than one authority. AoT brings authentication and access control to IoT, addressing IoT-specific requirements and constraints.

3.2 ECC and PBC on resource constrained devices

One of the first efforts to implement ECC into resource-constrained devices is presented by [46]. The work presents an implementation of ECC primitives based on two 128-bit elliptic curves. The authors use as target devices a family of 16-bit MSP430 microcontrollers with 1 KB of RAM and a maximum clock frequency of 3.8 MHz.

Discussion about PBC in a resource-constrained context, in turn, is presented by [90]. In this work, the authors claim that it is a real possibility to have PBC running in such type of devices by presenting estimated performance numbers of the Tate pairing computation over the 8-bit processor ATmega128L.

Although with a low level of security, the actual implementation of pairings on a resource-constrained device, a wireless sensor node, is first presented in TinyTate [89]. The target node is also the 8-bit ATmega128L microcontroller with 7.3828 MHz of clock frequency, and 4 KB of RAM. The cryptographic library TinyECC is the basis of implementation, and the results show that a Tate pairing computation takes about 30 s.

In the work NanoECC, [112] present improved numbers for pairing computation using NIST k163 Koblitz curve [87]. The run times to compute a pairing are 17.93 s on the 8-bit ATmega128L and 11.82 s on the 16-bit MSP430 with 8.192 MHz clock frequency, and 10 KB of RAM.

[43] implement PBC code specifically tailored to the family of 16-bit MSP430 micro-controller with 8MHz clock frequency. The authors use the same 100-bit security level BN curve used in AoT's prototype implementation (BN-254), and obtain a run time of 14.7 s to compute an optimal Ate pairing.

Using lazy reduction optimizations, [42] improve the computation of optimal Ate pairing at 100-bit security level (BN-254 curve) to 9.930 s in a 16-bit MSP430 microcontroller with 8 MHz, 8 KB RAM. Using a specialized 32-bit hardware multiplier, the result is further improved to 5.967 s.

In TinyPBC [92], the authors present run times of 1.90 s to compute η_T pairings on the 7.4MHz ATmega128L, 1.27 s on an 8 MHz MSP430, and 0.14 s on a 13 MHz ARM platform. These results are achieved using a supersingular 271-bit order curve, no longer consider secure [8]. In Secure-TWS [91], the authors evaluate, among others, the Boneh-Lynn-Shacham (BLS) scheme [16], a signature scheme based on pairings. The target devices are the MSP430-F2418 with 16-bit 16 MHz clock frequency, and 8 KB of RAM, and the ARM LPC2138 with 32-bit 60 MHz clock frequency, and 32 KB of RAM. In the BLS scheme there are 2 pairings computation, the authors used 80-bit security bit nonsupersingular MNT curve and Tate pairing. The run times of BLS are 1.59 s and 98.4 ms on MSP and ARM platforms, respectively. Unfortunately, there is no result of pairing computation alone.

[114] present a hardware approach to accelerate pairing computation. The authors equip a 32-bit processor equivalent to the ARM Cortex-M0+ (48 MHz clock frequency) with a dedicated hardware module to speed up arithmetic in the prime field. Using 100-bit security level (BN254 curve), the authors reach 164 ms for pairing computation.

From the above analysis, it is clear that the implementation of PBC has been taken to a high level of optimization to fit resource constrained-devices. However, not only the research community has put efforts in bringing down run time numbers but also devices with more computational power are becoming more and more affordable, allowing the implementation of PBC protocols and schemes such as AoT.

3.3 Authentication and access control for IoT

In the following sections we discuss works that do not consider a fine-grained access control mechanism for IoT (Section 3.3.1), proposals of access control models and frameworks for IoT (Section 3.3.2), and proposals that address authentication and fine-grained access control (Section 3.3.1). In Section 3.3.4, we present a comparison table among all the discussed works and AoT.

3.3.1 Authentication

Several works on authentication and access control for IoT enforce their access control scheme solely on the authentication mechanism, i.e., fine-grained access control is not considered. Below, we describe some work that, in a certain way, can be categorized as so.

In [91], the authors propose to authenticate communication from a device to multiple users. The proposal named Secure Tiny Web Services (Secure-TWS) investigates the resource overheads for digital signatures, notably, for the Elliptic Curve Digital Algorithm (ECDSA) and the Boneh-Lynn-Shacham (BLS) short signature schemes. The work contemplates a prototype on a set of resource-constrained devices.

[85] propose a scheme to provide authentication for smart domestic devices. The authors employ IBC and Wi-Fi as well as Bluetooth and NFC to authenticate message exchange between devices. NFC is used during *Bluetooth pairing* and key distribution. The authors present prototypes of their proposal for intermediate and powerful devices.

Yavuz proposes an ECC-based authentication scheme for IoT devices called Efficient and Tiny Authentication (ETA) [122]. Due to its ECC nature, the proposal provides authentication with a small key and signature sizes. Even though the author claims the scheme targets resource-constrained devices, the evaluation comprises only powerful devices.

In [98], it is proposed a two-phase authentication scheme for IoT. The proposed protocol relies on implicit certificates to provide mutual authentication in a heterogeneous architecture. In the first phase, the entities on the architecture obtain their credentials from a trusted party to, then, in the second phase, be able to mutually authenticate each other. The protocol is designed based on the ECQV implicit certificate and ECDH key exchange mechanisms.

[80] propose a federated end-to-end authentication scheme for IoT based on IBC. In this work, constrained IoT devices are located in sub-networks connected to the Internet through more powerful border gateways, which maintain a local trusted authority. The end-to-end authentication between nodes from different domains requires a federation between the gateways. The authors provide an experimental evaluation of their proposal on a resource-constrained device.

In [120], the authors propose a method for key agreement and authentication that allows two IoT devices physically close to each other to agree on a common key based on the Channel State Information (CSI) available from the Orthogonal Frequency Division Multiplexing (OFDM) of the current WiFi standard.

[99] propose a key management architecture for the resource- constrained devices in IoT that enables device-to-server authentication. The proposal allows a resource-constrained device to establish a pre-shared symmetric key with a resource server within the Datagram Transport Layer Security (DTLS) handshake.

In [108], it is proposed a lightweight and escrow-less authenticated key agreement for IoT that enables device-to-device authentication in such a context. The proposal protocol combines the SMQV (strengthened-Menezes-Qu-Vanstone) protocol combined with implicit certificates. The proposal ends-up in an efficient approach that avoids transmission and storage costs of traditional PKI-based certificates.

[60] propose an Elliptic Curve Cryptography (ECC)-based authentication protocol for a context where IoT devices, represented as resource-constrained devices, send data to a cloud server. The authors provide an analytical performance evaluation of their scheme and a security analysis using an automated tool.

In [48], the authors propose a decentralized authentication scheme for IoT based on blockchain. They call their system "bubbles of trust", where the bubbles are analogous to AoT's home domains. The authors evaluate their approach in a prototype comprised of powerful and intermediate (similar to the one we used in our evaluation) devices. Communication device-to-device is always dependent on the ledger.

In [33], the authors propose a lightweight symmetric-based user authentication scheme for IoT. The authors consider an architecture that comprises a registration authority, users, gateways, and IoT devices. In this case, a gateway distributes tokens generated by the registration authority for a pool of IoT devices it manages. The user also receives a token that will, then, allow her to authenticate to the devices. The authors provide an analytical performance evaluation and a security analysis of their scheme.

[86] propose a key management scheme for securing communications between IoT devices in an architecture formed by field nodes, gateways and a remote server. Their scheme is based on lightweight symmetric cryptographic and can be portable even in sensor nodes. The authors provide a experimental performance analysis on resource-constrained devices and

a security analysis using a automated tool.

[103] propose a federated lightweight authentication for IoT. Their protocol relies on implicit certificates and symmetric cryptography and, then, can be executed on resource-constrained devices without the dependency on an external entity. As a federated-based solution, their approach contemplates inter-domain communication. The authors also evaluate their approach on resource-constrained devices.

3.3.2 Access Control Frameworks

An authorization framework for IoT is proposed in [105]. The authors describe a framework that allows fine-grained and access control based on eXtensible Access Control Markup Language (XACML) to connected devices with limited processing power and memory. Their proposed architecture comprises a resource directory, where are kept the public and symmetric keys of the devices as well as the access policies, the resource-constrained device, the user that accesses the device resource, and a back-end authorization engine, which signs and verifies assertions of the users. The resource constrained-devices enforce the access control decision locally, however, the users always depend on the authorization engine to requests access to devices' operations. The authors provide a proof of concept of their proposal on a resource-constrained device.

[100] propose an schema that unifies the access control mechanism in an architecture comprises of intelligent agents, IoT devices and hybrid elements. Their schema abstracts the nature of the IoT entities in the access control policies, which allows the model to support different IoT contexts at the same time.

In [38], the authors present an architectural model to enables access control in IoT contexts. They propose the usage of OAuth2, which make their model compatible with RESTful services. In their approach, an IoT application needs to developed with a policy enforcement point and, then, be registered as a service in an Identity Provider to have the authorization layer automatically integrated.

[2] propose an access control model for virtual objects communication in IoT. Their model combines ACLs, RBAC, and ABAC, where ACLs control publish and subscribe rights of virtual objects and RBAC and ABAC govern the administrative rights on the ACL itself.

In [97], the authors present a blockchain-based architecture for IoT access authorizations. Their proposal architecture follows the principles of decentralization, resilience, off-line working, and low processor usage for authorization. The authors define a decoder that receives an existing access control model (RBAC, ABAC, or ACL) and translate it to mechanisms supported in their architecture.

3.3.3 Authentication and fine-grained Access Control

[74] propose an identity-based authentication and access control framework for IoT. The authors consider an architecture where things have a registration authority, and users have their exclusive registration authority. The communication between users and things is always dependent on the trusted authorities. In this case, OpenID is used to provide authentication and RBAC to govern access control.

In [52], it is presented an initial design that aims at making certificate-based authentication feasible for resource-constrained objects in IoT. This proposal is based on DTLS and uses certificate pre-validation which is computed on gateways, and a delegation procedure also executed on an external entity to reduce the computing burden on resource-constrained devices.

[123] propose an ECC-based authentication and access control mechanism for wireless sensor networks. In their work, the architecture comprises users' smart devices, sensor nodes, base stations, gateways, and a trusted certificate authority. In their proposal, users have to negotiate with the base stations the secret parameter to be used in the cryptosystem. The certificate authority, in turn, issues attribute certificates to users and sensors nodes. Hence, during access control, the user presents her certificate that must be verified by the node against the ABAC access policy before access to the data can be guaranteed. The proposal does not consider inter-domain communication nor provides an experimental evaluation of the protocols.

In [39], the authors evaluate the usage of OAuth 2.0 together with MQTT protocol as a model for federated identity and access management in IoT. The authors identify a number of issues, in open-source implementation of the protocols and in fundamentals, where propose further research. Besides authentication, the work provides access control, performance evaluation on a resource-constrained platform, and, as a federated approach, inter-domain communication.

[30] propose an architecture targeted to IoT scenarios for an external authorization service based on OAuth. In their proposal, the access control is delegated to an external service, which has to be invoked by the devices. The authors present an evaluation of their proposal on a resource-constrained device.

An authentication and authorization infrastructure for IoT applications based on IBC, SAML and XACML is proposed in [36]. The proposal provides two operational modes. The first one is specific for resource-constrained devices, in which case the access control is delegated to an external server. The other option, in turn, if the device itself has enough computational resources, it executes the access control processes.

In [55], the authors propose an OpenID-based authentication service for IoT. In their approach, the user's authentication is performed by an OpenID platform and user authorization to devices is performed by an access policy description component that executes on a relying party, which, in turn, intermediates the communication to the devices.

[76] propose a distributed access control solution for IoT based on a blockchain architec-

ture. In their proposal, gateways manage the user access to devices information and control the device communication. Besides, each gateway keeps a copy of the ledger, which, in turn, maintains the public key of each device on the environment. The authors provide an experimental evaluation in resource-constrained devices.

[72] also propose an authentication with fine-grained access control scheme for IoT based on blockchain. In their proposal, which targets the industrial context, the authors replace the ECDSA signature used in the Bitcoin system with ABS (they use the same scheme we adopt in the AoT prototype) and use secure certificateless multi-receiver encryption to send encrypted data from one entity to a group of pre-selected receivers. The transactions in their architecture carry instructions, such as querying, storing, and operating data signed with ABS, where the fine-grained access control is guaranteed. As their proposal targets the industry, the authors evaluate the performance of their prototype on a powerful device.

In [35], the authors propose an ABAC scheme based on blockchain for IoT. In their architecture, each device receives a set of attributes issued by a trusted authority. The trusted authority acts as a key generation center to issue an IBC key pair for IoT devices when they are registered on the system, and as an attribute authority when the devices apply for a set of attributes. Upon authorization, the attribute authority records each attribute as a transaction on the blockchain. Before enrolling in an access control protocol, a pair of devices mutually authenticate each other. Then, they execute a challenge-response protocol in which the requester must sign the challenge with a subset of attributes that satisfies that ABAC access policy. The signature is verified by the other party consulting the records on the blockchain.

A decentralized authentication scheme based on blockchain for IoT is proposed in [56]. In this case, the authors not only consider authentication but also access control defined on top of smart contracts. To communication to each other, the IoT devices always depend on the ledger. This work comprises a performance analysis based on a prototype that executes on powerful and intermediate devices.

3.3.4 Comparison

For the sake of comparison, we summarize the works we discussed above in Table 3.1. As comparison metrics we verify if the work covers: (i) the entire IoT device life-cycle; (ii) authentication; (iii) fine-grained access control; (iv) when the proposal contemplates access control we check if it demands an external server or the devices themselves can process the request; (v) inter-domain communication; (vi) the minimum computation capability of the device used on the proposal evaluation or prototype; and (vii) if the proposal is escrow resistant in the main domain (compared to AoT's home domain).

Work	IoT Device Life-Cycle	Authentication	Fine-grained Access Control	Access Control Server-independent	Inter-Domain Communication	IoT Device	Escrow Resistance
[91]	-	✓	-	-	-	constrained	-
[85]	-	✓	-	-	-	intermediate	-
[122]	-	✓	-	-	-	powerful	-
[98]	-	✓	-	-	-	constrained	-
[80]	-	✓	-	-	✓	constrained	-
[120]	-	✓	-	-	-	powerful	✓
[99]	-	✓	-	-	-	constrained	-
[108]	-	✓	-	-	-	constrained	✓
[60]	-	✓	-	-	-	-	-
[48]	-	✓	-	-	-	intermediate	✓
[33]	-	✓	-	-	✓	-	✓
[86]	-	✓	-	-	-	constrained	✓
[103]	-	✓	-	-	✓	constrained	✓
[105]	-	✓	✓	-	✓	constrained	-
[100]	-	-	✓	-	✓	-	-
[38]	-	✓	✓	-	✓	-	-
[2]	-	-	✓	-	✓	-	-
[97]	-	-	✓	✓	✓	-	✓
[74]	-	✓	✓	-	✓	-	-
[52]	-	✓	-	-	✓	constrained	✓
[123]	-	✓	✓	✓	✓	-	-
[39]	-	✓	✓	-	✓	constrained	✓
[30]	-	✓	✓	-	-	constrained	✓
[36]	-	✓	✓	✓	✓	powerful	-
[76]	-	✓	✓	-	✓	constrained	✓
[72]	-	✓	✓	-	-	powerful	-
[55]	-	✓	✓	-	✓	powerful	✓
[35]	-	✓	✓	-	✓	constrained	-
[56]	-	✓	✓	-	✓	intermediate	✓
AoT	✓	✓	✓	✓	✓	constrained	-

Table 3.1: Related work comparison.

We observe from table 3.1 that AoT has some advantages over related work on authentication and access control for IoT. First, AoT covers the entire IoT device life-cycle, which we point as a major contribution of our work. Another clear contribution is fine-grained access control, which is indeed considered in other works, however, they end up requiring devices with more computational resources than AoT does or depending on external entities to perform the decision making on behalf of the devices.

Chapter 4

Authentication of Things

In this chapter, we present AoT. AoT comprises a suite of protocols that cover all stages of an IoT device's life-cycle. We provide an overview of AoT (Section 4.1), present AoT's auxiliary (Section 4.2) and main protocols (Sections 4.3–4.7), and discuss its complementary features (Section 4.8).

4.1 Overview

4.1.1 Assumptions

In what follows, we assume (i) every message is numbered; (ii) every message carries the identifiers of the interlocutors, although we make it explicit in key agreement protocols for security analysis purposes; (iii) cryptographic material can be securely loaded into devices at the manufacturer's facility; (iv) PINs can be securely entered onto devices at home; (iv) cryptographic primitives are ideal—i.e., flawless—and can be treated as black boxes; (v) the device manufacturer is trusted.

4.1.2 Problem

In IoT, questions as how to enable authentication and fine-grained access control remain unanswered. On the one hand, traditional PKC is computationally expensive and most IoT devices cannot afford to run them. On the other hand, authentication schemes for IoT especially targeting resource-constrained devices, typically base their access control mechanism solely on authentication, which is also known as the all-or-nothing approach. Alternatives for fine-grained access control for resource-constrained devices in IoT, in turn, usually delegate the access control decision to an external trusted entity, creating a third-party dependency on a supposed direct device-to-device operation, which impacts user experience in cases of instability or unavailability on the authorization service. To make things worse, questions regarding the portability and mobility of “things” arise. Portability and mobility are typical of IoT and they reinforce the call for interoperation between local and guest devices. Last, there is a lack of options for authentication and access control solutions that cover the entire IoT device life-cycle, i.e., from device manufacturing to decommissioning.

4.1.3 Goal

Our ultimate goal is to design a holistic tailor-made authentication and fine-grained access control solution for the entire IoT device life-cycle. Our solution can be used in many smart contexts, however, we describe it as a smart house use case. AoT targets the highly heterogeneous and interoperable nature of IoT smart environments, where devices like home appliances: (i) interact with other home appliances, personal devices, and a home server in a local domain of trust, where the resources demand fine-grained access control permissions; (ii) do not have any dependency on third parties during the authentication and access control processes; (iii) can be operated by other (usually personal) devices from foreign domains; and (iv) interact with a remote cloud server in a manufacturer domain of trust, which represents the trust relationship between the appliance and its manufacturer’.

4.1.4 Approach

As usual, our approach concentrates (i) on key distribution to bootstrap security (ii) and on access control to govern permissions over device operations. However, we implement those in a novel way: IBC is used to distribute keys and ABC to control access to device operations.

Our key insight to tackle the key escrow problem of IBC (Section 2) is a two-domain architecture (Figure 4.1). More precisely, our solution comprises two distinct IBC setups, namely: a manufacturer (*Cloud*) setup and a local (*Home*) setup. They respectively define manufacturer-to-device and domestic device-to-device trust relationships. There is no overlap in these trust relationships and thus an artifact generated in the Cloud domain is invalid in the Home domain and vice-versa. Note that the key escrow still holds in each IBC setup individually; however, the escrow now is no longer a major problem. For the Cloud's IBC keys escrow, this is because the user's privacy is preserved in that requests originating from the Cloud domain are null in the Home domain. For the Home's IBC and ABC keys escrows, the context of where it takes place already deals with the problem. Nevertheless, the compromising of such a device leads to the compromising of the entire Home Domain, which means the device itself should integrate further protection mechanisms. In this sense, standard techniques can be used to protect the cryptographic material, like employing a Hardware Security Module (HSM) or Trusted Platform Module (TPM).

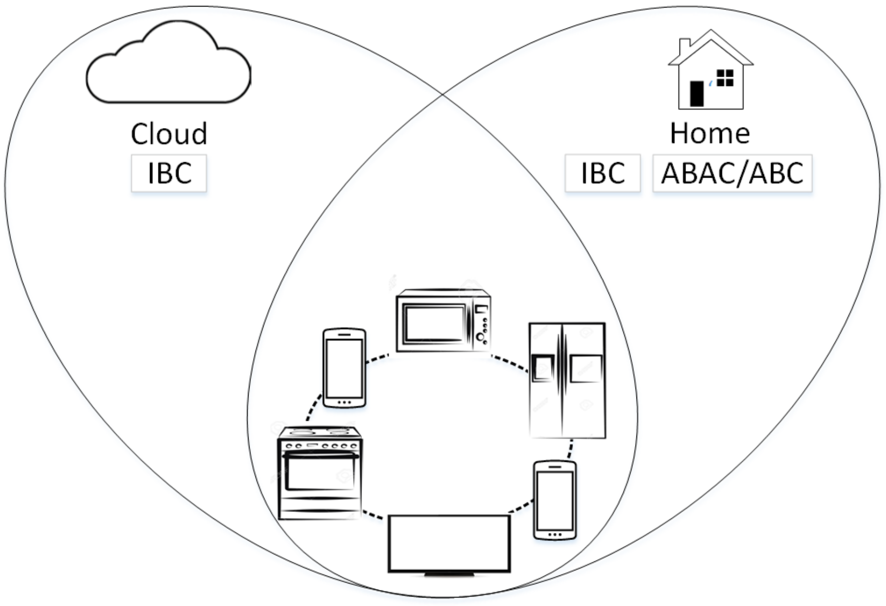


Figure 4.1: AoT two-domain architecture.

Our suite of protocols cryptographically provides authentication and access control over wireless mediums. One of AoT's key features is the ability to deploy devices without resorting to cabled connections. Instead, AoT leverages a home's physical protection and a reliable device (e.g., the Home domain manager's smartphone) to bridge the very first communication between a new device and the Home server to bootstrap security. This communication can happen over different encodings and transmission technologies. AoT delivers access control by combining ABAC and ABC so the former cryptographically enforces the later. Here, a permission for operating a device is ultimately tied to the ABC attributes of a requester; and the ABC attributes and predicates are governed by the higher-level ABAC policies.

4.1.5 Life-Cycle

In AoT, the life-cycle of an IoT device comprises five main stages, namely: (1) *pre-deployment*, (2) *ordering*, (3) *deployment*, (4) *functioning*, and (5) *retirement*. By the way of example, consider a given device life-cycle. In the *pre-deployment*, the cryptographic material of the Cloud domain is loaded into the device at the factory, i.e., during its manufacturing process. Next, in *ordering*, the (about to become) owner of the device purchases it and gets a PIN that grants the owner the initial access to the device. *Deployment*, as its name suggests, is when the device is deployed in its Home domain for the very first time and therefore is the stage responsible for bootstrapping security in such a context. During deployment, the owner uses the PIN to access the device, which puts it in setup mode. The device, in turn, uses the owner's personal device to establish a trust relationship with the Home server. (The Home server is the trusted home authority in charge of managing keys and orchestrating access control inside the home domain. For instance, the Home server issues IBC and ABC keys as well as advertises access permissions over the domestic network.) Finally, during this stage, the device is also bound to a user in the Cloud domain. Now, the way is paved for *functioning*, which corresponds to the daily operation of the device. At this point, users request the device's operations, and the latter reacts based on users' clearance levels. *Retirement* is the end point of a device life-cycle. It takes place whenever its owner will no longer use the device. During this stage, there is a wipeout cryptographic material held by the device and the owner is unbound from the device in the Cloud domain.

4.1.6 Complementary Features

AoT has also some complementary features. For instance, AoT enables inter-domain interactions between devices on different Home domains, meaning that devices from different Home domains may interoperate seamlessly as long as their respective Home servers have previously agreed on some parameters. This strategy is appealing because it neither violates the identity-based nature of AoT, nor requires key escrow across the participating domains. Our suite of protocols also address device reassignment, enabling a user to trade or give away one or more of his devices. Finally, we explain how AoT achieves key revocation.

4.2 Auxiliary Protocols

The auxiliary protocols *SessionKey* (Protocol 4.1), *PairwiseKeyAgreement* (Protocol 4.2), and *SessionKeyDerivation* (Protocol 4.3) compose the AoT's session key establishment strategy. Protocols *KeyIssue*, *Binding*, and *Unbinding* (Protocols 4.4, 4.5, and 4.6, respectively), in turn, assist the main protocols (Sections 4.3–4.7) in issuing a private key and binding and unbinding a device to a user in the Cloud domain, respectively.

4.2.1 SessionKey

In AoT, we use a context-dependent strategy to establish the session keys for its protocols. *SessionKey* protocol (Protocol 4.1) is used in contexts where forward secrecy is a requirement or extra computational burden to establish a session key can be tolerated. In practice, it is used as a session key establishment protocol in *KeyIssue*, *Binding*, *Unbinding*, and *Deployment*. *SessionKey* is based on the enhanced [29] identity-based key agreement protocol of [81]. A server S – Home server H or Cloud server C – sets up the identity-based cryptosystem I for the domain $\mathcal{Z}$ – Home domain $\mathcal{H}$ or Cloud domain $\mathcal{C}$ – as follows. It first inputs a security parameter η in a BDH parameter generator $\mathcal{B}$ (Section 2.5.3), which returns the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with order r, identity $\mathcal{O}$, and generators $G^{\mathrm{I}}_{1,\mathcal{Z}}$ and $G^{\mathrm{I}}_{2,\mathcal{Z}}$, respectively, the group $\mathbb{G}_T$ of order r and identity 1, and the admissible *bilinear pairing* $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Besides, X selects mapping functions $\textsc{Map}_1 : \{0,1\}^* \to \mathbb{Z}^*_r$ and $\textsc{Map}_2 : \{0,1\}^* \times \{0,1\}^* \times \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_T \to \{0,1\}^n$, randomly chooses an master secret $\mathrm{secret}^{\mathrm{I}}_{\mathcal{Z}} \in \mathbb{Z}^*_r$, and a sets the master public key (S's public key) $P^{\mathrm{I}}_{S,\mathcal{Z}} := \mathrm{secret}^{\mathrm{I}}_{\mathcal{Z}} \cdot G^{\mathrm{I}}_{1,\mathcal{Z}}$. The IBC public key of a device X is calculated as $P^{\mathrm{I}}_{X,\mathcal{Z}} := \textsc{Map}_1(id_{X,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}}$. X's IBC private key, in turn, is calculated as $S^{\mathrm{I}}_{X,\mathcal{Z}} := (\textsc{Map}_1(id_{X,\mathcal{Z}}) + \mathrm{secret}^{\mathrm{I}}_{\mathcal{Z}})^{-1} \cdot G^{\mathrm{I}}_{2,\mathcal{Z}}$.

Following the Protocol 4.1 step by step, when a device A has to establish a session key with device B, A calculates B's IBC public key $P^{\mathrm{I}}_{B,\mathcal{Z}}$ (step 1). A then generates a random scalar $\mathrm{x}_A \in \mathbb{Z}^*_r$ and associates it with B's IBC public key in a relation we call R_A through a scalar multiplication operation, i.e, $\mathsf{R}_A := \mathrm{x}_A \cdot P^{\mathrm{I}}_{B,\mathcal{Z}}$ (step 2). After that, A generates a nonce n_A, combines it in a message with the previously calculated term R_A and a session key establishment request label, and sends the message to device B (step 3)[1].

$$\begin{array}{rrl} 1. & A: & P^{\mathrm{I}}_{B,\mathcal{Z}} \coloneqq \textsc{Map}_1(id_{B,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}} \\ 2. & A: & \mathsf{R}_A \coloneqq \mathrm{x}_A \cdot P^{\mathrm{I}}_{B,\mathcal{Z}} \\ 3. & A \rightarrow B: & id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathrm{n}_A, \mathsf{R}_A, \mathsf{session\_req} \end{array}$$

Upon receiving such a message, B calculates A's IBC public key (step 4), generates its random scalar $\mathrm{x}_B \in \mathbb{Z}^*_r$ and associates it with A's IBC public key in the relation $\mathsf{R}_B \coloneqq \mathrm{x}_B \cdot P^{\mathrm{I}}_{A,\mathcal{Z}}$ (step 5). In step 6, B calculates the identity-based key $k^{\mathrm{ID}}_{B,A}$ applying the mapping $\textsc{Map}_2$ over the identities of A and B, the relations R_A and R_B, and the result of the product of the pairing of the relation R_A received from A with IBC private key $S^{\mathrm{I}}_{B,\mathcal{Z}}$ and the pairing of the IBC groups generators parameters and the random scalar x_B, i.e., $k^{\mathrm{ID}}_{B,A} \coloneqq \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_A, S^{\mathrm{I}}_{B,\mathcal{Z}})$ $\hat{e}(G_1, G_2)^{\mathrm{x}_B})$. Now, B has a key that A also will able to generate in step 10.

$$\begin{array}{rrl} 4. & B: & P^{\mathrm{I}}_{A,\mathcal{Z}} \coloneqq \textsc{Map}_1(id_{A,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}} \\ 5. & B: & \mathsf{R}_B \coloneqq \mathrm{x}_B \cdot P^{\mathrm{I}}_{A,\mathcal{Z}} \\ 6. & B: & k^{\mathrm{ID}}_{B,A} \coloneqq \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_A, S^{\mathrm{I}}_{B,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B}) \end{array}$$

Differently from the original protocols [29, 81], the subsequent messages of Protocol 4.1 also allow the devices A and B to (indirectly) confirm the agreed key. To confirm $k^{\mathrm{ID}}_{B,A}$, however, we first observe that the session key generated by the protocol must serve the cryptographic purpose of the session of the higher-level protocol. For instance, in *Binding* the session key is used for authentication, while in *KeyIssue* the session key is used for authenticated encryption. Following the best practice of single-purpose use of keys, we define that $k^{\mathrm{ID}}_{B,A}$ can be used only to derive new keys to perform arbitrary cryptographic operations. Hence, we use $k^{\mathrm{ID}}_{B,A}$ to derive: (i) a new MAC key $k^{\mathrm{MAC}}_{B,A}$ to perform the confirmation of $k^{\mathrm{ID}}_{B,A}$; and (ii) the actual session key $k_{B,A}$. Therefore, in step 7, B uses the pseudo-random function $\textsc{Prf}$ over the concatenation of n_B and n_A using $k^{\mathrm{ID}}_{B,A}$ to derive $k^{\mathrm{MAC}}_{B,A}$. In step 8, after B generates its nonce n_B, it uses $\textsc{Prf}$ over the concatenation of n_B and n_A using the $k^{\mathrm{ID}}_{B,A}$ to derive the session key $k_{B,A}$, which can be used by the higher-level protocol to fulfill the required cryptographic purpose. Then, in step 9, B responds A with the nonce n_B, the relation R_B generated in step 5, an acknowledgment label string, and uses key $k^{\mathrm{MAC}}_{B,A}$ to calculate the MAC of all this content concatenated with n_A received in step 3.

$$\begin{array}{rrl} 7. & B: & k^{\mathrm{MAC}}_{B,A} \coloneqq \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\mathrm{ID}}_{B,A}} \\ 8. & B: & k_{B,A} \coloneqq \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_{k^{\mathrm{ID}}_{B,A}} \\ 9. & B \rightarrow A: & id_{B,\mathcal{Z}}, id_{A,\mathcal{Z}}, \mathrm{n}_B, \mathsf{R}_B, \mathsf{session\_ack}, \textsc{Mac}(\mathrm{n}_A \mid \mathsf{R}_A)_{k^{\mathrm{MAC}}_{B,A}} \end{array}$$

After receiving such a message, in step 10, A calculates the key using the mapping $\textsc{Map}_2$ over the identities of A and B, the relations R_A and R_B, and the result of the product of the pairing of the relation R_B received from B with IBC private key $S^{\mathrm{I}}_{A,\mathcal{Z}}$ and the pairing of the IBC

[1]As we mentioned on the assumptions, in key agreement protocols we we make it explicit the identifiers of interlocutors on exchanged messages.

groups generators parameters and the random scalar x_A, i.e., $k^{\text{ID}}_{A,B} := \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_B, S^{\text{I}}_{A,\mathcal{Z}})\hat{e}(G_1, G_2)^{\mathrm{x}_A})$. After that, analogously to B, A derives the keys $k^{\text{MAC}}_{A,B}$ (step 11) and $k_{A,B}$ (step 12). To proceed with the protocol, A verifies the MAC of the message received in step 9 (not explicitly shown in the protocol). If the verification fails, A aborts the protocol execution. Otherwise, A sends B an acknowledgment label string and a MAC of it concatenated with n_A, R_A, n_B, and R_B (step 13).

$$\begin{array}{rrl}
10. & A: & k^{\text{ID}}_{A,B} := \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_B, S^{\text{I}}_{A,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A}) \\
11. & A: & k^{\text{MAC}}_{A,B} := \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\text{ID}}_{A,B}} \\
12. & A: & k_{A,B} := \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_{k^{\text{ID}}_{A,B}} \\
13. & A \to B: & id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \text{ session\_ack}, \textsc{Mac}(\mathrm{n}_A \mid \mathsf{R}_A \mid \mathrm{n}_B \mid \mathsf{R}_B)_{k^{\text{MAC}}_{A,B}}
\end{array}$$

Upon receiving such a message, B verifies the MAC (also not shown in the protocol), if it fails, B aborts the protocol execution. Otherwise, the session is established, and a higher-level protocol can use the session key for its cryptographic purpose. We also allow the higher-level protocols to use the nonces generated during session key establishment as a session stamp to authenticate the protocol messages until their conclusion, which we consider the end of a session.

The keys $k^{\text{ID}}_{B,A} := \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_A, S^{\text{I}}_{B,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B})$ and $k^{\text{ID}}_{A,B} := \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_B, S^{\text{I}}_{A,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A})$ calculated by devices B and A in steps 6 and 10 of the protocol, respectively, are the same. We show bellow, using pairing properties, that the computation of the last term on the mapping $\textsc{Map}_2$ performed by both devices end up in the same result.

Expanding the term $\hat{e}(\mathsf{R}_A, S^{\text{I}}_{B,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B}$:

$$\begin{aligned}
&\hat{e}(\mathsf{R}_A, S^{\text{I}}_{B,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \\
&\hat{e}(\mathrm{x}_A \cdot P^{\text{I}}_{B,\mathcal{Z}}, S^{\text{I}}_{B,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \\
&\hat{e}(\mathrm{x}_A \cdot (\textsc{Map}_1(id_{B,\mathcal{Z}}) \cdot G^{\text{I}}_{1,\mathcal{Z}} + P^{\text{I}}_{S,\mathcal{Z}}), S^{\text{I}}_{B,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \\
&\hat{e}(\mathrm{x}_A \cdot (\textsc{Map}_1(id_{B,\mathcal{Z}}) \cdot G^{\text{I}}_{1,\mathcal{Z}} + \text{secret}^{\text{I}}_{\mathcal{Z}} \cdot G^{\text{I}}_{1,\mathcal{Z}}), S^{\text{I}}_{B,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \\
&\hat{e}(\mathrm{x}_A \cdot (\textsc{Map}_1(id_{B,\mathcal{Z}}) + \text{secret}^{\text{I}}_{\mathcal{Z}}) \cdot G^{\text{I}}_{1,\mathcal{Z}}, S^{\text{I}}_{B,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \\
&\hat{e}(\mathrm{x}_A(\textsc{Map}_1(id_{B,\mathcal{Z}}) + \text{secret}^{\text{I}}_{\mathcal{Z}}) \cdot G^{\text{I}}_{1,\mathcal{Z}}, (\textsc{Map}_1(id_{B,\mathcal{Z}}) + \text{secret}^{\text{I}}_{\mathcal{Z}})^{-1} \cdot G^{\text{I}}_{2,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \\
&\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A}\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A+\mathrm{x}_B}
\end{aligned}$$

Expanding the term $\hat{e}(\mathsf{R}_B, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A}$:

$\hat{e}(\mathsf{R}_B, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A} =$

$\hat{e}(\mathrm{x}_B \cdot P^{\mathrm{I}}_{A,\mathcal{Z}}, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A} =$

$\hat{e}(\mathrm{x}_B \cdot (\textsc{Map}_1(id_{A,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}}), S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A} =$

$\hat{e}(\mathrm{x}_B \cdot (\textsc{Map}_1(id_{A,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + \mathsf{secret}^{\mathrm{I}}_{\mathcal{Z}} \cdot G^{\mathrm{I}}_{1,\mathcal{Z}}), S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A} =$

$\hat{e}(\mathrm{x}_B \cdot (\textsc{Map}_1(id_{A,\mathcal{Z}}) + \mathsf{secret}^{\mathrm{I}}_{\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}}, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A} =$

$\hat{e}(\mathrm{x}_B(\textsc{Map}_1(id_{A,\mathcal{Z}}) + \mathsf{secret}^{\mathrm{I}}_{\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}}, (\textsc{Map}_1(id_{A,\mathcal{Z}}) + \mathsf{secret}^{\mathrm{I}}_{\mathcal{Z}})^{-1} \cdot G^{\mathrm{I}}_{2,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A} =$

$\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B}\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A} = \hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A+\mathrm{x}_B}$

Therefore, $\hat{e}(\mathsf{R}_A, S^{\mathrm{I}}_{B,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B} = \hat{e}(\mathsf{R}_B, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A}$, i.e., all the terms in $\textsc{Map}_2$ are the same, which results $k^{\mathrm{ID}}_{B,A}$ and $k^{\mathrm{ID}}_{A,B}$ are equal. As the MAC and session keys are derived from the identity-based keys using the exchanged nonces, $k^{\mathrm{MAC}}_{B,A}$ and $k^{\mathrm{MAC}}_{A,B}$ are also equal, as well as are the keys $k_{B,A}$ and $k_{A,B}$.

```
SessionKey(device A, device B)
 1.          A :  $P^{\mathrm{I}}_{B,\mathcal{Z}} := \textsc{Map}_1(id_{B,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}}$
 2.          A :  $\mathsf{R}_A := \mathrm{x}_A \cdot P^{\mathrm{I}}_{B,\mathcal{Z}}$
 3.  A → B :  $id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathrm{n}_A, \mathsf{R}_A$, session_req
 4.          B :  $P^{\mathrm{I}}_{A,\mathcal{Z}} := \textsc{Map}_1(id_{A,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}}$
 5.          B :  $\mathsf{R}_B := \mathrm{x}_B \cdot P^{\mathrm{I}}_{A,\mathcal{Z}}$
 6.          B :  $k^{\mathrm{ID}}_{B,A} := \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_A, S^{\mathrm{I}}_{B,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B})$
 7.          B :  $k^{\mathrm{MAC}}_{B,A} := \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\mathrm{ID}}_{B,A}}$
 8.          B :  $k_{B,A} := \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_{k^{\mathrm{ID}}_{B,A}}$
 9.  B → A :  $id_{B,\mathcal{Z}}, id_{A,\mathcal{Z}}, \mathrm{n}_B, \mathsf{R}_B$, session_ack, $\textsc{Mac}(\mathrm{n}_A \mid \mathsf{R}_A)_{k^{\mathrm{MAC}}_{B,A}}$
10.          A :  $k^{\mathrm{ID}}_{A,B} := \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_B, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_A})$
11.          A :  $k^{\mathrm{MAC}}_{A,B} := \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\mathrm{ID}}_{A,B}}$
12.          A :  $k_{A,B} := \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_{k^{\mathrm{ID}}_{A,B}}$
13.  A → B :  $id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}$, session_ack, $\textsc{Mac}(\mathrm{n}_A \mid \mathsf{R}_A \mid \mathrm{n}_B \mid \mathsf{R}_B)_{k^{\mathrm{MAC}}_{A,B}}$
```

Where the symbols denote:

$P^{\mathrm{Y}}_{X,\mathcal{Z}}$:	X's public key of cryptosystem Y, domain $\mathcal{Z}$
I :	Identity-based cryptosystem
$\textsc{Map}_1(m), \textsc{Map}_2(m)$:	mapping functions over m
$id_{X,\mathcal{Z}}$:	X's identity in domain $\mathcal{Z}$
$G^{\mathrm{I}}_{1,\mathcal{Y}}, G^{\mathrm{I}}_{2,\mathcal{Z}}$:	domain $\mathcal{Z}$'s identity-based cryptosystem additively-written groups $\mathbb{G}_1$ and $\mathbb{G}_2$ generator points, respectively
R_X :	Device X's relation that associates a X's random scalar with the intended communication partner's Identity-based public key
x_X :	random scalar generated by X
$\rightarrow$:	unicast transmission
n_X :	nonce generated by X
session_req, _ack :	request and acknowledgment labels
$k^{\mathrm{ID}}_{X,Y}$:	identity-based key agreed by X and Y
$\hat{e}$:	a computable, non-degenerate *bilinear pairing* function
$S^{\mathrm{Y}}_{X,\mathcal{Z}}$:	X's private key of cryptosystem Y, domain $\mathcal{Z}$
$k^{\mathrm{MAC}}_{X,Y}$:	MAC key established by X and Y
$\textsc{Prf}(m)_k$:	Pseudo-random function over m using key k
$k_{X,Y}$:	session key agreed by X and Y
msg, $\textsc{Mac}(m)_k$:	MAC over the message msg appended to m using key k
$\|$:	concatenation

Protocol 4.1: SessionKey.

4.2.1.1 Security Considerations

Key agreement protocols are usually evaluated under the following security properties: (i) known key security – if a key generated in a specific session is compromised, it should not affect any key generated in another session; (ii) key-compromise impersonation resilience – if the long-term private key of an entity is compromised, it should not allow the adversary to impersonate other entities to the one whose key got leaked; (iii) unknown key-share resilience – an entity should not be coerced into establish a key with a second one thinking it is establishing a key with a third one; (iv) key control resilience – no attacker should be capable to force pre-selected values in an execution of the protocol; and (v) forward secrecy – the compromise of the long-term private keys of an any of the parties involved in the key agreement or the trusted authority should not compromise secrecy of past sessions, i.e., the adversary should not be able to generate past session keys if the long-term private key is compromised.

As we based the *SessionKey* protocol (Protocol 4.1) on the work of [81] and [29], we heritage all its security properties. Therefore, based on the proof presented in [29], the *Session-*

Key protocol guarantees known key security, and resilience to key-compromise impersonation, unknown key-share, and key control. The protocol, however, can guarantee forward secrecy only if an adversary compromises at most one of the two long-term private keys of the entities directly involved in the protocol, not both and neither the trusted authority's private key. In AoT context, the only protocol that requires forward secrecy is the *KeyIssue*, which allows a device D to request the server, i.e., the trusted authority, its sensitive cryptographic material (private key) related to its domain. In this case, the session key used to protect such sensitive data is established through the *SessionKey* protocol. We observe that, in this case, if only the long-term private-key of D is compromised, the adversary is not able to generate previously session keys due to *SessionKey* forward secrecy property, i.e., the adversary is not able to reveal the new private key D just received. However, if such an adversary manages to compromise the other device involved in the protocol, i.e., the trusted authority, it does not matter if the device is compromised or not because, as we mentioned in Section 4.1, the escrow inside each individual domain will make the whole domain, including the device D, already compromised. Therefore, in AoT context the *SessionKey* limitation is tolerated.

4.2.2 Long-TermKeyAgreement

In AoT's session key establishment strategy, the *Long-TermKeyAgreement* protocol (Protocol 4.2) is used to establish a common long-term pairwise key $k^{LT}_{A,B}$. In practice, devices A and B that operate each other in a Home domain, previously establish a common long-term pairwise key $k^{LT}_{A,B}$, which is the source of the non-expensive session key derivation process in each operation between the devices in the *Functioning* stage of their life-cycle. The *Long-TermKeyAgreement* protocol must be executed at least once, e.g., in the first occurrence of communication between two devices. Subsequent protocol executions are used to renew the key $k^{LT}_{A,B}$ from time to time. We do not detail each step of Protocol 4.2 because it is identical to the *SessionKey* protocol detailed in the previous section, we name it differently to make its purpose clear.

```
$\textsc{Long-TermKeyAgreement}(device\ A,\ device\ B)$
1.          $A$ : $P^{\mathrm{I}}_{B,\mathcal{H}} := \textsc{Map}_1(id_{B,\mathcal{H}}) \cdot G^{\mathrm{I}}_{1,\mathcal{H}} + P^{\mathrm{I}}_{H,\mathcal{H}}$
2.          $A$ : $\mathsf{R}_A := \mathrm{x}_A \cdot P^{\mathrm{I}}_{B,\mathcal{H}}$
3.  $A \rightarrow B$ : $id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathrm{n}_A, \mathsf{R}_A$, long_term_key_req
4.          $B$ : $P^{\mathrm{I}}_{A,\mathcal{H}} := \textsc{Map}_1(id_{A,\mathcal{H}}) \cdot G^{\mathrm{I}}_{1,\mathcal{H}} + P^{\mathrm{I}}_{H,\mathcal{H}}$
5.          $B$ : $\mathsf{R}_B := \mathrm{x}_B \cdot P^{\mathrm{I}}_{A,\mathcal{H}}$
6.          $B$ : $k^{\mathrm{ID}}_{B,A} := \textsc{Map}_2(id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_A, S^{\mathrm{I}}_{B,\mathcal{H}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{H}}, G^{\mathrm{I}}_{2,\mathcal{H}})^{\mathrm{x}_B})$
7.          $B$ : $k^{\mathrm{MAC}}_{B,A} := \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\mathrm{ID}}_{B,A}}$
8.          $B$ : $k^{LT}_{B,A} := \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_{k^{\mathrm{ID}}_{B,A}}$
9.  $B \rightarrow A$ : $id_{B,\mathcal{H}}, id_{A,\mathcal{H}}, \mathrm{n}_B, \mathsf{R}_B$, long_term_key_ack, $\textsc{Mac}(\mathrm{n}_A \mid \mathsf{R}_A)_{k^{\mathrm{MAC}}_{B,A}}$
10.         $A$ : $k^{\mathrm{ID}}_{A,B} := \textsc{Map}_2(id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_B, S^{\mathrm{I}}_{A,\mathcal{H}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{H}}, G^{\mathrm{I}}_{2,\mathcal{H}})^{\mathrm{x}_A})$
11.         $A$ : $k^{\mathrm{MAC}}_{A,B} := \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\mathrm{ID}}_{A,B}}$
12.         $A$ : $k^{LT}_{A,B} := \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_{k^{\mathrm{ID}}_{A,B}}$
13. $A \rightarrow B$ : $id_{A,\mathcal{H}}, id_{B,\mathcal{H}}$, long_term_key_ack, $\textsc{Mac}(\mathrm{n}_A \mid \mathsf{R}_A \mid \mathrm{n}_B \mid \mathsf{R}_B)_{k^{\mathrm{MAC}}_{A,B}}$
```

Where the new symbols denote:

$\mathcal{H}$:	Home domain
H :	Home server
long_term_key_req, _ack :	request and acknowledgment labels
$k^{LT}_{X,Y}$:	common long-term pairwise key established by X and Y

Protocol 4.2: Long-TermKeyAgreement.

4.2.3 SessionKeyDerivation

The *SessionKeyDerivation* protocol (Protocol 4.3), in turn, is a protocol used to derive session keys from a pairwise common key k. The protocol is conceived especially to be used in a context where extra computational burden to establish a session key cannot be tolerated, i.e., the higher-level protocol involved in the session already impose a considerable computational burden to resource-constrained devices. In practice, the *SessionKeyDerivation* protocol is used to derive a session key from the long-term pairwise key established by the *Long-TermKeyAgreement* protocol. We use the protocol also to derive a session key from an ephemeral key in a specific context of execution of the *KeyIssue* protocol during the *Deployment* stage, detailed in Section 4.2.4.

The protocol starts with a device A generating a nonce n_A and sending it with a session key derivation request label to the intended communication device B (step 1). Upon receiving such a message, B generates a nonce n_B and derives a confirmation MAC key $k^{\mathrm{MAC}}_{B,A}$ from k and the concatenation of n_A received in step 1 and n_B (step 2). Then, in step 3, B derives the session key $k_{B,A}$ from k and the concatenation of nonce n_B with n_A. In step 4, B responds A with the nonce n_B, the session key derivation acknowledgment label, and uses the MAC key $k^{\mathrm{MAC}}_{B,A}$ to calculate the MAC of all this content concatenated with nonce n_A received in step 1. After receiving the response, A derives the $k^{\mathrm{MAC}}_{A,B}$ using k and $\mathrm{n}_A \mid \mathrm{n}_B$ and verifies the MAC of the message received in step 5 (not explicitly shown in the protocol). If the verification fails, A aborts the protocol execution. Otherwise, A derives the session key $k_{A,B}$ from k and $\mathrm{n}_B \mid \mathrm{n}_A$ (step 6), and sends B an acknowledgment label and a MAC of it concatenated with n_A and n_B (step 7). Upon receiving the final message, B verifies the MAC (also not shown in the protocol), if it fails, B aborts the protocol execution. Otherwise, the session is established. Analogously to the case of the new *SessionKey* protocol, we also allow the higher-level protocols that use the *SessionKeyDerivation* protocol to leverage not only the session key $k_{A,B}$ but also the nonces generated during the session establishment as a session stamp to authenticate the protocol messages until their conclusion, which we consider the end of a session.

```
SESSIONKEYDERIVATION($device\ A$, $device\ B$, $key\ k$)
1. $A \rightarrow B$ : $id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathrm{n}_A$, session_key_derivation_req
2.            $B$ : $k^{\mathrm{MAC}}_{B,A} := \mathrm{PRF}(\mathrm{n}_A \mid \mathrm{n}_B)_k$
3.            $B$ : $k_{B,A} := \mathrm{PRF}(\mathrm{n}_B \mid \mathrm{n}_A)_k$
4. $B \rightarrow A$ : $id_{B,\mathcal{H}}, id_{A,\mathcal{H}}, \mathrm{n}_B$, session_key_derivation_ack, $\mathrm{MAC}(\mathrm{n}_A)_{k^{\mathrm{MAC}}_{B,A}}$
5.            $B$ : $k^{\mathrm{MAC}}_{A,B} := \mathrm{PRF}(\mathrm{n}_A \mid \mathrm{n}_B)_k$
6.            $B$ : $k_{A,B} := \mathrm{PRF}(\mathrm{n}_B \mid \mathrm{n}_A)_k$
7. $A \rightarrow B$ : $id_{A,\mathcal{H}}, id_{B,\mathcal{H}}$, session_key_derivation_ack, $\mathrm{MAC}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\mathrm{MAC}}_{A,B}}$
```

Where the symbols denote:

session_key_derivation_req, _ack :	request and acknowledgment labels

Protocol 4.3: SessionKeyDerivation.

4.2.3.1 Security Considerations

The protocol *SessionKeyDerivation* provides the following properties: (i) known key security – as the session key is created from random nonces, there is no connection between keys generated in different sessions. Therefore, if a key generated in a specific session is compromised, it does not affect any key generated in another session; (ii) unknown key-share resilience – a device cannot be coerced into establishing a session key with a second device thinking it is establishing a key with a third one because the long-term key is pairwise; and (iii) key control resilience – as both devices have their contributions to the session key confirmed in MACs, there is no attacker capable to force pre-selected values in execution of the protocol.

As discussed in works that propose strategies where the session key is derived from long-term keys [12], the *SessionKeyDerivation* protocol does not provide any forward secrecy, i.e., the compromise of the key k leads to the compromise of all past session keys established between A and B that were derived from k. Forward secrecy is indeed very important, however, it might not be a requirement in some scenarios, for instance, when only authentication is required [24]. As we mentioned, in AoT, the *SessionKeyDerivation* protocol is used to establish session keys in two scenarios. The most common one is during a *Functioning* stage (Section 4.6). In the context of *Functioning*, confidentiality is not a requirement, once the protocol that governs the stage must guarantee only authentication and access control over the execution of operations. Hence, session keys from the past give no advantage to an adversary, because the operation has already been executed or not. That is why the lack forward secrecy is not a problem.

The protocol *SessionKeyDerivation* also does not provide key-compromise impersonation resilience because once the long-term pairwise key is compromised, the adversary can impersonate the partner on the key to a device. Under a *Functioning* perspective, when an attacker compromises the long-term key, he has two possibilities: (i) the impersonation of the device A that requests the operation to the device B or (ii) the impersonation of the device B owner of the operation to the requester A. If an adversary impersonates A, as we present in *Functioning* (Stage 4.4), he stars by requesting an operation execution on the device B, however during the protocol execution he must prove that he has a subset of attributes needed to satisfy the operation's predicate. In this case, the adversary must compromise not only the long-term key, but also the attribute-based private key of A ($S^{\mathbb{A}}_{A,\mathcal{H}}$) to have the signature successfully checked and the operation executed. On the other hand, with $k^{LT}_{A,B}$ possession, the adversary is able to impersonate B and acknowledge to A the execution of any requested operation on B. However, in many cases, the user who is expecting the operation to be executed will be able to detect that it has actually not been executed and notice that something is wrong. As a mitigation, we propose to renew the long-term key $k^{LT}_{A,B}$ frequently, reducing the attack window opportunities to explore the security issues discussed. This approach can be applied depending on the application, in which during idle periods the devices can enroll on *Long-TermKeyAgreement* protocol to renew the long-term keys.

The other scenario where the *SessionKeyDerivation* protocol is used is during a *Deployment*, when a device still does not have any keys in the Home domain. In this case, the device being deployed generates, in the beginning of the stage, an ephemeral key that is shared with the Home server in a secure way. This key is then, used to derive the session key for the *KeyIssue* protocol. This scenario requires confidentiality. However, as an ephemeral key is used to establish this single session, there is no lack of forward secrecy neither key-compromise impersonation is a problem.

4.2.4 Key Issue

Auxiliary protocol *KeyIssue* (Protocol 4.4) objective is to allow a device D to request to the server S its sensitive cryptographic material related to the domain $\mathcal{Z}$ in the cryptosystem Y. The protocol starts by analyzing in which context it is being executed (step 1): during the *Deployment* stage (Stage 4.3), when the device D does not have the IBC key of domain $\mathcal{H}$, or when the sensitive information has already been issued to D. In the first case, the device D has no private key that allows it to enroll in a session key establishment protocol with server S using *SessionKey* protocol (Protocol 4.1). However, as the device D is under a *Deployment* execution, it has already generated and shared the ephemeral key $k_{D,S}$ with S (Stage 4.3 – steps 3 and 7, respectively). Then, D and S (home server H in this case) establish a session key using *SessionKeyDerivation* protocol (Protocol 4.3) with the ephemeral key $k_{D,S}$ as the derivation key (step 2 in the first block of the *if* clause). In the second case, a *KeyIssue* for the IBC in the domain $\mathcal{H}$ has already been executed between device D and server S, which means the device has already received its cryptographic material that allows it to enroll in a *SessionKey* protocol with server S to establish the session key (step 2 in the second block of the *if* clause).

```
KeyIssue($device\ D$, $server\ S$, $domain\ \mathcal{Z}$, $cryptosystem\ \mathrm{Y}$)
1.          $D$ :  if $S^{\mathrm{I}}_{D,S}$ does not exist:
2.       $D, S$ :  SessionKeyDerivation($D$, $S$, $k_{D,S}$)
                   otherwise:
2.       $D, S$ :  SessionKey($D$, $S$)

3. $D \rightarrow S$ :  issue_req, Auth-Enc$(\mathrm{n}_S \mid \mathrm{n}_D)_{k_{D,S}}$
4. $S \rightarrow D$ :  $S^{\mathrm{Y}}_{D,\mathcal{Z}}$, issue_ack, Auth-Enc$(\mathrm{n}_S \mid \mathrm{n}_D)_{k_{D,S}}$
```

Where the symbols denote:

issue_req, _ack :	request and acknowledgment labels
msg, Auth-Enc$(m)_k$:	authenticated encryption over the message msg appended to m using key k

Protocol 4.4: KeyIssue.

Having established the session, in step 3, device D sends an authenticated encrypted message formed by a key issue request label concatenated with nonces n_D and n_S. Upon receiving such a message, S runs the authenticated decryption algorithm over the message (not explicitly shown in the protocol), if the decryption and verification of the message fail, S aborts the protocol execution. Otherwise, S sends D an authenticated encrypted message formed by D's private key of domain $\mathcal{Z}$ related to cryptosystem Y and a key issue acknowledgment label appended to the nonces n_D and n_S (step 4). Upon receiving such a message, D runs the authenticated decryption algorithm over the message (not explicitly shown in the protocol), if the decryption and verification of the message fail, D aborts the protocol execution. Otherwise, D's private key has been successfully issued.

4.2.5 Binding and Unbiding

Binding (Protocol 4.5) binds a device D to a user U in the Cloud domain. In step 1, the protocol *SessionKey* is invoked to establish $k_{D,C}$ between device D and cloud server C. The nonces created at session establishment are used until end of the session. In step 2, the device D requests the clould server C to bind itself to the user U with identity $id_{U,\mathcal{C}}$ in the Cloud domain by sending a message signed with its IBC signing key in cloud domain containing the request label and D's identity concatenated with the nonces generated during session establishment. Upon receiving the message, and successfully verifying the signature, the cloud server C binds U to D (step 3). In step 4, C acknowledges the operation by authenticating the message with the session key.

$\textsc{Binding}(\mathit{device}\ D,\ \mathit{user}\ U)$

1.	$D, C:$	$\textsc{SessionKey}(D, C)$
2.	$D \rightarrow C:$	bind_req, $id_{U,\mathcal{C}}$, $\textsc{Sig}(\mathrm{n}_C \mid \mathrm{n}_D)_{S^{\mathrm{I}}_{D,\mathcal{C}}}$
3.	$C:$	binds U to D
4.	$C \rightarrow D:$	bind_ack, $\textsc{Mac}(\mathrm{n}_C \mid \mathrm{n}_D\ )_{k_{D,C}}$

Where the new symbols denote:

$C:$	Cloud server
bind_req, _ack :	request and acknowledgment labels
$\mathcal{C}:$	Cloud domain

Protocol 4.5: Binding.

Unbinding (Protocol 4.6) is analogous to and performs the same steps as the *Binding* protocol. The main difference is that it unbinds rather than binds device D and user U. Besides, in step 4 the cloud server C sends D a pin'_B, which is set by D as a new access pin for a new deployment in case of an ownership reassignment (Section 4.8.1).

```
UNBINDING($device\ D$, $user\ U$)
  1.      $D, C$ :  SESSIONKEY($D$, $C$)
  2.  $D \rightarrow C$ :  unbind_req, $id_{U,C}$, SIG($\mathrm{n}_C \mid \mathrm{n}_D$)$_{S^{\mathrm{I}}_{D,C}}$
  3.          $C$ :  unbinds $U$ to $D$
  4.  $C \rightarrow D$ :  unbind_ack,$\mathrm{pin}'_B$, MAC($\mathrm{n}_C \mid \mathrm{n}_D$ )$_{k_{D,C}}$
```

Where the new symbols denote:

unbind_req, _ack :	request and acknowledgment labels

Protocol 4.6: Unbinding.

4.3 Pre-Deployment

Pre-Deployment (Stage 4.1) takes place at the factory and loads the Cloud domain's cryptographic material into devices. This material will be used, further, to establish a remote communication channel between devices and the manufacturer. Such a channel allows the execution of critical procedures by the manufacturer (e.g., software or firmware updates). Besides, it allows devices to run the auxiliary protocols with the Cloud server.

First, the Cloud server C generates the device D's identity $id_{D,C}$ (step 1) and, from that, C derives D's IBC private key $S^{\mathrm{I}}_{D,C}$ (step 2). Next, the Cloud server C generates an access PIN pin_D and loads them in tandem with all the remaining cryptographic material into device D (step 3). Note that the loading is secured via a physical channel, ensuring the communication is both confidential and authenticated. Finally, the Cloud server C deletes $S^{\mathrm{I}}_{D,C}$ and ships D to its trader T_D (step 4).

```
PRE-DEPLOYMENT($Device\ D$)
  1.           $C$ :  $id_{D,C} := D's\ serial\#$
  2.           $C$ :  $S^{\mathrm{I}}_{D,C}$ := GEN$^{\mathrm{I},\mathcal{C}}$($\mathrm{secret}^{\mathrm{I}}_{\mathcal{C}}$, $id_{D,C}$)
  3.    $C \rightarrow D$ :  PHY($id_{D,C} \mid S^{\mathrm{I}}_{D,C} \mid \mathrm{pin}_D$)
  4.   $C \Rrightarrow T_D$ :  $D$
```

Where the new symbols denote:

GEN$^{\mathrm{X},\mathcal{Z}}(s, i)$:	private key generator of cryptosystem X, domain $\mathcal{Z}$ using secret s and information i
$\mathrm{secret}^{\mathrm{X}}_{\mathcal{Y}}$:	secret parameter of cryptosystem X, domain $\mathcal{Y}$
PHY(m) :	m secured via a physical channel
pin_X :	PIN to grant acess to X
$\Rrightarrow$:	secure shipping
T_X :	trader of X

Stage 4.1: Pre-deployment.

4.4 Ordering

Ordering (Stage 4.2) illustrates a user U ordering a device D from trader T_D. Here, all digital communication is secured via TLS. The user U places the order and pays for the device (step 1). The trader T_D, in exchange, sends the user U an acknowledgment (step 2). At this point, the Trader lets the Cloud server C know about the order (step 3), and the Cloud server, in turn, sends the user U a PIN pin_D (step 4). Last, the Trader ships the device D out to user U's home (step 5). The PIN pin_D will be used in the *Deployment* (Stage 4.3) to grant access to D.

```
ORDERING(device D, user U)
  1.  $U \rightarrow T_D$ :  TLS($ | order_req)
  2.  $T_D \rightarrow U$ :  TLS(order_ack)
  3.  $T_D \rightarrow C$ :  TLS(D | U)
  4.  $C \rightarrow U$ :  TLS($\text{pin}_D$)
  5.  $T_D \Rrightarrow U$ :  $D$
```

Where the new symbols denote:

TLS(m) :	m protected via TLS
\$:	payment
order_req, _ack :	request and acknowledgment labels

Stage 4.2: Ordering.

4.5 Deployment

Deployment (Stage 4.3) bootstraps the security of devices in their Home domains. It paves the way for the protocols and is thus paramount for AoT. Here, the *root* user U_r—most likely, the household owner—and his personal device D_{U_r} play a key role. More precisely, D_{U_r} acts like a trusted bridge between the device being deployed and the Home server.

To set up a new device D, U_r enters D's access PIN pin_D onto D itself, which puts D in deployment mode (step 1). Next, in step 2, user U_r uses her device D_{U_r} to send to device D the root user's identity $id_{U_r,\mathcal{H}}$ in the domain $\mathcal{H}$ and the Home server H IBC public key $P^{\text{I}}_{H,\mathcal{H}}$. In step 3, the device D generates an ephemeral (i.e., valid for only a short period) pairwise key $k_{D,H}$ at random, and prepares a message to be sent to D_{U_r} in step 4. First, D encrypts $k_{D,H}$ using H's IBC public key $P^{\text{I}}_{H,\mathcal{H}}$ and concatenates the resulting ciphertext to the information about all its available operations. The message is then sent in step 4. In step 5, the root user U_r sets in her device D_{U_r} the identity $id_{D,\mathcal{H}}$ of device D, the owner user U_D of D, the attributes $\mathbb{A}_D$ of the device, and the set of predicates $\mathbb{Y}_D$ that any other device in the domain needs to satisfy to execute each one of D's operations. (Recall from Section 2.4 a predicate is a signing policy.) We assume communication in these first steps is made via channels that are not only secure but adequate for the device D being deployed (e.g., wireless). As the *Deployment* protocol runs behind closed doors (i.e., at the device owner's home), transmission mechanisms like typing on a keyboard on the device or scanning QRCodes could be carefully employed to exchange data securely. (These are the mechanisms we use in our prototype, but other approaches are possible.) Note this strategy permits one to set up even bulky devices (like a fridge) without resorting to a cabled connection or requiring that the device is physically close to the Home server.

$$\begin{array}{rrl}
1. & U_r \rightarrow D: & \textsc{Phy}(\text{pin}_D) \\
2. & D_{U_r} \rightarrow D: & \textsc{Phy}(id_{U_r,\mathcal{H}} \mid P^{\text{I}}_{H,\mathcal{H}}) \\
3. & D: & \text{generates ephemeral } k_{D,H} \\
4. & D \rightarrow D_{U_r}: & \textsc{Phy}(info_D \mid \textsc{Enc}(k_{D,H})_{P^{\text{I}}_{H,\mathcal{H}}}) \\
5. & U_r \rightarrow D_{U_r}: & \textsc{Phy}(id_{D,\mathcal{H}} \mid U_D \mid \mathbb{A}_D \mid \mathbb{Y}_D)
\end{array}$$

The protocol proceeds with the root device D_{U_r} requesting the deployment of the device D to the server H. To this end, D_{U_r} first establishes a session key $k_{D_{U_r},H}$ with H using the *SessionKey* (Protocol 4.1). Besides the session key that will be used later, the nonces generated at session key establishment are also used until the end of the *Deployment*. In step 7, D_{U_r} requests H a deploy of D by sending a message signed with its IBC signing key containing the request label, all of D's information set by the the root user, the encrypted pairwise ephemeral key $k_{D,H}$ generated by D in step 3, and the nonces generated during session establishment. Our key insight in this stage is to use the root device D_{U_r} as an authentication bridge between the new device D and the Home server H. Particularly, the root device D_{U_r} "blindly signs"[2] the recently generated pairwise key $k_{D,H}$.

$$\begin{array}{rrl}
6. & D_{U_r}, H: & \textsc{SessionKey}(D_{U_r}, H) \\
7. & D_{U_r} \rightarrow H: & \text{deploy\_req}, id_{D,\mathcal{H}}, \mathbb{A}_D, \mathbb{Y}_D, info_D, \\
 & & \textsc{Enc}(k_{D,H})_{P^{\text{I}}_{H,\mathcal{H}}}, \textsc{Sig}(\text{n}_H \mid \text{n}_{D_{U_r}})_{S^{\text{I}}_{D_{U_r},\mathcal{H}}}
\end{array}$$

In steps 8, 9, and 10 the Home server H generates Home and "Inter" domain IBC and ABC private keys for the device D based on all the information received from D_{U_r} in step 7.

$$\begin{array}{rrl}
8. & H: & S^{\text{I}}_{D_U,\mathcal{H}} \coloneqq \textsc{Gen}^{\text{I},\mathcal{H}}(\text{secret}^{\text{I}}_{\mathcal{H}}, id_{D_U,\mathcal{H}}) \\
9. & H: & S^{\text{I}}_{D_U,\mathcal{I}} \coloneqq \textsc{Gen}^{\text{I},\mathcal{I}}(\text{secret}^{\text{I}}_{\mathcal{I}}, id_{D_U,\mathcal{H}}) \\
10. & H: & S^{\text{A}}_{D_U,\mathcal{H}} \coloneqq \textsc{Gen}^{\text{A},\mathcal{H}}(\text{secret}^{\text{A}}_{\mathcal{H}}, \mathbb{A}_{D_U})
\end{array}$$

Last, in steps 11, 12, and 13 the Home server H issues IBC and ABC private keys to the device D using *KeyIssue* protocol (Protocol 4.4). More specifically, as D does not have any

[2]We use double quotes because a *blind signature* refers to a specific cryptographic construction [109].

cryptographic material in the Home domain, the *KeyIssue* in step 11 is secured by the ephemeral key $k_{D,H}$ generated by device D in step 3. In steps 11 and 12, on the other hand, as the device D received its IBC private key for the Home domain in the previous step, the *KeyIssue* can already be executed in a session established by the *SessionKey* protocol (Protocol 4.1).

$$\begin{array}{rrl} 11. & D: & \textsc{KeyIssue}(D, H, \mathcal{H}, \mathrm{I}) \\ 12. & D: & \textsc{KeyIssue}(D, H, \mathcal{I}, \mathrm{I}) \\ 13. & D: & \textsc{KeyIssue}(D, H, \mathcal{H}, \mathrm{A}) \end{array}$$

Then, H broadcasts D's information (identity, attributes, operations, predicate sets, and owner) in the domain (step 14) to inform all devices about the recent deployed device D. At this point, device D is granted access to the Internet, e.g., through the Home domain's Wi-Fi router. Device D then binds itself (step 15) using *Binding* (Protocol 4.5) to its owner user U_D defined by U_r in step 5. Finally, H acknowledges D_{U_r} the deployment of D (step 16), concluding the session established in step 6.

$$\begin{array}{lrl} 14. & H \Rightarrow \mathbb{G}_H: & \mathbb{Y}_{\mathbb{G}_H}, info_{\mathbb{G}_H}, \textsc{Sig}_{S^{\mathrm{I}}_{H,\mathcal{H}}} \\ 15. & D: & \textsc{Binding}\ (D, U_D) \\ 16. & H \rightarrow D_{U_r}: & \mathsf{deploy\_ack}, \textsc{Mac}(\mathrm{n}_H \mid \mathrm{n}_{D_{U_r}})_{k_{D_{U_r},H}} \end{array}$$

Root device deployment. It is clear from the above description the root device D_{U_r} cannot itself follow the protocol, as it plays a key role in the whole process. In fact, D_{U_r}—and D_{U_r} alone—needs to be deployed using a secure channel to the server. We believe this is an easy task since D_{U_r} is most likely a smartphone and can be easily connected to the server by using a cabled connection like USB. Note also that this procedure is carried out only once.

```
$\textsc{Deployment}(\mathit{device}\ D)$
1.   $U_r \to D$ :           $\textsc{Phy}(\mathrm{pin}_D)$
2.   $D_{U_r} \to D$ :       $\textsc{Phy}(id_{U_r,\mathcal{H}} \mid P^{\mathrm{I}}_{H,\mathcal{H}})$
3.   $D$ :                   generates ephemeral $k_{D,H}$
4.   $D \to D_{U_r}$ :       $\textsc{Phy}(info_D \mid \textsc{Enc}(k_{D,H})_{P^{\mathrm{I}}_{H,\mathcal{H}}})$
5.   $U_r \to D_{U_r}$ :     $\textsc{Phy}(id_{D,\mathcal{H}} \mid U_D \mid \mathbb{A}_D \mid \mathbb{Y}_D)$
6.   $D_{U_r}, H$ :          $\textsc{SessionKey}(D_{U_r}, H)$
7.   $D_{U_r} \to H$ :       deploy_req, $id_{D,\mathcal{H}}, \mathbb{A}_D, \mathbb{Y}_D, info_D,$
                             $\textsc{Enc}(k_{D,H})_{P^{\mathrm{I}}_{H,\mathcal{H}}}, \textsc{Sig}(\mathrm{n}_H \mid \mathrm{n}_{D_{U_r}})_{S^{\mathrm{I}}_{D_{U_r},\mathcal{H}}}$
8.   $H$ :                   $S^{\mathrm{I}}_{D_U,\mathcal{H}} := \textsc{Gen}^{\mathrm{I},\mathcal{H}}(\mathrm{secret}^{\mathrm{I}}_{\mathcal{H}}, id_{D_U,\mathcal{H}})$
9.   $H$ :                   $S^{\mathrm{I}}_{D_U,\mathcal{I}} := \textsc{Gen}^{\mathrm{I},\mathcal{I}}(\mathrm{secret}^{\mathrm{I}}_{\mathcal{I}}, id_{D_U,\mathcal{H}})$
10.  $H$ :                   $S^{\mathrm{A}}_{D_U,\mathcal{H}} := \textsc{Gen}^{\mathrm{A},\mathcal{H}}(\mathrm{secret}^{\mathrm{A}}_{\mathcal{H}}, \mathbb{A}_{D_U})$
11.  $D$ :                   $\textsc{KeyIssue}(D, H, \mathcal{H}, \mathrm{I})$
12.  $D$ :                   $\textsc{KeyIssue}(D, H, \mathcal{I}, \mathrm{I})$
13.  $D$ :                   $\textsc{KeyIssue}(D, H, \mathcal{H}, \mathrm{A})$
14.  $H \Rightarrow \mathbb{G}_H$ :   $\mathbb{Y}_{\mathbb{G}_H}, info_{\mathbb{G}_H}, \textsc{Sig}_{S^{\mathrm{I}}_{H,\mathcal{H}}}$
15.  $D$ :                   $\textsc{Binding}\ (D, U_D)$
16.  $H \to D_{U_r}$ :       deploy_ack, $\textsc{Mac}(\mathrm{n}_H \mid \mathrm{n}_{D_{U_r}})_{k_{D_{U_r},H}}$
```

Where the new symbols denote:

U_r :	root user
D_{U_r} :	personal device of the root user
$info_X$:	type and supported operations of X
$\textsc{Enc}(m)_k$:	encryption over m using key k
U_X :	user of device X
$\mathbb{A}_X$:	X's set of attributes
$\mathbb{Y}_X$:	X's set of predicates
deploy_req, _ack :	request and acknowledgment labels
msg, $\textsc{Sig}(m)_k$:	signature over the message msg appended to m using key k
$\mathcal{I}$:	"Inter" domain
A :	Attribute-based cryptosystem
$\Rightarrow$:	broadcast transmission
$\mathbb{G}_X$:	domain's X group of devices

Stage 4.3: Deployment.

4.6 Functioning

Functioning (Stage 4.4) governs the normal operation of devices. In the protocol, a user U requests an operation op on device B using device A (step 1) (e.g., the user uses her smartphone to request the image from the internal camera of a smart fridge). In step 2, the device A verifies if the long-term pairwise key $k^{LT}_{A,B}$ exists, and if not, it invokes the *Long-TermKeyAgreement* protocol. Hence, in step 3, devices A and B establish the new session key $k_{A,B}$ using the *SessionKeyDerivation* protocol (Protocol 4.3). Analogously to the previously described protocols, *Functioning* also leverages the nonces created at session establishment until the end of the session to authenticate the messages (steps 4, 5, and 7). In step 4, device A requests the execution of operation op in B, which responds, in step 5, with the predicate Υ_{op}. The device A proves it has the rights to perform the operation by signing a message formed by the operation description op and the predicate Υ_{op} appended to the nonces created at session establishment with a (sub)set of attributes in the signing key that satisfies the predicate Υ_{op} (step 6). If device B successfully verifies the signature satisfies the predicate, then the requested operation is performed (not explicitly shown in the protocol), and A is acknowledged (step 7). Otherwise, B aborts the protocol execution.

```
FUNCTIONING($user\ U$, $device\ A$, $device\ B$, $operation$ op)
1.          $U$ :  uses $A$ to request op over $B$
2.          $A$ :  if $k^{LT}_{A,B}$ does not exist:
         $A, B$ :  LONG-TERMKEYAGREEMENT($A$, $B$)
3.       $A, B$ :  SESSIONKEYDERIVATION($A$, $B$, $k^{LT}_{A,B}$)
4. $A \to B$ :  op_req, op, $\text{MAC}(n_B \mid n_A)_{k_{A,B}}$
5. $B \to A$ :  op, $\Upsilon_{op}$, $\text{MAC}(n_B \mid n_A)_{k_{A,B}}$
6. $A \to B$ :  op, $\Upsilon_{op}$, $\text{SIG}(n_B \mid n_A)_{S^{A}_{A,\mathcal{H}}}$
7. $B \to A$ :  op_ack, op, $\text{MAC}(n_B \mid n_A)_{k_{A,B}}$
```

Where the new symbols denote:

op_req, _ack :	request and acknowledgment labels
Υ_{op} :	operation op's predicate

Stage 4.4: Functioning.

4.7 Retirement

Broadly speaking, *Retirement* is simply a special operation and the protocol (Stage 4.5) is thus analogous to *Functioning* (Stage 4.4). To run *Retirement*, a user U employs the device A to request the retirement of a device B. The retirement is performed if A's attributes satisfy the predicate. These steps are executed in the same way as other operations are in *Functioning* (Stage 4.4, steps 1–6). Here, we assume that B is owned by U. So, B unbinds itself from its owner U (step 2) and deletes all of its keys from both the Cloud and Home domains (step 3). The retirement is concluded with B signalizing it has been successfully retired, which plays the role of a retirement_ack (step 4), e.g., if the device has a screen, it can display “retired” on its screen. Note that a retired device can no longer be used in the home domain.

```
RETIREMENT($user\ U$, $device\ A$, $device\ B$)
1.  $A$ :  {*Requests retirement*}
2.  $B$ :  UNBINDING($B$, $U$)
3.  $B$ :  deletes $S^{\mathrm{I}}_{B,\mathcal{C}}$, $S^{\mathrm{I}}_{B,\mathcal{H}}$, $S^{\mathrm{A}}_{B,\mathcal{H}}$ and $\mathbb{R}_B$
4.  $B$ :  signalizes ‘retired’
```

Where the new symbol denotes:
$\mathbb{R}_X$: X’s pre-shared keys ring

Stage 4.5: Retirement.

4.8 Complementary Features

In this section, we describe other capabilities in AoT, in particular how to handle device ownership transfer (Section 4.8.1), access revocation (Section 4.8.2), and guest access (Section 4.8.3).

4.8.1 Device Ownership Reassignment

Reassignment (Protocol 4.7) allows a user U to transfer the ownership of a device B he owns to another user V. The protocol is similar to *Retirement* (Stage 4.5) and the same observations there apply. Here, however, the device B does not delete its Cloud domain keys.

In the beginning, U tells the Cloud server C about the reassignment which, in turn, sends the user V a new PIN pin'_B (Protocol 4.7, steps 1–2). The subsequent steps of the protocol are analogous to the first steps of any operation (Stage 4.4, steps 1–6). Next, B unbinds itself from U, which also makes B to receive from C the its new access pin pin'_B. Then, B deletes all of its keys from the Home domain (step 5). The protocol continues with B signalizing ‘ownership reassigned’, which plays the role of a reassignment_ack (step 6) e.g., if the device has a screen, it can display “ownership reassigned” on its screen. Finally, U ships B to V (step 7).

```
REASSIGNMENT($user\ U$, $device\ A$, $device\ B$, $user\ V$)
1.  $U \rightarrow C$ :  TLS(B | $V$)
2.  $C \rightarrow V$ :  TLS(pin$'_B$)
3.          $A$ :  {*Requests reassignment*}
4.          $B$ :  UNBINDING($B, U$)
5.          $B$ :  deletes $S^{\mathrm{I}}_{B,\mathcal{H}}$, $S^{\mathrm{A}}_{B,\mathcal{H}}$ and $\mathbb{R}_B$
6.          $B$ :  signalizes "ownership reassigned"
7.  $U \Rrightarrow V$ :  $B$
```

Where the new symbol denotes:
pin$'_X$: new PIN to grant acess to X

Protocol 4.7: Reassignment.

4.8.2 Key Revocation

Revocation is not the main focus of AoT. In spite of that, as a revocation mechanism in AoT, we follow the scheme proposed by Boneh and Franklin [19] and later improved by Boldyreva *et al.* [15]. The strategy consists of associating keys with their respective expiring dates (a *timestamp*), so they become invalid after a given period of time. In this setup, instead of using solely the map MAP($id_{D,\mathcal{X}}$) of the device D identity $id_{D,\mathcal{X}}$ to form D's keys, the identity is actually concatenated with a timestamp, i.e., MAP($id_{D,\mathcal{X}}$ | timestamp). The granularity of the timestamp can be set by the Home or Cloud servers. For instance, keys may be renewed on a daily basis. With this strategy, keys are instantly revoked as soon as their timestamps expire.

4.8.3 Inter-Domain (Guest) Operation

InterDomainKeyAgreement (Protocol 4.8) allows a device A from a foreign Home domain (i.e., a guest) to agree on a key and thus interoperate with device B from a local Home domain. The protocol is identical to the *SessionKey* protocol (Protocol 4.1), i.e., based on the enhanced [29] identity-based key agreement protocol of [81], which allows key agreement between entities from different domains.

Protocol *InterDomainKeyAgreement* requires the A and B's Home servers to have a new IBC setup, which we call "Inter" domain, with agreed common public parameters: (i) the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with their generators $G^{\mathrm{I}}_{1,\mathcal{I}_A} = G^{\mathrm{I}}_{1,\mathcal{I}_B}$, and $G^{\mathrm{I}}_{2,\mathcal{I}_A} = G^{\mathrm{I}}_{2,\mathcal{I}_B}$, respectively; (ii) the group $\mathbb{G}_T$; (iii) the *bilinear pairing* function $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$; and (iv) the mapping functions

$\textsc{Map}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_r^*$ and $\textsc{Map}_2 : \{0,1\}^* \times \{0,1\}^* \times \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_T \rightarrow \{0,1\}^n$. Besides, it is also required that both devices obtain the public key of their partners' "Inter" domain ($P^{\mathrm{I}}_{\mathcal{I}_A}$ and $P^{\mathrm{I}}_{\mathcal{I}_B}$) in an authenticated manner. We do not detail each step of Protocol 4.8 because it is identical to the *SessionKey* protocol (Protocol 4.1).

```
INTER-DOMAINSESSIONKEY(device A, device B)
1.            $A$ : $P^{\mathrm{I}}_{B,\mathcal{I}} := \textsc{Map}(id_{B,\mathcal{H}}) \cdot G^{\mathrm{I}}_{1,\mathcal{I}_B} + P^{\mathrm{I}}_{\mathcal{I}_B}$
2.            $A$ : $\mathsf{R}_A := \mathsf{x}_A \cdot P^{\mathrm{I}}_{B,\mathcal{I}}$
3.  $A \rightarrow B$ : $id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathsf{n}_A, \mathsf{R}_A$, inter_session_req
4.            $B$ : $P^{\mathrm{I}}_{A,\mathcal{I}} := \textsc{Map}(id_{A,\mathcal{H}}) \cdot G^{\mathrm{I}}_{1,\mathcal{I}_A} + P^{\mathrm{I}}_{\mathcal{I}_A}$
5.            $B$ : $\mathsf{R}_B := \mathsf{x}_B \cdot P^{\mathrm{I}}_{A,\mathcal{I}}$
6.            $B$ : $k^{\mathrm{ID}}_{B,A} := \textsc{Map}_2(id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_A, S^{\mathrm{I}}_{B,\mathcal{I}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{I}_B}, G^{\mathrm{I}}_{2,\mathcal{I}_B})^{\mathsf{x}_B})$
7.            $B$ : $k^{\mathrm{MAC}}_{B,A} := \textsc{Prf}(\mathsf{n}_A \mid \mathsf{n}_B)_{k^{\mathrm{ID}}_{B,A}}$
8.            $B$ : $k_{B,A} := \textsc{Prf}(\mathsf{n}_B \mid \mathsf{n}_A)_{k^{\mathrm{ID}}_{B,A}}$
9.  $B \rightarrow A$ : $id_{B,\mathcal{H}}, id_{A,\mathcal{H}}, \mathsf{n}_B, \mathsf{R}_B$, inter_session_ack, $\textsc{Mac}(\mathsf{n}_A \mid \mathsf{R}_A)_{k^{\mathrm{MAC}}_{B,A}}$
10.           $A$ : $k^{\mathrm{ID}}_{A,B} := \textsc{Map}_2(id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_B, S^{\mathrm{I}}_{A,\mathcal{I}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{I}_A}, G^{\mathrm{I}}_{2,\mathcal{I}_A})^{\mathsf{x}_A})$
11.           $A$ : $k^{\mathrm{MAC}}_{A,B} := \textsc{Prf}(\mathsf{n}_A \mid \mathsf{n}_B)_{k^{\mathrm{ID}}_{B,A}}$
12.           $A$ : $k_{A,B} := \textsc{Prf}(\mathsf{n}_B \mid \mathsf{n}_A)_{k^{\mathrm{ID}}_{B,A}}$
13. $A \rightarrow B$ : $id_{A,\mathcal{H}}, id_{B,\mathcal{H}}$, inter_session_ack, $\textsc{Mac}(\mathsf{n}_A \mid \mathsf{R}_A \mid \mathsf{n}_B \mid \mathsf{R}_B)_{k^{\mathrm{MAC}}_{A,B}}$
```

Where the new symbols denote:

$\mathcal{I}_X$	X's "Inter" domain
$G^{\mathrm{I}}_{1,\mathcal{I}_X}$:	Group $\mathbb{G}_1$ generator of IBC of X's "Inter" domain
$P^{\mathrm{I}}_{\mathcal{I}_X}$:	Public key of X's "Inter" domain
inter_session_req, _ack :	request and acknowledgment labels

Protocol 4.8: Inter-Domain Session Key.

Inter-DomainOperation stage (Stage 4.6) governs the operations of devices from different domains. Because guest users have no attributes in domains they visit, domains must define an attribute "guest" which entitles guest devices to a limited set of permissions in the domain. Foreign operation requests are done in a manner similar to conventional operation requests. The exception is that the device does not have to prove possession of a certain attribute set. Instead, the requested operation is permitted if guests are allowed to perform it. That is, if the predicate to perform the operation is satisfied by the attribute "guest". Besides, no signatures are used during this interaction as the Attribute-Based Cryptosystems of A and B do not interoperate. Instead, they make use of MACs generated using the session key to authenticate. Describing in details, a guest user U starts the process by requesting in her device A the execution of an operation op on a device B (step 1). In step 2, devices A and B establish the a session key $k_{A,B}$ using the *Inter-DomainSessionKey* protocol (Protocol 4.3). In step 3, device A requests the execution of operation op in B. Upon receiving such a message, if the predicate Υ_{op} to execute the operation op is not satisfied by the attribute guest alone (not explicitly shown in the protocol), device B aborts protocol execution. Otherwise, B performs the operation op and acknowledges it to A in step 4.

```
INTER-DOMAINOPERATION($user\ U$, $device\ A$, $device\ B$, $operation$ op)
1.          $U$ :  uses $A$ to request op over $B$
2.       $A, B$ :  INTER-DOMAINSESSIONKEY($A$, $B$)
3.  $A \to B$ :  inter_op_req, op, MAC($n_B \mid n_A)_{k_{A,B}}$
4.  $B \to A$ :  inter_op_ack, op, MAC($n_B \mid n_A)_{k_{A,B}}$
```

Where the new symbols denote:

inter_op_req, _ack :	request and acknowledgment labels

Stage 4.6: Inter-Domain Operation.

Chapter 5

Security Analysis

In this chapter, we perform the security analysis of the protocols that compose each stage of a device life-cycle in AoT. We start with an introduction of the Universal Composability paradigm (Section 5.1), the framework we use to analyze AoT. In Section 5.2, we present the specific model we use in our analysis. In Section 5.3, we describe a functionality for cryptographic primitives, which we extend in Section 5.4, and prove our construction in Section 5.5. Finally, we present in Section 5.6 a functionality that will allow us to detail the security analysis of AoT protocols in Section 5.7.

5.1 Universal Composability

The security analysis of a protocol involves, at its core, proposing a mathematical model to represent all aspects related to a given protocol execution: the definition of the security properties that the protocol aims to satisfy, the adversarial behavior, the environment, adversary, and other protocols interaction, and all the threats that come from the execution environment [23]. In this context, the Universal Composability (UC) [23, 95] is a general paradigm based on the simulation proof technique [84], also known as real/ideal model, that allows a modular design and analysis of cryptographic protocols.

In UC, a protocol execution is orchestrated by an entity called environment generates all the inputs to the parties in a protocol, reads all their outputs, and interacts with the adversary in an arbitrary way. Besides, the environment represents everything that is outside that particular execution and can, directly or indirectly, influence it, for instance, the concurrent execution of the same protocol, noises, other protocols' executions, users, attackers, etc. The idea behind this concept is to keep the protocol security analysis in a stand-alone setting, as used in the simulation-based paradigm, at the same time guaranteeing the secure composition of the protocol in a realistic scenario where an instance of the protocol is executed in concurrency with many other protocols, in a complex environment. This goal is achieved by formulating the security definitions such that they are preserved under a general composition operation [23]. The new operation allows different protocols to be securely composable as modules of wider protocol systems. Therefore, UC can be used as a protocol design technique to put together modules, smaller protocols, already proven universally composable secure inside a specific UC model to build a more complex system that is automatically secure based on the composition theorems of the chosen model. Besides, using the benefits of the modularity, UC also can be used to decompose a protocol system into smaller components, perform individual and simpler security analysis of such components, and then put together the results to prove the security of the whole system.

In UC, the objective of the security analysis is to show that a real protocol that aims to provide some specific secure property is at least as secure as an idealized protocol, called functionality, that provides the desired secure property. The goal is achieved by proving that for every protocol adversary there is an functionality adversary, called simulator, such that an environment cannot distinguish if it is interacting with the protocol and the real adversary or with the functionality and the simulator. Therefore, from the environment perspective, if there is no adversary that can cause any difference between the usage of a real protocol or its security property idealization, the real protocol indeed provides such a property.

There are different models for universal composability, each one with its own underlying computational model, notions of security, and theorems, such as the canonical UC model proposed by Ran Canetti [23], the GNUC [49], and the IITM [61]. In the next section, we detail the IITM, the model we selected to perform the security analysis of AoT protocols.

We chose to analyze AoT under an UC model due to its modular nature. We observe that AoT is designed as a composition of several cryptographic protocols and primitives, therefore, we show that each different component of AoT can be securely composable to be part of the wider system.

5.2 IITM Model

The model for universal composability used in our work is based on the IITM model proposed by [61] that has been being constantly extending in many aspects during the years [62, 63, 65, 66, 67, 68]. The IITM model was selected because, even not directly applicable to AoT protocols, we could extend an existing framework built over the model with great synergy with AoT protocols to support our security analysis.

5.2.1 Computational Model

The computational model used in IITM model is based on Inexhaustible Interactive Turing Machines, which are probabilistic polynomial-time Turing machines with named input and output tapes parameterized with a polynomial.

Analogously to a program source code, a machine M has its own programming composed of internal variables used to process messages received on input tapes and write messages on output tapes. An instance of a machine, in turn, is analogous to a program execution. In the IITM model, every component – environments, protocols, functionalities, and adversaries – are machines or a system of machines.

A system of machines, written as $\mathcal{M} = M_1 \mid \cdots \mid M_k \mid !M'_1 \cdots \mid !M'_{k'}$, is composed of machines M_i, $i \in \{1, ..., k\}$, and M'_j, $j \in \{1, ..., k'\}$, that communicate among themselves using named input and output tapes. If two machines, for instance, M_1 and M_2, have tapes with the same name, say t, they communicate with each other using t, where t is an input tape in M_1 and an output tape in M_2 or the opposite of that. In a system of machines, there are two different sets of tapes, the set in which tapes have matching names in two different machines, which models internal communication, and tapes that have unique names, called external tapes, which allow the communication among different systems.

A machine is activated upon receiving a message on an input tape. A run of a system $\mathcal{M}$ is started by the activation of a special machine called master machine. During the execution of $\mathcal{M}$, only one instance of a machine is active while all the others are waiting for an input. In every run of a system $\mathcal{M}$, an unbounded number of new instances of machines in the scope of a ! operator, called bang operator, can be created, while machines that are not on that scope can have only one instance created. The bang operator is used to model the concurrent execution of multiple instances of a protocol. Any machine M has two modes of execution. When it is activated, upon an input on any of its tapes, M is in the **CheckAddress** mode, which is a mechanism to address the specific instance of M in a system of machines. The computation results in an **accept** or in a **reject**. An **accept** makes the instance to proceed to the **Compute** mode, which is, in turn, concluded with a message written on an output tape, which activates a specific machine in the system, or without any output, which activates the master machine of the system with an empty input. A **reject**, on the other hand, means that the particular instance cannot handle that input, and there are different scenarios depending on the scope of the machine. For example, if a machine M of system $\mathcal{M}$ has an input tape named t, and, in a particular point of a specific run of $\mathcal{M}$, a current active instance of a machine, say M_1, writes the message m on tape t, there are two possible scenarios: (i) **case M**, the machine M is not in the scope of a bang operator; and (ii) **case $!M$**, machine M is in the scope of a bang operator.

(i) **case M**. The first instance of M created in the run of $\mathcal{M}$ is activated. If there is no such an instance created yet, the instance of M is created.

- The instance of M executes in **CheckAddress** mode with m as input on tape t. If the computation results in an **accept**, the instance proceeds to **Compute**. Otherwise, i.e., if the computation results in a **reject**, the master machine of $\mathcal{M}$ is activated with an empty input.

(ii) **case !M.** Suppose there are already some instances of M in the run of $\mathcal{M}$. The first instance of M created in the run of $\mathcal{M}$ is activated.

- The instance of M executes in **CheckAddress** mode with m as input on tape t.
- If the computation results in an **accept**, the instance proceeds to **Compute**. Otherwise, i.e., if the computation results in a **reject**, the next instance of M created on $\mathcal{M}$ is activated.
- The process is repeated until one of existent instances returns **accept** from the computation in **CheckAddress** mode with m as input on tape t.
- If m gets rejected by all current instances of M in $\mathcal{M}$, a new instance of M is created and activated.
- If the new instance of M executes in **CheckAddress** mode with m as input on tape t and returns **accept**, it proceeds to the **Compute**. Otherwise, i.e., if the new instance also rejects m, the new instance is destroyed and the master machine of $\mathcal{M}$ is activated with an empty input.

The term inexhaustible comes from the fact that in both modes of computation, an IITM cannot be exhausted, upon every activation it performs actions. The overall runtime of a system of IITMs is polynomially bounded in the security parameter plus the length of all inputs it processes.

Two systems of machines $\mathcal{M}$ and $\mathcal{N}$ are called indistinguishable ($\mathcal{M} \equiv \mathcal{N}$) if the difference between the probability that $\mathcal{M}$ and $\mathcal{N}$ write 1 on their the **`decision`** tape is negligible.

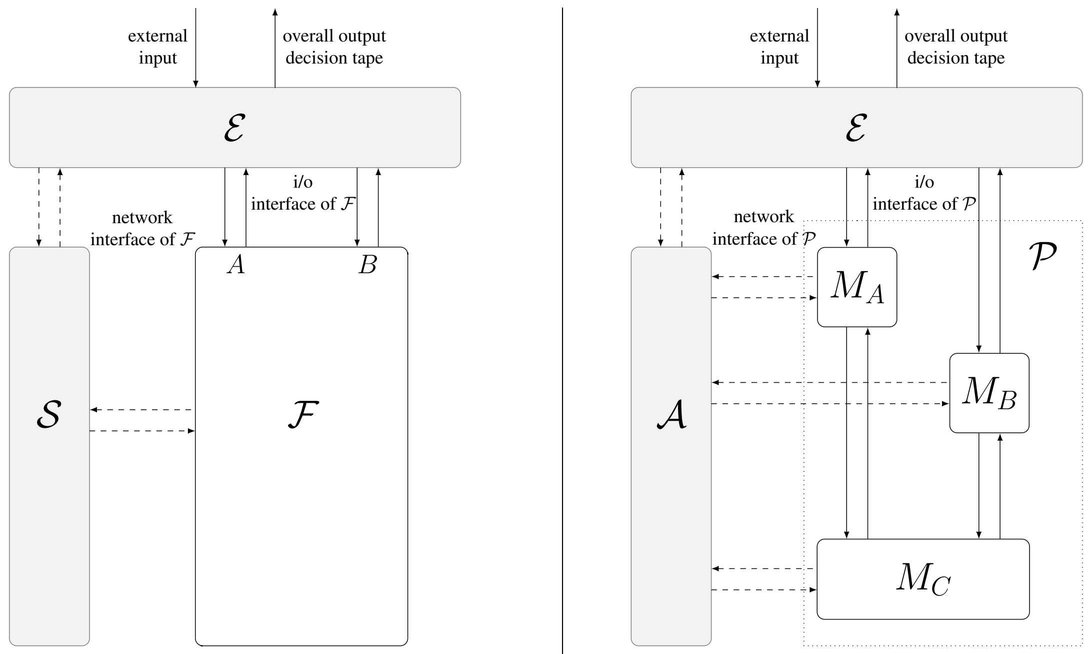
5.2.2 Security Notions

In this section, we present the underlying security notions of the IITM model, that is, the basic components that we will use to model the protocol execution, the adversarial behavior, and the bad interactions with concurrent instances of other, or even the same, protocols that run in a system as well as how we prove the security of a protocol. As there is no perfect model, the IITM poses limitations that will be discussed when it is appropriated.

The IITM model uses the following terminology. The external tapes of machines are divided into i/o tapes, or i/o interface, which models communication between users and protocols or functionalities as well as the internal communication among components of a wider system, and network tapes, or network interface, which model the untrusted network communication and allow the interaction between adversaries and simulators with protocols and functionalities, respectively. The model considers three types of systems: (i) protocol systems, which model real protocols and functionalities and have both i/o and network interfaces; (ii) adversarial systems, which model adversaries and simulators and have only the network interface; and (iii) environmental systems, which model the environments, i.e., the users of protocols and functionalities, and have both i/o and network interfaces. In the model, the environments are the master machines of a whole protocol execution modeling.

In the IITM model, adversary and environment systems are classified as responsive, which means they respond immediately upon receiving a special type of message, called restricting message, on the network. This is a mechanism that allows the exchange of control messages to model real-world behavior, expanding the capabilities of the model. It allows, for instance, to model a direct interference of the adversary on a protocol execution. In a certain point of a execution, for example, the adversary may be asked if a specific instance of a machine or a particular key should be corrupted before continue its execution. Of course this type of request does not exist in practice, i.e., no protocol will contact the adversary to harm its users, however, the model must allow the modeling of corrupted users.

In Figure 5.1a, we depict an IITM system $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$, where the functionality $\mathcal{F}$ ideally implements a two-party protocol with roles A and B. $\mathcal{F}$ has two pairs of i/o tapes (solid lines) to interface with the environment $\mathcal{E}$, one for each protocol role. The network tapes (dashed lines) allow communication between $\mathcal{F}$ and the simulator $\mathcal{S}$. Figure 5.1b, in turn, shows an IITM system $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P}$, where $\mathcal{P} = M_A \mid M_B \mid M_C$ is a two-party real protocol modeling with three machines M_A, M_B, and M_C. The machines M_A and M_B represent, respectively, the roles A and B on the protocol. They have i/o tapes to interface with $\mathcal{E}$ and network tapes that allow communication with the adversary $\mathcal{A}$. The tapes that connect a protocol machine with the adversary are called network because they model an untrusted communication channel, which is controlled by the adversary. Therefore, when there is a message exchange between the roles in the protocol specification ($A \rightarrow B$, for instance), in the modeling the instance M_A writes the message on its network tape, i.e., it gives the message to $\mathcal{A}$. The machine M_C, on the other hand, represents an internal component that acts as a service provider for M_A and M_B using internal i/o interface, but it is not directly accessible for $\mathcal{E}$.


external input
overall output decision tape
$\mathcal{E}$
i/o interface of $\mathcal{F}$
network interface of $\mathcal{F}$
A
B
$\mathcal{S}$
$\mathcal{F}$
external input
overall output decision tape
$\mathcal{E}$
i/o interface of $\mathcal{P}$
network interface of $\mathcal{P}$
$\mathcal{P}$
M_A
$\mathcal{A}$
M_B
M_C

(a) $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$. (b) $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P}$.

Figure 5.1: IITM ideal (a) and real (b) settings modeling.

In this work, we deal with both single- and multi-session versions of protocols and functionalities. In the example above, we represent $\mathcal{P} = M_A \mid M_B \mid M_C$ as the single-session version of a protocol, because neither of the machines of protocol roles A and B are in the scope of the ! operator. This means in a run of $\mathcal{P}$, only one instance of machines M_A and M_B can be created. Therefore, each one of the instances can correspond to a single party in $\mathcal{E}$. Hence, the instances will execute the protocol until it ends and nothing more. In the multi-session modeling of $\mathcal{P}$, denoted by $\mathcal{P} = !M_A \mid !M_B \mid M_C$, an unbounded number of instances of M_A and M_B can be created, i.e., multiple concurrent executions (sessions) of the protocol can be handled, as illustrated in Figure 5.2. Notice that M_C is not in the scope of the bang operator. That is because, in many cases, there is the need to share a service, usually a functionally, among all concurrent executions of a protocol, this topic will be discussed in more details in Section 5.3.6. A functionality $\mathcal{F}$, in turn, as it is an idealization, can implement single- and multi-session versions of a protocol without being in the scope of the bang operator, the system just need an appropriate user identification scheme.

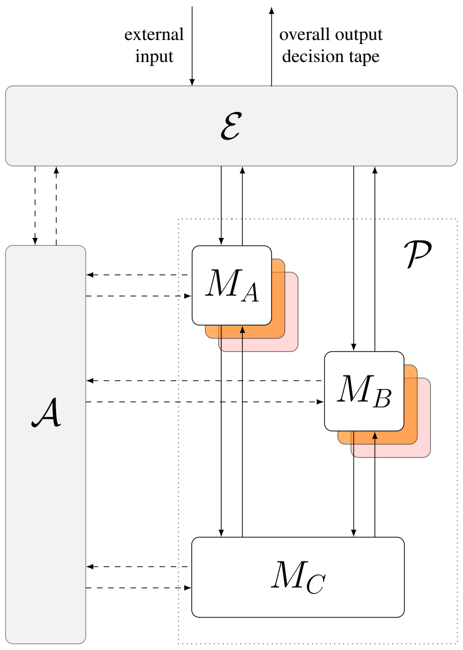


Figure 5.2: Representation of a multi-session two-party real protocol $\mathcal{P} = !M_A \mid !M_B \mid M_C$.

The user identification scheme that we use in this work is based on party identifiers (*pid*) and local session identifiers (*lsid*) locally chosen and managed by the party, i.e., by the environment. In a run of a system, all the instances of machines expect inputs to be prefix with a tuple like (*pid*, *lsid*) from the environment, adversaries, and other machines. Analogously, all of their output messages are prefix with (*pid*, *lsid*). With this in mind, if an instance of a machine M_A that represents a role A in a two-party multi-session protocol outputs in any of its tapes a message with prefix (*pid′*, *lsid′*), it means that, at some point in the run of the system, this instance has executed in **CheckAddress** mode with an input from user (*pid′*, *lsid′*) that returned **accept** and in **Compute** mode the instance wrote the message on an output tape. Therefore, in a run of such a protocol, there will be at most one instance of M_A representing the user (*pid′*, *lsid′*). We usually say such a user is fully identified by the tuple (*pid′*, *lsid′*, *A*). A functionality $\mathcal{F}$, in turn, since it has a specific pair of tapes for each role for the protocol it ideally implements, can run in **CheckAddress** and **accept** messages from all the users (*pid*, *lsid*), i.e., it can handle in a single machine instance all the users in the system. The same way, it can fully identify the users by the tuple (*pid*, *lsid*, *r*), where *r* is the specific role that has its dedicated pair of tapes in $\mathcal{F}$.

We now present how we use the IITM model to prove the security of a real protocol. In a high level, consider that in a real setting, the environment $\mathcal{E}$ interacts with a protocol and the adversary in an arbitrary way, choosing the inputs to the parties in their sessions, and instructing $\mathcal{A}$ which ones to corrupt. At the end of the system run, $\mathcal{E}$ outputs in its decision tape a single bit, which tells whether the environment thinks it has interacted with a protocol $\mathcal{P}$ or with a functionality $\mathcal{F}$ and a simulator $\mathcal{S}$. If a simulator $\mathcal{S}$ can be designed such that no environment $\mathcal{E}$ can tell with non-negligible probability whether it is interacting with $\mathcal{P}$ and $\mathcal{A}$

or $\mathcal{F}$ and $\mathcal{S}$, then $\mathcal{P}$ is as secure as $\mathcal{F}$. A more formal definition of this security notion, called strong simulatability, illustrated in Figure 5.3, is: let $\mathcal{P}$ and $\mathcal{F}$ be, respectively, a real protocol and a functionality modeled in IITM as protocol systems with the same i/o interfaces. Then, $\mathcal{P}$ is at least as secure as $\mathcal{F}$, or $\mathcal{P}$ realizes $\mathcal{F}$ ($\mathcal{P} \leq \mathcal{F}$), if there exists a simulator $\mathcal{S}$ modeled in IITM as an adversarial system, such that $\mathcal{P}$ and $\mathcal{S} \mid \mathcal{F}$ have the same i/o interfaces, and for all environment $\mathcal{E}$ modeled in IITM as an environmental system that connects to the i/o interface of $\mathcal{P}$ and of $\mathcal{S} \mid \mathcal{F}$, it holds true that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$. The relation $\leq$ is reflexive and transitive. Notice that in the strong simulatability, as $\mathcal{E}$ chooses everything for the adversary, the adversary is absorbed by the environment, which plays both roles.

In a security analysis of a protocol $\mathcal{P}$, the general idea to prove $\mathcal{P} \leq \mathcal{F}$ is to build a proper $\mathcal{S}$ that internally simulates $\mathcal{P}$ and then show that for every use case in $\mathcal{E}$ the result is the same.

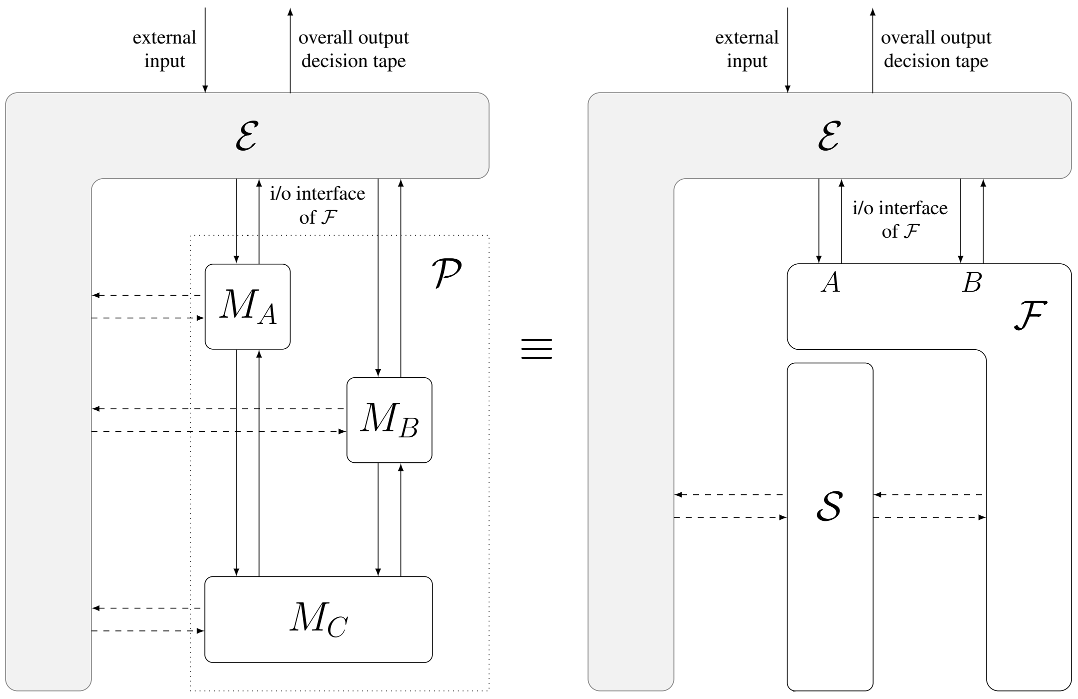


Figure 5.3: IITM security notion.

5.2.3 Theorems

The IITM model has many useful composition theorems. Theorem 5.1, for instance, handles the composition of a limited number of protocol systems.

Theorem 5.1 *([61]) Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be real protocol systems and $\mathcal{F}_1$ and $\mathcal{F}_2$ functionalities such that $\mathcal{P}_1$ realizes $\mathcal{F}_1$ ($\mathcal{P}_1 \leq \mathcal{F}_1$) and $\mathcal{P}_2$ realizes $\mathcal{F}_2$ ($\mathcal{P}_2 \leq \mathcal{F}_2$). Then it holds true that* *the composition of $\mathcal{P}_1$ and $\mathcal{P}_2$ also realizes the composition of the functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$: $\mathcal{P}_1 \mid \mathcal{P}_2 \leq \mathcal{F}_1 \mid \mathcal{F}_2$.*

Theorem 5.2, in turn, guarantees the secure composition of an unbounded number of instances of a real protocol that realizes a functionality. This means the security analysis of a multi-session protocol can be proved by performing the analysis of the single-session version of the protocol. However, the Theorem 5.2 assumes that concurrent sessions of the protocol have disjoint states, i.e., there is no information shared among the sessions. This is a limitation for our purposes, and, as we will need such a result to prove the security of one of AoT protocols, we state another joint state composition theorem in Section 5.3.6.

Theorem 5.2 *([61]) Let $\mathcal{P}$ be the single-session version of a real protocol system and $\mathcal{F}$ a single-session version of a functionality such that $\mathcal{P} \leq \mathcal{F}$. Then it holds true that $!\mathcal{P} \leq !\mathcal{F}$.*

As a corollary of the IITM composition theorems, and using that the relation $\leq$ is reflexive and transitive, we have that: if a real protocol $\mathcal{P}_1$ that has already been proven to realize a functionality $\mathcal{F}_1$ ($\mathcal{P}_1 \leq \mathcal{F}_1$) is composed with another protocol $\mathcal{P}$ to build a more complex protocol $\mathcal{Q} = \mathcal{P} \mid \mathcal{P}_1$, which, in turn, has to be proven to realize a functionality $\mathcal{F}$, we can analyze $\mathcal{Q}$ as $\mathcal{P} \mid \mathcal{F}_1 \leq \mathcal{F}$, where $\mathcal{P}$ ideally uses the functionality $\mathcal{F}_1$. After the analysis, the functionality $\mathcal{F}_1$ can be directly changed to its realization $\mathcal{P}_1$, which gives a real implementation of $\mathcal{Q}$.

5.3 Functionality for Cryptographic Primitives

As discussed in the Section 5.1, the UC models can be used to analyze the security of communication protocols in a modular way, allowing the composition of previously proven protocols to build more complex systems. This is especially interesting if we could support the analysis of real protocols based on a previously proven secure set of cryptographic primitives inside the model. That is the key idea behind the framework proposed by [66]. The authors propose a functionality called $\mathcal{F}_{\text{crypto}}$ to support the execution of cryptographic primitives in an ideal way, which is proved in a realization $\mathcal{P}_{\text{crypto}}$ based on standard cryptographic assumptions. Based on their results, it is possible, for instance, to analyze the security of a real protocol $\mathcal{P}$ that intends to realize a specific functionality $\mathcal{F}$ using $\mathcal{F}_{\text{crypto}}$ as the cryptographic primitives' provider for $\mathcal{P}$ without any reduction proofs regarding the cryptographic primitives, that have already been conducted in the analysis of $\mathcal{P}_{\text{crypto}}$. $\mathcal{F}_{\text{crypto}}$ allows a protocol such as $\mathcal{P}$ to ideally: generate and derive symmetric keys, use symmetric and asymmetric encryption schemes, generate and verify MACs, use a digital signature scheme, and generate fresh nonces.

Although the functionality $\mathcal{F}_{\text{crypto}}$ is equipped with most basic cryptographic primitives, it needs to be extended when a protocol being analyzed in the framework needs an unsupported cryptographic primitive. This is done, for instance, in [62]. The authors add to $\mathcal{F}_{\text{crypto}}$ the mechanisms to support standard Diffie-Hellman key exchange. Similarly, we present in Section 5.4 an extension of $\mathcal{F}_{\text{crypto}}$ to support a set of operations that, in turn, will enable the usage of $\mathcal{F}_{\text{crypto}}$ to build identity-based authenticated key agreement protocols [29, 70, 81, 121] categorized into the same family of protocols proposed by [81].

5.3.1 $\mathcal{F}_{\textbf{crypto}}$ Modeling

In this section, we present the $\mathcal{F}_{\text{crypto}}$ modeling: the general $\mathcal{F}_{\text{crypto}}$ machine interface (Section 5.3.1.1), the identification of users (Section 5.3.1.2), how cryptographic keys are handled (Section 5.3.1.3), how the Diffie-Hellman key exchange is implemented (Section 5.3.1.4), the existent cryptographic operations (Section 5.3.2), how it is parameterized (Section 5.3.5), and initialized (Section 5.3.3).

5.3.1.1 $\mathcal{F}_{\textbf{crypto}}$ Interface

The multi-session version of the functionality for cryptographic primitives [62, 65, 66] is modeled by a single machine $\mathcal{F}_{\text{crypto}}$ that interfaces with an n-party higher-level protocol. The machine $\mathcal{F}_{\text{crypto}}$ has n pairs of i/o tapes, one pair for each protocol role, and a network interface for adversary communication. In Figure 5.4, for instance, we show a diagram representing a two-party multi-session protocol $\mathcal{P} = !M_A \mid !M_B$ with roles A and B, respectively. In this case, M_A and M_B implement their respective roles following the higher-level protocol specification and uses the i/o interfaces of $\mathcal{F}_{\text{crypto}}$ to request the cryptographic operations they need.

5.3.1.2 Users

The functionality $\mathcal{F}_{\text{crypto}}$ expects to receive messages prefixed with party and local session identifiers (*pid*, *lsid*). As the messages are sent and received in tapes specific for each

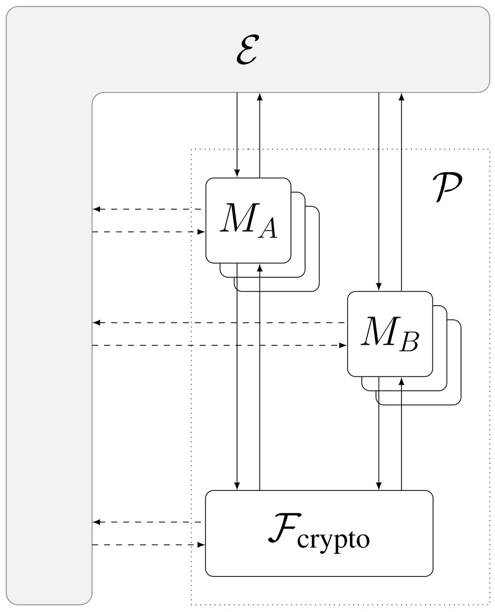


Figure 5.4: Modeling the implementation of a multi-session two-party protocol $\mathcal{P}$ which uses $\mathcal{F}_{\text{crypto}}$ as a cryptographic provider.

protocol role, any user is fully identified by the tuple (*pid*, *lsid*, *r*), where *r* is the role in the protocol.

5.3.1.3 Cryptographic Keys

In $\mathcal{F}_{\text{crypto}}$, a symmetric key, besides its actual value, also have a specific type, which models the security requirement of single-purpose usage of keys. The type `pre-key` represents keys that can be used only to derive other symmetric keys of arbitrary types. Keys of `unauthenc-key` type, in turn, can only perform unauthenticated encryption and decryption. Similarly, `authenc-key` keys can perform only authenticated encryption and decryption, `mac-key` keys can only generate and verify MACs, and `dh-key` represents keys established by a Diffie-Hellman key exchange, which can be used only to derive new symmetric keys of arbitrary types.

A key also has its corruption status, which determines if the corresponding cryptographic operation is performed ideally or without security guarantees. To provide such guarantees, $\mathcal{F}_{\text{crypto}}$ does not reveal the actual value of keys directly to their users, otherwise, $\mathcal{F}_{\text{crypto}}$ could not tell if a key has been "seen" or not by any other user than the keys' owner. Instead, $\mathcal{F}_{\text{crypto}}$ gives the users pointers that reference the keys. Thus, the pointers can be used as parameters of the operations that are performed inside the functionality. Even though, a user is able to retrieve the actual value of a key, which makes the key known outside $\mathcal{F}_{\text{crypto}}$, i.e., corrupted. Therefore, in the case of symmetric keys, $\mathcal{F}_{\text{crypto}}$ needs to keep track whether a key is known outside its domains or not. For that, it uses two sets. The set Keys contains all the keys controlled by the functionality, generated by it or inserted by the users. The set $\mathsf{Keys}_{\mathsf{known}} \subseteq \mathsf{Keys}$, in turn, contains the keys that have become known outside $\mathcal{F}_{\text{crypto}}$ by being explicitly retrieved by its owner, the keys that have been encrypted with a corrupted key, the keys that have not been generated by $\mathcal{F}_{\text{crypto}}$, but explicitly inserted by a user, and the keys that have been corrupted by the adversary.

In $\mathcal{F}_{\text{crypto}}$, the adversary (simulator) provides the actual values of the keys[1] generated in the functionality. However, the corruption status of a key is what tells $\mathcal{F}_{\text{crypto}}$ whether the corresponding cryptographic operation is performed ideally (uncorrupted key) or without security guarantees (corrupted key). In the realization $\mathcal{P}_{\text{crypto}}$ of $\mathcal{F}_{\text{crypto}}$, however, an uncorrupted key will actually not be known by the adversary or the environment. After receiving a key k from the adversary, $\mathcal{F}_{\text{crypto}}$ first checks if there is any collision with its current keys, i.e., if $k \notin \mathsf{Keys}$, and asks for another key until the check passes. Similarly, whenever a user insert a key k in $\mathcal{F}_{\text{crypto}}$, the functionality prevents key guessing of unknown keys by rejecting k if $k \in \mathsf{Keys} \setminus \mathsf{Keys}_{\mathsf{known}}$. In fact, one of the main objective of these sets, besides keeping the track of (un)known keys, is to give $\mathcal{F}_{\text{crypto}}$ a mechanism to guarantee the generation of fresh keys and resistance against guessing of unknown keys, because the functionality has no dependency on a particular algorithm for generating keys, which, in turn, must be implemented by its realization.

$\mathcal{F}_{\text{crypto}}$ also allows higher-level protocols to model setup assumptions. Two or more users can create a pre-shared symmetric key of type `pre-key`, `unauthenc-key`, `authenc-key`, or `mac-key` using a specific operation that has as a parameter the name of the key. Hence, if the users use a common name and type, they will get pointers to the same key in $\mathcal{F}_{\text{crypto}}$.

Asymmetric keys, in turn, are modeled a little bit differently. First of all, there is no interface to generate asymmetric keys using $\mathcal{F}_{\text{crypto}}$. The functionality assumes there is a key distribution scheme, then all the asymmetric keys are provided by the adversary in the initialization phase of $\mathcal{F}_{\text{crypto}}$. Furthermore, the private keys are not accessible to users, not even through pointers. These keys are kept inside the functionality and can be indirectly used by their owners with the corresponding cryptographic operation, specifically decryption or signing. Besides, there is no option to retrieve the actual value of private keys. Public keys, on the other hand, have the actual values returned to users when requested. Last, private keys can be corrupted only by the adversary.

[1]The adversary also provides the values of other secret cryptographic information and also nonces.

5.3.1.4 Diffie-Hellman Key Exchange

Diffie-Hellman (DH) key exchange modeling in $\mathcal{F}_{\text{crypto}}$ is based on general cyclic groups $\mathbb{G}$ of order n and generator G, exponents $e \in \mathbb{Z}_n^*$, and group elements, also called DH shares, $H = G^e \in \mathbb{G}$. In $\mathcal{F}_{\text{crypto}}$, exponents are modeled analogously to keys, the users do not receive the actual value e of an exponents, but a pointer to it. However, the users can get the actual value of its corresponding DH share $H = G^e$. A user with access to an exponent e can combine it with an arbitrary DH share $H = G^d$, not necessarily generated in $\mathcal{F}_{\text{crypto}}$, to create a new symmetric key $k = G^{ed}$. $\mathcal{F}_{\text{crypto}}$ guarantees that if the exponents e and d were created in $\mathcal{F}_{\text{crypto}}$, then the resulting key k is only be accessible by the owners of e and d. The corruption status of exponents are also tracked by sets. The general set Exp contains all exponents, and the set $\mathsf{Exp}_{\mathsf{known}} \subseteq \mathsf{Exp}$ keeps the exponents that become known to the environment. Hence, if an exponent e is unknown, $e \in \mathsf{Exp} \backslash \mathsf{Exp}_{\mathsf{known}}$. An auxiliary set, called $\mathsf{BlockedElements}$, is used to maintain all DH shares that cannot be generated in $\mathcal{F}_{\text{crypto}}$. This set is used to avoid that an exponent being created in $\mathcal{F}_{\text{crypto}}$ does not result in a DH share that has not been created in the functionality, but received by higher-level protocol that wants to use it inside $\mathcal{F}_{\text{crypto}}$ and, hence, has been explicitly blocked in $\mathcal{F}_{\text{crypto}}$ by the higher-level protocol.

Similarly to keys modeling, the adversary also provides the actual value of exponents for $\mathcal{F}_{\text{crypto}}$, which means that the corruption status of the exponents determine if the key derived from them is corrupted or not. In the realization $\mathcal{P}_{\text{crypto}}$ of $\mathcal{F}_{\text{crypto}}$, however, an uncorrupted exponent will actually not be known by the adversary or the environment. Whenever $\mathcal{F}_{\text{crypto}}$ receives a new exponent e from the adversary, it performs some verification before accepting the value. First, it checks if $e \in \mathbb{Z}_n^*$ to guarantee consistency with the DH group. Then, $\mathcal{F}_{\text{crypto}}$ checks if $e \notin \mathsf{Exp}$, modeling fresh exponent generation. Last, $\mathcal{F}_{\text{crypto}}$ verifies if $G^e \notin \mathsf{BlockedElements}$, i.e., it checks if the resulted DH share derived from the generated exponent has not already been blocked in the functionality. If any of the mentioned verification fails, $\mathcal{F}_{\text{crypto}}$ will request the adversary a new value for the exponent until all verification pass successfully.

Keys that are generated using the DH key exchange capability in $\mathcal{F}_{\text{crypto}}$ have the type `dh-key`. These keys can be used only to derive new symmetric keys of other types. In addition, they can be generated only by the `GenDHKey` and `Store` operations. The corruption status of exponents (known or unknown) is what determines the if a a new key of type `dh-key` created from them if corrupted (known) or not. $\mathcal{F}_{\text{crypto}}$ marks a kew key as uncorrupted only if it is generated from two unknown exponents. That is, if any of them is known, or if the DH share used to create the key is not generated in $\mathcal{F}_{\text{crypto}}$, the key is corrupted.

5.3.2 Cryptographic Operations

Bellow, we present the list of cryptographic operations provided by $\mathcal{F}_{\text{crypto}}$, a detailed description of $\mathcal{F}_{\text{crypto}}$'s behavior upon each command is provided in [66] and [62].

- Generate a fresh symmetric key
  Command: $(\texttt{New}, \texttt{type})$
  Users can ask $\mathcal{F}_{\text{crypto}}$ to generate a fresh symmetric key of type $\texttt{type} \in \{\texttt{pre-key}, \texttt{unauthenc-key}, \texttt{authenc-key}, \texttt{mac-key}\}$.

- Establish a pre-shared key
  Command: $(\texttt{GetPSK}, \texttt{type}, \texttt{name})$
  Users can establish pre-shared symmetric keys of type $\texttt{type} \in \{\texttt{pre-key}, \texttt{unauthenc-key}, \texttt{authenc-key}, \texttt{mac-key}\}$ and identified by $\texttt{name} \in \{0,1\}^*$ to model setup assumptions. Different users that use the command with same `type` and `name` parameters have a common pre-shared key.

- Store a key
  Command: $(\texttt{Store}, \texttt{type}, k)$
  Users can store a key k of type `type` in $\mathcal{F}_{\text{crypto}}$. This command allows higher-level protocols to use the functionality with keys that are not created by it. As the key is `known` outside $\mathcal{F}_{\text{crypto}}$, after making sure there is no key guessing of unknown keys, $\mathcal{F}_{\text{crypto}}$ adds the key k to the set $\mathsf{Keys}_{\mathsf{known}}$.

- Retrieve a key value
  Command: $(\texttt{Retrieve}, \texttt{ptr})$
  Users can retrieve the actual value of a key referred by the pointer `ptr`, which makes $\mathcal{F}_{\text{crypto}}$ to add the key to the set $\mathsf{Keys}_{\mathsf{known}}$ and inform the adversary about it.

- Check if two keys are equal
  Command:$(\texttt{Equal?}, \texttt{ptr}, \texttt{ptr}')$
  Users can check if two pointers, `ptr` and $\texttt{ptr}'$, refer to the same key.

- Check if a key is corrupted
  Command:$(\texttt{Corrupted?}, \texttt{ptr})$
  Users can check if a key k referred by a pointer `ptr` is corrupted.

- Key derivation
  Command: $(\texttt{Derive}, \texttt{ptr}, \texttt{type}, \texttt{seed})$
  Users can ask $\mathcal{F}_{\text{crypto}}$ to derive a new symmetric key of type $\texttt{type} \in \{\texttt{pre-key},$

`unauthenc-key`, `authenc-key`, `mac-key`} from the seed `seed` and the symmetric key referred by `ptr`, which, in turn, must have type `pre-key` or `dh-key`.

- Symmetric encryption
  Command: (`Enc`, `ptr`, `msg`)
  Users can encrypt a message `msg` using a key referred by pointer `ptr`. The type of the referred key, `unauthenc-key` or `authenc-key`, determines if the operation will be unauthenticated or authenticated, respectively.

- Symmetric decryption
  Command: (`Dec`, `ptr`, `ciph`)
  Users can decrypt a ciphertext `ciph` using a key referred by pointer `ptr`. Similarly to encryption, the type of the referred key, `unauthenc-key` or `authenc-key`, determines if the operation will be unauthenticated or authenticated, respectively.

- Generate a MAC
  Command: (`Mac`, `ptr` , `msg`)
  Users can generate a MAC on a message `msg` with a key of type `mac-key` referred by `ptr`.

- Verify a MAC
  Command: (`MacVerify`, `ptr`, `msg`, σ)
  Users can verify if a MAC σ corresponds to the MAC generated on the message `msg` with the key of type `mac-key` referred by `ptr`.

- Encryption public key request
  Command: (`GetPubKeyPKE`, pid')
  A user (pid, $lsid$, r) can request the encryption public key of party pid'.

- Signature verification public key request
  Command: (`GetPubKeySig`, pid')
  A user (pid, $lsid$, r) can request the signature verification public key of party pid'.

- Asymmetric encryption
  Command: (`PKEnc`, pid', $P_{pid'}$, `msg`)
  A user (pid, $lsid$, r) can encrypt a message `msg` under the encryption public key $P_{pid'}$ of party pid'.

- Decryption under private keys
  Command: (`PKDec`, `ciph`)
  Users can decrypt a ciphertext `ciph` using her private key.

- Generate a signature
  Command: (`Sign`, `msg`)
  Users can sign a message `msg` with her private key.

- Verify a signature
  Command: (`SigVerify`, `msg`, *pid*′, $P_{pid'}$, `msg`, σ)
  A user (*pid*, *lsid*, *r*) can verify if a signature σ corresponds to the signature generated on the message `msg` with the signing key related to the public key $P_{pid'}$ of party *pid*′.

- Corruption status request for an asymmetric (encryption or signing) key
  Command: (`CorruptPKE?`, *pid*′) and (`CorruptSig?`, *pid*′)
  A user (*pid*, *lsid*, *r*) can check the corruption status of the private (encryption and signing, respectively) key of party *pid*′.

- Generate fresh nonces
  Command: (`NewNonce`)
  Users can generate fresh nonces that does not collide with any nonce previously generated in $\mathcal{F}_{\text{crypto}}$. Nonces are also provided by the adversary, and $\mathcal{F}_{\text{crypto}}$ guarantees their freshness using a set N, analogously to keys. However, as nonces are not secrets, there is no need to keep the known and unknown status.

- Get the DH group parameters
  Command: (`GetDHGroup`)
  Users can ask for the DH group parameters $\mathbb{G}$, n, and G generated in $\mathcal{F}_{\text{crypto}}$ during initialization.

- Generate fresh exponents
  Command: (`GenExp`)
  Users can request a pointer to a new exponent e that does not collide with any other exponent previous generated or inserted in $\mathcal{F}_{\text{crypto}}$ and the corresponding DH share G^e.

- Block a DH share to be genereated
  Command: (`BlockGroupElement`, H)
  Users can block DH shares $H = G^e$ from being generated upon a `GenExp` command.

- Retrieve an exponent
  Command: (`RetrieveExp`, `ptr`)
  Users can retrieve the exponent referred by a pointer `ptr`, which makes $\mathcal{F}_{\text{crypto}}$ to add the referred exponent to the set $\mathsf{Exp}_{\mathsf{known}}$, and inform the adversary about it.

- Store an exponent
  Command: (`StoreExp`, e)
  Users can store an exponent e in $\mathcal{F}_{\text{crypto}}$. This command allows higher-level protocols

use $\mathcal{F}_{\text{crypto}}$ with exponents that are not created by it. As the exponent is known outside the functionality, after making sure there is no exponent guessing of unknown exponents, $\mathcal{F}_{\text{crypto}}$ adds the exponent to the set $\mathsf{Exp}_{\mathsf{known}}$, and informs the adversary.

- Generate a new Diffie-Hellman key
  Command: (`GenDHKey`, `ptr`, H)
  Users can generate new keys of type `dh-key` derived from a DH share H and the exponent referred by pointer `ptr`.

5.3.3 Initialization

$\mathcal{F}_{\text{crypto}}$ can be activated by either a message on the network tape or on an input tape. Upon such a command, $\mathcal{F}_{\text{crypto}}$ executes the `GroupGen`(1^η) algorithm to support the Diffie-Hellman key exchange and stores the resulting group and parameters ($\mathbb{G}$, n, G). Then, $\mathcal{F}_{\text{crypto}}$ sends this result to the adversary together with a request of all of cryptographic algorithms and pairs of keys of the encryption and signing asymmetric schemes. After receiving the response, the initialization is complete. Depending on the tape that initialization was requested, $\mathcal{F}_{\text{crypto}}$ resumes the message processing (initialization on the input tape) or gives the control back to the adversary (initialization on the network tape).

5.3.4 Corruption Model

To describe the corruption model of $\mathcal{F}_{\text{crypto}}$, assume it as a cryptographic module that can keep the long-term keys of a party at the same time it can generate fresh cryptographic material. That being said, the corruption model of cryptographic secrets is such that during the initialization, the adversary indicates in each pair of asymmetric keys if it is corrupted or not. After that, private encryption keys can be corrupted by the adversary before they are used for the first time but not afterwards. Signing keys, on the other hand, can be corrupted at any point in time. Pre-shared keys, in turn, can be corrupted when they are retrieved, that is, upon a `GetPSK` command, $\mathcal{F}_{\text{crypto}}$ asks the adversary for the value of key, the adversary then indicates together with the provided value if such a key is corrupted or not. Notice that we can refer to the keys we mentioned so far as long-term information of parties. This information could have been exposed throughout time and that is the reason the adversary can corrupt these keys directly in

$\mathcal{F}_{\text{crypto}}$ and not in the higher-level protocol that uses the functionality. On the other hand, fresh secret generated in $\mathcal{F}_{\text{crypto}}$ is considered local computation of the user, thus it is not corrupted by the adversary, which is the case of exponents with the `GenExp` command and new keys with command `New`. Therefore, the corruption model of the higher-level protocol that uses $\mathcal{F}_{\text{crypto}}$ must consider these assumptions. Besides, any secret that can become known outside $\mathcal{F}_{\text{crypto}}$ gets corrupted. Similarly, every key that is derived from components that can be known outside $\mathcal{F}_{\text{crypto}}$, using commands `Derive` and `GenDHKey`, has the corruption status dependent on the corruption status of those components.

5.3.5 Parameterization

The functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ is parameterized with a security parameter η and two algorithms. Recall that the status of a key determines if the cryptographic operation is performed ideally or security cannot be guaranteed by $\mathcal{F}_{\text{crypto}}$. Regarding encryption, there is a need for a leakage algorithm $\mathcal{L}$ that leaks some information about the plaintext that is being ideally encrypted in case of an unknown key. This algorithm can, for instance, leak the length of the plaintext ($\mathcal{L}(\texttt{msg}, \eta) = 1^{|\texttt{msg}|}$). To handle the DH key exchange, in turn, $\mathcal{F}_{\text{crypto}}$ needs an algorithm $\texttt{GroupGen}(1^{\eta})$ to setup the DH group $\mathbb{G}$.

5.3.6 Joint State Composition Theorem

In this section, we state a theorem that we use to analyze the security of *SessionKey-Derivation* protocol in Section 5.7.2. We need this, because the theorems presented in Section 5.2.3 that guarantee that the security of a multi-session protocol can be proved by performing the analysis of the single-session version of the protocol, which is clearly a simpler task, assume that concurrent sessions of the protocol have disjoint states, i.e., there is no information shared among the sessions. This is a considerable limitation for analyzing protocols that use long-term pairwise keys or asymmetric keys, since they would be required, for instance, to use a fresh long-term pairwise key in each session. As a solution to that, the Theorem 5.3 [65] states that under specific conditions on a real protocol $\mathcal{P}$, called implicit disjointness, we can prove that the $\mathcal{P}$'s multi-session version, where all sessions use the same ideal cryptographic functionality $\mathcal{F}_{\text{crypto}}$, realizes a functionality $\mathcal{F}$, by performing the security analysis of the a single-session version of $\mathcal{P}$. In reality, implicit disjointness is a condition that guarantees that different sessions of $\mathcal{P}$ do not interfere with each other even when they share information in the same functionality $\mathcal{F}_{\text{crypto}}$, such as pre-shared and asymmetric keys. Before stating the theorem, however, we need to define the condition of implicit disjointness, which demands the introduction of partnering functions and explicitly shared keys. As in the proofs of our protocols we only use MAC requests to establish the sessions, we omit the details about the other types of requests that can be used to guarantee implicit disjointness, although they are very similar.

Partnering Function [65] – A partnering function τ is a function that can groups users (*pid*, *lsid*, *r*) of a protocol $\mathcal{P}$ into sessions, i.e., if $\mathcal{P}$ is a two-party protocol with roles A and B, in a run of $\mathcal{P}$, τ group instances of machine M_A with instances of machines M_B. In practice, partnering functions are usually determined based on the nonces that parties exchange on a protocol. Formally, the partnering function τ for a protocol that uses $\mathcal{F}_{\text{crypto}}$ ($\mathcal{P} \mid \mathcal{F}_{\text{crypto}}$) is a polynomial-time computable partial function that maps every sequence of configurations α of a machine instance M_r in a run of $\mathcal{P}$ to a string and returns either a sub-string of α that represents the session, i.e., a session stamp, or $\perp$. For every environment $\mathcal{E}$, a (partial) run ρ of $\mathcal{E} \mid \mathcal{P} \mid \mathcal{F}_{\text{crypto}}$, and every instance (*pid*, *lsid*, *r*), it is defined $\tau_{(pid,lsid,r)}(\rho) = \tau(\alpha)$ where α is the projection of ρ to the sequence of configurations of M_r with party identifier *pid* and local session identifier *lsid*. Instances (*pid*, *lsid*, *r*) and (*pid′*, *lsid′*, *r′*) belong to the same session, or are partners, in a (parcial) run ρ if $\tau_{(pid,lsid,r)}(\rho) = \tau_{(pid',lsid',r')}(\rho) \neq \perp$. The partnering function τ is valid if, for every environment $\mathcal{E}$ that interacts with $\mathcal{P} \mid \mathcal{F}_{\text{crypto}}$, the following holds (with overwhelming probability over runs ρ of the system): (i) once a session stamp is assigned, it is fixed; (ii) corrupted instances do not belong to sessions, i.e., if (*pid′*, *lsid′*, *r′*) is corrupted $\tau_{(pid,lsid,r)}(\rho) = \perp$; (iii) Every session contains at most one user per role, i.e., for every partner (*pid′*, *lsid′*, *r′*) of (*pid*, *lsid*, *r*) in ρ, it holds that $r \neq r'$ or $(pid', lsid', r') = (pid, lsid, r)$.

Explicitly Shared Keys [65] – Explicitly shared keys are pre-shared keys or keys derived from them in different sessions with the same parameters.

Implicit Disjointness [65] – Let $\mathcal{P}$ be a multi-session protocol that uses $\mathcal{F}_{\text{crypto}}$ and τ a valid partnering function for $\mathcal{P} \mid \mathcal{F}_{\text{crypto}}$. Then, $\mathcal{P}$ satisfies implicit session disjointness with respect to τ if for every environment $\mathcal{E}$ for $\mathcal{P} \mid \mathcal{F}_{\text{crypto}}$ the following holds with overwhelming probability for runs ρ of $\mathcal{E} \mid \mathcal{P} \mid \mathcal{F}_{\text{crypto}}$: (i) every explicitly shared key is either always marked unknown or always marked known in $\mathcal{F}_{\text{crypto}}$; (ii) whenever an instance (*pid′*, *lsid′*, *r′*) uses an explicitly unknown shared key to successfully verify a MAC σ in $\mathcal{F}_{\text{crypto}}$ at some partial run ρ' point in ρ, there exists a specific instance (*pid*, *lsid*, *r*) that sent a MAC generation request to $\mathcal{F}_{\text{crypto}}$ which resulted in σ such that both users are partners or both users are corrupted in ρ'.

Theorem 5.3 *([65]) Let F_{single} be a machine that acts as a bridge between the environment and protocol systems (protocols and functionalities) which allows an environment to create at most one session of the system, i.e., only one instance per role on the system. Let $\mathcal{F}$ be multi-session version of a functionality and $\mathcal{P}$ a multi-session version of a protocol that uses $\mathcal{F}_{crypto}$*

and satisfies implicit disjointness w.r.t. τ, *if* $F_{single} \mid \mathcal{P} \mid \mathcal{F}_{crypto} \leq^{\tau} F_{single} \mid \mathcal{F}$, *then it holds true that* $\mathcal{P} \mid \mathcal{F}_{crypto} \leq \mathcal{F}$.

5.4 $\mathcal{F}_{\text{crypto}}$ Extension

In this section, we describe how we extend $\mathcal{F}_{\text{crypto}}$ to support a set of cryptographic primitives that, in turn, enable the usage of $\mathcal{F}_{\text{crypto}}$ to build identity-based authenticated key agreement protocols categorized into the same family of protocols proposed by [81]. In a high level, a detailed description is presented in *SessionKey* protocol (Section 4.2.1), a user in an identity-based key agreement protocols of such a category needs to generate unique secret scalar $\mathsf{x} \in \mathbb{Z}_r^*$ and associate it in a public relation $\mathsf{R} \in \mathbb{G}_1$ with the intended communication party identity using a scalar point multiplication operation. This information is, then, shared in a public network with the intended communication party. The partner generates analogous information and also shares it with the user. Having received their corresponding public relations, the users put their secret scalar in a pairing operation and are able to derive a common key.

In a high level modeling, our extension of $\mathcal{F}_{\text{crypto}}$ allows two users (*pid*, *lsid*, *r*) and (*pid'*, *lsid'*, *r'*) to generate unique secret scalars ($\mathsf{x} \in \mathbb{Z}_r^*$) x_{pid} and $\mathsf{x}_{pid'}$, respectively, tied to the identity of each other in public relations ($\mathsf{R} \in \mathbb{G}_1$) R_{pid} and $\mathsf{R}_{pid'}$, respectively. The users can then exchange these public relations in the network to create a common symmetric key k derived from these public relations and their respective scalars. $\mathcal{F}_{\text{crypto}}$ guarantees that if the relations were created in $\mathcal{F}_{\text{crypto}}$, the resulting key k is only accessible by the owners of scalars x_{pid} and $\mathsf{x}_{pid'}$. Therefore, we add to the functionality $\mathcal{F}_{\text{crypto}}$ an Identity-Based Cryptosystem, secret scalars, public relations, and a new type of key.

We add to $\mathcal{F}_{\text{crypto}}$ initialization (Section 5.3.3), more specifically to the phase in which $\mathcal{F}_{\text{crypto}}$ requests the cryptographic algorithms and pairs of asymmetric keys to the adversary, the request of the pairs of identity public and private keys of each party *pid* (P^{I}_{pid} and S^{I}_{pid}, respectively) in the environment and the Identity-Based Cryptosystem public parameters ($\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, r), where $\mathbb{G}_1$ and $\mathbb{G}_2$ are the descriptions of the two additively-written groups of order r, and $\mathbb{G}_T$ is the description of the multiplicatively-written group of order r. Analogously to the other asymmetric cryptosystems in $\mathcal{F}_{\text{crypto}}$, we assume the keys have been distributed and that the identity private keys have a corruption status to represent if they are corrupted or not.

Our scalar modeling in $\mathcal{F}_{\text{crypto}}$ is similar to DH exponents. The scalars belong to the IBC set $\mathbb{Z}_r^*$, their actual value of scalars are not revealed to users unless explicitly requested. Instead, the users receive pointers that refer to the scalars they generate. On the other hand, the public relations $\mathsf{R} \in \mathbb{G}_1$ between scalars and users identities are indeed public, and can be directly accessed. The corruption of scalars is also tracked using sets. The new set Scalar

keeps all the scalars controlled in $\mathcal{F}_{\text{crypto}}$. The set $\mathsf{Scalar}_{\mathsf{known}} \subseteq \mathsf{Scalar}$, in turn, maintains all corrupted scalars. Hence, if a scalar x is uncorrupted, $\mathrm{x} \in \mathsf{Scalar} \backslash \mathsf{Scalar}_{\mathsf{known}}$. Analogously to fresh keys and exponents in $\mathcal{F}_{\text{crypto}}$, the corruption model of scalars is such that the adversary cannot directly corrupt them, scalars only become corrupted when they have their actual value retrieved by the users. The scalars generation is handled by the new command (`NewScalar`, *pid'*), explained in details in the next section.

We add to $\mathcal{F}_{\text{crypto}}$ the new type of keys `id-key` to represent the keys established using secret scalars and public relations. We define that `id-key` keys can only be generated by the new command `GenIDKey`, described in details in the next section, and can only be used to derive new keys of arbitrary types. Analogously to other keys in $\mathcal{F}_{\text{crypto}}$, the corruption status of an `id-key` key is tracked in the existing sets Keys and $\mathsf{Keys}_{\mathsf{known}}$. As we add the new type `id-key` in the set of symmetric keys in $\mathcal{F}_{\text{crypto}}$, we extend following existent operations to also consider the `id-key` keys: (`Retrieve`, `ptr`), (`Equal?`, `ptr` , `ptr`$'$), (`Corrupted?`, `ptr`), (`Derive`, `ptr`, `type`, `seed`), and (`Store`, `type`, k).

5.4.1 $\mathcal{F}_{\text{crypto}}$ ID-Based Operations

In this section, we describe in details the new identity-based operations we add to $\mathcal{F}_{\text{crypto}}$. As the operations make sense only in the IBC context, in all of them, $\mathcal{F}_{\text{crypto}}$ first verifies if the requester and the intended communication partner (if the commands requires so) have IBC keys, and if not, an error message is returned to the user, otherwise, it proceeds.

- Corruption status request of the identity private key
  Command: (`CorruptPID?`, *pid'*)
  Return: (`Corrupted`, `bool`)
  A user (*pid*, *lsid*, *r*) can request the corruption status of the identity private key of party *pid'*. $\mathcal{F}_{\text{crypto}}$ returns the message (`Corrupted`, `bool`), where the boolean `bool` contains `true` if the identity private key of party *pid'* is corrupted, otherwise, `bool` = `false`.

- Generate a fresh scalar associated with an identity
  Command: (`NewScalar`, *pid'*)
  Return: (`ScalarPointer`, $\texttt{ptr}_{\mathrm{x}_{pid}}$, R_{pid})
  A user (*pid*, *lsid*, *r*) can request a pointer to a fresh scalar related to an intended communication partner *pid'*. Upon such request, $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new scalar x_{pid} with the message (`ProvideScalar`, *pid'*) on the network interface.

  After receiving the scalar value x_{pid} and a relation R_{pid} from the adversary, $\mathcal{F}_{\text{crypto}}$ first verifies if $\mathrm{x}_{pid} \in \mathbb{Z}_r^*$ and $\mathsf{R}_{pid} \in \mathbb{G}_1$ to guarantee consistency with the IBC groups. Then, it checks if $\mathsf{x}_{pid} \notin \mathsf{Scalars}$. If any of the checks fail, $\mathcal{F}_{\text{crypto}}$ requests the adversary new values until everything checks successfully, which models that a new scalar do not collide with any previously generated values. As a result, $\mathcal{F}_{\text{crypto}}$ adds the scalar to the set Scalars and internally registry the entry $((pid, lsid, r), \texttt{ptr}_{\mathsf{x}_{pid}}, \mathsf{x}_{pid}, \mathsf{R}_{pid}, pid')$, which represents the user $(pid, lsid, r)$, through scalar pointer $\texttt{ptr}_{\mathsf{x}_{pid}}$, has access to the scalar x_{pid}, which is tied with the identity of party pid' in the relation R_{pid}. Then, $\mathcal{F}_{\text{crypto}}$ returns the message $(\texttt{ScalarPointer}, \texttt{ptr}_{\mathsf{x}_{pid}}, \mathsf{R}_{pid})$ to the user.

- Generate an ID-Based key
  Command: $(\texttt{GenIDKey}, \texttt{ptr}_{\mathsf{x}_{pid}}, \mathsf{R}, pid')$
  Return: $(\texttt{IDKey}, \texttt{ptr}_k)$
  A user $(pid, lsid, r)$ can request $\mathcal{F}_{\text{crypto}}$ a pointer $\texttt{ptr}_k$ that refers to a new key of type `id-key` derived from the scalar x_{pid} refereed by $\texttt{ptr}_{\mathsf{x}_{pid}}$ and from the relation R received from user pid'. Upon such a request, $\mathcal{F}_{\text{crypto}}$ first checks if $\mathsf{R}_{pid}, \mathsf{R} \in \mathbb{G}_1$ to keep consistency with the IBC settings and returns an error if the check fails.

The entry $((pid, lsid, r), \texttt{ptr}_{\mathsf{x}_{pid}}, \mathsf{x}_{pid}, \mathsf{R}_{pid}, q)$ exists[2] in $\mathcal{F}_{\text{crypto}}$, i.e., the scalar x_{pid} that belongs to user $(pid, lsid, r)$ is tied to the identity of a party q. Then, $\mathcal{F}_{\text{crypto}}$ verifies if an entry such as $((pid', lsid', r'), \texttt{ptr}_{\mathsf{x}_{pid'}}, \mathsf{x}_{pid'}, \mathsf{R}, p)$ exists, i.e., the relation R that ties an $\mathsf{x}_{pid'}$ and an identity of a party p was created by a user $(pid', lsid', r')$ in $\mathcal{F}_{\text{crypto}}$. Below we discuss all the possibilities.

(i) The entry $((pid', lsid', r'), \texttt{ptr}_{\mathsf{x}_{pid'}}, \mathsf{x}_{pid'}, \mathsf{R}, p)$ exists, $q = pid'$, and $p = pid$. That is, the requester $(pid, lsid, r)$ generated a scalar x_{pid} tied to the identity of pid', which, in turn, generated a relation R that ties the scalar scalar $\mathsf{x}_{pid'}$ and the identity of pid. In this case, $\mathcal{F}_{\text{crypto}}$ generate a common key for $(pid, lsid, r)$ and $(pid', lsid', r')$ guaranteed to be accessible only by the scalars' owners.

    If there already exists an `id-key` key k derived from pid, pid', x_{pid}, and R, $\mathcal{F}_{\text{crypto}}$ defines $\texttt{ptr}_k$ to reference such a key, and returns $(\texttt{IDKey}, \texttt{ptr}_k)$ to the user.

    If the key does not exist yet, $\mathcal{F}_{\text{crypto}}$ must generate it. An `id-key` key is dependent on the IBC in the environment, i.e., it is derived from the secret scalars together with the identity of the users. We define that the corruption status of the private identity key of the users has precedence over the scalars' when determining the corruption status of a resulting `id-key` key, which models that if the private key is corrupted, uncorrupted scalars cannot guarantee the security of the resulting key. Therefore, $\mathcal{F}_{\text{crypto}}$ checks if both scalars are corrupted, or the identity private key of pid or pid' is corrupted. If any of the conditions is true, the key k is corrupted, i.e., known. Hence, $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new `id-key` key with the message

[2]For the sake of simplicity, we consider that pointers are always valid and comes from their real owners, otherwise, $\mathcal{F}_{\text{crypto}}$ could just return an error.

(`ProvideIDKey`, `known`, *pid*, x_{pid}, *r*, *pid′*, R, $\mathsf{x}_{pid'}$) on the network interface. Otherwise, the key k is uncorrupted, i.e., unknown, and $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new key with the message (`ProvideIDKey`, `unknown`, *pid*, x_{pid}, *r*, *pid′*, R, $\mathsf{x}_{pid'}$) on the network interface. $\mathcal{F}_{\text{crypto}}$ keeps asking for a new key k until $k \notin \mathsf{Keys}$, which models that a new key never collides with any previously generated key. Once $\mathcal{F}_{\text{crypto}}$ accepts the key, it adds k to the set Keys, registers k as derived from *pid*, *pid′*, x_{pid}, and R sets the pointer $\mathtt{ptr}_k$ to refer k, and adds k to the set $\mathsf{Keys}_{\mathsf{known}}$ if k is known.

In the end, $\mathcal{F}_{\text{crypto}}$ returns the message (`IDKey`, $\mathtt{ptr}_k$) to the user.

(ii) The entry ((*pid′*, *lsid′*, *r′*), $\mathtt{ptr}_{\mathsf{x}_{pid'}}$, $\mathsf{x}_{pid'}$, R, *p*) exists, $q \neq pid'$, and $p = pid$. That is, the requester (*pid*, *lsid*, *r*) generated a scalar x_{pid} tied to an identity that is not *pid′*, the generator of R. However, the relation R was created using the identity of the requester *pid* tied to the scalar $\mathsf{x}_{pid'}$.

This means that even the user (*pid*, *lsid*, *r*) being the owner of the scalar x_{pid} and the public relation R being address to it, the party (*pid′*, *lsid′*, *r′*) will never received a relation that contains the scalar x_{pid} tied to its identity because this scalar, when generated, was already tied to the identity of another party. However, $\mathcal{F}_{\text{crypto}}$ must generate the key anyway.

Analogously to the previous case, $\mathcal{F}_{\text{crypto}}$ checks if both scalars are corrupted, or the identity private key of *pid* or *pid′* is corrupted. If any of the conditions is true, the key k is corrupted, i.e., known. Hence, $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new key with the message (`ProvideIDKey`, `known`, *pid*, x_{pid}, *r*, *pid′*, R, $\mathsf{x}_{pid'}$) on the network interface. Otherwise, the key k is uncorrupted, i.e., unknown, and $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new key with the message (`ProvideIDKey`, `unknown`, *pid*, x_{pid}, *r*, *pid′*, R, $\mathsf{x}_{pid'}$) on the network interface. $\mathcal{F}_{\text{crypto}}$ keeps asking for a new key k until $k \notin \mathsf{Keys}$. Once $\mathcal{F}_{\text{crypto}}$ accepts the key, it adds k to the set Keys, registers k as derived from *pid*, *pid′*, x_{pid}, and R, sets the pointer $\mathtt{ptr}_k$ to refer k, and adds k to the set $\mathsf{Keys}_{\mathsf{known}}$ if k is known. In the end, $\mathcal{F}_{\text{crypto}}$ returns the message (`IDKey`, $\mathtt{ptr}_k$) to the user.

Here we highlight an aspect of our modeling. A secret scalar can be seen as a confidential nonce, that is, it should not be revealed and used only once to derive a single key. The first requirement we model by considering the scalars a secret in $\mathcal{F}_{\text{crypto}}$, i.e., using the pointers instead of returning their actual values to users. The second requirement, in turn, we model by tying, at generation, a scalar to the intended communication partner identity public key, which partially fulfill the requirement, because, in spite of a user not being able to use a scalar generated to communicate to one party to communicate to another one, two parties can reuse scalars they generate to communicate with each other.

(iii) The entry $((pid', lsid', r'), \texttt{ptr}\mathrm{x}_{pid'}, \mathrm{x}_{pid'}, \mathsf{R}, p)$ exists, and $p \neq pid$. Therefore, the relation R was created using an identity of another party than *pid*. This key will make sense only for user $(pid, lsid, r)$. However, $\mathcal{F}_{\text{crypto}}$ must generate the key anyway.

This new key cannot be derived from two scalars, besides the relation R is public, therefore, the corruption status of the resulting key depends solely on the identity private key of the requester. Then, $\mathcal{F}_{\text{crypto}}$ checks if the identity private key of *pid* is corrupted. If so, $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new key with the message $(\texttt{ProvideIDKey}, \texttt{known}, pid, \mathrm{x}_{pid}, r, pid', \mathsf{R})$ on the network interface. Otherwise, the key k is uncorrupted, i.e., unknown, and $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new key with the message $(\texttt{ProvideIDKey}, \texttt{unknown}, pid, \mathrm{x}_{pid}, r, pid', \mathsf{R})$ on the network interface. $\mathcal{F}_{\text{crypto}}$ keeps asking for a new key k until $k \notin \mathsf{Keys}$. Once $\mathcal{F}_{\text{crypto}}$ accepts the key, it adds k to the set Keys, registers k as derived from pid, pid', x_{pid}, and R, sets the pointer $\texttt{ptr}_k$ to refer k, and adds k to the set $\mathsf{Keys}_{\mathsf{known}}$ if k is known. In the end, $\mathcal{F}_{\text{crypto}}$ returns the message $(\texttt{IDKey}, \texttt{ptr}_k)$ to the user.

(iv) The entry $((pid', lsid', r'), \texttt{ptr}\mathrm{x}_{pid'}, \mathrm{x}_{pid'}, \mathsf{R}, p)$ does not exist in $\mathcal{F}_{\text{crypto}}$. Therefore, R was not created in $\mathcal{F}_{\text{crypto}}$. This key will make sense only for user $(pid, lsid, r)$. However, $\mathcal{F}_{\text{crypto}}$ must generate the key anyway.

Analogously to the previous case, this new key cannot be derived from two scalars, and the relation R is public, therefore the corruption status of the resulting key depends solely on the identity private key of the requester. Then, $\mathcal{F}_{\text{crypto}}$ checks if the identity private key of *pid* is corrupted. If so, $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new key with the message $(\texttt{ProvideIDKey}, \texttt{known}, pid, \mathrm{x}_{pid}, \mathsf{R})$ on the network interface. Otherwise, the key k is uncorrupted, i.e., unknown, and $\mathcal{F}_{\text{crypto}}$ asks the adversary for a new key with the message $(\texttt{ProvideIDKey}, \texttt{unknown}, pid, \mathrm{x}_{pid}, \mathsf{R})$ on the network interface. $\mathcal{F}_{\text{crypto}}$ keeps asking for a new key k until $k \notin \mathsf{Keys}$. Once $\mathcal{F}_{\text{crypto}}$ accepts the key, it adds k to the set Keys, registers k as derived from pid, pid', x_{pid}, and R, sets the pointer $\texttt{ptr}_k$ to refer k, and adds k to the set $\mathsf{Keys}_{\mathsf{known}}$ if k is known. In the end, $\mathcal{F}_{\text{crypto}}$ returns the message $(\texttt{IDKey}, \texttt{ptr}_k)$ to the user.

- Retrieve the actual value of a scalar
  Command: $(\texttt{RetrieveScalar}, \texttt{ptr}\mathrm{x}_{pid})$
  Return: $(\texttt{Scalar}, \mathrm{x}_{pid})$
  A user $(pid, lsid, r)$ can request the actual value x_{pid} of the scalar referred by $\texttt{ptr}\mathrm{x}_{pid}$. As a result, $\mathcal{F}_{\text{crypto}}$ adds the scalar x_{pid} to $\mathsf{Scalar}_{\mathsf{known}}$ as a corrupted scalar, informs the

adversary about it, and returns the message (`Scalar`, x_{pid}) to the user.

5.4.2 Corruption Model

In our extension we add to $\mathcal{F}_{\text{crypto}}$ three types of confidential information that can be affected by corruption, the identity private keys, scalars, and `id-key` keys. We define that the adversary can statically corrupt identity private keys at the moment they are provided to $\mathcal{F}_{\text{crypto}}$ or before they are used for the first time, i.e., when the first `id-key` key is derived, but not afterwards. Similarly to other asymmetric settings in $\mathcal{F}_{\text{crypto}}$, the identity private keys cannot be corrupted by being requested by the users. The scalars, in turn, as fresh confidential information generated in $\mathcal{F}_{\text{crypto}}$, are considered a local computation of the user, thus cannot be corrupted by the adversary. Last, as a key that is derived from other components that can be known outside $\mathcal{F}_{\text{crypto}}$, the corruption status of `id-key` keys depend on the corruption status of those components at generation phase, as described in the `GenIDKey` command.

5.5 Realization $\mathcal{P}_{\text{crypto}}$ of $\mathcal{F}_{\text{crypto}}$

In this section, we show how we can extend $\mathcal{P}_{\text{crypto}}$ to realize our $\mathcal{F}_{\text{crypto}}$ extension. Different from the original proposal of $\mathcal{P}_{\text{crypto}}$ [66], however, we do not intend to provide a detailed proof of our extension based on standard cryptographic assumptions, we want to show that $\mathcal{P}_{\text{crypto}}$ realizes our extension using the particular enhanced [29] identity-based cryptosystem from [81]. Therefore, when it is appropriate we signalize the results we are reusing without going into further details and formalism.

5.5.1 $\mathcal{P}_{\text{crypto}}$ Definition

As first detailed in [66] and extended in [62] $\mathcal{P}_{\text{crypto}}$ has the same i/o interface as $\mathcal{F}_{\text{crypto}}$ and it is parameterized with authenticated and unauthenticated symmetric encryption schemes, $\Sigma_{\texttt{authenc}}$ and $\Sigma_{\texttt{unauthenc}}$, respectively, a public-key encryption mechanism $\Sigma_{\texttt{pub}}$, a digital signature scheme $\Sigma_{\texttt{sig}}$, a MAC scheme $\Sigma_{\texttt{MAC}}$, the algorithm $\texttt{GroupGen}(1^\eta)$ to generate the DH group, and two families of pseudo-random functions to derive keys, $\texttt{F}$ for the general case and $\texttt{F}'$ for the $\texttt{GroupGen}$ algorithm. We extend $\mathcal{P}_{\text{crypto}}$ adding to it the identity-based cryptosystem $\Sigma_{\texttt{IBC}}$ [29, 81]. The cryptosystem $\Sigma_{\texttt{IBC}}$ is initialized with the input of the security parameter η into the BDH parameter generator $\mathcal{B}$ (Section 2.5.3), which returns the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with order r, identity $\mathcal{O}$, and generators G_1 and G_2, respectively, the group $\mathbb{G}_T$ of order r and identity 1, and the admissible *bilinear pairing* $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. Besides, $\Sigma_{\texttt{IBC}}$ selects mapping functions $\textsc{Map}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_r^*$ and $\textsc{Map}_2 : \{0,1\}^* \times \{0,1\}^* \times \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_T \rightarrow \{0,1\}^n$ and a randomly chosen master secret $s \in \mathbb{Z}_r^*$ that defines a master public key $P^{\text{I}} = s \cdot G_1$. The identity public key of a party *pid* in the system is calculated as $P^{\text{I}}_{pid} = (\ \textsc{Map}_1(pid) \cdot G_1 + P^{\text{I}}\ )$, and the identity private key of such a party is calculated as $S^{\text{I}}_{pid} = ((\ \textsc{Map}_1(pid) + s)^{-1} \cdot G_2)$.

Upon activation, $\mathcal{P}_{\text{crypto}}$ initializes itself. First, it executes the $\texttt{GroupGen}$ algorithm and stores the results. Then, it has to set up public encryption and digital signature schemes. In these cases, the adversary has to send requests to $\mathcal{P}_{\text{crypto}}$ to generate the key pairs for each party *pid* in the environment that should have a key pair. In each request, the adversary indicates if a private key is corrupted or not. If so, the adversary provides together the value of the key pair, otherwise, $\mathcal{P}_{\text{crypto}}$ generates the pair of keys following the appropriate key generation algorithm from $\Sigma_{\texttt{pub}}$ or $\Sigma_{\texttt{sig}}$. In the end, $\mathcal{P}_{\text{crypto}}$ stores the value of the keys together with their corruption status.

In our extension, after concluding the setup of existing asymmetric cryptosystems, $\mathcal{P}_{\text{crypto}}$ initializes the $\Sigma_{\texttt{IBC}}$ cryptosystem. It first executes the BDH algorithm $\mathcal{B}$, sets up $\textsc{Map}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_r^*$, $\textsc{Map}_2 : \{0,1\}^* \times \{0,1\}^* \times \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_T \rightarrow \{0,1\}^n$, the master secret $s \in \mathbb{Z}_r^*$ randomly chosen and the master public key $P^{\text{I}} = s \cdot G_1$. Then, $\mathcal{P}_{\text{crypto}}$ informs the adversary that it can start requesting the generation of IBC keys. Analogously to the other pair of keys, the adversary sends requests to $\mathcal{P}_{\text{crypto}}$ to generate the IBC keys of each party. In each request it is also informed if the private key should be corrupted or not and, if so, the adversary provide the value of the keys in the same message. Otherwise, i.e., if the private key is not corrupted, $\mathcal{P}_{\text{crypto}}$ generates the key pair as defined in $\Sigma_{\texttt{IBC}}$. Last, $\mathcal{P}_{\text{crypto}}$ stores the keys for each party together with the corruption status of the private key.

We also add to $\mathcal{P}_{\text{crypto}}$ the ability to track the (un)known status of scalars, and the new type of keys $\texttt{id-key}$ to represent keys established using the identity-based operations. As in $\mathcal{F}_{\text{crypto}}$, $\mathcal{P}_{\text{crypto}}$ stores the types of symmetric keys and use them to verify if a requested operation can be executed with a given pointer, and maintains the corruption status of all other confidential material (keys, exponents, and scalars), which allows it to respond to the users appropriately when requested, as in $\mathcal{F}_{\text{crypto}}$. However, as in its original proposal, $\mathcal{P}_{\text{crypto}}$ does not need the sets used in $\mathcal{F}_{\text{crypto}}$ to guarantee freshness and resistance to collision and key guessing, because, as a real implementation, $\mathcal{P}_{\text{crypto}}$ must do so based on the mathematical properties of its cryptosystems and not by verifying whether an element already belongs to a set or not.

5.5.2 $\mathcal{P}_{\text{crypto}}$ ID-Based Operations

Now we detail how each id-based operation defined for our extension is actually implemented in $\mathcal{P}_{\text{crypto}}$. As the operations only make sense in an IBC context, in all new operations, $\mathcal{P}_{\text{crypto}}$ first verifies if the requester and the intended communication partner (if the commands requires so) have keys created in $\Sigma_{\texttt{IBC}}$, and if not, an error message is returned to the user, otherwise it proceeds.

- Generate a fresh scalar associated with an identity
  Command: ($\texttt{NewScalar}$, *pid′*)
  Return: ($\texttt{ScalarPointer}$, $\texttt{ptr}_{\mathsf{x}_{pid}}$, R_{pid})
  A user (*pid*, *lsid*,*r*) requests $\mathcal{P}_{\text{crypto}}$ a new pointer to a fresh scalar related to an intended communication partner *pid′* with message ($\texttt{NewScalar}$, *pid′*). Upon such request, $\mathcal{P}_{\text{crypto}}$ randomly selects $\mathsf{x}_{pid} \in \mathbb{Z}_r^*$, creates a pointer $\texttt{ptr}_{\mathsf{x}_{pid}}$ that refers x_{pid}, calculates $\mathsf{R}_{pid} = \mathsf{x}_{pid} \cdot P^{\mathrm{I}}_{pid'}$, where $P^{\mathrm{I}}_{pid'}$ is the identity public key of *pid′*, and returns ($\texttt{ScalarPointer}$, $\texttt{ptr}_{\mathsf{x}_{pid}}$, R_{pid}) to the user.

- Retrieve the actual value of a scalar
  Command: ($\texttt{RetrieveScalar}$, $\texttt{ptr}_{\mathsf{x}_{pid}}$)
  Return: ($\texttt{Scalar}$, x_{pid})
  A user (*pid*, *lsid*,*r*) requests the actual value x_{pid} of the scalar referred by $\texttt{ptr}_{\mathsf{x}_{pid}}$ with message ($\texttt{RetrieveScalar}$, $\texttt{ptr}_{\mathsf{x}_{pid}}$). $\mathcal{P}_{\text{crypto}}$ returns the message ($\texttt{Scalar}$, x_{pid}) to the user.

- Generate an ID-Based key
  Command: ($\texttt{GenIDKey}$, $\texttt{ptr}_{\mathsf{x}_{pid}}$, R, *pid′*)
  Return: ($\texttt{IDKey}$, $\texttt{ptr}_k$)
  A user (*pid*, *lsid*, *r*) requests $\mathcal{P}_{\text{crypto}}$ a new key of type $\texttt{id-key}$ derived from scalar x_{pid} refereed by $\texttt{ptr}_{\mathsf{x}_{pid}}$ and the relation R received from user *pid′* with message ($\texttt{GenIDKey}$, $\texttt{ptr}_{\mathsf{x}_{pid}}$, R, *pid′*). Upon such a request, $\mathcal{P}_{\text{crypto}}$ checks if $\mathsf{R} \in \mathbb{G}_1$ to keep consistency with $\Sigma_{\texttt{IBC}}$, and if the verification fails, $\mathcal{P}_{\text{crypto}}$ returns an error message to the user. Otherwise, $\mathcal{P}_{\text{crypto}}$ calculates the a key k of type $\texttt{id-key}$ as follows:

  $k = \textsc{Map}_2(pid, pid', \mathsf{x}_{pid} \cdot P^{\mathrm{I}}_{pid'}, \mathsf{R}, \hat{e}(\mathsf{R}, S^{\mathrm{I}}_{pid})(\mathbb{G}_1, \mathbb{G}_2)^{\mathsf{x}_{pid}})$ if the user has the role (*r*) initiator on the higher-level protocol; otherwise,

  $k = \textsc{Map}_2(pid', pid, \mathsf{R}, \mathsf{x}_{pid} \cdot P^{\mathrm{I}}_{pid'}, \hat{e}(\mathsf{R}, S^{\mathrm{I}}_{pid})(\mathbb{G}_1, \mathbb{G}_2)^{\mathsf{x}_{pid}})$, i.e., if the user has the role (*r*) responder on the higher-level protocol.

  Then, $\mathcal{P}_{\text{crypto}}$ creates a pointer $\texttt{ptr}_k$ to k, and returns ($\texttt{IDKey}$, $\texttt{ptr}_k$) to the user.

5.5.3 $\mathcal{P}_{\text{crypto}}$ realizes $\mathcal{F}_{\text{crypto}}$

In this section, we present some restrictions that the environments that interact with $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$ must respect and show that our extension can be realized using the [81] identity-based cryptosystem $\Sigma_{\texttt{IBC}}$.

5.5.3.1 Environment Restrictions

As we are expanding $\mathcal{P}_{\text{crypto}}$, we have to consider the same restrictions imposed in its original proposal and extensions [62, 64, 66], which restrict the class of environments $\mathcal{E}$ that interact with $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$ to well-behaved environments. Besides, as our extension involves a new key that is used to derive other keys and scalars that are also used to derive a key, the restrictions also apply to the elements we add. More specifically, to be considered a well-behaved environment, an environment $\mathcal{E}$ must not cause the commitment problem and it must be used-order respecting. An environment $\mathcal{E}$ does not cause the commitment problem if the cases in which an unknown key k, now including the new type `id-key`, used at time t in a key derivation operation becomes known at time $t' > t$, or the cases where an unknown scalar x used to derive an unknown `id-key` k at time t becomes known at time $t' > t$, happen with negligible probability. $\mathcal{E}$ is used-order respecting, in turn, if the case in which an unknown key k, now including the new type `id-key`, used at time t in a key derivation operation is encrypted at time $t' > t$ by another unknown key, happens with negligible probability (which avoid key cycles). In all of AoT protocols where we intend to use our $\mathcal{F}_{\text{crypto}}$ extension to perform the security analysis, the environments, i.e., the higher-level protocols that use $\mathcal{F}_{\text{crypto}}$ do not cause the commitment problem. In the case of scalars, for instance, they are generated, used exactly once to derive an `id-key` key, and then erased from the memory. Similarly, the protocols in AoT are all used-order respecting, because none of them encrypts a key after it has been already used.

5.5.3.2 Proof $\mathcal{P}_{\text{crypto}}$ realizes $\mathcal{F}_{\text{crypto}}$

Our goal in this section is to show our operations in $\mathcal{F}_{\text{crypto}}$ can be realized in $\mathcal{P}_{\text{crypto}}$ using the particular enhanced [29] identity-based cryptosystem from [81].

As in the original proofs [62, 66], we use an intermediate machine $\mathcal{F}^*$ that acts as a bridge between the environment and protocol systems communication to guarantee that all requirements with respect to the well-behaved environments are fulfilled, which means $\mathcal{F}^*$ blocks if any of them is not respected. Therefore, let $\Sigma_{\texttt{IBC}}$ the identity-based scheme from [81], $\mathcal{F}_{\text{crypto}}$ the functionality for cryptographic primitives described in [66], extended in [62] and extended by us as described in Section 5.4, $\mathcal{P}_{\text{crypto}}$ be the realization of $\mathcal{F}_{\text{crypto}}$ proven in [62, 66] and extended by us as described in Section 5.5.2, and $\mathcal{F}^*$ be a machine that in any point during the protocol systems communication blocks if the environment cause the commitment problem or does not respect the used-order requirement. We want to show that, if the hardness of the problem BDDH holds true for the BDH generator $\mathcal{B}_t$ of $\Sigma_{\texttt{IBC}}$, then

$$\mathcal{F}^* \mid \mathcal{P}_{\text{crypto}} \leq \mathcal{F}^* \mid \mathcal{F}_{\text{crypto}} \tag{5.1}$$

We must show $\mathcal{E} \mid \mathcal{P}_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}_{\text{crypto}}$ to prove (5.1). Then, we start by designing a simulator $\mathcal{S}$ that internally simulates $\mathcal{P}_{\text{crypto}}$ and, being the adversary of $\mathcal{F}_{\text{crypto}}$, responds to all of its requests, keeping the consistency defined in the ideal definition of the cryptographic primitives in the primitive based on the mathematical properties of the assumptions assumed in $\mathcal{P}_{\text{crypto}}$. As we want to reuse the original proofs of $\mathcal{P}_{\text{crypto}}$, our simulator $\mathcal{S}$ is the same as in [62]. First of all, as the adversary of $\mathcal{F}_{\text{crypto}}$, $\mathcal{S}$ initializes $\mathcal{F}_{\text{crypto}}$, which, in turn, generates the DH group and sends it to $\mathcal{S}$ together with a request for the cryptographic algorithms, and key pairs. Then, $\mathcal{S}$ initializes its internal simulation of $\mathcal{P}_{\text{crypto}}$, which includes $\Sigma_{\texttt{IBC}}$. After that, $\mathcal{S}$ returns all the cryptographic algorithms, keys, corruption status and public information to $\mathcal{F}_{\text{crypto}}$, including all $\Sigma_{\texttt{IBC}}$-related material, the public parameters ($\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, q) and the pair of keys (P^{I}, S^{I}) with their corresponding corruption status.

After initialization, upon a new scalar request from $\mathcal{F}_{\text{crypto}}$ with a (`ProvideScalar`, *pid*) message, $\mathcal{S}$ responds with (`ScalarPointer`, $\texttt{ptr}_{\mathsf{X}}$, $\mathsf{R} \cdot P^{\text{I}}_{pid}$), where x is chosen at random from $\mathbb{Z}_r^*$. If $\mathcal{F}_{\text{crypto}}$ refuses a value generated by $\mathcal{S}$ to prevent collision and asks $\mathcal{S}$ to provide another one, $\mathcal{S}$ stops its execution blocking any subsequent message. If $\mathcal{F}_{\text{crypto}}$ asks $\mathcal{S}$ for a new `id-key` key based on scalars with a message (`ProvideIDKey`, `unknown` or `known`, *pid*, x, *r*, *pid′*, R, y), $\mathcal{S}$ returns $k = \textsc{Map}_2(pid, pid', \mathsf{x} \cdot P^{\text{I}}_{pid'}, \mathsf{R}, \hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\mathsf{x}+\mathsf{y}})$ if the *r* indicates an initiator role; otherwise, $k = \textsc{Map}_2(pid', pid, \mathsf{R}, \mathsf{x} \cdot P^{\text{I}}_{pid'}, \hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\mathsf{x}+\mathsf{y}})$, i.e., if the role *r* indicates a responder role. On the other hand, if $\mathcal{F}_{\text{crypto}}$ requests for a new key `id-key` key with message (`ProvideIDKey`, `unknown` or `known`, *pid*, x, *r*, *pid′*, R), $\mathcal{S}$ returns $k = \textsc{Map}_2(pid, pid', \mathsf{x} \cdot P^{\text{I}}_{pid'}, \mathsf{R}, \hat{e}(\mathsf{R}, S^{\text{I}}_{\text{pid}})\hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\mathsf{x}})$ if the *r* indicates an initiator role; otherwise, $k = \textsc{Map}_2(pid', pid, \mathsf{R}, \mathsf{x} \cdot P^{\text{I}}_{pid'}, \hat{e}(\mathsf{R}, S^{\text{I}}_{\text{pid}})\hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\mathsf{x}})$, i.e., if the role *r* indicates a responder role. Analogously to scalars, if $\mathcal{F}_{\text{crypto}}$ refuses a key value generated by $\mathcal{S}$ to prevent collision and requests another one, $\mathcal{S}$ stops its execution blocking any subsequent message. When $\mathcal{F}_{\text{crypto}}$ asks for a key derivation based on a `id-key`, $\mathcal{S}$ behaves the same as it does for the `pre-share` keys.

As $\mathcal{P}_{\text{crypto}}$ encompasses different cryptographic primitives, the original proof strategy [66] is built through a series of hybrid systems, in which the cryptographic primitives in $\mathcal{P}_{\text{crypto}}$ are individually proven to realize its respective idealization in $\mathcal{F}_{\text{crypto}}$. The analysis starts from the equivalence $\mathcal{E} \mid \mathcal{P}_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{P}_{\text{crypto}}$ that is obviously true. Then, the first relevant equivalence $\mathcal{E} \mid \mathcal{P}_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{P}^1_{\text{crypto}}$ must be proved. In this case, $\mathcal{P}^1_{\text{crypto}}$ is a hybrid copy of $\mathcal{P}_{\text{crypto}}$ in which the idealization (defined in $\mathcal{F}_{\text{crypto}}$) of the first cryptographic primitive being analyzed substitutes its real implementation. In each one of the next steps, the equivalence $\mathcal{E} \mid \mathcal{P}^i_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{P}^{i+1}_{\text{crypto}}$ must be proven, until all n cryptographic primitives are substituted by their $\mathcal{F}_{\text{crypto}}$ idealization in $\mathcal{P}^n_{\text{crypto}}$, that is, $\mathcal{E} \mid \mathcal{P}^{n-1}_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{P}^n_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}_{\text{crypto}}$.

Step 1 In the first step, we define the hybrid system $\mathcal{P}^1_{\text{crypto}}$ that is the same as $\mathcal{P}_{\text{crypto}}$ except for signature handling, asymmetric encryption and decryption, nonce generation, creation and storage of DH exponents, and creation and storage of DH key, where we substitute all those operations by their idealization in $\mathcal{F}_{\text{crypto}}$, and have to show that

$$\mathcal{E} \mid \mathcal{F}^* \mid \mathcal{P}_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^* \mid \mathcal{P}^1_{\text{crypto}} \tag{1}$$

As we do not propose any change on such operations, the original proof from [62] still holds.

Step 2 In the second step, we change the real implementation of scalars in the hybrid system $\mathcal{P}^2_{\text{crypto}}$ to behave as in $\mathcal{F}_{\text{crypto}}$, i.e., $\mathcal{P}^2_{\text{crypto}}$ controls the scalars status with sets Scalar and $\mathsf{Scalar}_{\mathsf{known}}$, and prevent scalar collisions by checking the contents of the sets. In this step, we have to show that

$$\mathcal{E} \mid \mathcal{F}^* \mid \mathcal{P}^1_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^* \mid \mathcal{P}^2_{\text{crypto}} \tag{2}$$

Whenever $\mathcal{P}^2_{\text{crypto}}$ asks for a new scalar and its idealized control of scalars detects a collision, it asks for another scalar. In this case, by the design of $\mathcal{S}$, it blocks, that is, as $\mathcal{P}^1_{\text{crypto}}$ cannot prevent collision it does not realize $\mathcal{P}^2_{\text{crypto}}$. Our idea is to show that if a generated scalar collides, we can build an adversary on the BDDH assumption (Section 2.5.3) with non-negligible advantage. Since $\mathcal{P}^1_{\text{crypto}}$ and $\mathcal{P}^2_{\text{crypto}}$ have the same behavior in any case other than this one, if we prove that collisions occur with negligible probability, we prove (2).

Suppose that the probability that $\mathcal{P}^2_{\text{crypto}}$ rejects a scalar provided by $\mathcal{S}$ occurs is non-negligible. The, we can say that some scalar x is not fresh in a non-negligible set of executions of $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^* \mid \mathcal{P}^2_{\text{cypto}}$. As the runtime of all machines is polynomially bounded, the number of scalars generated during an execution is also polynomially bounded. Therefore, when a scalar x is generated, it can collide with at most a polynomial number of scalars, which means the probability of a collision is non-negligible. This allows us to build a polynomial time adversary A on the BDDH assumption. An algorithm for A is shown in Algorithm 1. A receives as input the security parameter η, the BDH groups description $(G_1, G_2, r, \hat{e})$ and a challenge (a $\cdot$ G_1,

$\text{a} \cdot G_2$, $\text{b} \cdot G_1$, $\text{c} \cdot G_1$, $\text{c} \cdot G_2$, Z) where $Z = \hat{e}(G_1, G_2)^{\text{a}\cdot\text{b}\cdot\text{c}}$ and, in this case the experiment is BDDH-0, or $Z = \hat{e}(G_1, G_2)^{\text{z}}$ where $\text{z} \in \mathbb{Z}_r^*$ is random, and, in this case, the experiment is BDDH-1. The adversary A has to correctly guess which is the experiment (0 or 1). Upon receiving its input, A generates a random scalar $\text{x} \in \mathbb{Z}_r^*$ and calculates $\text{x} \cdot G_1$. Hence, A verifies if $\text{x} \cdot G_1$ equals $\text{a} \cdot G_1$, $\text{b} \cdot G_1$, or $\text{c} \cdot G_1$, and, if it does not, i.e., there was no collision, A resigns by returning 1. If any verification succeeds, i.e., there was a collision. In this case, A is always able to correctly guess the experiment by checking if Z equals $\hat{e}(G_1, G_2)^{\text{a}\cdot\text{b}\cdot\text{c}}$, because if $\text{x} \cdot G_1 = \text{a} \cdot G_1$, then $\hat{e}(\text{b} \cdot G_1, \text{c} \cdot G_2)^{\text{x} = \text{a}} = \hat{e}(G_1, G_2)^{\text{a}\cdot\text{b}\cdot\text{c}}$, if $\text{x} \cdot G_1 = \text{b} \cdot G_1$, then $\hat{e}(\text{a} \cdot G_1, \text{c} \cdot G_2)^{\text{x} = \text{b}} = \hat{e}(G_1, G_2)^{\text{a}\cdot\text{b}\cdot\text{c}}$, if $\text{x} \cdot G_1 = \text{c} \cdot G_1$, then $\hat{e}(\text{b} \cdot G_1, \text{a} \cdot G_2)^{\text{x} = \text{c}} = \hat{e}(G_1, G_2)^{\text{a}\cdot\text{b}\cdot\text{c}}$. Therefore, if Z equals $\hat{e}(G_1, G_2)^{\text{a}\cdot\text{b}\cdot\text{c}}$, A returns 0. Otherwise, it returns 1. This adversary A is a polynomially bounded algorithm. Notice that the cases A resigns, i.e., without collisions, happen with same probability in both experiments, without influence on the overall advantage of A. When there is a collision, on the other hand, A always responds correctly in both experiments. As the probability of a collision is non-negligible, the advantage $\mathsf{Adv}_{A,\mathcal{B}_t}^{DDH}$ is non-negligible, which violates the DDH assumption. Therefore, (2) holds.

Algorithm 1: Adversary A

```
Input  : $\eta, (G_1, G_2, q, \hat{e}), \text{a} \cdot G_1, \text{a} \cdot G_2, \text{b} \cdot G_1, \text{c} \cdot G_1, \text{c} \cdot G_2, Z$
Output: 0, 1
$\text{x} \xleftarrow{R} \mathbb{Z}_r^*$
if $\text{x} \cdot G_1 = \text{a} \cdot G_1$ then
    if $Z = \hat{e}(\text{b} \cdot G_1, \text{c} \cdot G_2)^{\text{x}}$ then
        return 0 // Correctly Guess
    else
        return 1 // Correctly Guess
else
    if $\text{x} \cdot G_1 = \text{b} \cdot G_1$ then
        if $Z = \hat{e}(\text{a} \cdot G_1, \text{c} \cdot G_2)^{\text{x}}$ then
            return 0 // Correctly Guess
        else
            return 1 // Correctly Guess
    else
        if $\text{x} \cdot G_1 = \text{c} \cdot G_1$ then
            if $Z = \hat{e}(\text{a} \cdot G_1, \text{c} \cdot G_2)^{\text{x}}$ then
                return 0 // Correctly Guess
            else
                return 1 // Correctly Guess
        else
            return 1 // Resigns
```

Step 3 In the third step, we define the hybrid system $\mathcal{P}^3_{\text{crypto}}$, copy of $\mathcal{P}^2_{\text{crypto}}$ except that we change the real implementation of `id-key` keys generation to behave ideally, as defined in $\mathcal{F}_{\text{crypto}}$. This means that, upon a `GenIDKey` request, $\mathcal{P}^3_{\text{crypto}}$ asks $\mathcal{S}$ for the new key k. Upon receiving k, $\mathcal{P}^3_{\text{crypto}}$ rejects if it finds a collision of the key in the Keys set. We now have to show that

$$\mathcal{E} \mid \mathcal{F}^* \mid \mathcal{P}^2_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^* \mid \mathcal{P}^3_{\text{crypto}} \tag{3}$$

In a high level, we have to show that there is no collisions on `id-key` keys, i.e., $\mathcal{P}^3_{\text{crypto}}$ never rejects the keys generated by $\mathcal{S}$. Hence, we have to argue that, (i) upon a command (`ProvideIDKey`, `unknown` or `known`, *pid*, x, *r*, *pid′*, R, y), which is responded by $\mathcal{S}$ with k resulted of a mapping which the last term is $\hat{e}(G_1, G_2)^{\mathrm{x}+\mathrm{y}}$ is never rejected by $\mathcal{P}^3_{\text{crypto}}$; and (ii) upon a command (`ProvideIDKey`, `unknown` or `known`, *pid*, x, *r*, *pid′*, R), which is responded by $\mathcal{S}$ with k resulted of a mapping which the last term is $\hat{e}(\mathsf{R}, S^{\mathrm{I}}_{pid})\hat{e}(G_1, G_2)^{\mathrm{x}}$ is never rejected. Since $\mathcal{P}^2_{\text{crypto}}$ and $\mathcal{P}^3_{\text{crypto}}$ have the same behavior in any case other than these ones, if we prove cases (i) and (ii), we prove (3).

In case (i), $\mathcal{F}_{\text{crypto}}$ requests a key that is formed by two scalars. We known by the design of $\mathcal{F}_{\text{crypto}}$, that it only requests a key if there is no pointer yet created to such a key, therefore, if such request is made, there is no key created between the same scalars before. As scalars in $\mathcal{P}_{\text{crypto}}$ are randomly chosen, and there is no collision, as we just show in the proof of $\mathcal{P}^2_{\text{crypto}}$, the sum of two scalars is also random. Since the exponent used in the paring function to generate the key is the sum of two scalars, and $\hat{e}(G_1, G_2)$ is a generator of the group $\mathbb{G}_T$, the term $\hat{e}(G_1, G_2)^{\mathrm{x}+\mathrm{y}}$ will be uniformly distributed over the group $\mathbb{G}_T$, making the mapping, i.e., k, uniformly distributed, which proves (i).

Case (ii), is actually similar to case (i). First, by the design of $\mathcal{F}_{\text{crypto}}$, it only requests a single key formed by a specific scalar and a specific relation. Besides, the parameters, R and S^{I}_{pid} of the function are such that $\mathsf{R} \in \mathbb{G}_1$ and $S^{\mathrm{I}}_{pid} \in \mathbb{G}_2$, i.e., they can be generated by the groups generators G_1 and G_2. Therefore, as x is uniformly distributed over $\mathbb{Z}^*_r$, the term $\hat{e}(\mathsf{R}, S^{\mathrm{I}}_{pid})\hat{e}(G_1, G_2)^{\mathrm{x}}$ is uniformly distributed over $\mathbb{G}_T$ and has never been generated by $\mathcal{P}_{\text{crypto}}$, making the mapping, i.e., k, uniformly distributed, which proves (ii). Hence, in fact, the keys generated in cases (i) and (ii) are fresh, which proves (3).

Step 4 In the fourth step, we change the real implementation of symmetric encryption and decryption and key derivation in the hybrid system $\mathcal{P}^4_{\text{crypto}}$ to behave as in $\mathcal{F}_{\text{crypto}}$, i.e., these operations are performed ideally. We now have to show that

$$\mathcal{E} \mid \mathcal{F}^* \mid \mathcal{P}^3_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^* \mid \mathcal{P}^4_{\text{crypto}} \tag{4}$$

In this case, comparing to the original $\mathcal{P}_{\text{crypto}}$, we do not propose any change other than adding the key derivation of `id-key` keys. However, `id-key` keys can be used to derive other keys as any existent `pre-key` in $\mathcal{P}_{\text{crypto}}$. Therefore, based on the results from [62], (4) holds.

Step 5 In the fifth step, we change the real implementation of MAC generation and verification in the hybrid system $\mathcal{P}^5_{\text{crypto}}$ to behave as in $\mathcal{F}_{\text{crypto}}$.

$$\mathcal{E} \mid \mathcal{F}^* \mid \mathcal{P}^4_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^* \mid \mathcal{P}^5_{\text{crypto}} \tag{5}$$

As we do not propose any change on such operations, the original proof from [62] still holds, therefore (5) holds.

With the results in (1), (2), (3), (4), and (5) we have that our extension in $\mathcal{F}_{\text{crypto}}$ can be realized by $\mathcal{P}_{\text{crypto}}$ with the cryptosystem $\Sigma_{\texttt{IBC}}$:

$$\mathcal{F}^* \mid \mathcal{P}_{\text{crypto}} \leq \mathcal{F}^* \mid \mathcal{F}_{\text{crypto}}$$

5.6 Functionality to be realized

After having extended the functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ and proving its realization $\mathcal{P}_{\text{crypto}}$ to support the identity-based operations, we have in the IITM model ideal and real implementations to support all cryptographic primitives that we need to analyze the security of AoT protocols. Hence, in this section, we present a functionality that ideally provides the universally composable security properties of key agreement protocols. In the end, our goal is to prove that AoT protocols, which are a composition of higher-level protocols with key agreement protocols, are indeed securely composable.

5.6.1 Mutually Authenticated Key Exchange

According to [83], a key exchange protocol is compliant if the honest parties who comply with the protocol specification always complete the protocol having computed a common key and knowledge of the identities of the parties with whom the key is shared. Besides capturing such an objective, the standard universally composable security notions of key exchange protocols [25] also capture the security in scenarios with or without perfect forward secrecy. That is because, although perfect forward secrecy is an important security property, it is not required in all contexts [24]. Indeed, as discussed in Section 4.2.1, we don't have in AoT any context where perfect forward secrecy is a requirement. In this section, we present $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, an idealization for a two-party mutually authenticated key exchange. Our functionality is based on the general $\mathcal{F}^{\text{MA}}_{\text{key-use}}$ proposal from [62]. As in the standard universally composable security notions of key exchange protocols only perfect forward secrecy can be modeled, we describe $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ as a general functionality that follows the standard without contemplating perfect forward secrecy [25].

In a high level, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ acts as an idealized trusted party that allows two parties, initiator and responder, to send a command to establish a fresh session with each other, which is responded by the functionality with a session key pointer. $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ guarantees that when it outputs a session key pointer, the particular instance of the party that gets access to the key is uncorrupted and in a session with its intended communication partner. Analogously, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ guarantees that the particular user instance of the partner only gets access to the same key if it is also uncorrupted.

An important aspect of $\mathcal{F}^{\text{MA}}_{\text{key-use}}$ that we use in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ is that it outputs to the environment session key pointers instead of the session key itself. The idea is to offer the users a way to perform ideal cryptographic operations in $\mathcal{F}_{\text{crypto}}$ with the established key while they have an active session in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. This is a feature that significantly simplifies the analysis of higher-level protocols that can not only establish the session key but also use it in an ideal way. Our functionality differs from the original $\mathcal{F}^{\text{MA}}_{\text{key-use}}$ because, while in session, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ gives the users access to a digital signature functionality. Besides, we define that $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ return not only the session key pointer but also a fresh stamp of the session that can be used by higher-level protocols during the session. Therefore, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ can be seen as the composition of the functionality for two-party mutually authenticated key exchange with key usability, which in modeled as $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}_{\text{crypto}}$, and a functionality for digital signatures, which, in turn, can also be provided by $\mathcal{F}_{\text{crypto}}$.

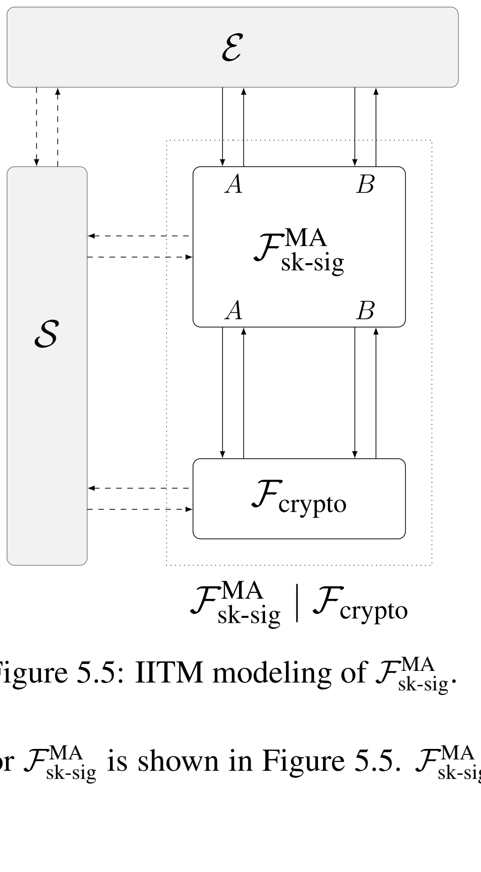

$\mathcal{E}$
$\mathcal{S}$
A
B
$\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
A
B
$\mathcal{F}_{\text{crypto}}$
$\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}_{\text{crypto}}$

Figure 5.5: IITM modeling of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$.

The IITM modeling for $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ is shown in Figure 5.5. $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ comprises one single machine with two pairs of i/o tapes to interface with the environment, one for each protocol role, the initiator A and the responder B, the network tapes to communicate with the simulator $\mathcal{S}$, and two pairs of i/o tapes to use $\mathcal{F}_{\text{crypto}}$ as the cryptographic primitives' provider, one for each protocol role. $\mathcal{F}_{\text{crypto}}$ also has its network interface to communicate with $\mathcal{S}$, but it does not interface directly with the environment. $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ is parameterized with a symmetric key type $t \in \{\texttt{authenc-key}, \texttt{mac-key}\}$ to determine the type of the key established using the functionality, i.e., the type of cryptographic operation that will be performed with the key while the users are in an active session. As in AoT our protocols only use session keys to perform MAC- and authenticated encryption-related operations, we restrict the types of keys we allow in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, however we could allow other types of keys without major considerations.

Similarly to $\mathcal{F}_{\text{crypto}}$, the users in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ are fully identified by the tuple (*pid*, *lsid*, *r*), with party identification *pid* $\in \{0,1\}^*$, local session identification *lsid* $\in \{0,1\}^*$, and role *r* $\in \{A, B\}$. Besides, each message in the i/o tapes of the system is prefixed with (*pid*, *lsid*).

The functionality $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ keeps an internal register *state*(*pid*, *lsid*, *r*) $\rightarrow \{\perp$, `started`, `inSession`, `exchangeFinished`, `sessionClosed`, `corrupted`$\}$, which represents the state of each user (*pid*, *lsid*, *r*) in the key exchange and it is initially set to $\perp$. The intended communication party of each user is also kept registered in a map *partner*(*pid*,*lsid*,*r*) $\rightarrow$*pid*′.

As any other instance of $\mathcal{F}_{\text{crypto}}$, when initialized by $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, it receives the digital signature scheme pair of keys of the users from the simulator, as described in Section 5.3.3. In the following we describe, including graphically, how $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ behaves, its interaction with environment, simulator, and $\mathcal{F}_{\text{crypto}}$.

- The user (*pid*, *lsid*) starts a key exchange with the intended communication party *pid*′ by sending the command (`InitKE`, *pid*′) on the input interface of role A in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ (message ① in Figure 5.6). If *state*(*pid*, *lsid*, A) $= \perp$, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ sets *state*(*pid*, *lsid*, A) $=$ `started`, records the intended communication partner of the instance *partner*(*pid*,*lsid*,A) $=$*pid*′, and informs the simulator about the request by sending the message ((`InitKE`, *pid*′), (*pid*, *lsid*, A)) on the network interface (message ② in Figure 5.6). The analogous scenario of a user (*pid*′, *lsid*′) with role B and indented communication party *pid* is shown in Figure 5.6, messages ①′ and ②′.

- The simulator may declare two users (*pid*, *lsid*, A) and (*pid*′, *lsid*′, B) as partners in a session by sending the message (`GroupSession`, (*pid*, *lsid*, A), (*pid*′, *lsid*′, B)) to $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ (message ① in Figure 5.7). The functionality only creates such a session if the users' states are `started` or `corrupted` and that neither of them are already part of another recorded session. $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ allows the simulator to group an uncorrupted user, i.e., with state `started`, with a corrupted one, however, as we present in the next steps, a corrupted user do not get a pointer to the session key.

  Hence, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ uses the command `GetPSK` to ask $\mathcal{F}_{\text{crypto}}$ for a pointer to an unknown key of type t (parameter of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$) and name `name` $=$ ((*pid*, *lsid*, A), (*pid*′, *lsid*′, B)) (messages

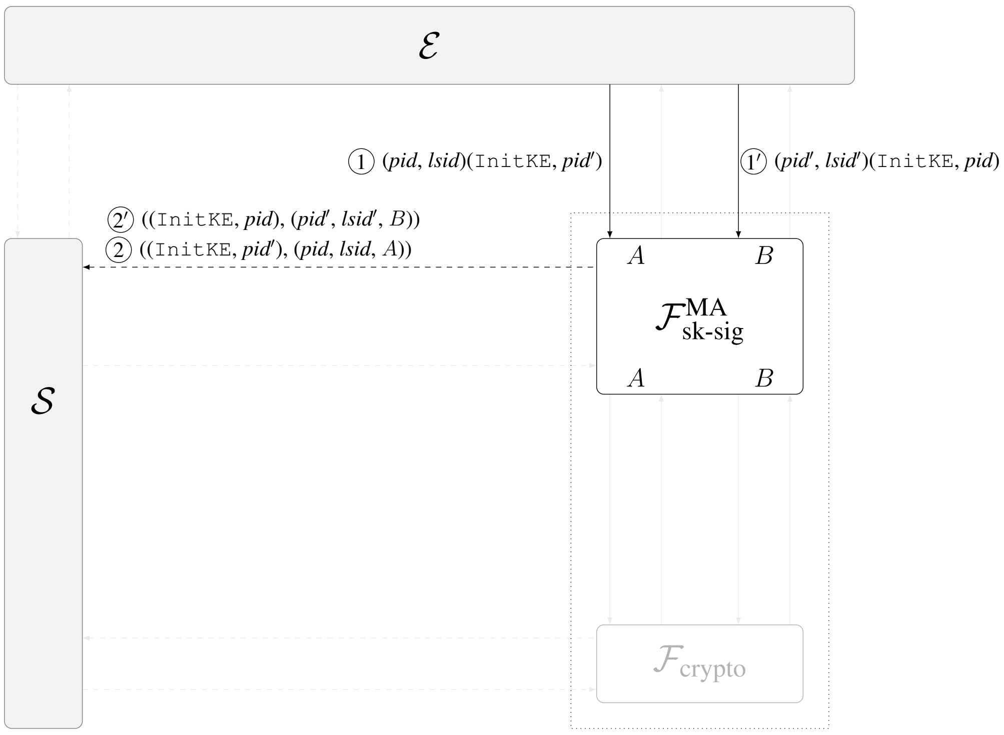
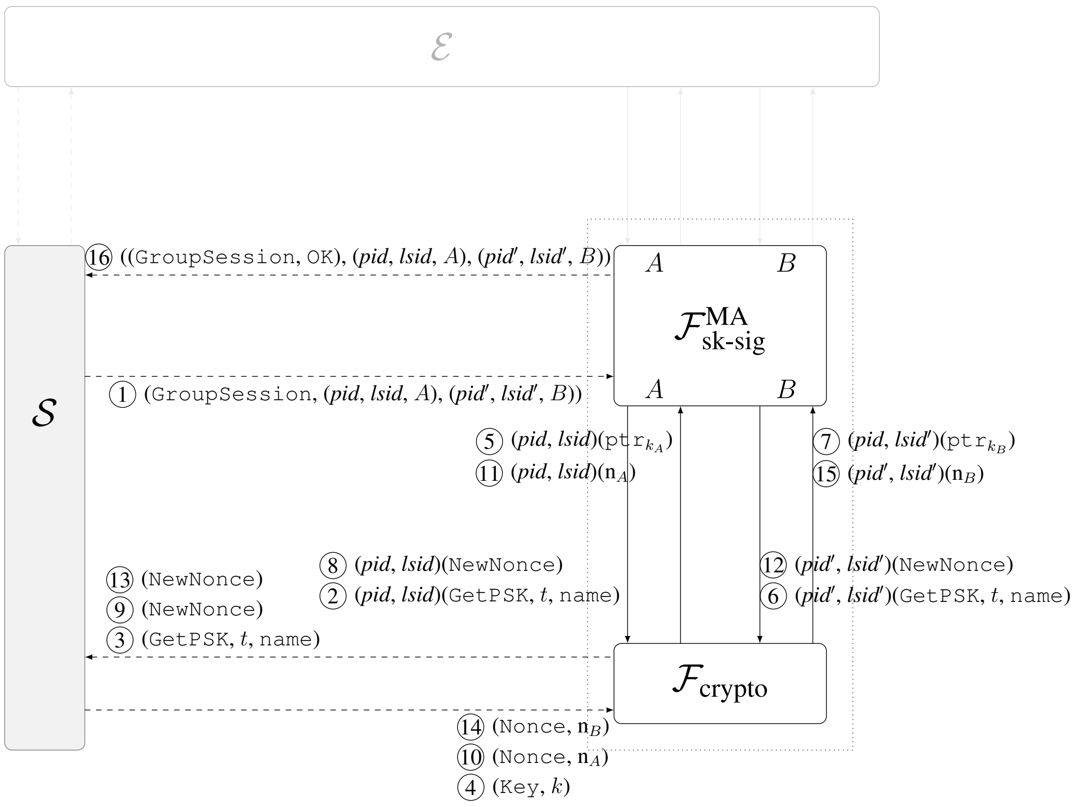

$\mathcal{E}$
① $(pid, lsid)(\texttt{InitKE}, pid')$
①′ $(pid', lsid')(\texttt{InitKE}, pid)$
②′ $((\texttt{InitKE}, pid), (pid', lsid', B))$
② $((\texttt{InitKE}, pid'), (pid, lsid, A))$
A B
$\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
A B
$\mathcal{S}$
$\mathcal{F}_{\text{crypto}}$

Figure 5.6: Key exchange initialization in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$.

② and ⑥ in Figure 5.7). According to the design of $\mathcal{F}_{\text{crypto}}$, explained in Section 5.3 and detailed in the original proposal [66], the functionality actually asks the adversary (simulator) for the value of keys (message ③ in Figure 5.7), this is an important aspect that will be used to prove our protocols realize $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Then, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ uses the command `NewNonce`, once for each role, to ask $\mathcal{F}_{\text{crypto}}$ fresh nonces to be used as a fresh stamp for the session (messages ⑧ and ⑫ in Figure 5.7). Similarly to keys, $\mathcal{F}_{\text{crypto}}$ also asks the simulator for the value of new nonces (messages ⑨ and ⑬ in Figure 5.7). After receiving the nonces from $\mathcal{F}_{\text{crypto}}$ (messages ⑪ and ⑮ in Figure 5.7), $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ changes the uncorrupted instances stages to `inSession`, records (`Session`, $(pid, lsid, A)$, $\texttt{ptr}_{k_A}$, n_A, $(pid', lsid', B)$, $\texttt{ptr}_{k_B}$, n_B) as a session, and responds `OK` to the simulator (⑯ in Figure 5.7).

This idea of letting the simulator in charge of grouping instances in sessions in the functionality is presented in [65]. As in the IITM model the simulator internally executes the protocol that needs to be shown to realize the functionality, it internally simulates the protocol execution with the same entries the functionality gets from the environment. Therefore, the protocol needs a mechanism that correctly groups the users in sessions and showing that such a mechanism works is part of the security analysis.

- The simulator can request $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to complete the key exchange for a user $(pid, lsid, A)$. To do so, it sends (`FinishKE`, $(pid, lsid, A)$) to $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ (message ① in Figure 5.8). $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ checks if $state(pid, lsid, A) = \texttt{inSession}$ and that there is a record of a session between $(pid, lsid, A)$ and an instance of its intended communication partner, for instance, there



$\mathcal{E}$
$\mathcal{S}$
$\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
A B
A B
$\mathcal{F}_{\text{crypto}}$
⑯ ((GroupSession, OK), (pid, lsid, A), (pid′, lsid′, B))
① (GroupSession, (pid, lsid, A), (pid′, lsid′, B))
⑤ (pid, lsid)(ptr$_{k_A}$)
⑪ (pid, lsid)(n$_A$)
⑦ (pid, lsid′)(ptr$_{k_B}$)
⑮ (pid′, lsid′)(n$_B$)
⑧ (pid, lsid)(NewNonce)
② (pid, lsid)(GetPSK, t, name)
⑫ (pid′, lsid′)(NewNonce)
⑥ (pid′, lsid′)(GetPSK, t, name)
⑬ (NewNonce)
⑨ (NewNonce)
③ (GetPSK, t, name)
⑭ (Nonce, n$_B$)
⑩ (Nonce, n$_A$)
④ (Key, k)

Figure 5.7: Grouping instances into a session in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$.

exists a record such as (`Session`, (*pid*, *lsid*, A), `ptr`$_{k_A}$, n$_A$, (*pid*′, *lsid*′, B), `ptr`$_{k_B}$, n$_B$) and *partner*(*pid*,*lsid*,A) →*pid*′. It not, i.e., the instance is corrupted or there is no session for the instance, the user does not receive a session key pointer. Otherwise, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ sets the *state*(*pid*, *lsid*, A) = `exchangeFinished`, and outputs the message (`Established`, `ptr`$_{k_A}$, (n$_A$, n$_B$)) to the user (*pid*, *lsid*) on the environment (message ② in Figure 5.8), where `ptr`$_{k_A}$ is the pointer to the established key and (n$_A$, n$_B$) is what we call the session stamp. The analogous scenario where a key exchange for a user (*pid*′, *lsid*′) with role B is completed and the session key pointer `ptr`$_{k_B}$ is output is shown in Figure 5.8 in messages ①′ and ②′, respectively.

- A user (*pid*, *lsid*) which has *state*(*pid*, *lsid*, *r*) = `exchangeFinished` has access to the established key through pointer `ptr`$_{k_r}$, $r \in \{A, B\}$, and can use $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to perform cryptographic operations using $\mathcal{F}_{\text{crypto}}$, including the ones related to the digital signature scheme. $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ receives the cryptographic commands from the user (messages ① and ①′ in Figure 5.9), forwards them to $\mathcal{F}_{\text{crypto}}$ to be processed (messages ② and ②′ Figure 5.9), receives the results from $\mathcal{F}_{\text{crypto}}$ (messages ③ and ③′ in Figure 5.9), and returns the result to the users (messages ④ and ④′ in Figure 5.9). The symmetric operations $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ can intermediate in $\mathcal{F}_{\text{crypto}}$ for user (*pid*, *lsid*, *r*) are: `Enc`, `Dec`, `Mac`, `MacVerify`, `Corrupted?`, and `Equal?`. The digital signature scheme operations $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ exposes, in turn, are: `GetPubKeySig`, `Sign`, `SigVerify`, and `CorruptSig?`.

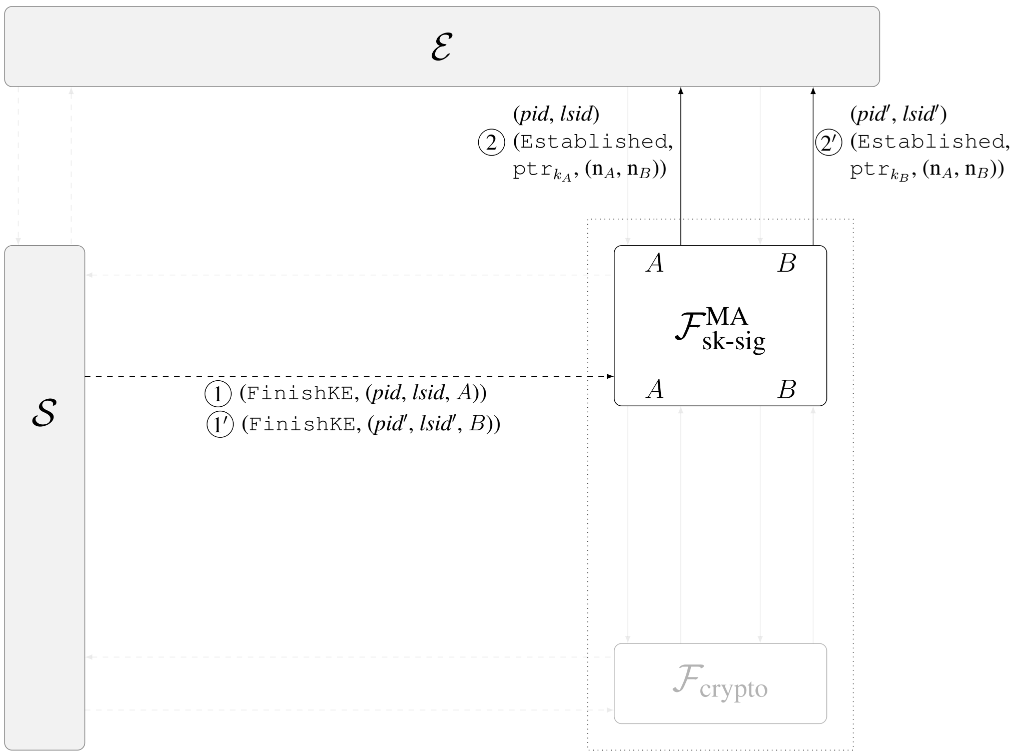
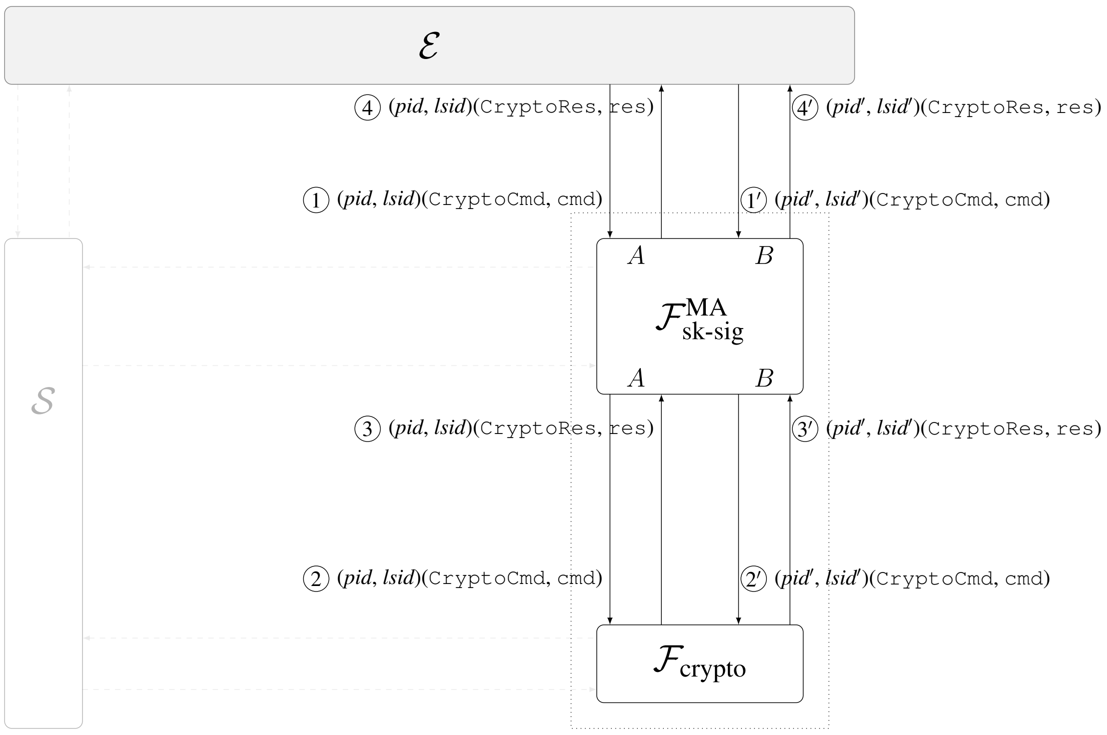

$\mathcal{E}$
$(pid, lsid)$
② (Established, $\texttt{ptr}_{k_A}$, $(\mathrm{n}_A, \mathrm{n}_B)$)
$(pid', lsid')$
②′ (Established, $\texttt{ptr}_{k_B}$, $(\mathrm{n}_A, \mathrm{n}_B)$)
$\mathcal{S}$
A
B
$\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$
A
B
① (FinishKE, $(pid, lsid, A)$)
①′ (FinishKE, $(pid', lsid', B)$)
$\mathcal{F}_{\text{crypto}}$

Figure 5.8: Finishing the key exchange in $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$.

- The user $(pid, lsid)$ which has $state(pid, lsid, r) = \texttt{exchangeFinished}$ can close the session in $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ by sending the message (CloseSession) (messages ① and ①′ in Figure 5.10). $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ then sets $state(pid, lsid, r) = \texttt{sessionClosed}$, revokes the access of $(pid, lsid, r)$ to the established key, and informs the simulator about the event by sending (CloseSession, $(pid, lsid, r)$) on the network tape (messages ② and ②′ in Figure 5.10). After receiving an acknowledge from the simulator ((messages ③ and ③′ in Figure 5.10), $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ responds OK to the user (messages ④ and ④′ in Figure 5.10).

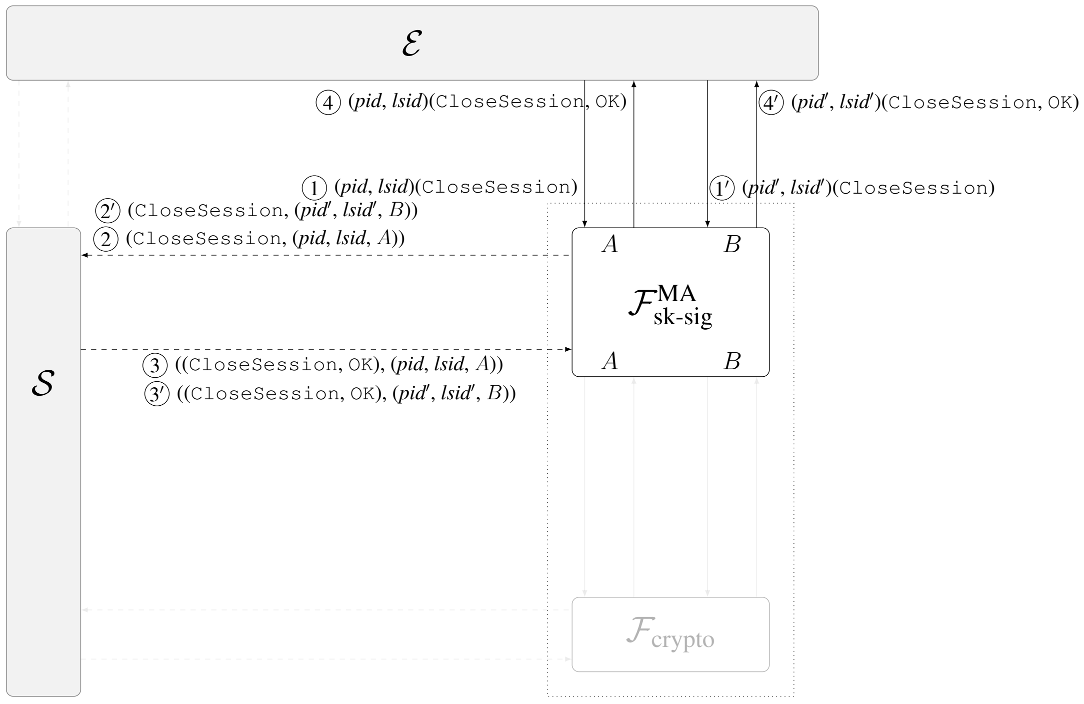
- A user $(pid, lsid)$ in the environment can request its corruption status in $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ by sending the (Corrupt?) command (messages ① and ①′ in Figure 5.11). If $state(pid, lsid, r) = \perp$, i.e., if the key exchange has not yet been started, before answering the user, $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ first sends the message (CorruptUser?, $(pid, lsid, r)$) to the simulator asking if the user should be corrupted (messages ② and ②′ in Figure 5.11), and sets $state(pid, lsid, r) = \texttt{corrupted}$ if the answer if the corruption flag is set to true in the response message (messages ③ and ③′ in Figure 5.11). Last, $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ sets bool accordingly and returns (Corrupt, bool) to the user (messages ④ and ④′ in Figure 5.11).

In $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ the corruption model is such that the simulator can corrupt a user before a key exchange starts and after the session is close, but not during the session. More exactly, a user $(pid, lsid, r)$ with $state(pid, lsid, r) = \perp$ or sessionClosed is corrupted by the simulator when it sends the message (Corrupt, $(pid, lsid, r)$, bool) with bool = true to $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ or, when asked by $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ at the beginning of the key exchange, as explained in the corruption status



$\mathcal{E}$
④ (*pid*, *lsid*)(CryptoRes, res)
④′ (*pid*′, *lsid*′)(CryptoRes, res)
① (*pid*, *lsid*)(CryptoCmd, cmd)
①′ (*pid*′, *lsid*′)(CryptoCmd, cmd)
A B
$\mathcal{F}_{\text{sk-sig}}^{\text{MA}}$
A B
$\mathcal{S}$
③ (*pid*, *lsid*)(CryptoRes, res)
③′ (*pid*′, *lsid*′)(CryptoRes, res)
② (*pid*, *lsid*)(CryptoCmd, cmd)
②′ (*pid*′, *lsid*′)(CryptoCmd, cmd)
$\mathcal{F}_{\text{crypto}}$

Figure 5.9: Performing cryptographic operations in $\mathcal{F}_{\text{crypto}}$ through $\mathcal{F}_{\text{sk-sig}}^{\text{MA}}$.



$\mathcal{E}$
④ (*pid*, *lsid*)(CloseSession, OK)
④′ (*pid*′, *lsid*′)(CloseSession, OK)
① (*pid*, *lsid*)(CloseSession)
①′ (*pid*′, *lsid*′)(CloseSession)
②′ (CloseSession, (*pid*′, *lsid*′, B))
② (CloseSession, (*pid*, *lsid*, A))
A B
$\mathcal{F}_{\text{sk-sig}}^{\text{MA}}$
A B
$\mathcal{S}$
③ ((CloseSession, OK), (*pid*, *lsid*, A))
③′ ((CloseSession, OK), (*pid*′, *lsid*′, B))
$\mathcal{F}_{\text{crypto}}$

Figure 5.10: Closing a session in $\mathcal{F}_{\text{sk-sig}}^{\text{MA}}$.

request. All messages to or from corrupted users in i/o interfaces are forwarded by $\mathcal{F}_{\text{sk-sig}}^{\text{MA}}$ to the simulator on the network tape. To model the lack of perfect forward secrecy, $\mathcal{F}_{\text{sk-sig}}^{\text{MA}}$ allows the simulator to execute operations in $\mathcal{F}_{\text{crypto}}$ using the corrupted instance (*pid*, *lsid*, *r*) in the name of user (*pid*, *lsid*). However, the operations available in such case are those that have direct relation with the session keys. For example, the simulator can execute the GetPSK command

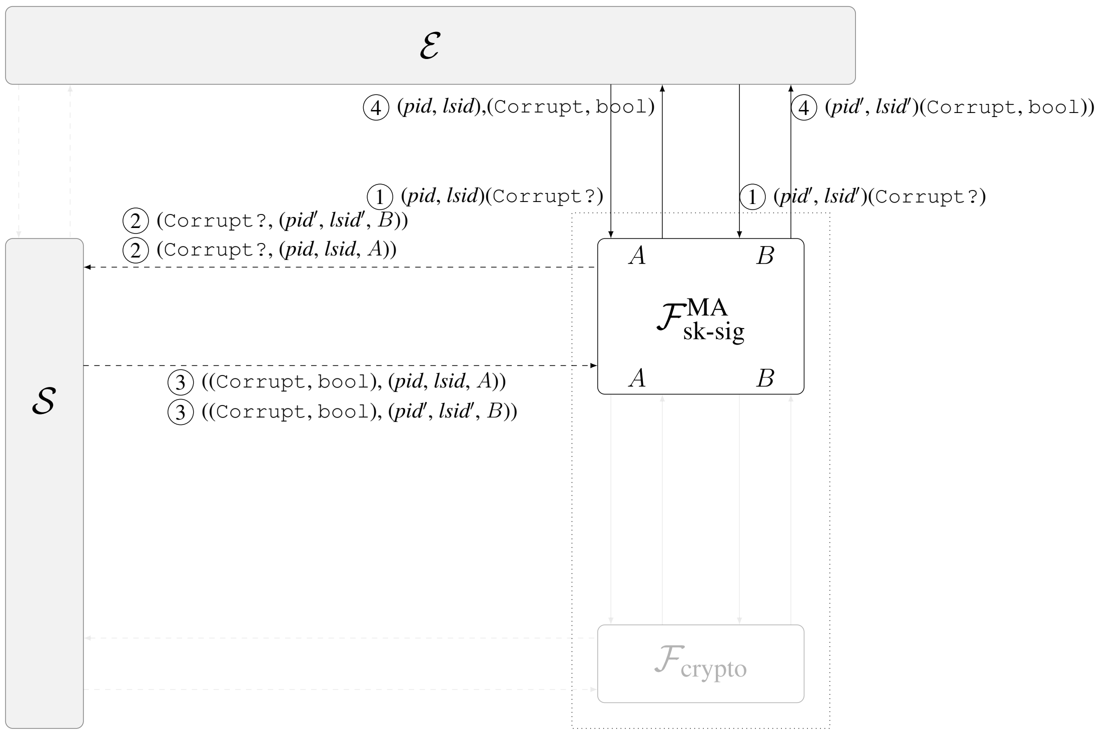

$\mathcal{E}$
④ (*pid*, *lsid*),(Corrupt, bool)
④ (*pid*′, *lsid*′)(Corrupt, bool))
① (*pid*, *lsid*)(Corrupt?)
① (*pid*′, *lsid*′)(Corrupt?)
② (Corrupt?, (*pid*′, *lsid*′, *B*))
② (Corrupt?, (*pid*, *lsid*, *A*))
$\mathcal{S}$
A B
$\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
A B
③ ((Corrupt, bool), (*pid*, *lsid*, *A*))
③ ((Corrupt, bool), (*pid*′, *lsid*′, *B*))
$\mathcal{F}_{\text{crypto}}$

Figure 5.11: Corruption status request.

using an appropriate name of the key to get a pointer to key established in a past session and use the key as in the higher-level protocol, potentially revealing the secrets of such session. On the other hand, the simulator cannot access the digital signature scheme operations using a corrupted instance, otherwise, we would have to require that for corrupting an instance, the signing key of the party would have to be previously corrupted.

5.7 Security Analysis of AoT

In this section, we present security analysis of AoT in the universal composability IITM model. Our idea is to first show that AoT's key establishment protocols *SessionKey* (Section 4.2.1) and *SessionKeyDerivation* (Section 4.2.3) realize the general universally composable security properties of key agreement protocols, i.e., the functionality $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Then, based on the modular design of AoT's protocols, we show that every module is a composition of a higher-level protocol with a key agreement protocol with key usability ($\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$) and, as AoT is cryptosystem agnostic, any standard cryptographic algorithm that fulfills $\mathcal{F}_{\text{crypto}}$ requirements can be used to instantiate a secure version of AoT, i.e., AoT is securely universally composable. We start with the analysis of *SessionKey* in Section 5.7.1, then we analyze protocol *SessionKey-Derivation* in Section 4.2.3, and argue about AoT's prototols in Section 5.7.3.

5.7.1 SessionKey Protocol

The security analysis of *SessionKey* protocol (Section 4.1, Protocol 4.1) consists of proving that the protocol is as secure as the functionality for mutually authenticated key exchange with key usability and in-session access to digital signatures $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, described in Section 5.6.1. We present the protocol modeling in Section 5.7.1.1, describe its execution in Section 5.7.1.2, and develop the proof in Section 5.7.1.4.

5.7.1.1 SessionKey IITM Modeling

The *SessionKey* protocol (Protocol 4.1), referred as $\mathcal{P}_{\text{SK}}$, has a straightforward two-party multi-session protocol modeling $\mathcal{P}_{\text{SK}} = !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}}$, which consists of two machines M_A and M_B, one for each protocol role, A as the initiator device and B as the responder one. M_A and M_B have i/o tapes to interface with the environment $\mathcal{E}$, the network interface for adversary $\mathcal{A}$ communication, and i/o tapes to ideally use $\mathcal{F}_{\text{crypto}}$ as the cryptographic primitives provider, as shown in Figure 5.12.

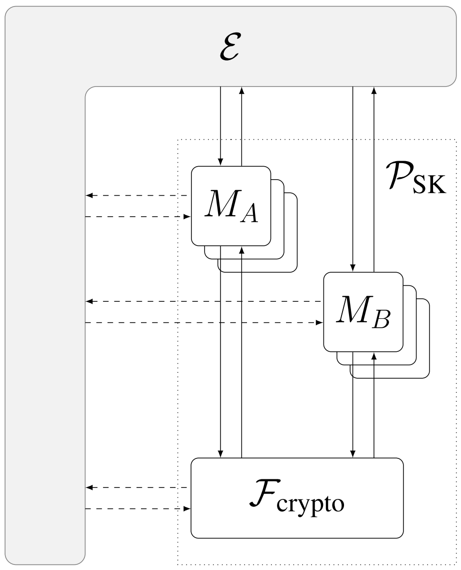


Figure 5.12: *SessionKey* ($\mathcal{P}_{\text{SK}}$) protocol modeling.

Users in $\mathcal{P}_{\text{SK}}$ are fully identified by the tuple (*pid*, *lsid*, *r*), where *pid* and *lsid* are party and local session identifiers, totally managed by the environment, and *r* $\in \{A, B\}$ is the protocol role. Therefore, in a run of $\mathcal{P}_{\text{SK}}$, there is a single instance (*pid*, *lsid*, *r*) of a machine M_r per user (*pid*, *lsid*) in the environment. All messages in the i/o tapes are prefixed with (*pid*, *lsid*), making the modeling totally compatible with $\mathcal{F}_{\text{crypto}}$ identification scheme. $\mathcal{P}_{\text{SK}}$ is parameterized with a symmetric key type $t \in \{$`authenc-key`, `mac-key`$\}$ to determine the type of the key established using the protocol.

Each machine maintains an internal variable to control its current state in execution of the protocol. This variable allows a machine to parse the messages received on its tapes according to what has been executed up to that particular point. As the machines play different roles, they have different stages in an execution, M_A has a control *state*$_A \in \{\perp$, `started`, `finished`, `sessionClosed`$\}$, initially set to $\perp$, M_B, in turn, has a control *state*$_B \in \{\perp$, `started`, `secondMessageSent`, `finished`, `sessionClosed`$\}$, initially set to $\perp$. The machines also keep their corruption status (`false` or `true`) initially set to `false`, and a register with the party identifier of the intended communication partner, initially empty.

5.7.1.2 SessionKey Execution

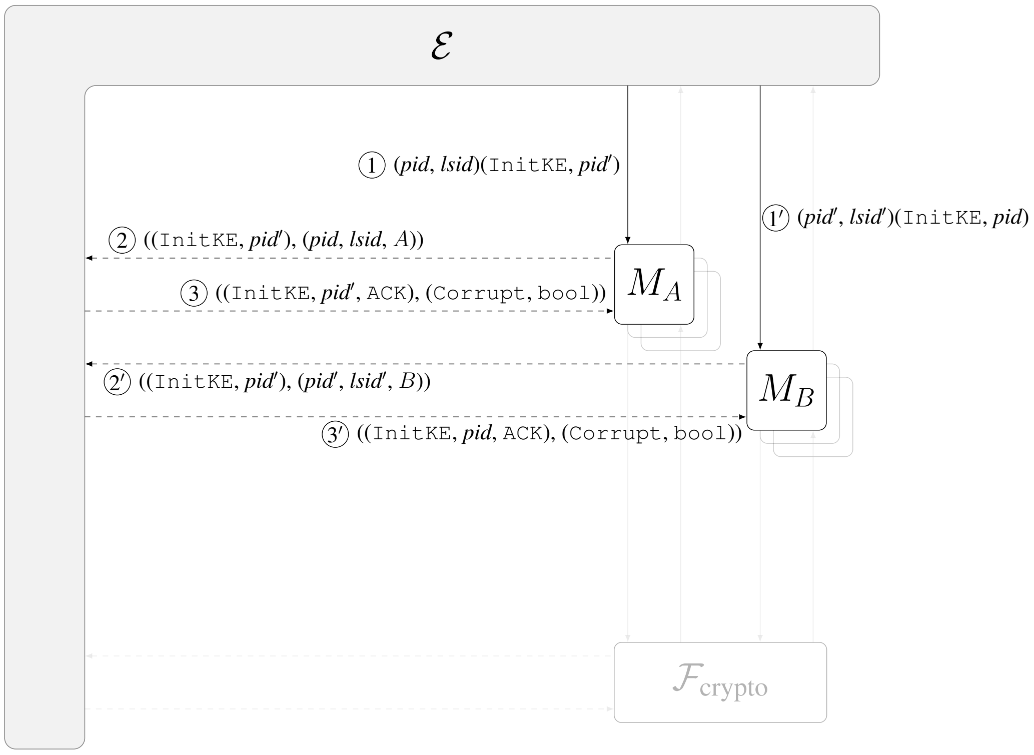
In terms of code, the machines M_A and M_B implement the specification of the corresponding role in the *SessionKey* protocol (Protocol 4.1) with some modeling necessities that we describe during the section. The first necessity is to provide to the environment the same interface as the functionality *SessionKey* wants to realize. As we want to prove $\mathcal{P}_{\text{SK}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}_{\text{crypto}}$, M_A and M_B must provide to $\mathcal{E}$ the same interface as $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Therefore, a user (*pid*, *lsid*) in $\mathcal{E}$ must be able to:

- send the command (`InitKE`, *pid'*) to start a key exchange with desired party *pid'*;
- Perform the desired cryptographic operations in $\mathcal{F}_{\text{crypto}}$ after the session key pointer is returned. Analogously to $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, the available symmetric operations are: `Enc`, `Dec`, `Mac`, `MacVerify`, `Corrupted?`, and `Equal?`, and the digital signature scheme operations are: `GetPubKeySig`, `Sign`, `SigVerify`, and `CorruptSig?`.
- Send the command (`CloseSession`) after performing the desired cryptographic operations in the session;
- Send the command (`Corrupt?`) to get its corruption status.

In the following, we describe, including graphically, the code of machines M_A and M_B and their behaviors in a normal run of $\mathcal{P}_{\text{SK}}$. The machines are implemented such that all messages are sent in **Compute** mode. Therefore, if a message is sent, a machine has executed in **CheckAddress** mode and returned **accept**. This means that all the protocol messages are built and verified in the **CheckAddress** mode piece.

- Key exchange initialization. The machines have internal state set to $\perp$.

– Machines M_A and M_B have *state* = $\perp$, and receive (one at a time) messages (*pid*, *lsid*)(`InitKE`, *pid′*) and (*pid′*, *lsid′*)(`InitKE`, *pid*) (messages ① and ①′ in Figure 5.13, respectively) from the environment, which start key exchanges for users (*pid*, *lsid*) and (*pid′*, *lsid′*), that have as the intended communication partners the parties *pid′* and *pid*, respectively. The machines **accept** the messages, i.e., they are now user instances (*pid*, *lsid*, A) and (*pid′*, *lsid′*, B) in the protocol, and proceed to **Compute** mode to output a message.



$\mathcal{E}$
① (*pid*, *lsid*)(`InitKE`, *pid′*)
①′ (*pid′*, *lsid′*)(`InitKE`, *pid*)
② ((`InitKE`, *pid′*), (*pid*, *lsid*, A))
③ ((`InitKE`, *pid′*, `ACK`), (`Corrupt`, `bool`))
M_A
②′ ((`InitKE`, *pid′*), (*pid′*, *lsid′*, B))
M_B
③′ ((`InitKE`, *pid*, `ACK`), (`Corrupt`, `bool`))
$\mathcal{F}_{\text{crypto}}$

Figure 5.13: Key exchange initialization in $\mathcal{P}_{\text{SK}}$.

– M_A and M_B inform the key exchange request to the adversary by sending, respectively, ((*pid*, *lsid*), (`InitKE`, *pid′*)) and ((*pid′*, *lsid′*), (`InitKE`, *pid*)) on their output network tapes (messages ② and ②′ in Figure 5.13, respectively).

– The adversary acknowledges the machines about the initialization, which, in turn, can start the protocol execution. This specific acknowledge message on the beginning can be used by the adversary to corrupt the users by setting the corruption flag `bool` as `true`, as shown in messages ③ and ③′ in Figure 5.13. If such request is made, the corruption status of the machine is set to `true`, and the machine stops without producing output, which makes the environment (the master machine of the system) to be activated. Otherwise, M_A sets *state*$_A$ = `started`, M_B sets *state*$_B$ = `started`, they record the intended communication partner (*pid′* in M_A and *pid* in M_B), and continue with protocol execution. M_A prepares to send the first protocol message, while M_B concludes the **CheckAddress** mode returning **accept**, proceeds to **Compute** mode, which is concluded without outputting a message, activating the environment.

- M_A continues in **CheckAddress** mode from the initialization acknowledgment message and sends the first message of the protocol on its output network tape.

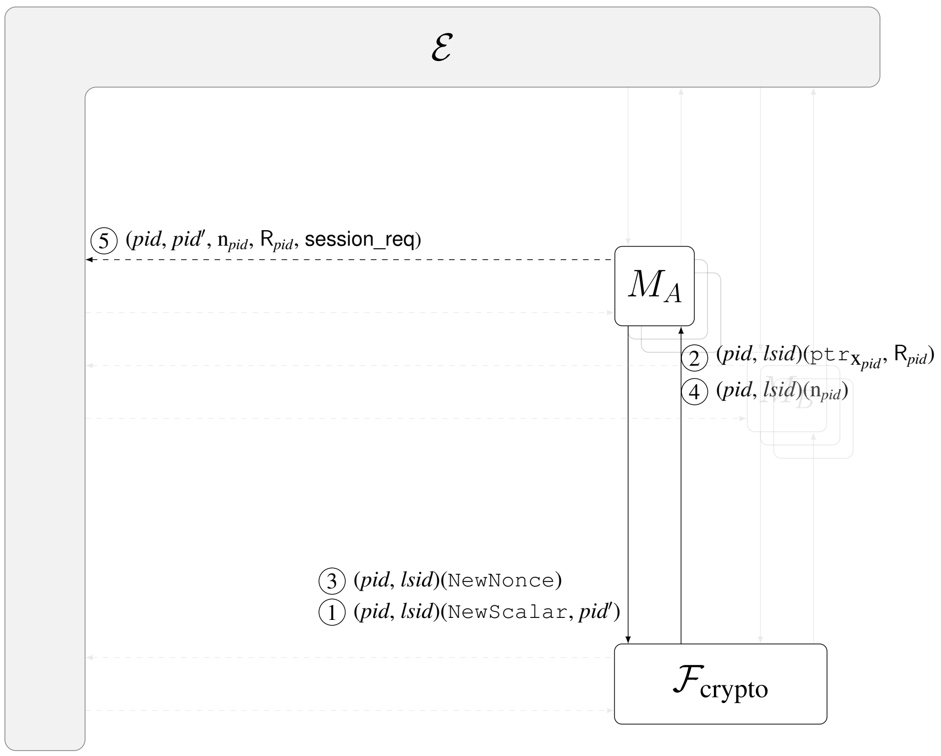

$\mathcal{E}$
⑤ (*pid*, *pid′*, n$_{pid}$, R$_{pid}$, session_req)
M_A
② (*pid*, *lsid*)(ptrx$_{pid}$, R$_{pid}$)
④ (*pid*, *lsid*)(n$_{pid}$)
③ (*pid*, *lsid*)(NewNonce)
① (*pid*, *lsid*)(NewScalar, *pid′*)
$\mathcal{F}_{\text{crypto}}$

Figure 5.14: First message of *SessionKey* protocol.

  - Protocol specification:

$$\begin{aligned} A: &\quad P^{\mathrm{I}}_{B,\mathcal{Z}} := \textsc{Map}_1(id_{B,\mathcal{Z}}) \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}} \\ A: &\quad \mathsf{R}_A := \mathrm{x}_A \cdot P^{\mathrm{I}}_{B,\mathcal{Z}} \\ A \to B: &\quad id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathrm{n}_A, \mathsf{R}_A, \textsf{session\_req} \end{aligned}$$

  - M_A asks $\mathcal{F}_{\text{crypto}}$ for a scalar x_{pid} related to the identity of the recorded intended communication partner *pid′* through the command (`NewScalar`, *pid′*), obtaining the scalar pointer `ptr`x_{pid} and the public relation R_{pid} (messages ① and ② in Figure 5.14)[3].
  - M_A uses the command (`NewNonce`) to ask $\mathcal{F}_{\text{crypto}}$ for a fresh nonce, and receives n_{pid} (messages ③ and ④ in Figure 5.14) and concludes the **CheckAddress** mode returning **accept**.
  - In **Compute** mode, M_A sends the first protocol message (*pid*, *pid′*, n_{pid}, R_{pid}, session_req) on its output network tape (message ⑤ in Figure 5.14).

[3]For the sake of simplicity, we omit the full identification of the user instance on the components generated in $\mathcal{F}_{\text{crypto}}$. For instance, the best representation of a scalar pointer in a session would be `ptr`$\mathrm{x}_{pid,lsid}$ and of the public relation would be $\mathsf{R}_{pid,lsid}$, because they are always tied to a specific session.

- M_B receives the first message of the protocol on its input network tape, it has *state*$_B$ = `started`, it prepares the second message of the protocol, and sends it on its output network tape.

  – Protocol specification:

$$\begin{aligned}
B:&\ \ P^{\mathrm{I}}_{A,\mathcal{Z}} \coloneqq \textsc{Map}_1(id_{A,\mathcal{Z}})\ \cdot G^{\mathrm{I}}_{1,\mathcal{Z}} + P^{\mathrm{I}}_{S,\mathcal{Z}}\\
B:&\ \ \mathsf{R}_B \coloneqq \mathrm{x}_B \cdot P^{\mathrm{I}}_{A,\mathcal{Z}}\\
B:&\ \ k^{\mathrm{ID}}_{B,A} \coloneqq \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_A, S^{\mathrm{I}}_{B,\mathcal{Z}})\hat{e}(G^{\mathrm{I}}_{1,\mathcal{Z}}, G^{\mathrm{I}}_{2,\mathcal{Z}})^{\mathrm{x}_B})\\
B:&\ \ k^{\mathrm{MAC}}_{B,A} \coloneqq\ \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_{k^{\mathrm{ID}}_{B,A}}\\
B:&\ \ k_{B,A} \coloneqq\ \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_{k^{\mathrm{ID}}_{B,A}}\\
B \to A:&\ \ id_{B,\mathcal{Z}}, id_{A,\mathcal{Z}}, \mathrm{n}_B,\ \mathsf{R}_B, \textsf{session\_ack}, \textsc{Mac}(\mathrm{n}_A \mid \mathsf{R}_A)_{k^{\mathrm{MAC}}_{B,A}}
\end{aligned}$$

  – Eventually, the adversary delivers the protocol message to M_B (message ① in Figure 5.15). M_B parses the message and checks if it corresponds to the expected format and if the parties on the message correspond to the party identifiers of the recorded intended communication partner *pid* and the party identifier of the user *pid′*, respectively. If the checks fail, M_B in **CheckAddress** mode returns **reject** to the message. Otherwise, it continues processing in **CheckAddress** mode.

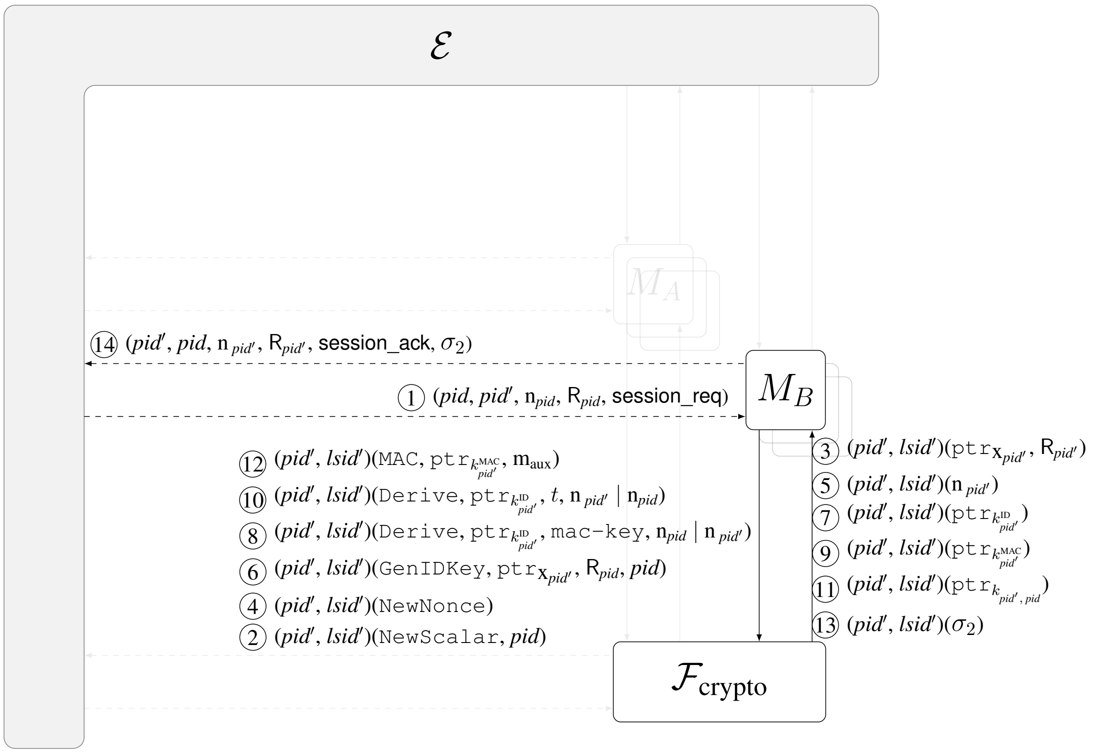

$\mathcal{E}$
M_A
⑭ $(pid', pid, \mathrm{n}_{pid'}, \mathsf{R}_{pid'}, \textsf{session\_ack}, \sigma_2)$
① $(pid, pid', \mathrm{n}_{pid}, \mathsf{R}_{pid}, \textsf{session\_req})$
M_B
⑫ $(pid', lsid')(\texttt{MAC}, \texttt{ptr}_{k^{\mathrm{MAC}}_{pid'}}, \mathrm{m}_{\mathrm{aux}})$
⑩ $(pid', lsid')(\texttt{Derive}, \texttt{ptr}_{k^{\mathrm{ID}}_{pid'}}, t, \mathrm{n}_{pid'} \mid \mathrm{n}_{pid})$
⑧ $(pid', lsid')(\texttt{Derive}, \texttt{ptr}_{k^{\mathrm{ID}}_{pid'}}, \texttt{mac-key}, \mathrm{n}_{pid} \mid \mathrm{n}_{pid'})$
⑥ $(pid', lsid')(\texttt{GenIDKey}, \texttt{ptr}_{\mathrm{x}_{pid'}}, \mathsf{R}_{pid}, pid)$
④ $(pid', lsid')(\texttt{NewNonce})$
② $(pid', lsid')(\texttt{NewScalar}, pid)$
③ $(pid', lsid')(\texttt{ptr}_{\mathrm{x}_{pid'}}, \mathsf{R}_{pid'})$
⑤ $(pid', lsid')(\mathrm{n}_{pid'})$
⑦ $(pid', lsid')(\texttt{ptr}_{k^{\mathrm{ID}}_{pid'}})$
⑨ $(pid', lsid')(\texttt{ptr}_{k^{\mathrm{MAC}}_{pid'}})$
⑪ $(pid', lsid')(\texttt{ptr}_{k_{pid', pid}})$
⑬ $(pid', lsid')(\sigma_2)$
$\mathcal{F}_{\mathrm{crypto}}$

Figure 5.15: Second message of *SessionKey* protocol.

  – M_B asks $\mathcal{F}_{\mathrm{crypto}}$ for a scalar $\mathrm{x}_{pid'}$ related to the identity of the partner *pid* through the command (`NewScalar`, *pid*), obtaining the scalar pointer $\texttt{ptr}_{\mathrm{x}_{\mathrm{pid}'}}$ and the public relation $\mathsf{R}_{pid'}$ (messages ② and ③ in Figure 5.15, respectively).

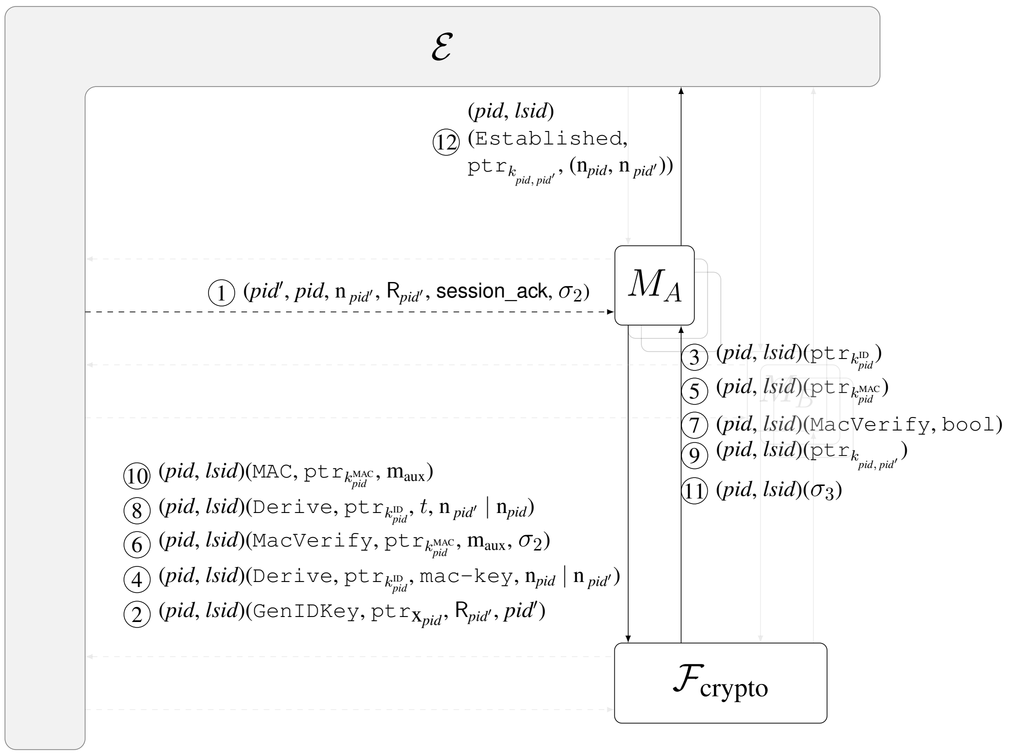
  - M_B uses the command (`NewNonce`) to ask $\mathcal{F}_{\text{crypto}}$ for a fresh nonce, and receives $\mathsf{n}_{pid'}$ (messages ④ and ⑤ in Figure 5.15, respectively).
  - M_B uses the command (`GenIDKey`, $\mathtt{ptr}_{\mathsf{x}_{pid'}}$, R_{pid}, *pid*) to ask $\mathcal{F}_{\text{crypto}}$ for a new `id-key` derived from the scalar $\mathsf{x}_{pid'}$ and the public relation R_{pid} received in the first message of the protocol from party *pid*. $\mathcal{F}_{\text{crypto}}$ returns the pointer $\mathtt{ptr}_{k^{\text{ID}}_{pid'}}$ (messages ⑥ and ⑦ in Figure 5.15, respectively).
  - M_B uses the command (`Derive`, $\mathtt{ptr}_{k^{\text{ID}}_{pid'}}$, `mac-key`, $\mathsf{n}_{pid} \mid \mathsf{n}_{pid'}$) to ask $\mathcal{F}_{\text{crypto}}$ to derive a new `mac-key` derived from the `id-key` referred by $\mathtt{ptr}_{k^{\text{ID}}_{pid'}}$ and nonce n_{pid} concatenated with $\mathsf{n}_{pid'}$. $\mathcal{F}_{\text{crypto}}$ returns the pointer $\mathtt{ptr}_{k^{\text{MAC}}_{pid'}}$ (messages ⑧ and ⑨ in Figure 5.15, respectively).
  - M_B uses the command (`Derive`, $\mathtt{ptr}_{k^{\text{ID}}_{pid'}}$, t, $\mathsf{n}_{pid'} \mid \mathsf{n}_{pid}$) to ask $\mathcal{F}_{\text{crypto}}$ to derive the session key of type t from the `id-key` referred by $\mathtt{ptr}_{k^{\text{ID}}_{pid'}}$ and nonce $\mathsf{n}_{pid'}$ concatenated with n_{pid}. $\mathcal{F}_{\text{crypto}}$ returns the pointer $\mathtt{ptr}_{k_{pid,pid'}}$ (messages ⑩ and ⑪ in Figure 5.15, respectively).
  - M_B uses the command (`MAC`, $\mathtt{ptr}_{k^{\text{MAC}}_{pid'}}$, m_{aux}) to ask $\mathcal{F}_{\text{crypto}}$ to calculate the MAC over the message m_{aux} using the key referred by pointer $\mathtt{ptr}_{k^{\text{MAC}}_{pid'}}$, with m_{aux} = (*pid′*, *pid*, $\mathsf{n}_{pid'}$, $\mathsf{R}_{pid'}$, session_ack, n_{pid}, R_{pid}), which is responded by $\mathcal{F}_{\text{crypto}}$ with MAC σ_2 (messages ⑫ and ⑬ in Figure 5.15, respectively).
  - M_B sets *state*$_B$ = `secondMessageSent` and concludes the **CheckAddress** mode returning **accept**.
  - In **Compute** mode, M_B sends the second message of the protocol (*pid′*, *pid*, $\mathsf{n}_{pid'}$, $\mathsf{R}_{pid'}$, session_ack, σ_2) on its output network tape (message ⑭ in Figure 5.15).

- M_A receives the second message of the protocol on its input network tape, it has *state*$_A$ = `started`, it prepares the third message of the protocol and outputs the session key pointer to the user.

  - Protocol specification:

$$\begin{aligned}
A:\ & k^{\text{ID}}_{A,B} := \text{MAP}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{R}_A, \mathsf{R}_B, \hat{e}(\mathsf{R}_B, S^{\text{I}}_{A,\mathcal{Z}})\hat{e}(G^{\text{I}}_{1,\mathcal{Z}}, G^{\text{I}}_{2,\mathcal{Z}})^{\mathsf{x}_A})\\
A:\ & k^{\text{MAC}}_{A,B} := \text{PRF}(\mathsf{n}_A \mid \mathsf{n}_B)_{k^{\text{ID}}_{A,B}}\\
A:\ & k_{A,B} := \text{PRF}(\mathsf{n}_B \mid \mathsf{n}_A)_{k^{\text{ID}}_{A,B}}
\end{aligned}$$

  - Eventually, the adversary delivers the protocol message to M_A (message ① in Figure 5.16). M_A parses the message and checks if it corresponds to the expected format and if the parties on the message correspond to the party identifiers of the recorded intended communication partner *pid′* and the party identifier of the user *pid*, respectively. If the checks fail, M_A in **CheckAddress** mode returns **reject** to the message. Otherwise, it continues processing in **CheckAddress** mode.



$\mathcal{E}$
$(pid, lsid)$ ⑫ $(\texttt{Established}, \texttt{ptr}_{k_{pid,pid'}}, (\mathrm{n}_{pid}, \mathrm{n}_{pid'}))$
M_A
① $(pid', pid, \mathrm{n}_{pid'}, \mathsf{R}_{pid'}, \mathsf{session\_ack}, \sigma_2)$
③ $(pid, lsid)(\texttt{ptr}_{k^{\mathrm{ID}}_{pid}})$
⑤ $(pid, lsid)(\texttt{ptr}_{k^{\mathrm{MAC}}_{pid}})$
⑦ $(pid, lsid)(\texttt{MacVerify}, \texttt{bool})$
⑨ $(pid, lsid)(\texttt{ptr}_{k_{pid,pid'}})$
⑪ $(pid, lsid)(\sigma_3)$
⑩ $(pid, lsid)(\texttt{MAC}, \texttt{ptr}_{k^{\mathrm{MAC}}_{pid}}, \mathrm{m_{aux}})$
⑧ $(pid, lsid)(\texttt{Derive}, \texttt{ptr}_{k^{\mathrm{ID}}_{pid}}, t, \mathrm{n}_{pid'} \mid \mathrm{n}_{pid})$
⑥ $(pid, lsid)(\texttt{MacVerify}, \texttt{ptr}_{k^{\mathrm{MAC}}_{pid}}, \mathrm{m_{aux}}, \sigma_2)$
④ $(pid, lsid)(\texttt{Derive}, \texttt{ptr}_{k^{\mathrm{ID}}_{pid}}, \texttt{mac-key}, \mathrm{n}_{pid} \mid \mathrm{n}_{pid'})$
② $(pid, lsid)(\texttt{GenIDKey}, \texttt{ptr}_{\mathrm{x}_{pid}}, \mathsf{R}_{pid'}, pid')$
$\mathcal{F}_{\mathrm{crypto}}$

Figure 5.16: M_A outputs the session key pointer to the user.

– M_A uses the command $(\texttt{GenIDKey}, \texttt{ptr}_{\mathrm{x}_{pid}}, \mathsf{R}_{pid'}, pid')$ to ask $\mathcal{F}_{\mathrm{crypto}}$ for a new `id-key` derived from the scalar x_{pid} and the public relation $\mathsf{R}_{pid'}$ received in the second message of the protocol. $\mathcal{F}_{\mathrm{crypto}}$ returns the pointer $\texttt{ptr}_{k^{\mathrm{ID}}_{pid}}$ (messages ② and ③ in Figure 5.16, respectively).

– M_A uses the command $(\texttt{Derive}, \texttt{ptr}_{k^{\mathrm{ID}}_{pid}}, \texttt{mac-key}, \mathrm{n}_{pid} \mid \mathrm{n}_{pid'})$ to ask $\mathcal{F}_{\mathrm{crypto}}$ to derive a new `mac-key` derived from the `id-key` referred by $\texttt{ptr}_{k^{\mathrm{ID}}_{pid}}$ and nonce n_{pid} concatenated with $\mathrm{n}_{pid'}$. $\mathcal{F}_{\mathrm{crypto}}$ returns the pointer $\texttt{ptr}_{k^{\mathrm{MAC}}_{pid}}$ (messages ④ and ⑤ in Figure 5.16, respectively).

– M_A asks $\mathcal{F}_{\mathrm{crypto}}$ to verify the MAC σ_2 using the key referred by the pointer $\texttt{ptr}_{k^{\mathrm{MAC}}_{pid}}$ through the command $(\texttt{MacVerify}, \texttt{ptr}_{k^{\mathrm{MAC}}_{pid}}, \mathrm{m_{aux}}, \sigma_2)$, where $\mathrm{m_{aux}} = (pid', pid, \mathrm{n}_{pid'}, \mathsf{R}_{pid'}, \mathsf{session\_ack}, \mathrm{n}_{pid}, \mathsf{R}_{pid})$ (message ⑥ in Figure 5.16). If the boolean returned by $\mathcal{F}_{\mathrm{crypto}}$ (message ⑦ in Figure 5.16) is `false`, i.e, the verification failed, M_A in **CheckAddress** mode returns **reject** to this message. Otherwise, i.e., if the MAC is successfully verified, we say M_A accepted the key exchanged for $(pid, lsid)$ and pid'.

– M_A uses the command $(\texttt{Derive}, \texttt{ptr}_{k^{\mathrm{ID}}_{pid}}, t, \mathrm{n}_{pid'} \mid \mathrm{n}_{pid})$ to ask $\mathcal{F}_{\mathrm{crypto}}$ to derive the session key of type t from the `id-key` referred by $\texttt{ptr}_{k^{\mathrm{ID}}_{pid}}$ and nonce $\mathrm{n}_{pid'}$ concatenated with n_{pid}. $\mathcal{F}_{\mathrm{crypto}}$ returns the pointer $\texttt{ptr}_{k_{pid,pid'}}$ (messages ⑧ and ⑨ in Figure 5.16, respectively).

– M_A uses the command $(\texttt{MAC}, \texttt{ptr}_{k^{\mathrm{MAC}}_{pid}}, \mathrm{m_{aux}})$ to ask $\mathcal{F}_{\mathrm{crypto}}$ to calculate the MAC over the message $\mathrm{m_{aux}}$ using the key referred by pointer $\texttt{ptr}_{k^{\mathrm{MAC}}_{pid}}$, with $\mathrm{m_{aux}} = (pid,$

pid′, session_ack, n$_{pid}$, R$_{pid}$, n$_{pid'}$, R$_{pid'}$), which is responded by $\mathcal{F}_{\text{crypto}}$ with MAC σ_3 (messages ⑩ and ⑪ in Figure 5.16, respectively).

- M_A sets *state*$_A$ = `finished` and concludes the **CheckAddress** mode returning **accept**.
- In **Compute** mode, M_A outputs the session key pointer $\texttt{ptr}_{k_{pid,\,pid'}}$ and the stamp (n$_{pid}$, n$_{pid'}$) of the session to the user through the message (`Established`, $\texttt{ptr}_{k_{\text{pid, pid}'}}$, (n$_{pid}$,n$_{pid'}$)) (message ⑫ in Figure 5.16). Here we have another necessity on the modeling. Ideally, M_A should output the result to the user on the environment at the same time that it sends the final protocol message on its output network tape. However, machines cannot write messages in two tapes simultaneously. This limitation is not restricted to IITM but to all models in universal composability. Usually, this restriction is overcome by modeling the machine to first output the message to the user and wait for an adversary command on the network for the subsequent protocol message. This is exactly the approach that we adopt.

- M_A receives the request from the adversary to conclude the protocol, it has *state*$_A$ = `finished` and the message ready to be sent, which makes the machine to send the third protocol message on its output network tape.
  - Protocol specification:

$$A \to B: \quad id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}},\ \text{session\_ack}, \textsc{Mac}(\text{n}_A \mid \text{R}_A \mid \text{n}_B \mid \text{R}_B)_{k^{\text{MAC}}_{A,B}}$$

  - The adversary requests M_A the final protocol message (message ① in Figure 5.17). M_A parses the message and checks if it corresponds to the expected format. If the check fails, M_A in **CheckAddress** mode returns **reject** to the message. Otherwise, it returns **accept** and proceed to **Compute** mode.

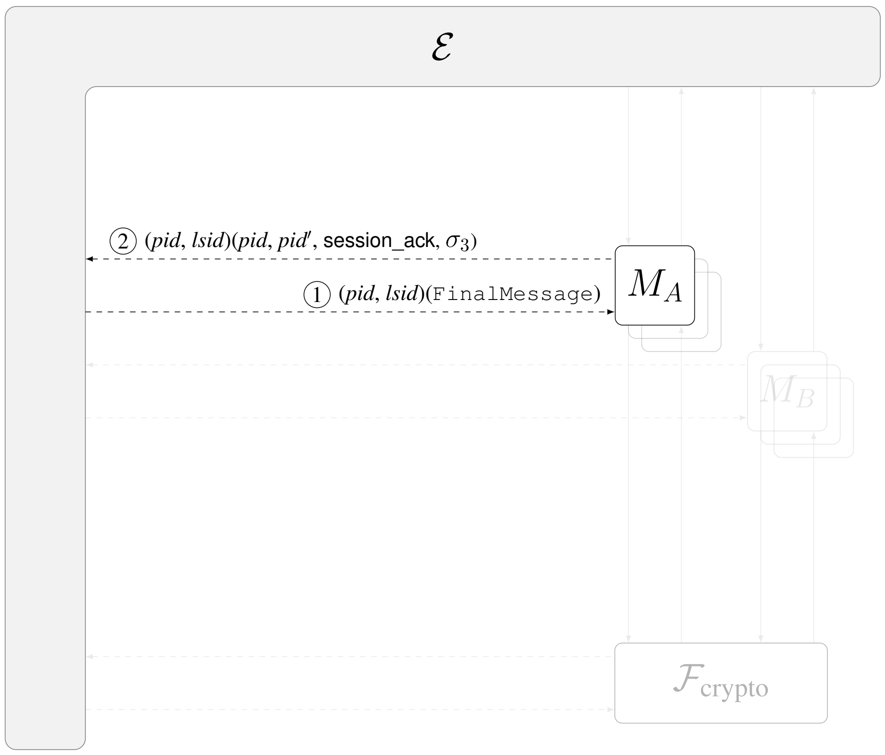

$\mathcal{E}$
② $(pid, lsid)(pid, pid', \text{session\_ack}, \sigma_3)$
① $(pid, lsid)$(FinalMessage)
M_A
M_B
$\mathcal{F}_{\text{crypto}}$

Figure 5.17: Third message of *SessionKey* protocol.

- In **Compute** mode, M_A sends the third protocol message $(pid, pid', \text{session\_ack}, \sigma_3)$ on its output network tape (message ② in Figure 5.17).

• M_B receives the third message of the protocol on its input network tape, it has *state*$_B$ = `secondMessageSent`, it outputs the session key pointer to the user.

- Eventually, the adversary delivers the protocol message to M_B (message ① in Figure 5.18). M_B parses the message and checks if it corresponds to the expected format and if the parties on the message correspond to the party identifiers of the recorded intended communication partner *pid* and the party identifier of the user *pid*′, respectively. If the checks fail, M_B in **CheckAddress** mode returns **reject** to the message. Otherwise, it continues processing in **CheckAddress** mode.

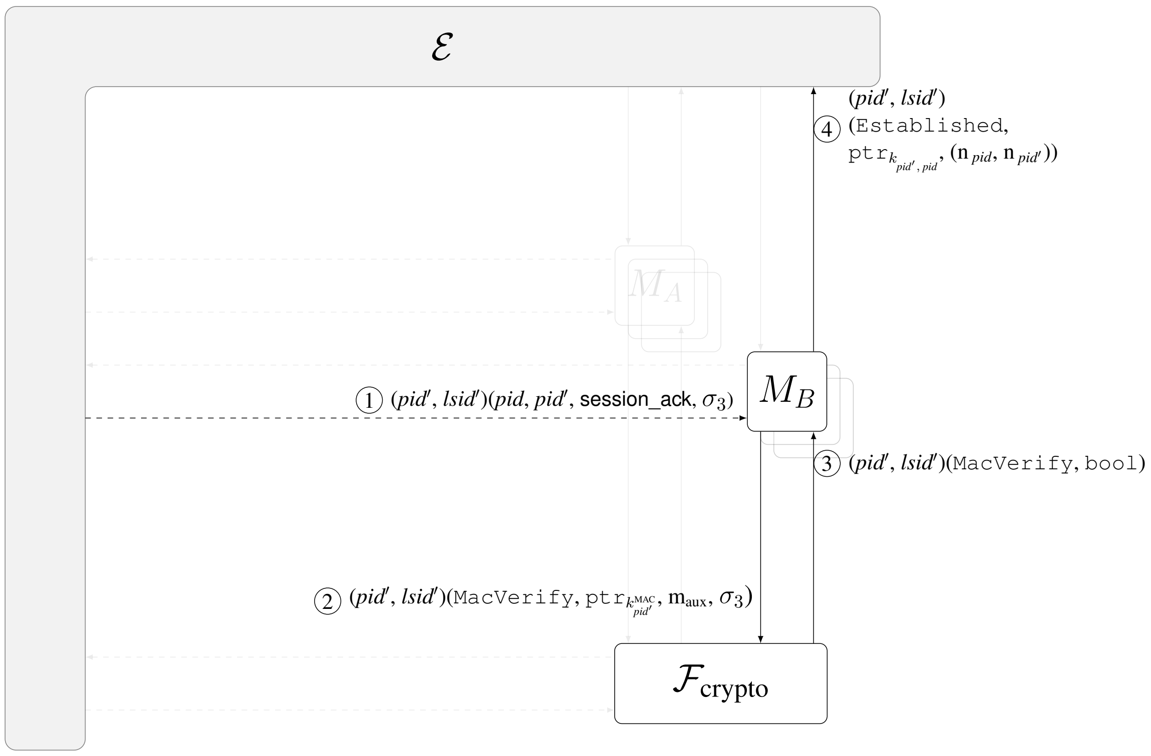

$\mathcal{E}$
④ $(pid', lsid')$ (Established, $\mathtt{ptr}_{k_{pid',pid}}$, $(\mathrm{n}_{pid}, \mathrm{n}_{pid'})$)
M_A
① $(pid', lsid')(pid, pid'$, session_ack, $\sigma_3)$
M_B
③ $(pid', lsid')$(MacVerify, bool)
② $(pid', lsid')$(MacVerify, $\mathtt{ptr}_{k^{\mathrm{MAC}}_{pid'}}$, $\mathrm{m}_{\mathrm{aux}}$, σ_3)
$\mathcal{F}_{\text{crypto}}$

Figure 5.18: M_B outputs the session key pointer to the user.

- M_B asks $\mathcal{F}_{\text{crypto}}$ to verify the MAC σ_3 using the key referred by pointer $\mathtt{ptr}_{k^{\mathrm{MAC}}_{pid'}}$ through the command (`MacVerify`, $\mathtt{ptr}_{k^{\mathrm{MAC}}_{pid'}}$, $\mathrm{m}_{\mathrm{aux}}$, σ_3), where $\mathrm{m}_{\mathrm{aux}} = (pid, pid'$, session_ack, n_{pid}, R_{pid}, $\mathrm{n}_{pid'}$, $\mathsf{R}_{pid'})$ (message ② in Figure 5.18). If the boolean returned by $\mathcal{F}_{\text{crypto}}$ (message ③ in Figure 5.18) is `false`, i.e, the verification failed, M_B in **CheckAddress** mode returns **reject** to this message. Otherwise, i.e., if the MAC is successfully verified, we say M_B accepted the key exchanged for $(pid', lsid')$ and pid.
- M_B sets $state_B$ = `finished` and concludes the **CheckAddress** mode returning **accept**.
- In **Compute** mode, M_B outputs the session key pointer $\mathtt{ptr}_{k_{pid,pid'}}$ and the stamp $(\mathrm{n}_{pid}, \mathrm{n}_{pid'})$ of the session to the user through the message (`Established`, $\mathtt{ptr}_{k_{pid,pid'}}$, $(\mathrm{n}_{pid}, \mathrm{n}_{pid'})$) (message ④ in Figure 5.18).

• Executing cryptographic operations. The machines have $state_A$ = `finished` and $state_B$ = `finished`, they receive requests, forward to $\mathcal{F}_{\text{crypto}}$ and return the results to the user.

  - Users $(pid, lsid, A)$ and $(pid, lsid, B)$ with state `finished` has access to the same key through pointers $\mathtt{ptr}_{k_{pid,pid'}}$ and $\mathtt{ptr}_{k_{pid',pid}}$, respectively. Under a request (messages ① and ①′ in Figure 5.19) to execute ideal cryptographic operations in $\mathcal{F}_{\text{crypto}}$, the machines forward the requests to $\mathcal{F}_{\text{crypto}}$ (message ② and ②′ in Figure 5.19), and send the results received from $\mathcal{F}_{\text{crypto}}$ to the user (messages ③ and ③′, and ④ and ④′ in Figure 5.19, respectively).

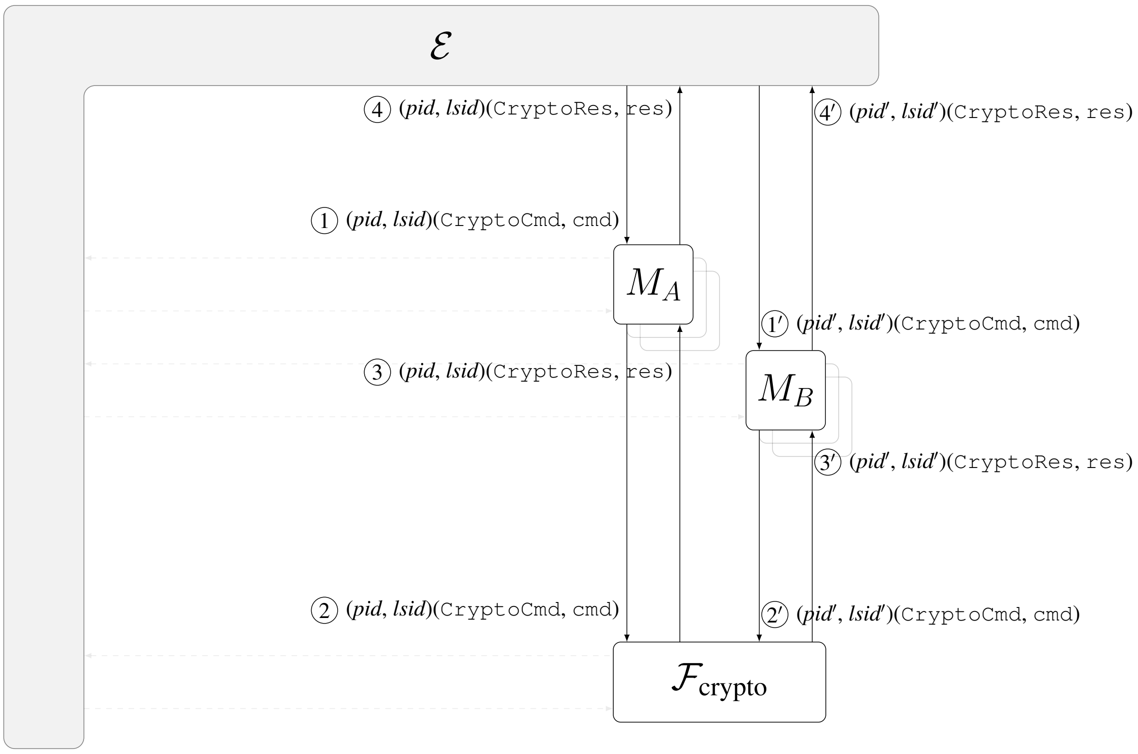

$\mathcal{E}$
④ (*pid*, *lsid*)(CryptoRes, res)
④′ (*pid*′, *lsid*′)(CryptoRes, res)
① (*pid*, *lsid*)(CryptoCmd, cmd)
M_A
①′ (*pid*′, *lsid*′)(CryptoCmd, cmd)
③ (*pid*, *lsid*)(CryptoRes, res)
M_B
③′ (*pid*′, *lsid*′)(CryptoRes, res)
② (*pid*, *lsid*)(CryptoCmd, cmd)
②′ (*pid*′, *lsid*′)(CryptoCmd, cmd)
$\mathcal{F}_{\text{crypto}}$

Figure 5.19: Cryptographic operations in an established session.

- Closing a session. The machines have *state*$_A$ = `finished` and *state*$_B$ = `finished`, they receive the request to close the session, inform the adversary about the session being closed, receive an acknowledgment from the adversary, revoke the access of the user to the cryptographic material generated during the session, and acknowledge the user.

  – After performing the desired cryptographic operations, an instance (*pid*, *lsid*,*r*) with state `finished` can be requested to close the session with the `CloseSession` command (messages ① and ①′ in Figure 5.20). The machine informs the adversary about the request and waits for an acknowledgment message (messages ② and ②′, and ③ and ③′ in Figure 5.20, respectively). The machine then sets its state to `sessionClosed`, revokes the access of (*pid*, *lsid*, *r*) to all the cryptographic material generated by itself for the key exchange and during the session, and acknowledges the user (messages ④ in Figure 5.20).

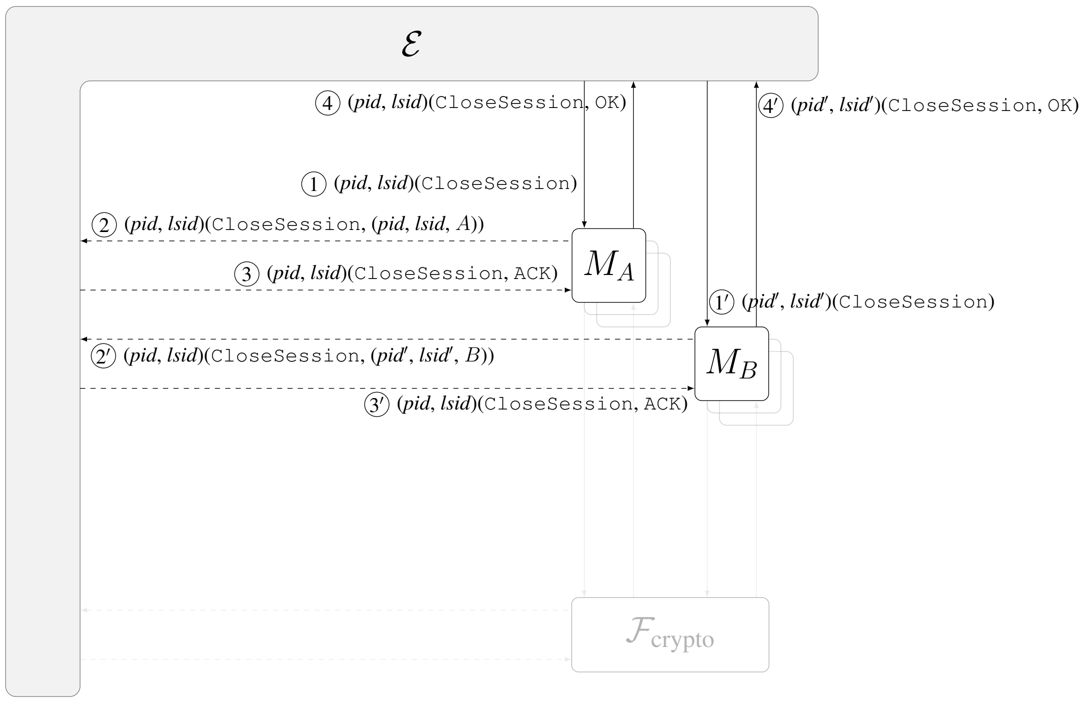

$\mathcal{E}$
④ (pid, lsid)(CloseSession, OK)
④′ (pid′, lsid′)(CloseSession, OK)
① (pid, lsid)(CloseSession)
② (pid, lsid)(CloseSession, (pid, lsid, A))
③ (pid, lsid)(CloseSession, ACK)
M_A
①′ (pid′, lsid′)(CloseSession)
②′ (pid, lsid)(CloseSession, (pid′, lsid′, B))
M_B
③′ (pid, lsid)(CloseSession, ACK)
$\mathcal{F}_{\text{crypto}}$

Figure 5.20: Closing a session.

- Corruption status request.
  - At any point, the user may request the corruption status of an instance of M_A or M_B (messages ① and ①′ in Figure 5.21). The machines return the value of its corruption status to the environment (messages ② and ②′ in Figure 5.21) considering the corruption model discussed in further details in the next section.

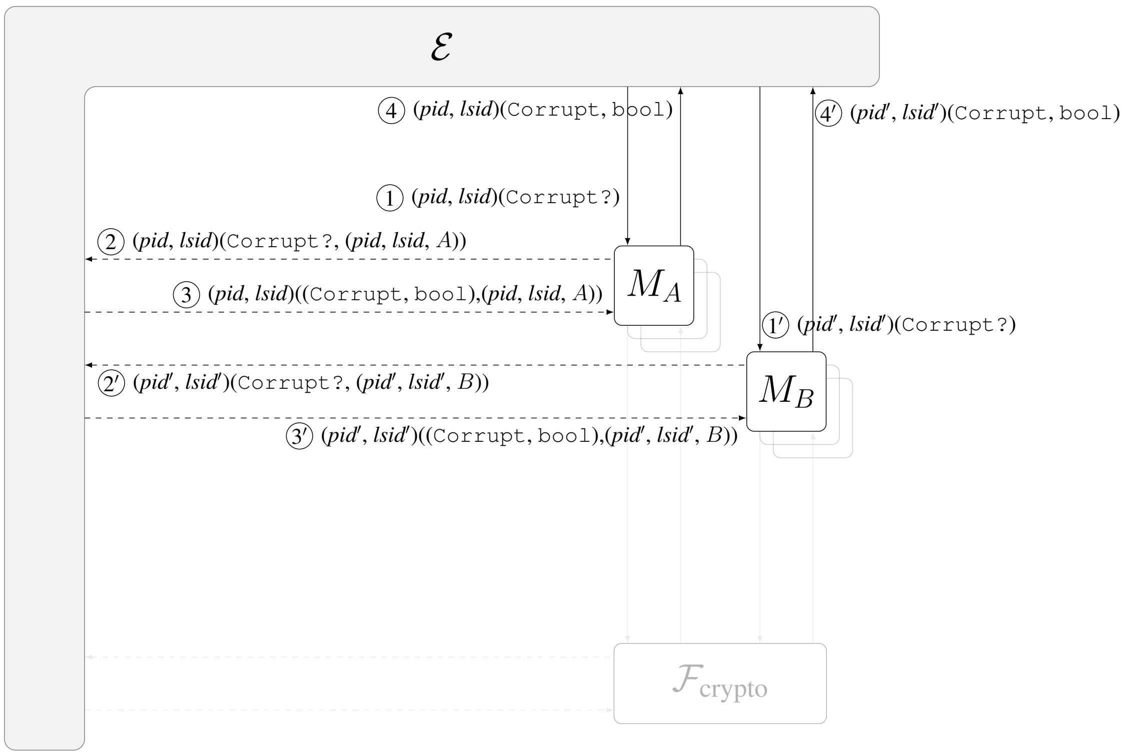

$\mathcal{E}$
④ (*pid*, *lsid*)(Corrupt, bool)
④′ (*pid*′, *lsid*′)(Corrupt, bool)
① (*pid*, *lsid*)(Corrupt?)
② (*pid*, *lsid*)(Corrupt?, (*pid*, *lsid*, *A*))
③ (*pid*, *lsid*)((Corrupt, bool),(*pid*, *lsid*, *A*))
M_A
①′ (*pid*′, *lsid*′)(Corrupt?)
②′ (*pid*′, *lsid*′)(Corrupt?, (*pid*′, *lsid*′, *B*))
M_B
③′ (*pid*′, *lsid*′)((Corrupt, bool),(*pid*′, *lsid*′, *B*))
$\mathcal{F}_{\text{crypto}}$

Figure 5.21: Corruption status request.

5.7.1.3 SessionKey Corruption Model

The corruption model of the *SessionKey* protocol is such that:

- The adversary can send a special message to corrupt an instance of the protocol (*pid*, *lsid*, *r*), *r* $\in \{A, B\}$, before a key exchange starts and after the session is closed, but not during the key exchange or the session.
- A corrupted instance (*pid*, *lsid*, *r*) forwards all messages on its i/o tapes originated from or destined to its corresponding user (*pid*, *lsid*) to the adversary on the network tape.
- The adversary can execute operations in $\mathcal{F}_{\text{crypto}}$ using the corrupted instance (*pid*, *lsid*, *r*) in the name of user (*pid*, *lsid*). The $\mathcal{F}_{\text{crypto}}$ operations available for the adversary are all operations exposed to $\mathcal{E}$ and those used in the *SessionKey* protocol implementation, all other are blocked. In this case, all secret the a corrupted instance creates in $\mathcal{F}_{\text{crypto}}$ must be known.
- The adversary can corrupt an instance (*pid*, *lsid*, *r*) only if the identity private key of party *pid* is corrupted. As the session is dependent on the identity-based cryptosystem

and the corrupted instances give full control to the adversary in $\mathcal{F}_{\text{crypto}}$, we assume this is reasonable modeling.

- The adversary gets access to secret components of a closed session after directly corrupting an instance because the information of a session is not erased when the session is closed, which models the lack of perfect forward secrecy of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$.
- An instance (*pid*, *lsid*, *r*) in a run of *SessionKey* with an instance of party *pid′* that has not output a session key pointer also considers itself corrupted if its identity private key or the identity private key of *pid′* is corrupted in $\mathcal{F}_{\text{crypto}}$, even without being explicitly corrupted by the adversary. This means that no security can be guaranteed if any of the secret keys from which the session key is derived is corrupted.
- Our modeling allows an instance to be corrupted only before a key exchange starts and after the session is closed. However, as previously put, the corruption status of an instance can also be determined by the identity private key's corruption status of the parties involved in the key exchange, and these keys might become corrupted during the key exchange process. We then define that an instance define its initial status based on the response received from the adversary at initialization and on the corruption status of the identity private key of the parties involved in the key exchange. Even not corrupted, the instance only outputs the session key pointer to its user if right before such an event an additional check on the corruption status of the identity private keys shows that their status have not change since the first check. Otherwise, the instance blocks all requests.
- The `id-key` key established by the protocol, consequently its session key, can also be derived from the secret scalars generated by the instances that are exchanging the key. However, the id-based key agreement commands are not exposed to the environment. This includes the scalar retrieval command that makes a scalar `known` outside $\mathcal{F}_{\text{crypto}}$. Therefore, the only way to have a `known` scalar in a run of the protocol is if the instance that owns the scalar is actually controlled by the adversary, i.e., corrupted. In this case, we model that the identity private key of the party has to be already corrupted, consequently the resulting `id-key` and session key will be known.

5.7.1.4 SessionKey Proof

The goal of this section is to prove the Theorem 5.4, which states that the *SessionKey* protocol (Protocol 4.1) realizes the functionality for mutually authenticated key exchange with key usability and in-session access to digital signatures $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}_{\text{crypto}}$.

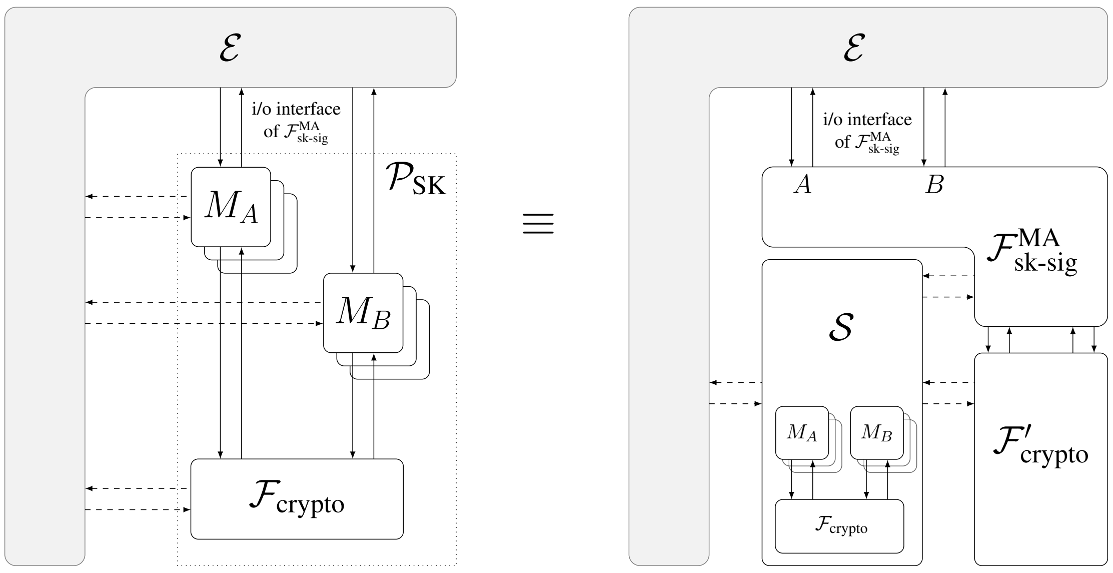
Theorem 5.4 *Let*

- $\mathcal{P}_{SK} = !M_A \mid !M_B \mid \mathcal{F}_{crypto}$ *be the modeling of the SessionKey protocol (Protocol 4.1) as described in Section 5.7.1.1;*
- M_A *and* M_B *to behave as described in Section 5.7.1.2;*
- $\mathcal{F}_{crypto}$ *and* $\mathcal{F}'_{crypto}$ *two different versions of the functionality for cryptographic primitives extended in this work (Section 5.4);*
- $\mathcal{F}^{MA}_{sk\text{-}sig}$ *functionality for mutually authenticated key exchange with key usability with key usability and in-session access to digital signatures, described in Section 5.6.1.*

Then it holds true that

$$\mathcal{P}_{SK} \mid \mathcal{F}_{crypto} \leq \mathcal{F}^{MA}_{sk\text{-}sig} \mid \mathcal{F}'_{crypto}.$$

In the case of the *SessionKey* protocol, we perform our analysis directly on the multi-session version of the protocol. We have to specify a simulator $\mathcal{S}$ such as, for every environment $\mathcal{E}$, it holds that $\mathcal{E} \mid !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$. In practice, $\mathcal{S}$ internally simulates the execution of $\mathcal{P}_{\text{SK}} = !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}}$ and, as the adversary of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, it interacts with $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, such that no $\mathcal{E}$ can distinguish between the real and ideal settings, as illustrated in Figure 5.22. More specifically, any instance (*pid*, *lsid*) in $\mathcal{E}$, initiator or responder, corrupted or uncorrupted, will get the same result when interacting with $!M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}}$ or $\mathcal{S} \mid \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$. We define $\mathcal{S}$ as follows:



$\mathcal{E}$
i/o interface of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
$\mathcal{P}_{\text{SK}}$
M_A
M_B
$\mathcal{F}_{\text{crypto}}$
$\equiv$
$\mathcal{E}$
i/o interface of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
A
B
$\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
$\mathcal{S}$
M_A
M_B
$\mathcal{F}_{\text{crypto}}$
$\mathcal{F}'_{\text{crypto}}$

Figure 5.22: Simulation of $!M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}}$ to prove $\mathcal{E} \mid !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$

- $\mathcal{S}$ maintains synchronized the corruption status of any instance (*pid*, *lsid*, *r*) in $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$ with corresponding instance (*pid*, *lsid*, *r*) in the simulation of $\mathcal{P}_{\mathrm{SK}} = !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}}$;

- Being the adversary of $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, $\mathcal{S}$ initializes $\mathcal{F}'_{\text{crypto}}$. As a result, $\mathcal{S}$ receives all of $\mathcal{F}'_{\text{crypto}}$ public parameters, which can be used in its internal simulation of $\mathcal{F}_{\text{crypto}}$. Together with the parameters response, $\mathcal{S}$ is asked to provide all cryptographic algorithms and asymmetric keys related to the digital signature scheme for $\mathcal{F}'_{\text{crypto}}$. $\mathcal{S}$ forwards the request to $\mathcal{E}$ and returns the algorithms and all the pair of keys to $\mathcal{F}'_{\text{crypto}}$;

- Whenever an instance (*pid*, *lsid*, *r*) initiates a key exchange with party *pid′* in $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$, as $\mathcal{S}$ is informed about it by $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ with the message ((`InitKE`, *pid′*), (*pid*, *lsid*, *r*)) (shown in Figure 5.6), $\mathcal{S}$ can initiate the key exchange in its internal simulation;

- In its simulation of $\mathcal{P}_{\mathrm{SK}}$, when an uncorrupted instance (*pid*, *lsid*, A) accepts the key exchange by verifying the MAC of the second protocol message, $\mathcal{S}$ sends the command (`GroupSession`, (*pid*, *lsid*, A), (*pid′*, *lsid′*, B)) to $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ (shown in Figure 5.7), instructing the functionality to create a session between the simulated users (*pid*, *lsid*, A) and (*pid′*, *lsid′*, B), that, respectively verified the MAC on the second protocol message and generated such a MAC;

- Upon the `GroupSession` command, $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ uses the command `GetPSK` to request $\mathcal{F}'_{\text{crypto}}$ a pointer to a new key of type t (parameter of $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$). $\mathcal{F}'_{\text{crypto}}$, in turn, asks $\mathcal{S}$ for the value of the key. Then, $\mathcal{S}$ responds with the value of the session key calculated in its internal simulation of $\mathcal{P}_{\mathrm{SK}}$. Similarly, when $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ uses the command `NewNonce` to request $\mathcal{F}'_{\text{crypto}}$ new fresh nonces for the session, $\mathcal{F}'_{\text{crypto}}$ also asks $\mathcal{S}$ to provide the value of the nonces, which also are provided by $\mathcal{S}$ based on the internal simulation of $\mathcal{P}_{\mathrm{SK}}$. After $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ receives the pointers to the session key and the nonces from $\mathcal{F}'_{\text{crypto}}$, it acknowledges $\mathcal{S}$ about the success of the `GroupSession` command (shown in Figure 5.7). Hence, $\mathcal{S}$ instructs $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ to output the session key pointer to user (*pid*, *lsid*, A) using the command (`FinishKE`, (*pid*, *lsid*, A)) (shown in Figure 5.8);

- In its internal simulation of $\mathcal{P}_{\mathrm{SK}}$, when an uncorrupted instance (*pid′*, *lsid′*, B) accepts the key exchange by verifying the MAC on the third protocol message and outputs the session key pointer, $\mathcal{S}$ instructs $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ to output the session key pointer to user (*pid′*, *lsid′*, B) using the command (`FinishKE`, (*pid′*, *lsid′*, B)) (shown in Figure 5.8);

- Once the key exchange is completed, the users can request $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ the execution of the available cryptographic operations in $\mathcal{F}'_{\text{crypto}}$. In the set of the symmetric cryptographic operations, none of them requires that $\mathcal{F}'_{\text{crypto}}$ requests new values to the adversary (simulator), for instance, for new keys, nonces, scalars, or exponents, i.e., there is no dependency on $\mathcal{F}_{\text{crypto}}$. However, there exists a dependency regarding the digital signatures scheme, because the adversary can corrupt the signing keys at any time. Nevertheless, $\mathcal{S}$ is able to capture such kind of corruption in $\mathcal{F}_{\text{crypto}}$ and replicate to $\mathcal{F}'_{\text{crypto}}$.

- When $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ informs $\mathcal{S}$ that an instance (*pid*, *lsid*, *r*) asked to close its session, $\mathcal{S}$ updates the internal simulation removing the access to keys and responds `OK` to $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ (shown in Figure 5.10);

We have to show that $\mathcal{S}$ guarantees that, from the perspective of any $\mathcal{E}$, there are no differences between the real or the ideal settings. We start by arguing that, as the adversary of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, $\mathcal{S}$ can keep $\mathcal{F}'_{\text{crypto}}$ consistent with $\mathcal{F}_{\text{crypto}}$. First, we have that all new symmetric keys in $\mathcal{F}'_{\text{crypto}}$ are created due to `GetPSK` commands from $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to $\mathcal{F}'_{\text{crypto}}$, which causes $\mathcal{F}'_{\text{crypto}}$ to ask the adversary $\mathcal{S}$ for the values of the keys. $\mathcal{S}$, in turn, can provide the keys based on its internal simulation of $\mathcal{F}_{\text{crypto}}$, with the same value and status. As the $\mathcal{F}'_{\text{crypto}}$ is not exposed to users, neither are the commands `Store` and `Retrieve`, there is no user in $\mathcal{E}$ capable of inserting new symmetric keys or making keys known to the environment. We have a similar scenario for nonces. All nonces used in $\mathcal{F}'_{\text{crypto}}$ are created due to `NewNonce` commands from $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to $\mathcal{F}'_{\text{crypto}}$, which causes $\mathcal{F}'_{\text{crypto}}$ to ask the adversary $\mathcal{S}$ for the values of nonces. $\mathcal{S}$, in turn, can provide the same values from its internal simulation of $\mathcal{F}_{\text{crypto}}$. Last, $\mathcal{S}$ receives the signature scheme and related key pairs from $\mathcal{E}$ and forwards it to $\mathcal{F}'_{\text{crypto}}$ at initialization, therefore the cryptosystem and all the keys exist in both $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$. According to the corruption model of $\mathcal{F}_{\text{crypto}}$ (Section 5.3.4), signing keys can be corrupted at any point in time, therefore, as soon as the simulator receives a request from the adversary to corrupt a signing key, as the adversary of $\mathcal{F}'_{\text{crypto}}$, it can corrupt such key in $\mathcal{F}'_{\text{crypto}}$, allowing $\mathcal{S}$ to keep also the digital signature setting synchronized in both $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$. Hence, $\mathcal{S}$ maintains $\mathcal{F}'_{\text{crypto}}$ always consistent with $\mathcal{F}_{\text{crypto}}$, which means no request such as for corruption request to $\mathcal{F}'_{\text{crypto}}$ will cause different results in the real or ideal settings.

Now we argue that $\mathcal{S}$ correctly handles the following use cases: (1) uncorrupted initiator instances and (2) uncorrupted responder instances during the key exchange; (3) uncorrupted instances after the key exchange; and (4) corrupted instances. During the analysis, some key points from the corruption model of *SessionKey* (Section 5.7.1.3) are constantly mentioned, we reinforce them bellow:

- The adversary can explicitly corrupt an instance (*pid*, *lsid*, *r*) before key exchange starts and after the session is closed, but not during the key exchange or the session.
- An instance (*pid*, *lsid*, *r*) of party *pid* can be explicitly corrupted only if its identity private key is corrupted.
- An instance (*pid*, *lsid*, *r*) in a run of *SessionKey* with an instance of party *pid′* that has not output a session key pointer also considers itself corrupted if its identity private key or the identity private key of its partner *pid′* is corrupted. Even with an uncorrupted status, right

before outputting a session key pointer, such an instance checks the corruption status of its identity private key and the identity private key of its partner pid' again. If any of the keys has become corrupted, the instance blocks.

Now we analyze each one of the four use cases.

(1) Uncorrupted initiator instance during the key exchange.

Let $(pid_A, lsid_A, A)$ be an uncorrupted instance of an initiator that wants to exchange a key with party pid'. This instance can be simulated by $\mathcal{S}$ using $\mathcal{F}_{\text{crypto}}$ until it outputs a session key after verifying the MAC on the second message of the protocol without dependency on any information or operation in $\mathcal{F}'_{\text{crypto}}$. We have to show that $\mathcal{S}$ pairs with $(pid_A, lsid_A, A)$ a single instance of a responder which party identifier is pid'.

As $(pid_A, lsid_A, A)$ is uncorrupted and outputs the session key pointer, the identity private key of parties pid_A and pid' must be uncorrupted, otherwise this instance would block according to the corruption model. Therefore, any instance of pid' cannot have been explicitly corrupted by the adversary.

By outputting the key pointer, the instance $(pid_A, lsid_A, A)$ has successfully verified and accepted the MAC over the message $\mathrm{m} = (pid', pid_A, \mathrm{n}, \mathsf{R}, \mathsf{session\_ack}, \mathrm{n}_{pid_A}, \mathsf{R}_{pid_A})$, where $\mathsf{R} = \mathrm{x} \cdot P^{\mathrm{I}}_{pid_A}$ and $\mathsf{R}_{pid_A} = \mathrm{x}_{pid_A} \cdot P^{\mathrm{I}}_{pid'}$. The key $(pid_A, lsid_A, A)$ uses to verify the MAC is necessarily derived from the `id-key` that comes from: (i) the public relation R that carries a scalar x tied to the identity public key of pid_A; (ii) a scalar x_{pid_A} created by instance $(pid_A, lsid_A, A)$ that is tied to the identity public key of pid' on the public relation R_{pid_A}; and (iii) the identity private key of pid_A. Analogously, the key used to create such MAC is necessarily derived from the `id-key` that comes from: (i) the public relation R_{pid_A} that carries the identity public key of pid'; (ii) the scalar x that is tied to identity public key of pid_A on the public relation R; and (iii) the identity private key of pid'. No other party than pid' could generate such a key, hence the MAC on the second protocol message must have been created by an instance of pid', which we identify as $(pid', lsid', r)$.

We just showed pid_A is the session partner of $(pid', lsid', r)$. As the identity private key of pid_A is uncorrupted, the instance $(pid', lsid', r)$ cannot consider itself corrupted. Therefore, $(pid', lsid', r)$ is uncorrupted.

Now we have to show that the instance $(pid', lsid', r)$ is indeed a responder. Suppose $(pid', lsid', r)$ is an initiator. As such, it must have already accepted a second protocol message by verifying a MAC of a message $\mathrm{m}' = (pid_A, pid', \mathrm{n}_{pid_A}, \mathsf{R}_{pid_A}, \mathsf{session\_ack}, \mathrm{n}, \mathsf{R})$, using a key that is necessarily derived from the `id-key` that comes from: (i) the public relation R_{pid_A} that carries the scalar x_{pid_A} tied to the identity public key of pid'; (ii) the scalar x created by $(pid', lsid', r)$ that is tied to the identity public key of pid_A on the public relation R; and (iii) the identity private key of pid'. Analogously, the key used to create such MAC is necessarily derived from the `id-key` that comes from: (i) the public relation R that carries the identity public key of pid_A; (ii) the scalar x_{pid_A}; and (iii) the identity private key of pid_A. No other instance than (pid_A, $lsid_A$, A) could generate such a key, because only this instance has access to the secret scalar x_{pid_A}. As (pid_A, $lsid_A$, A) does not create any MAC before accepting the second message of the protocol, (pid', $lsid'$, r) is indeed a responder, i.e, $r = B$.

Now we have to show that (pid', $lsid'$, B) is assigned to a session in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ with no other instance than (pid_A, $lsid_A$, A).

Based on the design of $\mathcal{S}$, a `GroupSession` command to create the session between the uncorrupted responder instance (pid', $lsid'$, B) and an uncorrupted initiator instance is sent only when the initiator successfully verifies and accepts the MAC over m = (pid', pid_A, n, R, session_ack, n_{pid_A}, R_{pid_A}) on the second protocol message. As the scalars are unique and this MAC is only verified and accepted by the initiator instance that created the scalar x_{pid_A}, which is the instance (pid_A, $lsid_A$, A), the only possible `GroupSession` command to create the global session for (pid', $lsid'$, B) has necessarily the instance (pid_A, $lsid_A$, A) as initiator.

Last, as the scalars x and x_{pid_A} are ideally created and the identity private keys of parties pid_A and pid' are uncorrupted, the resulting `id-key` in the simulation is unknown in $\mathcal{F}_{\text{crypto}}$, which results in an unknown session key. Hence, as the sets of keys are synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$, the simulator can indeed provide the exact same key from the simulation to $\mathcal{F}'_{\text{crypto}}$. Besides, this key can only be accessed by instances (pid_A, $lsid_A$, A) and (pid', $lsid'$, B) which is the expected behavior in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Similarly, as the nonces n and n_{pid_A} are ideally created in $\mathcal{F}_{\text{crypto}}$, the simulator can indeed provide the nonces from the internal simulation to $\mathcal{F}'_{\text{crypto}}$ and keep also the set of nonces synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$.

Therefore, real and ideal settings behave identically from the perspective of an uncorrupted initiator (pid_A, $lsid_A$).

(2) Uncorrupted responder instance during the key exchange.

Let (pid_B, $lsid_B$, B) be an uncorrupted instance of a responder that wants to exchange a key with party pid'. This instance can be simulated by $\mathcal{S}$ using $\mathcal{F}_{\text{crypto}}$ until it outputs a session key pointer after verifying the MAC on the third message of the protocol without dependency on any information or operation in $\mathcal{F}'_{\text{crypto}}$. We have to show that $\mathcal{S}$ has paired (pid_B, $lsid_B$, B) with a single instance of an initiator which party identifier is pid'.

If (pid_B, $lsid_B$, B) is uncorrupted and outputs the session key pointer, the identity private key of parties pid_B and pid' must be uncorrupted, uncorrupted, otherwise this instance would block according to the corruption model. Therefore, any instance of pid' cannot have been explicitly corrupted by the adversary.

By outputting the key pointer, the instance (pid_B, $lsid_B$, B) has successfully verified and accepted the MAC over the message m = (pid', pid_B, session_ack, n, R, n_{pid_B}, R_{pid_B}), where R = x · $P^{I}_{pid_B}$ and R_{pid_B} = x_{pid_B} · $P^{I}_{pid'}$. The key (pid_B, $lsid_B$, B) uses to verify the MAC is necessarily derived from the `id-key` that comes from: (i) the public relation R that carries a scalar x tied to the identity public key of pid_B; (ii) a scalar x_{pid_B} created by instance (pid_B, $lsid_B$, B) that is tied to the identity public key of pid' on the public relation R_{pid_B}; and (iii) the identity private key of pid_B. Analogously, the MAC key used to create such MAC is necessarily derived from the `id-key` that comes from: (i) the public relation R_{pid_B} that carries the identity public key of pid'; (ii) the scalar x that is tied to identity public key of pid_B on the public relation R; and (iii) the identity private key of pid'. No other party than pid' could generate such a key, hence the MAC on the third protocol message must have been created by an instance of pid', which we identify as (pid', $lsid'$, r).

We just showed pid_B is the session partner of (pid', $lsid'$, r). As the identity private key of pid_B is uncorrupted, the instance (pid', $lsid'$, r) cannot consider itself corrupted. Therefore, (pid', $lsid'$, r) is uncorrupted.

We have to show that the instance (pid', $lsid'$, r) is indeed an initiator. Suppose (pid', $lsid'$, r) is a responder. As such, the instance (pid', $lsid'$, r) creates the MAC over a message m′ = (pid', pid_B, n, R session_ack, n_{pid_B}, R_{pid_B}) sent on a second protocol message. The key (pid', $lsid'$, r) uses to create such MAC is necessarily derived from the `id-key` that comes from: (i) the public relation R_{pid_B} that carries the scalar x_{pid_B} tied to the identity public key of pid'; (ii) the scalar x that is tied to identity public key of pid_B on the public relation R; and (iii) the identity private key of pid'. However, the public relation R_{pid_B} is created by instance (pid_B, $lsid_B$, B) upon scalar x_{pid_B} generation, which it performed by (pid_B, $lsid_B$, B) after it receives its first protocol message. As a responder, the second protocol message is send by (pid', $lsid'$, r) only after it receives its fist protocol message. Therefore, (pid', $lsid'$, r) cannot have been received R_{pid_B} in its first protocol message before (pid_B, $lsid_B$, B) has received its first protocol message. Analogously, to successfully verify such a MAC, the key (pid_B, $lsid_B$, B) needs is necessarily derived from the `id-key` that comes from the scalar x_{pid_B} that is generated after pid_B receives its first message. However, the scalar x_{pid_B} is already MACed on the message. Then, as scalars are implemented ideally, (pid_B, $lsid_B$, B) is not able to create its scalar x_{pid_B}. Last, we have that $x_{pid_B} \neq$ x due to collision resistance of scalars in $\mathcal{F}_{\text{crypto}}$, which guarantees ($pid_B$, $lsid_B$, B) $\neq$ (pid', $lsid'$, r). Therefore, (pid', $lsid'$, r) is indeed an initiator, i.e, $r = A$.

Now we have to show that (pid', $lsid'$, B) is in a global session with (pid', $lsid'$, A) in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$.

Having generated a MAC on the third message, the initiator (pid', $lsid'$, A) accepted the MAC over m = (pid_B, pid', n_{pid_B}, R_{pid_B}, session_ack, n, R) on the second message of the protocol and, consequently, output a session key pointer. The only instances capable of generating such a MAC are (pid_B, $lsid_B$, B) and (pid', $lsid'$, A), the instances that have access to scalars x_{pid_B} and x, respectively. However, being the initiator, (*pid′*, *lsid′*, A) could not have created any MAC up to that point in the protocol. Therefore, as (*pid′*, *lsid′*, A) and (*pid*$_B$, *lsid*$_B$, B) are uncorrupted instances, when (*pid′*, *lsid′*, A) accepts the key exchange, $\mathcal{S}$ creates in $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ a group session for the responder instance (*pid*$_B$, *lsid*$_B$, B) with initiator (*pid′*, *lsid′*, A).

Last, as the scalars x and x_{pid_B} are ideally created and the identity private keys of parties *pid*$_A$ and *pid′* are uncorrupted, the resulting `id-key` in the simulation is unknown in $\mathcal{F}_{\text{crypto}}$, which results in an unknown session key. Hence, as the sets of keys are synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$, the simulator can indeed provide the exact same key from the simulation to $\mathcal{F}'_{\text{crypto}}$. Besides, this key can only be accessed by instances (*pid′*, *lsid′*, A) and (*pid*$_B$, *lsid*$_B$, B) which is the expected behavior in $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$. Similarly, as the nonces n and n_{pid_B} are ideally created in $\mathcal{F}_{\text{crypto}}$, the simulator can indeed provide the nonces from the internal simulation to $\mathcal{F}'_{\text{crypto}}$ and keep also the set of nonces synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$.

Therefore, real and ideal settings behave identically from the perspective of an uncorrupted responder (*pid*$_B$, *lsid*$_B$).

(3) Uncorrupted instances after the key exchange.

Let (*pid*$_A$, *lsid*$_A$, A) and (*pid*$_B$, *lsid*$_B$, B) be uncorrupted instances of an initiator and a responder, respectively, with an established unknown session key.

As shown in the previous cases, $\mathcal{S}$ can perfectly simulate the instances to the point when the instances get the pointers to the unknown session key in $\mathcal{F}_{\text{crypto}}$. Besides, no instance other than the ones in the session has access to the session key pointer. By the design of $\mathcal{S}$, the session key is provided by $\mathcal{S}$ to $\mathcal{F}'_{\text{crypto}}$ and $\mathcal{F}^{\mathrm{MA}}_{\text{sk-sig}}$ is instructed to output session key pointers to (*pid*$_A$, *lsid*$_A$, A) and (*pid*$_B$, *lsid*$_B$, B). Therefore, after the key establishment phase, during the usage of the session key and other cryptographic operations, the real and ideal settings behave identically from the perspective of uncorrupted instances.

(4) Corrupted instances.

Let (*pid*, *lsid*, *r*), $r \in \{A, B\}$ be a corrupted instance. The simulator controls the i/o interface of such an instance. This instance is corrupted due to one of the two reasons:

(i) it was explicitly corrupted by the adversary or (ii) the identity private key of one of the parties involved in the key exchange is corrupted. In case (i), the adversary controls the instance before the key exchange starts or after it finishes. In any of these stages, the adversary gets access only to known keys in $\mathcal{F}_{\text{crypto}}$, which, in fact, do not exist in $\mathcal{F}'_{\text{crypto}}$. Therefore, $\mathcal{S}$ can simulate the same behavior of $\mathcal{F}_{\text{crypto}}$ for this kind of corrupted instance. In case (ii), the simulator also has to simulate unknown keys in $\mathcal{F}_{\text{crypto}}$. However, as we shown during our analysis, an uncorrupted instance actually will never established a session key with a corrupted user, therefore, the unknown keys related to this case will never be inserted in $\mathcal{F}'_{\text{crypto}}$. Hence, $\mathcal{S}$ can also simulate this kind of corrupted instance.

With the analysis of the four possible cases, we conclude the proof of Theorem 5.4.

Now we present Corollary 5.1, that comes from Theorem 5.1, the proof that our extension $\mathcal{P}_{\text{crypto}}$ realizes ou extension $\mathcal{F}_{\text{crypto}}$ (Section 5.5) and Theorem 5.4 we just proved. We use this result to replace $\mathcal{F}_{\text{crypto}}$ by our extension $\mathcal{P}_{\text{crypto}}$, obtaining a real implementation of the *SessionKey* protocol (Protocol 4.1), a universally composable mutual authenticated key exchange protocol.

Corollary 5.1 *Let $\mathcal{P}_{SK} = !M_A \mid !M_B$ be the IITM modeling of the* SessionKey *protocol (Protocol 4.1) such that M_A and M_B behave as described in Section 5.7.1.2; $\mathcal{F}_{crypto}$ and $\mathcal{P}_{crypto}$ defined as in Section 5.5; $\mathcal{F}^{MA}_{sk\text{-}sig}$ be the functionality for mutually authenticated key exchange with key usability and in-session access to digital signatures, as described in Section 5.6.1; $\mathcal{F}^*$ be a machine such that in any point during the protocol systems communication blocks if the environment cause the commitment problem or does not respect the used-order requirement. Then it holds true that:*

$$\mathcal{F}^* \mid \mathcal{P}_{SK} \mid \mathcal{P}_{crypto} \leq \mathcal{F}^* \mid \mathcal{F}^{MA}_{sk\text{-}sig} \mid \mathcal{F}_{crypto}.$$

5.7.2 SessionKeyDerivation Protocol

Analogously to the security analysis of the *SessionKey* protocol, in our analysis of *SessionKeyDerivation* protocol (Section 4.3) we also need to proof that the protocol is as secure as the functionality for mutually authenticated key exchange with key usability and in-session access to digital signatures $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, described in Section 5.6.1. We present the protocol modeling in Section 5.7.2.1, describe its execution in Section 5.7.2.2, and develop the proof in Section 5.7.2.4.

5.7.2.1 SessionKeyDerivation IITM Modeling

The *SessionKeyDerivation* protocol (Protocol 4.3), referred as $\mathcal{P}_{\text{SKD}}$, analogously to $\mathcal{P}_{\text{SK}}$, has a straightforward two-party multi-session protocol modeling $\mathcal{P}_{\text{SKD}} = !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}}$, which consists of two machines M_A and M_B, one for each protocol role, A as the initiator device and B as the responder one. M_A and M_B have i/o tapes to interface with the environment $\mathcal{E}$, the network interface for adversary $\mathcal{A}$ communication, and i/o tapes to ideally use $\mathcal{F}_{\text{crypto}}$ as the cryptographic primitives provider, as shown in Figure 5.23.

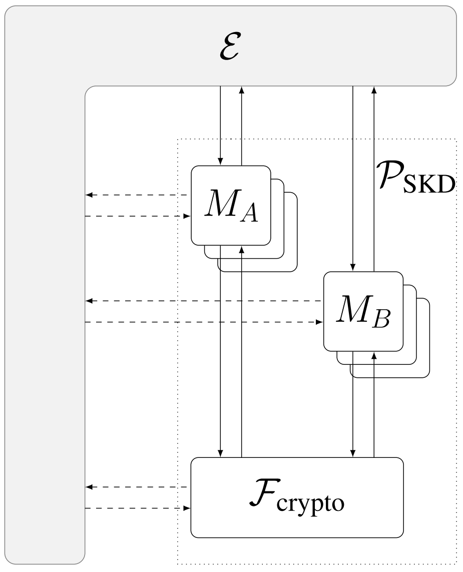


Figure 5.23: *SessionKeyDerivation* ($\mathcal{P}_{\text{SKD}}$) protocol modeling.

Users in $\mathcal{P}_{\text{SKD}}$ are fully identified by the tuple (*pid*, *lsid*, *r*), where *pid* and *lpid* are party and local session identifiers, totally managed by the environment, and *r* $\in \{A, B\}$ is the protocol role. Therefore, in a run of $\mathcal{P}_{\text{SKD}}$, there is a single instance (*pid*, *lsid*, *r*) of a machine M_r per user (*pid*, *lsid*) in the environment. All messages in the i/o tapes are prefixed with (*pid*, *lsid*), making the modeling totally compatible with $\mathcal{F}_{\text{crypto}}$ identification scheme. $\mathcal{P}_{\text{SK}}$ is parameterized with a symmetric key type $t \in \{$`authenc-key`, `mac-key`$\}$ to determine the type of the key established using the protocol.

Each machine maintains an internal variable to control its current state in execution of the protocol. This variable allows a machine to parse the messages received on its tapes according to what has been executed up to that particular point. As the machines play different roles, they have different stages in an execution, M_A has a control *state*$_A \in \{\perp$, `started`, `finished`, `sessionClosed`$\}$, initially set to $\perp$, M_B, in turn, has a control *state*$_B \in \{\perp$, `started`, `secondMessageSent`, `finished`, `sessionClosed`$\}$, initially set to $\perp$. The machines also keep their corruption status (`false` or `true`) initially set to `false`, and a register with the party identifier of the intended communication partner, initially empty.

5.7.2.2 SessionKeyDerivation Execution

In terms of code, the machines M_A and M_B implement the specification of the corresponding role in the *SessionKeyDerivation* protocol (Protocol 4.3) with the same modeling necessities we have to model the *SessionKey* protocol. The first necessity is to provide to the environment the same interface as the functionality *SessionKeyDerivation* wants to realize. As we want to prove $\mathcal{P}_{\text{SK}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}_{\text{crypto}}$, M_A and M_B must provide to $\mathcal{E}$ the same interface as $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Therefore, a user (*pid*, *lsid*) in $\mathcal{E}$ must be able to:

- send the command (`InitKE`, *pid′*) to start a key exchange with desired party *pid′*;
- Perform the desired cryptographic operations in $\mathcal{F}_{\text{crypto}}$ after the session key pointer is returned. Analogously to $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, the available symmetric operations are: `Enc`, `Dec`, `Mac`, `MacVerify`, `Corrupted?`, and `Equal?`, and the digital signature scheme operations are: `GetPubKeySig`, `Sign`, `SigVerify`, and `CorruptSig?`.
- Send the command (`CloseSession`) after performing the desired cryptographic operations in the session;
- Send the command (`Corrupt?`) to get its corruption status.

In the following, we describe, including graphically, the code of machines M_A and M_B and their behaviors in a normal run of $\mathcal{P}_{\text{SKD}}$. The machines are implemented such that all messages are sent in **Compute** mode. Therefore, if a message is sent, a machine has executed in **CheckAddress** mode and returned **accept**. This means that all the protocol messages are built and verified in the **CheckAddress** mode piece.

- Key exchange initialization. The machines have internal state set to $\perp$.
  - Machines M_A and M_B have *state* = $\perp$, and receive (one at a time) messages (*pid*, *lsid*)(`InitKE`, *pid′*) and (*pid′*, *lsid′*)(`InitKE`, *pid*) (messages ① and $\textcircled{1'}$ in Figure 5.13, respectively) from the environment, which start key exchanges for users (*pid*, *lsid*) and (*pid′*, *lsid′*), that have as the intended communication partners the parties *pid′* and *pid*, respectively. The machines **accept** the messages, i.e., they are now user instances (*pid*, *lsid*, *A*) and (*pid′*, *lsid′*, *B*) in the protocol, and proceed to **Compute** mode to output a message.
  - M_A and M_B inform the key exchange request to the adversary by sending, respectively, ((*pid*, *lsid*), (`InitKE`, *pid′*)) and ((*pid′*, *lsid′*), (`InitKE`, *pid*)) on their output network tapes (messages ② and $\textcircled{2'}$ in Figure 5.24, respectively).
  - The adversary acknowledges the machines about the initialization, which, in turn, can start the protocol execution. This specific acknowledge message on the beginning can be used by the adversary to corrupt the users by setting the corruption flag `bool` as `true`, as shown in messages ③ and $\textcircled{3'}$ in Figure 5.13. If such request is made, the corruption status of the machine is set to `true`, and the machine stops without producing output, which makes the environment (the master machine of the

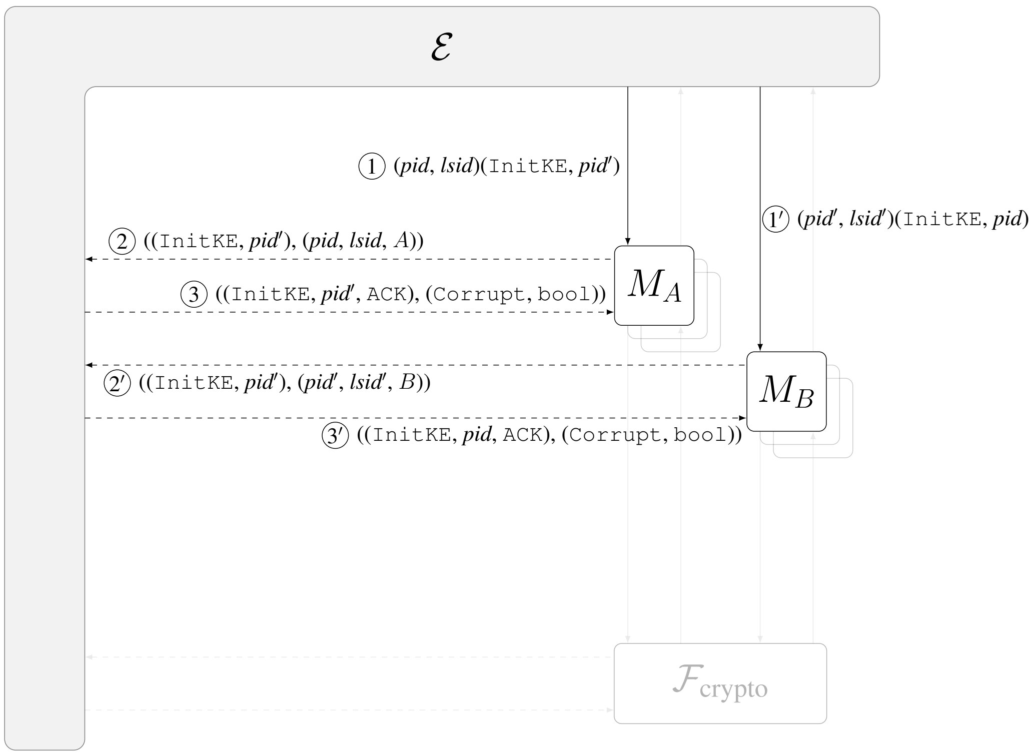

$\mathcal{E}$
① (*pid*, *lsid*)(InitKE, *pid*′)
② ((InitKE, *pid*′), (*pid*, *lsid*, A))
③ ((InitKE, *pid*′, ACK), (Corrupt, bool))
M_A
①′ (*pid*′, *lsid*′)(InitKE, *pid*)
②′ ((InitKE, *pid*′), (*pid*′, *lsid*′, B))
③′ ((InitKE, *pid*, ACK), (Corrupt, bool))
M_B
$\mathcal{F}_{\text{crypto}}$

Figure 5.24: Key exchange initialization in $\mathcal{P}_{\text{SKD}}$.

system) to be activated. Otherwise, M_A sets *state*$_A$ = started, M_B sets *state*$_B$ = started, they record the intended communication partner (*pid*′ in M_A and *pid* in M_B), and continue with protocol execution. M_A prepares to send the first protocol message, while M_B concludes the **CheckAddress** mode returning **accept**, proceeds to **Compute** mode, which is concluded without outputting a message, activating the environment.

- M_A continues in **CheckAddress** mode from the initialization acknowledgment message and sends the first message of the protocol on its output network tape.
  - Protocol specification:

$$A \rightarrow B: \quad id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \mathrm{n}_A, \text{session\_key\_derivation\_req}$$

  - M_A uses the command (NewNonce) to ask $\mathcal{F}_{\text{crypto}}$ for a fresh nonce, and receives n_{pid} (messages ① and ② in Figure 5.25). M_A concludes the **CheckAddress** mode returning **accept**.
  - In **Compute** mode, M_A sends (*pid*, *pid*′, n_{pid}, session_key_derivation_req), the first protocol message, on its output network tape (message ③ in Figure 5.25).

- M_B receives the first message of the protocol on its input network tape, it has *state*$_B$ = started, it prepares the second message of the protocol, and sends it on its output network tape.

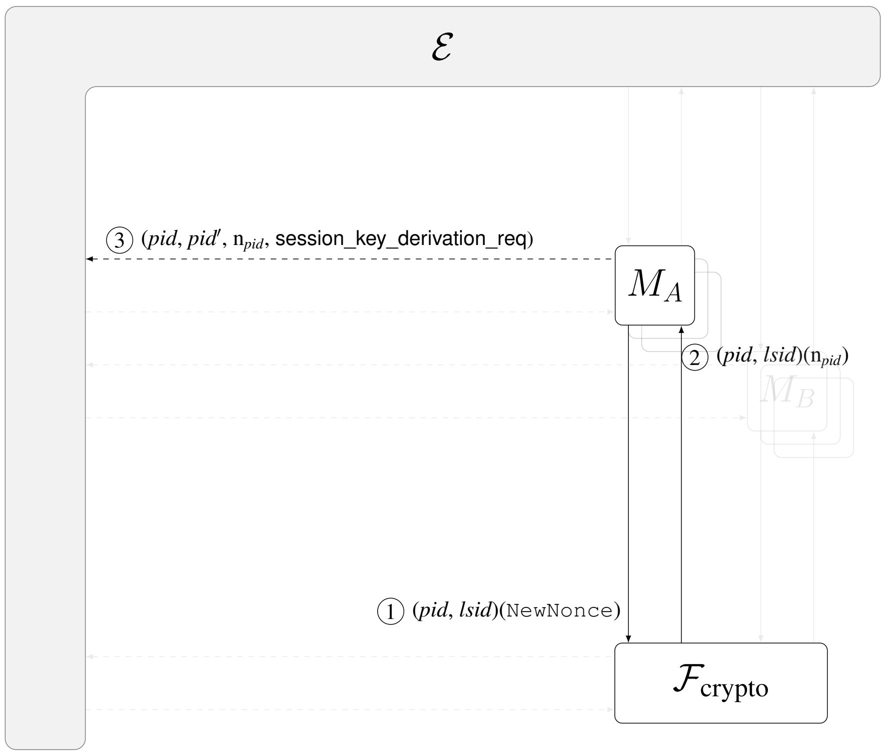

$\mathcal{E}$
③ (*pid*, *pid'*, n$_{pid}$, session_key_derivation_req)
M_A
② (*pid*, *lsid*)(n$_{pid}$)
M_B
① (*pid*, *lsid*)(NewNonce)
$\mathcal{F}_{\text{crypto}}$

Figure 5.25: First message of *SessionKeyDerivation* protocol.

– Protocol specification:

$$\begin{aligned} B:&\quad k^{\text{MAC}}_{B,A} := \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_k \\ B:&\quad k_{B,A} := \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_k \\ B \to A:&\quad id_{B,\mathcal{H}}, id_{A,\mathcal{H}}, \mathrm{n}_B, \text{session\_key\_derivation\_ack}, \textsc{Mac}(\mathrm{n}_A)_{k^{\text{MAC}}_{B,A}} \end{aligned}$$

– Eventually, the adversary delivers the protocol message to M_B (message ① in Figure 5.26). M_B parses the message and checks if it corresponds to the expected format and if the parties on the message correspond to the party identifiers of the recorded intended communication partner *pid* and the party identifier of the user *pid'*, respectively. If the checks fail, M_B in **CheckAddress** mode returns **reject** to the message. Otherwise, it continues processing in **CheckAddress** mode.

– M_B uses the command (`NewNonce`) to ask $\mathcal{F}_{\text{crypto}}$ for a fresh nonce, and receives n$_{pid'}$ (messages ② and ③ in Figure 5.26, respectively).

– M_B then requests $\mathcal{F}_{\text{crypto}}$ a pointer to a `pre-key` established between parties *pid* and *pid'* using command (`GetPSK`, `pre-key`, *pid* | *pid'*) (message ④ in Figure 5.26), which models the setup assumptions in AoT. This key is either the key that has been established using the *Long-TermKeyAgreement* protocol (Protocol 4.2) for deriving the key for access control during a functioning stage or it is the ephemeral key that has been shared between the device being deployed and the home server in the beginning of the deployment stage to be used in the first execution of the *KeyIs-*

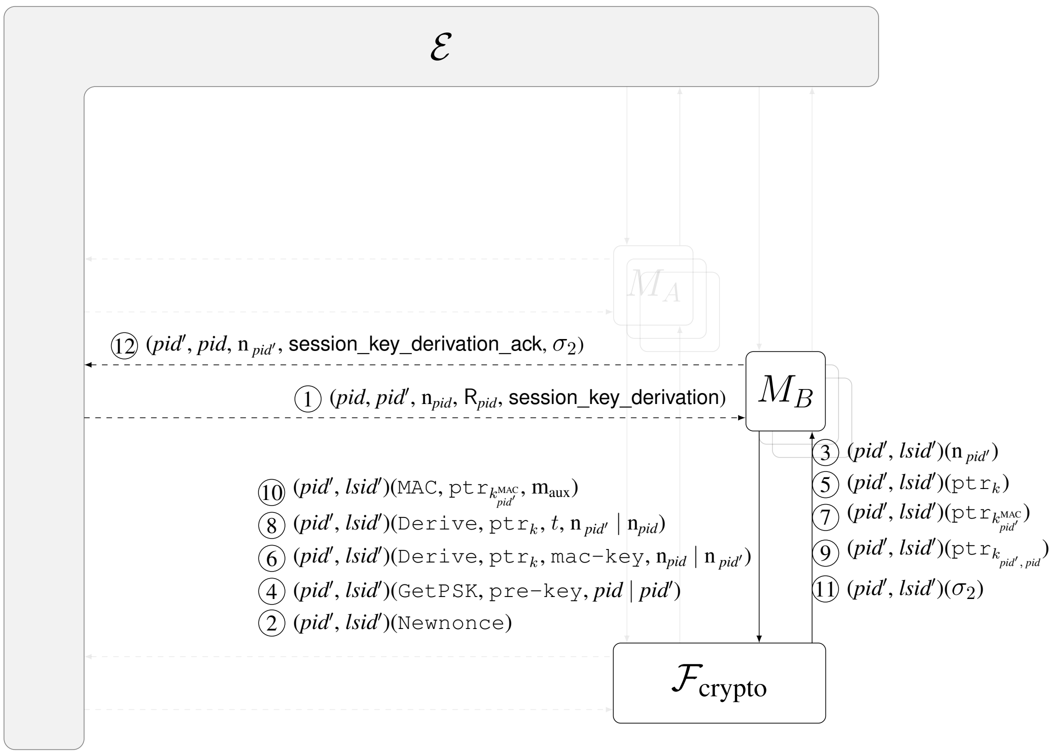

$\mathcal{E}$
M_A
⑫ (pid', pid, n$_{pid'}$, session_key_derivation_ack, σ_2)
① (pid, pid', n$_{pid}$, R$_{pid}$, session_key_derivation)
M_B
③ (pid', $lsid'$)(n$_{pid'}$)
⑤ (pid', $lsid'$)(ptr$_k$)
⑦ (pid', $lsid'$)(ptr$_{k^{\text{MAC}}_{pid'}}$)
⑨ (pid', $lsid'$)(ptr$_{k_{pid',pid}}$)
⑪ (pid', $lsid'$)(σ_2)
⑩ (pid', $lsid'$)(MAC, ptr$_{k^{\text{MAC}}_{pid'}}$, m$_{\text{aux}}$)
⑧ (pid', $lsid'$)(Derive, ptr$_k$, t, n$_{pid'}$ | n$_{pid}$)
⑥ (pid', $lsid'$)(Derive, ptr$_k$, mac-key, n$_{pid}$ | n$_{pid'}$)
④ (pid', $lsid'$)(GetPSK, pre-key, pid | pid')
② (pid', $lsid'$)(Newnonce)
$\mathcal{F}_{\text{crypto}}$

Figure 5.26: Second message of *SessionKeyDerivation* protocol.

sue protocol (Protocol 4.4). $\mathcal{F}_{\text{crypto}}$ returns the pointer ptr$_k$ to the pre-shared key (message ⑤ in Figure 5.26).

- M_B uses the command (Derive, ptr$_k$, mac-key, n$_{pid}$ | n$_{pid'}$) to ask $\mathcal{F}_{\text{crypto}}$ to derive a new mac-key derived from the pre-key referred by ptr$_k$ and nonce n$_{pid}$ concatenated with n$_{pid'}$. $\mathcal{F}_{\text{crypto}}$ returns the pointer ptr$_{k^{\text{MAC}}_{pid'}}$ (messages ⑥ and ⑦ in Figure 5.26, respectively).
- M_B uses the command (Derive, ptr$_k$, t, n$_{pid'}$ | n$_{pid}$) to ask $\mathcal{F}_{\text{crypto}}$ to derive the session key of type t from the pre-key referred by ptr$_k$ and nonce n$_{pid'}$ concatenated with n$_{pid}$. $\mathcal{F}_{\text{crypto}}$ returns the pointer ptr$_{k_{pid,pid'}}$ (messages ⑧ and ⑨ in Figure 5.26, respectively).
- M_B uses the command (MAC, ptr$_{k^{\text{MAC}}_{pid'}}$, m$_{\text{aux}}$) to ask $\mathcal{F}_{\text{crypto}}$ to calculate the MAC over the message m$_{\text{aux}}$ using the key referred by pointer ptr$_{k^{\text{MAC}}_{pid'}}$, with m$_{\text{aux}}$ = (pid', pid, n$_{pid'}$, session_key_derivation_ack, n$_{pid}$), which is responded by $\mathcal{F}_{\text{crypto}}$ with MAC σ_2 (messages ⑩ and ⑪ in Figure 5.26, respectively).
- M_B sets *state*$_B$ = secondMessageSent and concludes the **CheckAddress** mode returning **accept**.
- In **Compute** mode, M_B sends the second message of the protocol (pid', pid, n$_{pid'}$, session_key_derivation_ack, σ_2) on its output network tape (message ⑫ in Figure 5.26).

• M_A receives the second message of the protocol on its input network tape, it has *state*$_A$

= `started`, it prepares the third message of the protocol and outputs the session key pointer to the user.

– Protocol specification:

$$B: \quad k_{A,B}^{\text{MAC}} := \textsc{Prf}(\mathrm{n}_A \mid \mathrm{n}_B)_k$$
$$B: \quad k_{A,B} := \textsc{Prf}(\mathrm{n}_B \mid \mathrm{n}_A)_k$$

– Eventually, the adversary delivers the protocol message to M_A (message ① in Figure 5.27). M_A parses the message and checks if it corresponds to the expected format and if the parties on the message correspond to the party identifiers of the recorded intended communication partner *pid′* and the party identifier of the user *pid*, respectively. If the checks fail, M_A in **CheckAddress** mode returns **reject** to the message. Otherwise, it continues processing in **CheckAddress** mode.

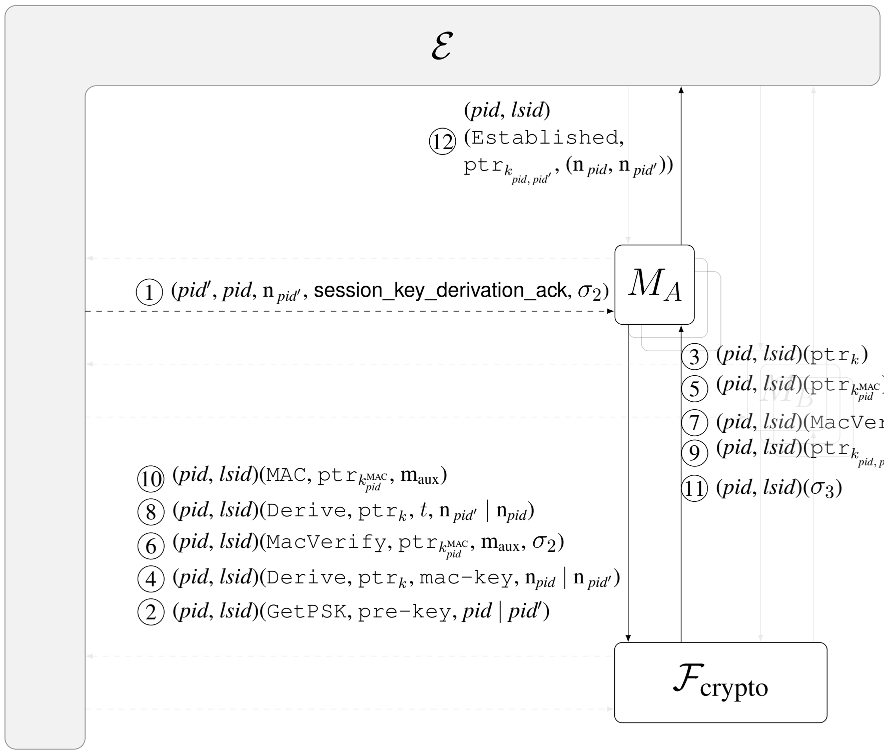

$\mathcal{E}$
⑫ $(pid, lsid)$ (Established, $\texttt{ptr}_{k_{pid,pid'}}$, $(\mathrm{n}_{pid}, \mathrm{n}_{pid'})$)
M_A
① $(pid', pid, \mathrm{n}_{pid'}$, session_key_derivation_ack, $\sigma_2)$
③ $(pid, lsid)(\texttt{ptr}_k)$
⑤ $(pid, lsid)(\texttt{ptr}_{k^{\text{MAC}}_{pid}})$
⑦ $(pid, lsid)(\texttt{MacVerify}, \texttt{bool})$
⑨ $(pid, lsid)(\texttt{ptr}_{k_{pid,pid'}})$
⑪ $(pid, lsid)(\sigma_3)$
⑩ $(pid, lsid)(\texttt{MAC}, \texttt{ptr}_{k^{\text{MAC}}_{pid}}, \mathrm{m}_{\text{aux}})$
⑧ $(pid, lsid)(\texttt{Derive}, \texttt{ptr}_k, t, \mathrm{n}_{pid'} \mid \mathrm{n}_{pid})$
⑥ $(pid, lsid)(\texttt{MacVerify}, \texttt{ptr}_{k^{\text{MAC}}_{pid}}, \mathrm{m}_{\text{aux}}, \sigma_2)$
④ $(pid, lsid)(\texttt{Derive}, \texttt{ptr}_k, \texttt{mac-key}, \mathrm{n}_{pid} \mid \mathrm{n}_{pid'})$
② $(pid, lsid)(\texttt{GetPSK}, \texttt{pre-key}, pid \mid pid')$
$\mathcal{F}_{\text{crypto}}$

Figure 5.27: M_A outputs the session key pointer to the user.

– M_A requests $\mathcal{F}_{\text{crypto}}$ a pointer to the `pre-key` established between parties *pid* and *pid'* using the command (`GetPSK`, `pre-key`, *pid* | *pid'*) (message ② in Figure 5.27). $\mathcal{F}_{\text{crypto}}$ returns $\texttt{ptr}_k$ (message ③ in Figure 5.27).

– M_A uses the command (`Derive`, $\texttt{ptr}_k$, `mac-key`, $\mathrm{n}_{pid} \mid \mathrm{n}_{pid'}$) to ask $\mathcal{F}_{\text{crypto}}$ to derive a new `mac-key` derived from the `pre-key` referred by $\texttt{ptr}_k$ and nonce n_{pid} concatenated with $\mathrm{n}_{pid'}$. $\mathcal{F}_{\text{crypto}}$ returns the pointer $\texttt{ptr}_{k^{\text{MAC}}_{pid}}$ (messages ④ and ⑤ in Figure 5.27, respectively).

- M_A asks $\mathcal{F}_{\text{crypto}}$ to verify the MAC σ_2 using the key referred by the pointer $\texttt{ptr}_{k^{\text{MAC}}_{pid}}$ through the command ($\texttt{MacVerify}$, $\texttt{ptr}_{k^{\text{MAC}}_{pid}}$, m_{aux}, σ_2), where m_{aux} = (*pid′*, *pid*, $\text{n}_{pid'}$, session_key_derivation_ack, n_{pid}) (message ⑥ in Figure 5.27). If the boolean returned by $\mathcal{F}_{\text{crypto}}$ (message ⑦ in Figure 5.27) is $\texttt{false}$, i.e, the verification failed, M_A in **CheckAddress** mode returns **reject** to this message. Otherwise, i.e., if the MAC is successfully verified, we say M_A accepted the key exchanged for (*pid*, *lsid*) and *pid′*.
- M_A uses the command ($\texttt{Derive}$, $\texttt{ptr}_k$, t, $\text{n}_{pid'} \mid \text{n}_{pid}$) to ask $\mathcal{F}_{\text{crypto}}$ to derive the session key of type t from the $\texttt{pre-key}$ referred by $\texttt{ptr}_k$ and nonce $\text{n}_{pid'}$ concatenated with n_{pid}. $\mathcal{F}_{\text{crypto}}$ returns the pointer $\texttt{ptr}_{k_{pid,pid'}}$ (messages ⑧ and ⑨ in Figure 5.27, respectively).
- M_A uses the command ($\texttt{MAC}$, $\texttt{ptr}_{k^{\text{MAC}}_{pid}}$, m_{aux}) to ask $\mathcal{F}_{\text{crypto}}$ to calculate the MAC over the message m_{aux} using the key referred by pointer $\texttt{ptr}_{k^{\text{MAC}}_{pid}}$, with m_{aux} = (*pid*, *pid′*, session_key_derivation_ack, n_{pid}, $\text{n}_{pid'}$), which is responded by $\mathcal{F}_{\text{crypto}}$ with MAC σ_3 (messages ⑩ and ⑪ in Figure 5.27, respectively).
- M_A sets $state_A$ = $\texttt{finished}$ and concludes the **CheckAddress** mode returning **accept**.
- In **Compute** mode, M_A outputs the session key pointer $\texttt{ptr}_{k_{pid,pid'}}$ and the stamp (n_{pid}, $\text{n}_{pid'}$) of the session to the user through the message to the user through the message ($\texttt{Established}$, $\texttt{ptr}_{k_{\text{pid},\text{pid}'}}$, ($\text{n}_{pid}$, $\text{n}_{pid'}$)) (message ⑫ in Figure 5.27). Here we use the same approach we used in *SessionKey* execution modeling (Section 5.7.1.2), i.e., M_A outputs results to the user and waits for a final protocol message request from the adversary on the network tape.

• M_A receives the request from the adversary to conclude the protocol, it has $state_A$ = $\texttt{finished}$ and the message ready to be sent, which makes the machine to send the third protocol message on its output network tape.

- Protocol specification:

$$A \to B: \quad id_{A,\mathcal{H}}, id_{B,\mathcal{H}}, \text{ session\_key\_derivation\_ack}, \text{MAC}(\text{n}_A \mid \text{n}_B)_{k^{\text{MAC}}_{A,B}}$$

- The adversary requests M_A the final protocol message (message ① in Figure 5.28). M_A parses the message and checks if it corresponds to the expected format. If the check fails, M_A in **CheckAddress** mode returns **reject** to the message. Otherwise, it returns **accept** and proceed to **Compute** mode.
- In **Compute** mode, M_A sends (*pid*, *pid′*, session_key_derivation_ack, σ_3), the third protocol message, on its output network tape (message ② in Figure 5.28).

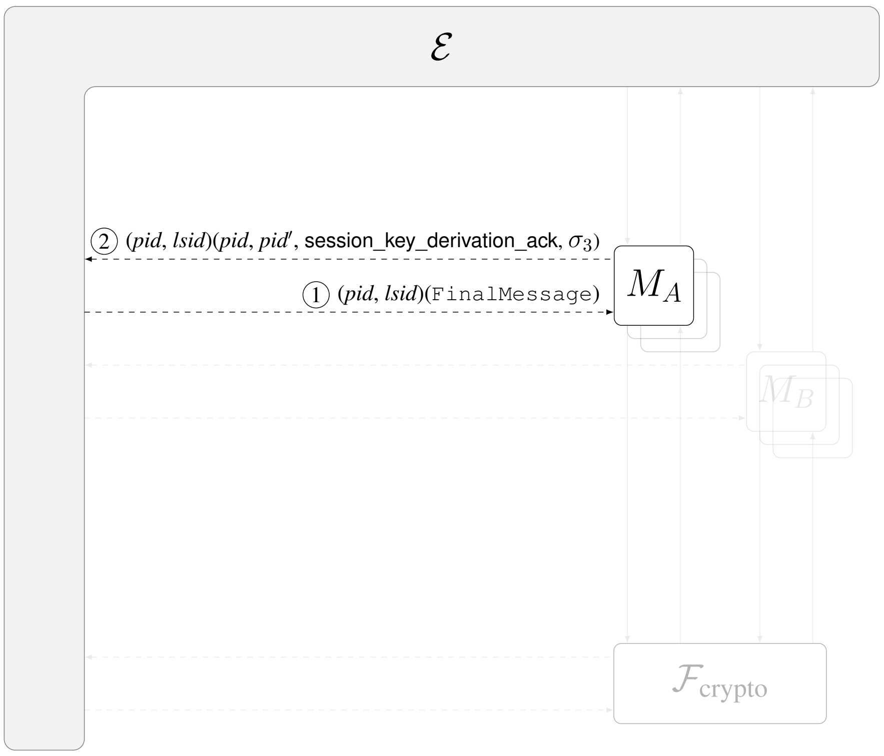
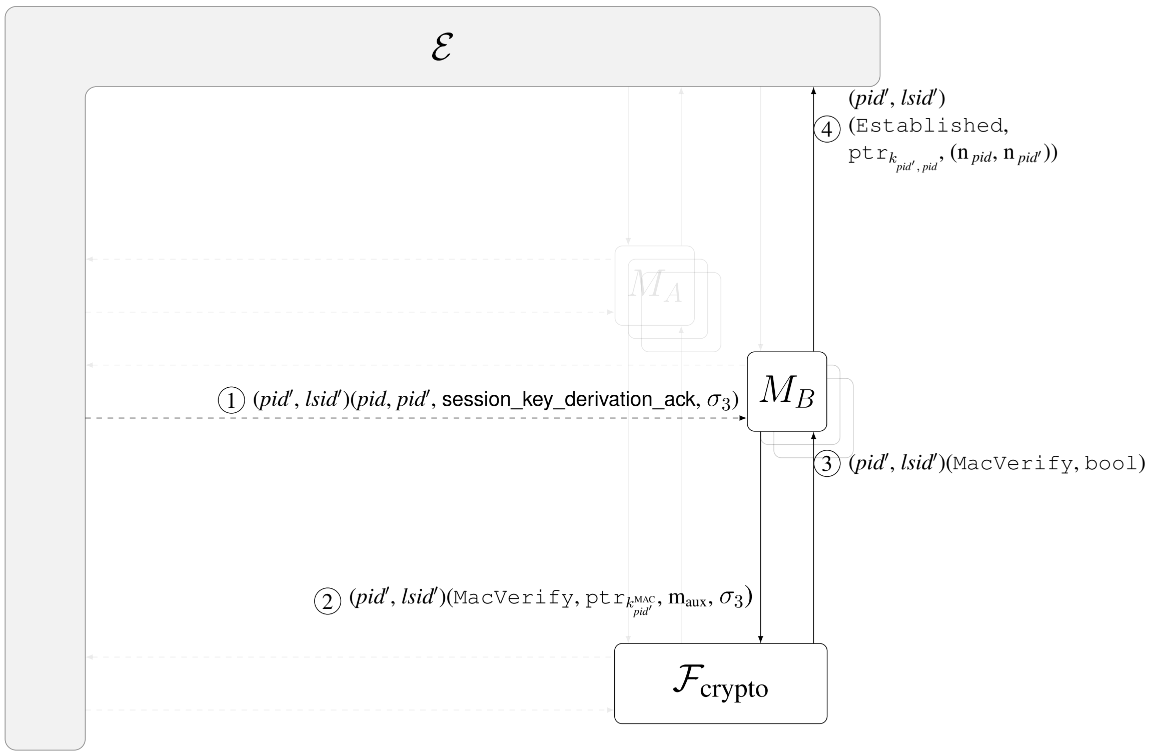

$\mathcal{E}$
② (*pid*, *lsid*)(*pid*, *pid′*, session_key_derivation_ack, σ_3)
① (*pid*, *lsid*)(FinalMessage)
M_A
M_B
$\mathcal{F}_{\text{crypto}}$

Figure 5.28: Third message of *SessionKeyDerivation* protocol.

- M_B receives the third message of the protocol on its input network tape, it has *state*$_B$ = `secondMessageSent`, it outputs the session key pointer to the user.
  - Eventually, the adversary delivers the protocol message to M_B (message ① in Figure 5.29). M_B parses the message and checks if it corresponds to the expected format and if the parties on the message correspond to the party identifiers of the recorded intended communication partner *pid* and the party identifier of the user *pid′*, respectively. If the checks fail, M_B in **CheckAddress** mode returns **reject** to the message. Otherwise, it continues processing in **CheckAddress** mode.
  - M_B asks $\mathcal{F}_{\text{crypto}}$ to verify the MAC σ_3 using the key referred by pointer $\texttt{ptr}_{k^{\text{MAC}}_{pid'}}$ through the command (`MacVerify`, $\texttt{ptr}_{k^{\text{MAC}}_{pid'}}$, m$_{\text{aux}}$, σ_3), where m$_{\text{aux}}$ = (*pid*, *pid′*, session_key_derivation_ack, n$_{pid}$, n$_{pid'}$) (message ② in Figure 5.29). If the boolean returned by $\mathcal{F}_{\text{crypto}}$ (message ③ in Figure 5.29) is `false`, i.e, the verification failed, M_B in **CheckAddress** mode returns **reject** to this message. Otherwise, i.e., if the MAC is successfully verified, we say M_B accepted the key exchanged for (*pid′*, *lsid′*) and *pid*.
  - M_B sets *state*$_B$ = `finished` and concludes the **CheckAddress** mode returning **accept**.
  - In **Compute** mode, M_B outputs the session key pointer $\texttt{ptr}_{k_{pid,pid'}}$ and the stamp (n$_{pid}$, n$_{pid'}$) of the session to the user through the message (`Established`, $\texttt{ptr}_{k_{pid,pid'}}$, (n$_{pid}$, n$_{pid'}$)) (message ④ in Figure 5.29).



$\mathcal{E}$
M_A
M_B
$\mathcal{F}_{\text{crypto}}$
① (pid′, lsid′)(pid, pid′, session_key_derivation_ack, σ_3)
② (pid′, lsid′)(MacVerify, $\mathtt{ptr}_{k^{\text{MAC}}_{pid'}}$, m_{aux}, σ_3)
③ (pid′, lsid′)(MacVerify, bool)
④ (pid′, lsid′) (Established, $\mathtt{ptr}_{k_{pid',pid}}$, ($n_{pid}$, $n_{pid'}$))

Figure 5.29: M_B outputs the session key pointer to the user.

- Executing cryptographic operations. The machines have $state_A$ = `finished` and $state_B$ = `finished`, they receive requests, forward to $\mathcal{F}_{\text{crypto}}$ and return the results to the user.
  - Users (*pid*, *lsid*,*A*) and (*pid*, *lsid*,*B*) with state `finished` has access to the same key through pointers $\mathtt{ptr}_{k_{pid,pid'}}$ and $\mathtt{ptr}_{k_{pid',pid}}$, respectively. Under a request (messages ① and ①′ in Figure 5.30) to execute ideal cryptographic operations in $\mathcal{F}_{\text{crypto}}$, the machines forward the requests to $\mathcal{F}_{\text{crypto}}$ (message ② and ②′ in Figure 5.30), and send the results received from $\mathcal{F}_{\text{crypto}}$ to the user (messages ③ and ③′, and ④ and ④′ in Figure 5.30, respectively).

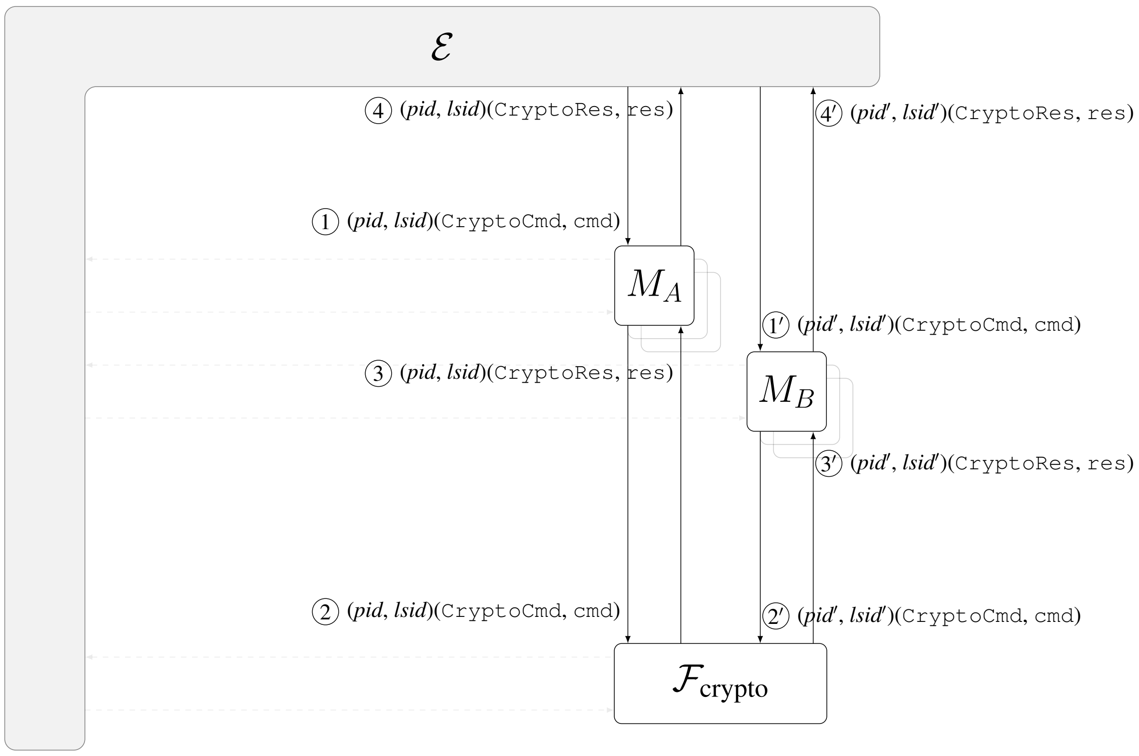

$\mathcal{E}$
④ (*pid*, *lsid*)(CryptoRes, res)
① (*pid*, *lsid*)(CryptoCmd, cmd)
M_A
③ (*pid*, *lsid*)(CryptoRes, res)
② (*pid*, *lsid*)(CryptoCmd, cmd)
④′ (*pid*′, *lsid*′)(CryptoRes, res)
①′ (*pid*′, *lsid*′)(CryptoCmd, cmd)
M_B
③′ (*pid*′, *lsid*′)(CryptoRes, res)
②′ (*pid*′, *lsid*′)(CryptoCmd, cmd)
$\mathcal{F}_{\text{crypto}}$

Figure 5.30: Cryptographic operations in an established session.

- Closing a session. The machines have *state*$_A$ = `finished` and *state*$_B$ = `finished`, they receive the request to close the session, inform the adversary about the session being closed, receive an acknowledgment from the adversary, revoke the access of the user to the cryptographic material generated during the session, and acknowledge the user.

  – After performing the desired cryptographic operations, an instance (*pid*, *lsid*,*r*) with state `finished` can be requested to close the session with the `CloseSession` command (messages ① and ①′ in Figure 5.31). The machine informs the adversary about the request and waits for an acknowledgment message (messages ② and ②′, and ③ and ③′ in Figure 5.31, respectively). The machine then sets its state to `sessionClosed`, revokes all cryptographic material generated by itself for the key exchange, and acknowledges the user (messages ④ in Figure 5.31).

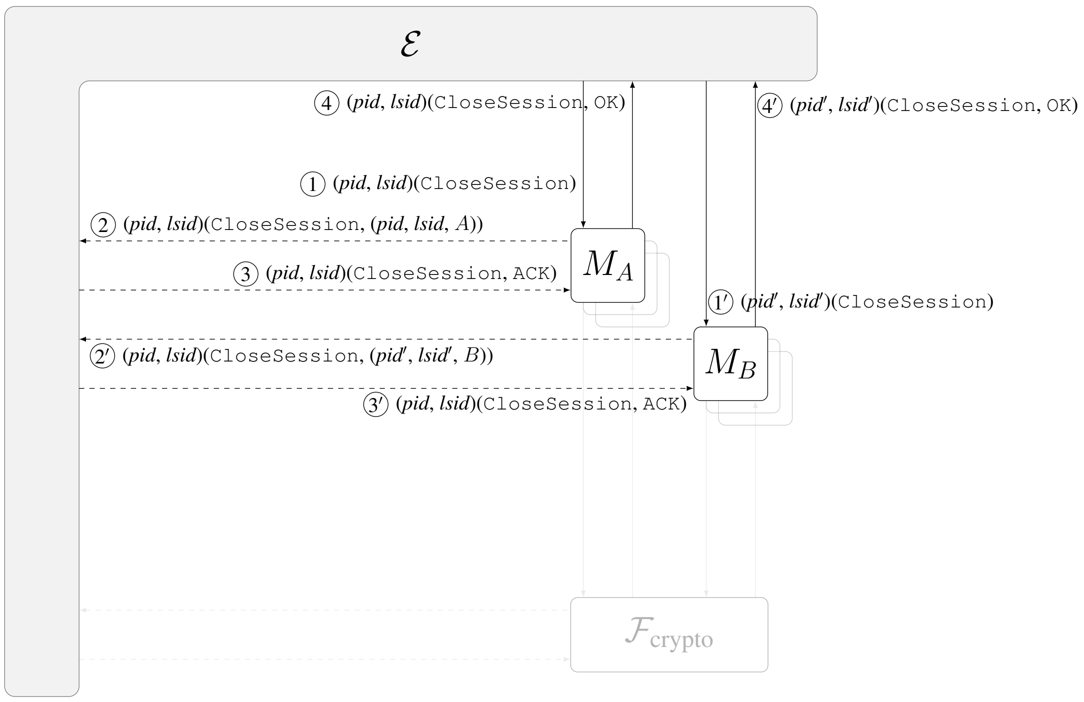

$\mathcal{E}$
④ (*pid*, *lsid*)(CloseSession, OK)
④′ (*pid*′, *lsid*′)(CloseSession, OK)
① (*pid*, *lsid*)(CloseSession)
② (*pid*, *lsid*)(CloseSession, (*pid*, *lsid*, A))
③ (*pid*, *lsid*)(CloseSession, ACK)
M_A
①′ (*pid*′, *lsid*′)(CloseSession)
②′ (*pid*, *lsid*)(CloseSession, (*pid*′, *lsid*′, B))
M_B
③′ (*pid*, *lsid*)(CloseSession, ACK)
$\mathcal{F}_{\text{crypto}}$

Figure 5.31: Closing a session.

- Corruption status request.
  - At any point, the user may request the corruption status of an instance of M_A or M_B (messages ① and ①′ in Figure 5.32). The machines return the value of its corruption status to the environment (messages ② and ②′ in Figure 5.32) considering the corruption model discussed in further details in the next section.

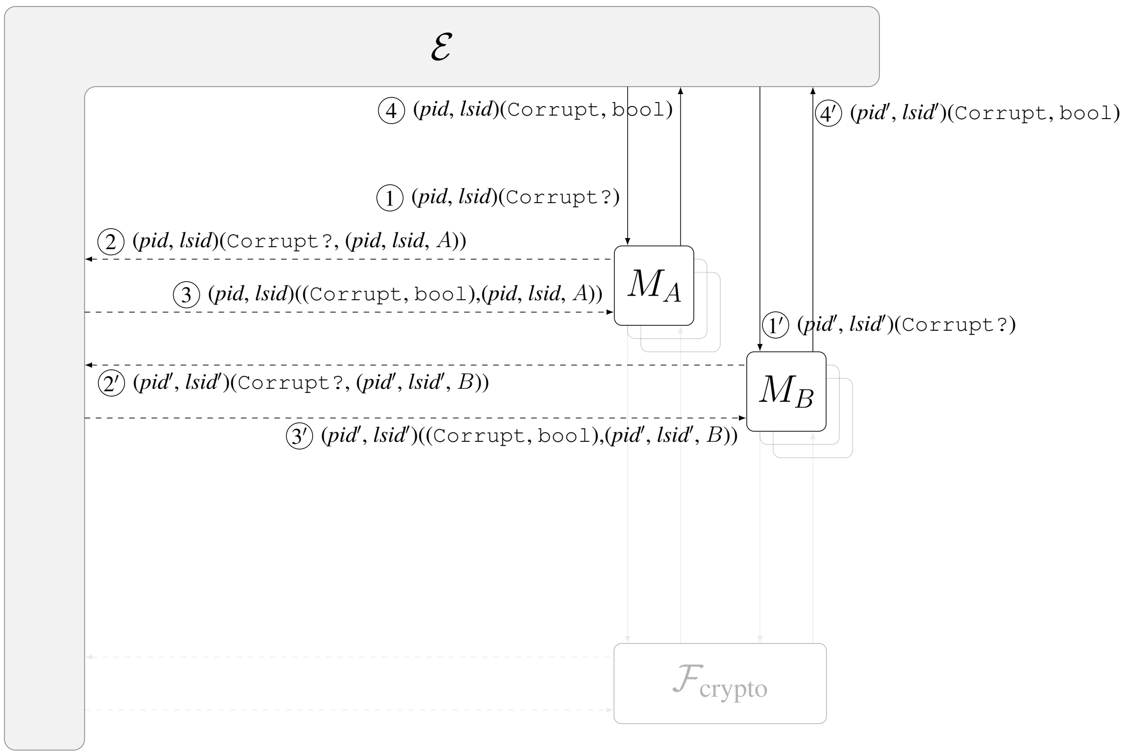

$\mathcal{E}$
④ (*pid*, *lsid*)(Corrupt, bool)
④′ (*pid*′, *lsid*′)(Corrupt, bool)
① (*pid*, *lsid*)(Corrupt?)
② (*pid*, *lsid*)(Corrupt?, (*pid*, *lsid*, *A*))
③ (*pid*, *lsid*)((Corrupt, bool),(*pid*, *lsid*, *A*))
M_A
①′ (*pid*′, *lsid*′)(Corrupt?)
②′ (*pid*′, *lsid*′)(Corrupt?, (*pid*′, *lsid*′, *B*))
M_B
③′ (*pid*′, *lsid*′)((Corrupt, bool),(*pid*′, *lsid*′, *B*))
$\mathcal{F}_{\text{crypto}}$

Figure 5.32: Corruption status request.

5.7.2.3 SessionKeyDerivation Corruption Model

The corruption model of *SessionKeyDerivation* protocol is such that:

- The adversary can send a special message to corrupt an instance of the protocol (*pid*, *lsid*, *r*), $r \in \{A, B\}$, before a key exchange starts and after the session is closed, but not during the key exchange or the session.
- A corrupted instance (*pid*, *lsid*, *r*) forwards all messages on its i/o tapes originated from or destined to its corresponding user (*pid*, *lsid*) to the adversary on the network tape.
- The adversary can execute operations in $\mathcal{F}_{\text{crypto}}$ using the corrupted instance (*pid*, *lsid*, *r*) in the name of user (*pid*, *lsid*). The $\mathcal{F}_{\text{crypto}}$ operations available for the adversary are all operations exposed to $\mathcal{E}$ and those used in the *SessionKeyDerivation* protocol implementation, all other are blocked. In this case, all secret the corrupted instance creates in $\mathcal{F}_{\text{crypto}}$ must be known.
- The adversary can corrupt an instance (*pid*, *lsid*, *r*) only if the key previously shared between *pid* and its intended communication partner is corrupted, i.e., known in $\mathcal{F}_{\text{crypto}}$,

which models that if the adversary can access this key through the corrupted instance, such a key must be corrupted.

- The adversary gets access to secret components of a closed session after directly corrupting an instance because the information of a session is not erased when the session is closed, which models the lack of perfect forward secrecy of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$.
- An instance (*pid*, *lsid*, *r*) in a run of *SessionKeyDerivation* with an instance of party *pid'* that has not output a session key pointer also considers itself corrupted if the key previously shared between *pid* and *pid'* is corrupted (known) in $\mathcal{F}_{\text{crypto}}$, even is this instance has not been explicitly corrupted by the adversary. This means that no security can be guaranteed if the secret key from which the session key is derived is corrupted.
- Our modeling allows an instance to be corrupted only before a key exchange starts and after the session is closed. However, the corruption status of an instance can also be determined by the corruption status of the key previously shared between the parties involved in the key exchange, and this key might become corrupted during the key exchange process. We define that an instance define its initial status based on the response received from the adversary at initialization and on the corruption status of the key previously shared with its intended communication party. Even not corrupted, the instance only outputs the session key pointer to its user if right before such an event an additional check on the corruption status of the pre-shared key shows that its status has not change since the first check. Otherwise, the instance blocks all requests.

5.7.2.4 SessionKeyDerivation Proof

The goal of this section goal is to prove that our multi-session of the *SessionKeyDerivation* protocol (Protocol 4.3) is as secure as the two-party mutually authenticated key exchange with key usability and in-session access to digital signatures functionality $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. However, the direct analysis on the multi-session version of the protocol is complex. Hence, we use Theorem 5.3 to analyze the security of *SessionKeyDerivation* in a single-session version and directly obtain the security for its multi-session one. Therefore, our goal is to prove the Theorem 5.5.

Theorem 5.5 *Let*

- $\mathcal{P}_{SKD} = !M_A \mid !M_B \mid \mathcal{F}_{crypto}$ *be the multi-session version modeling of the SessionKeyDerivation protocol (Protocol 4.3) as described in Section 5.7.2.1 that uses* $\mathcal{F}_{crypto}$ *and satisfies implicit disjointness w.r.t. a partnering function* τ*.*

- *M_A and M_B to behave as described in Section 5.7.2.2;*
- *$\mathcal{F}_{crypto}$ and $\mathcal{F}'_{crypto}$ two different versions of the functionality for cryptographic primitives extended in this work (Section 5.4);*
- *$\mathcal{F}^{MA}_{sk\text{-}sig}$ be multi-session version of the functionality for mutually authenticated key exchange with key usability and in-session access to digital signatures, as described in Section 5.6.1.*
- *Let F_{single} be a machine that acts as a bridge between the environment and $\mathcal{P}_{SKD}$ and $\mathcal{F}^{MA}_{sk\text{-}sig}$, which allows the environment to create at most one session in $\mathcal{P}_{SKD}$ and in $\mathcal{F}^{MA}_{sk\text{-}sig}$.*

Then it holds true that

$$F_{single} \mid \mathcal{P}_{SKD} \mid \mathcal{F}_{crypto} \leq^{\tau} F_{single} \mid \mathcal{F}^{MA}_{sk\text{-}sig} \mid \mathcal{F}'_{crypto}$$

The first step of this proof is to find a partnering function τ for the $\mathcal{P}_{\text{SKD}}$ protocol and show it is valid and satisfies implicit disjointness. As we mentioned in Section 5.3.6, a partnering function τ maps every sequence of configurations α of a machine instance M_r in a run of a real protocol $\mathcal{P}$ to a string and finds in α a sub-string that represents the session. A possible partnering function for $\mathcal{P}_{\text{SKD}}$ is the pair of nonces the roles exchange in the protocol. In a run ρ of $\mathcal{E} \mid \mathcal{P}_{\text{SKD}} \mid \mathcal{F}_{\text{crypto}}$, let α be the sequence of configurations of an instance of M_r, i.e., (*pid*$_r$, *lsid*$_r$, *r*), $r \in \{A, B\}$. If (*pid*$_r$, *lsid*$_r$, *r*) is corrupted, $\tau(\alpha) = \perp$ because, due to our modeling, there is no protocol execution to have a sequence α where a session stamp for (*pid*$_r$, *lsid*$_r$, *r*) can be found. Otherwise, i.e., if (*pid*$_r$, *lsid*$_r$, *r*) is uncorrupted, in the case *r* = A and α contains the first two messages of the protocol, then τ = (n_{pid_A}, n_{pid_B}), where n_{pid_A} is the nonce generated by (*pid*$_A$, *lsid*$_A$, A) and n_{pid_B} is the nonce received by this instance. Analogously, in the case *r* = B and α contains the first two messages of the protocol, τ is also (n_A, n_B), where n_{pid_A} is the nonce received by (*pid*$_B$, *lsid*$_B$, B) and n_{pid_B} is the nonce generated by this instance. As the nonces do not collide in $\mathcal{F}_{\text{crypto}}$, τ = (n_{pid_A}, n_{pid_B}) is a valid partnering function for $\mathcal{P}_{\text{SKD}}$.

We still have to show τ = (n_{pid_A}, n_{pid_B}) satisfies implicit disjointness, which implies to show that in a run ρ of the system (i) every explicitly shared key is either always marked unknown or always marked known in $\mathcal{F}_{\text{crypto}}$; and (ii) whenever an instance (*pid'*, *lsid'*, *r'*) uses an explicitly unknown shared key to successfully verify a MAC σ in $\mathcal{F}_{\text{crypto}}$ at some point in a run of the system, there exists a specific instance (*pid*, *lsid*, *r*) that sent a MAC generation request to $\mathcal{F}_{\text{crypto}}$ which resulted in σ such that both users are partners or both users are corrupted in this particular run. Assume k to be the shared key between parties *pid*$_A$ and *pid*$_B$ in ρ. The key k is an explicitly shared key. Since k is never encrypted or retrieved by the environment in ρ, k is either always corrupted or always uncorrupted in ρ. As k is a pre-shared key, all other keys derived from it follows the corruption status of k, i.e., all other keys are always known or always unknown in ρ. Therefore, the requirement (i) is satisfied for all the keys. We have to show that the MAC key derived from k only successfully verifies a MAC σ if σ has been generated in the same session according to the partnering function τ. Considering the MAC of message m = (pid_B, pid_A, n_{pid_B}, session_key_derivation_ack, n_{pid_A}) on the second message of the protocol, as it has the contents of the partnering function itself, the condition (ii) is satisfied. We have the same for the MAC of message m = (pid_A, pid_B, session_key_derivation_ack, n_{pid_A}, n_{pid_B}) on the third message of the protocol, because it also carries the contents of the partnering function, hence the condition (ii) is also satisfied. Therefore, we conclude that $\tau_{(pid_A, lsid_A, A)}(\rho) = \tau_{(pid_B, lsid_B, B)}(\rho) =$ (n_{pid_A}, n_{pid_B}), which means the users (pid_A, pid_A, A) and (pid_B, pid_A, B) are partners in ρ and the function τ = (n_{pid_A}, n_{pid_B}) is a valid partnering function for $\mathcal{P}_{\text{SKD}}$ and it satisfies implicit disjointness.

Now we have to show $F_{single} \mid \mathcal{P}_{\text{SKD}} \mid \mathcal{F}_{\text{crypto}} \leq^{\tau} F_{single} \mid \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, i.e., the single-session version of $\mathcal{P}_{\text{SKD}}$ realizes the single-session version of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Analogously to any other simulation proof, we have to specify a simulator $\mathcal{S}$ such as, for every environment $\mathcal{E}$, it holds that $\mathcal{E} \mid F_{single} \mid !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}} \equiv F_{single} \mid \mathcal{S} \mid \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$. In this case, as we have F_{single}, we can be more restrict and defined $\mathcal{S}$ to internally simulate the execution of the single-session version of $\mathcal{P}_{\text{SKD}} = M_A \mid M_B \mid \mathcal{F}_{\text{crypto}}$. Besides, as the adversary of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, $\mathcal{S}$ also interacts with a single session of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$. As usual, we have to show that no $\mathcal{E}$ can distinguish between the real and ideal settings, as illustrated in Figure 5.33. More specifically, the two users (pid_A, $lsid_A$) and (pid_B, $lsid_B$) in $\mathcal{E}$, corrupted or uncorrupted, will get the same result when interacting with $F_{single} \mid !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}}$ or $F_{single} \mid \mathcal{S} \mid \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$. We define $\mathcal{S}$ as follows:

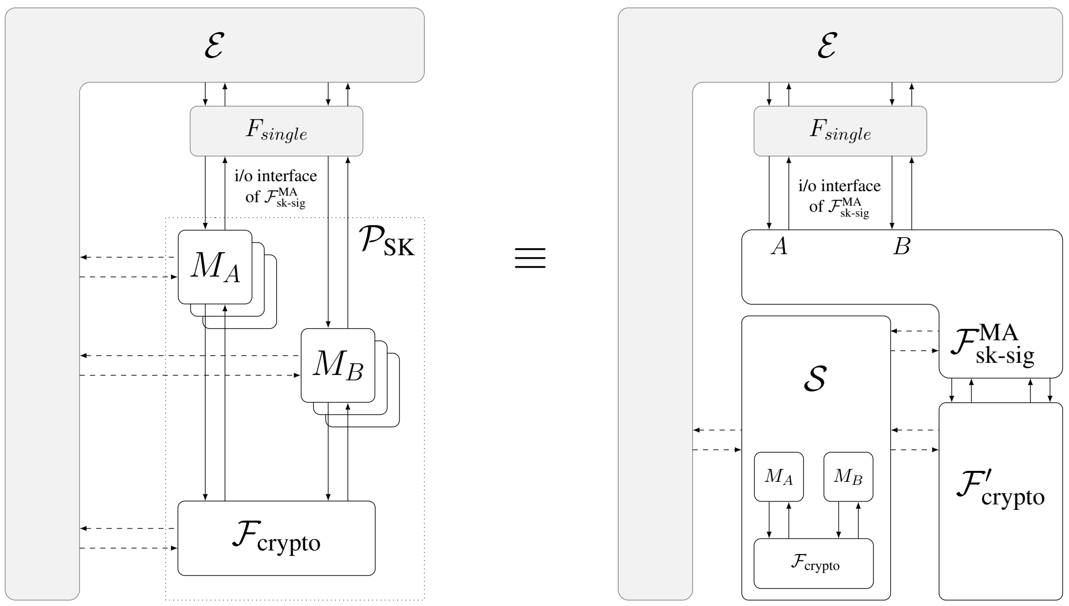

$\mathcal{E}$
F_{single}
i/o interface of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
$\mathcal{P}_{\text{SK}}$
M_A
M_B
$\mathcal{F}_{\text{crypto}}$
$\equiv$
$\mathcal{E}$
F_{single}
i/o interface of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
A
B
$\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$
$\mathcal{S}$
M_A
M_B
$\mathcal{F}_{\text{crypto}}$
$\mathcal{F}'_{\text{crypto}}$

Figure 5.33: Simulation of $M_A \mid M_B \mid \mathcal{F}_{\text{crypto}}$ to prove $\mathcal{E} \mid F_{single} \mid !M_A \mid !M_B \mid \mathcal{F}_{\text{crypto}} \equiv \mathcal{E} \mid F_{single} \mid \mathcal{S} \mid \mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$

- $\mathcal{S}$ maintains synchronized the corruption status of users (pid_A, $lsid_A$, A) and (pid_B, $lsid_B$, B) in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$ with the corresponding users (pid_A, $lsid_A$, A) and (pid_B, $lsid_B$, B) in the simulation of $\mathcal{P}_{\text{SKD}} = M_A \mid M_B \mid \mathcal{F}_{\text{crypto}}$;

- Being the adversary of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, $\mathcal{S}$ initializes $\mathcal{F}'_{\text{crypto}}$. As a result, $\mathcal{S}$ receives all of $\mathcal{F}'_{\text{crypto}}$ public parameters, which can be used in its internal simulation of $\mathcal{F}_{\text{crypto}}$. Together with the parameters response, $\mathcal{S}$ is asked to provide all cryptographic algorithms and asymmetric keys related to the digital signature scheme for $\mathcal{F}'_{\text{crypto}}$. $\mathcal{S}$ forwards the request to $\mathcal{E}$ and returns the algorithms and all the pair of keys to $\mathcal{F}'_{\text{crypto}}$;

- Whenever the users (*pid*$_A$, *lsid*$_A$, A) and (*pid*$_B$, *lsid*$_B$, B) initiates a key exchange with each other in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, as $\mathcal{S}$ is informed about it by $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ with the message ((`InitKE`, *pid*$_B$), (*pid*$_A$, *lsid*$_A$, A)) and ((`InitKE`, *pid*$_A$), (*pid*$_B$, *lsid*$_B$, B)), respectively, $\mathcal{S}$ can initiate the key exchange in its internal simulation (shown in Figure 5.6).

- In its simulation, when the uncorrupted user (*pid*$_A$, *lsid*$_A$, A) accepts the key exchange by verifying the MAC of the second protocol message, $\mathcal{S}$ sends (`GroupSession`, (*pid*$_A$, *lsid*$_A$, A), (*pid*$_B$, *lsid*$_B$, B)) to $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ (shown in Figure 5.7), instructing the functionality to create a session between the simulated users (*pid*$_A$, *lsid*$_A$, A) and (*pid*$_B$, *lsid*$_B$, B), that, respectively verified the MAC on the second protocol message and generated such a MAC;

- Upon the `GroupSession` command, $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ uses the command `GetPSK` to request $\mathcal{F}'_{\text{crypto}}$ a pointer to a new key of type t (parameter of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$). $\mathcal{F}'_{\text{crypto}}$, in turn, asks $\mathcal{S}$ for the value of the key. Then, $\mathcal{S}$ responds with the value of the session key calculated in its internal simulation of $\mathcal{P}_{\text{SKD}}$. Similarly, when $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ uses the command `NewNonce` to request $\mathcal{F}'_{\text{crypto}}$ new fresh nonces for the session, $\mathcal{F}'_{\text{crypto}}$ also asks $\mathcal{S}$ to provide the value of the nonces, which also are provided by $\mathcal{S}$ based on the internal simulation of $\mathcal{P}_{\text{SKD}}$. After $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ receives the pointers to the session key and the nonces from $\mathcal{F}'_{\text{crypto}}$, it acknowledges $\mathcal{S}$ about the success of the `GroupSession` command (shown in Figure 5.7). Hence, $\mathcal{S}$ instructs $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to output the session key pointer to user (*pid*$_A$, *lsid*$_A$) using the command (`FinishKE`, (*pid*$_A$, *lsid*$_A$, A)) (shown in Figure 5.8);

- In its internal simulation of $\mathcal{P}_{\text{SKD}}$, when the uncorrupted user (*pid*$_B$, *lsid*$_B$, B) accepts the key exchange by verifying the MAC on the third protocol message and outputs the session key pointer, $\mathcal{S}$ instructs $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to output the session key pointer to user (*pid*$_B$, *lsid*$_B$) using the command (`FinishKE`, (*pid*$_B$, *lsid*$_B$, B)) (shown in Figure 5.8);

- Once the key exchange is completed, the users can request $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ the execution of the available cryptographic operations in $\mathcal{F}'_{\text{crypto}}$. In the set of the symmetric cryptographic operations, none of them requires that $\mathcal{F}'_{\text{crypto}}$ requests new values to the adversary (simulator), for instance, for new keys, nonces, scalars, or exponents, i.e., there is no dependency on $\mathcal{F}_{\text{crypto}}$. However, there exists a dependency regarding the digital signatures scheme, because the adversary can corrupt the signing keys at any time. Nevertheless, $\mathcal{S}$ is able to capture such kind of corruption in $\mathcal{F}_{\text{crypto}}$ and replicate to $\mathcal{F}'_{\text{crypto}}$.

- When $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ informs $\mathcal{S}$ that users (*pid*$_A$, *lsid*$_A$, A) or (*pid*$_B$, *lsid*$_B$, B) asked to close the session, $\mathcal{S}$ updates the internal simulation removing the access to keys and responds `OK` to $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ (shown in Figure 5.10).

We have to show that $\mathcal{S}$ guarantees that, from the perspective of any $\mathcal{E}$, there are no differences between the real or the ideal settings. We start by arguing that, as the adversary of $\mathcal{F}^{\text{MA}}_{\text{sk-sig}} \mid \mathcal{F}'_{\text{crypto}}$, $\mathcal{S}$ can keep $\mathcal{F}'_{\text{crypto}}$ consistent with $\mathcal{F}_{\text{crypto}}$. First, we have that the only new symmetric key in $\mathcal{F}'_{\text{crypto}}$ is created due to the `GetPSK` command from $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to $\mathcal{F}'_{\text{crypto}}$, which causes $\mathcal{F}'_{\text{crypto}}$ to ask the adversary $\mathcal{S}$ for the value of the key. $\mathcal{S}$, in turn, can provide the key based on its internal simulation of $\mathcal{F}_{\text{crypto}}$, with the same value and status. As the $\mathcal{F}'_{\text{crypto}}$ is not exposed to users, neither are the commands `Store` and `Retrieve`, there is no user in $\mathcal{E}$ capable of inserting new symmetric keys or making the key known to the environment. We have a similar scenario for nonces. All nonces used in $\mathcal{F}'_{\text{crypto}}$ are created due to `NewNonce` commands from $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ to $\mathcal{F}'_{\text{crypto}}$, which causes $\mathcal{F}'_{\text{crypto}}$ to ask the adversary $\mathcal{S}$ for the values of nonces. $\mathcal{S}$, in turn, can provide the same values from its internal simulation of $\mathcal{F}_{\text{crypto}}$. Last, $\mathcal{S}$ receives the signature scheme and related key pairs from $\mathcal{E}$ and forwards it to $\mathcal{F}'_{\text{crypto}}$ at initialization, therefore the cryptosystem and all the keys exist in both $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$. According to the corruption model of $\mathcal{F}_{\text{crypto}}$ (Section 5.3.4), signing keys can be corrupted at any point in time, therefore, as soon as the simulator receives a request from the adversary to corrupt a signing key, as the adversary of $\mathcal{F}'_{\text{crypto}}$, it can corrupt such key in $\mathcal{F}'_{\text{crypto}}$, allowing $\mathcal{S}$ to keep also the digital signature setting synchronized in both $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$. Hence, $\mathcal{S}$ maintains $\mathcal{F}'_{\text{crypto}}$ always consistent with $\mathcal{F}_{\text{crypto}}$, which means no request such as for corruption request to $\mathcal{F}'_{\text{crypto}}$ will cause different results in the real or ideal settings.

Now we argue that $\mathcal{S}$ correctly handles the following use cases: (1) uncorrupted initiator user and (2) uncorrupted responder user during the key exchange; (3) uncorrupted users after the key exchange; and (4) corrupted users. During the analysis, some key points from the corruption model of *SessionKeyDerivation* (Section 5.7.2.3) are constantly mentioned are summarized as follows:

- The adversary can explicitly corrupt a user (*pid*$_r$, *lsid*$_r$, *r*) before key exchange starts and after the session is closed, but not during the key exchange or the session.
- A user (*pid*$_r$, *lsid*$_r$, *r*) can be explicitly corrupted only if the key previously shared with its intended communication party is corrupted.
- A user (*pid*$_r$, *lsid*$_r$, *r*) in a run of *SessionKeyDerivation* that has not output a session key pointer also considers itself corrupted if the key it previously shared with its intended

communication partner is corrupted. Even with an uncorrupted status, right before outputting a session key pointer, such user checks the status of this key, and it blocks if there is any change.

(1) Uncorrupted initiator user during the key exchange.

Let (pid_A, $lsid_A$, A) be an uncorrupted initiator user that wants to exchange a key with party pid'. This user can be simulated by $\mathcal{S}$ using $\mathcal{F}_{\text{crypto}}$ until it outputs a session key after verifying the MAC on the second message of the protocol without dependency on any information or operation in $\mathcal{F}'_{\text{crypto}}$. As we are analyzing the single-session version of the protocol, we have to show that $\mathcal{S}$ pairs with (pid_A, $lsid_A$, A) the user (pid', $lsid'$, B).

As (pid_A, $lsid_A$, A) is uncorrupted and outputs the session key pointer, the key previously shared between parties pid_A and pid' must be uncorrupted, otherwise this instance would block according to the corruption model. Therefore, user (pid', $lsid'$, B) cannot have been explicitly corrupted by the adversary.

By outputting the key pointer, the user (pid_A, $lsid_A$, A) has successfully verified and accepted the MAC over the message m = (pid', pid_A, n, session_key_derivation_ack, n_{pid_A}), where n_{pid_A} is the nonce generated by (pid_A, $lsid_A$, A). The key (pid_A, $lsid_A$, A) uses to verify the MAC is derived from the `pre-key` shared between pid_A and pid', i.e., no other party than pid' could generate such a key, hence the MAC on the second protocol message must have been created by user (pid', $lsid'$, B).

We just showed pid_A is the session partner of (pid', $lsid'$, B). As the key pid_A and pid' share is uncorrupted, the user (pid', $lsid'$, B) cannot consider itself corrupted. Therefore, the user (pid', $lsid'$, B) is uncorrupted.

We do not we to show that the instance (pid', $lsid'$, B) is indeed a responder, because as we are considering the single-session version of the protocol and (pid_A, $lsid_A$, A) is already an initiator, (pid', $lsid'$, B) must be a responder.

Now we have to show that (pid', $lsid'$, B) is assigned to a session in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ with (pid_A, $lsid_A$, A).

Based on the design of $\mathcal{S}$, a `GroupSession` command to create the session between the uncorrupted user (pid', $lsid'$, B) and the uncorrupted initiator user is sent only when the initiator successfully verifies and accepts the MAC on the second protocol message which is calculated over m = (pid', pid_A, n, session_key_derivation_ack, n_{pid_A}). As there is no other session, the only possible `GroupSession` command to create the global session for (pid', $lsid'$, B) has necessarily the user (pid_A, $lsid_A$, A) as initiator.

Last, as the previously shared key between pid_A and pid' is uncorrupted, the resulting session key is also uncorrupted, i.e., unknown. Hence, as the sets of keys are synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$, the simulator can indeed provide the exact same key from the simulation to $\mathcal{F}'_{\text{crypto}}$. Besides, this key can only be accessed by users (pid_A, $lsid_A$, A) and (pid', $lsid'$, B) which is the expected behavior in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Similarly, as the nonces n and n_{pid_A} are ideally created in $\mathcal{F}_{\text{crypto}}$, the simulator can indeed provide the nonces from the internal simulation to $\mathcal{F}'_{\text{crypto}}$ and keep also the set of nonces synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$.

Therefore, real and ideal settings behave identically from the perspective of an uncorrupted initiator (pid_A, $lsid_A$).

(2) Uncorrupted responder instance during the key exchange.

Let (pid_B, $lsid_B$, B) be an uncorrupted responder user that wants to exchange a key with party pid'. This user can be simulated by $\mathcal{S}$ using $\mathcal{F}_{\text{crypto}}$ until it outputs a session key pointer after verifying the MAC on the third message of the protocol without dependency on any information or operation in $\mathcal{F}'_{\text{crypto}}$. As we are analyzing the single-session version of the protocol, we have to show that $\mathcal{S}$ has paired (pid_B, $lsid_B$, B) with the user (pid', $lsid'$, A).

As (pid_B, $lsid_B$, B) is uncorrupted and outputs the session key pointer, the key previously shared between parties pid_B and pid' must be uncorrupted, otherwise this instance would block according to the corruption model. Therefore, user (pid', $lsid'$, A) cannot have been explicitly corrupted by the adversary.

By outputting the key pointer, the user (pid_B, $lsid_B$, B) has successfully verified and accepted the MAC over the message m = (pid', pid_B, session_key_derivation_ack, n, n_{pid_B}), where n_{pid_B} is the nonce generated by (pid_B, $lsid_B$, B). The key (pid_B, $lsid_B$, B) uses to verify the MAC is derived from the `pre-key` shared between pid_B and pid', i.e., no other party than pid' could generate such a key, hence the MAC on the third protocol message must have been created by user (pid', $lsid'$, A).

We just showed pid_B is the session partner of (pid', $lsid'$, A). As the key pid_B and pid' share is uncorrupted, the user (pid', $lsid'$, A) cannot consider itself corrupted. Therefore, the user (pid', $lsid'$, A) is uncorrupted.

We do not we to show that the instance (pid', $lsid'$, A) is indeed an initiator, because as we are considering the single-session version of the protocol and (pid_B, $lsid_B$, B) is already a responder, (pid', $lsid'$, A) must be an initiator.

Now we have to show that (pid_B, $lsid_B$, B) is in a global session with (pid', $lsid'$, A) in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Based on the design of $\mathcal{S}$, a `GroupSession` command to create the session between the uncorrupted user (pid', $lsid'$, B) and the uncorrupted initiator user is sent when the initiator successfully verifies and accepts the MAC on the second protocol message. The initiator only sends the third message of the protocol after it successfully verifies and accepts such MAC. Therefore, a `GroupSession` was necessarily trigger. As there is no other session, the `GroupSession` command created the global session for (pid', $lsid'$, B) and (pid', $lsid'$, A).

Last, as the previously shared key between pid_B and pid' is uncorrupted, the resulting session key is also uncorrupted, i.e., unknown. Hence, as the sets of keys are synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$, the simulator can indeed provide the exact same key from the simulation to $\mathcal{F}'_{\text{crypto}}$. Besides, this key can only be accessed by users (pid_B, $lsid_B$, B) and (pid', $lsid'$, A) which is the expected behavior in $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. Similarly, as the nonces n and n_{pid_B} are ideally created in $\mathcal{F}_{\text{crypto}}$, the simulator can indeed provide the nonces from the internal simulation to $\mathcal{F}'_{\text{crypto}}$ and keep also the set of nonces synchronized in $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$.

Therefore, real and ideal settings behave identically from the perspective of an uncorrupted responder (pid_B, $lsid_B$).

(3) Uncorrupted instances after the key exchange.

Let (pid_A, $lsid_A$, A) and (pid_B, $lsid_B$, B) be uncorrupted instances of an initiator and a responder, respectively, with an established unknown session key.

As shown in the previous cases, $\mathcal{S}$ can perfectly simulate the instances to the point when the instances get the pointers to the unknown session key in $\mathcal{F}_{\text{crypto}}$. Besides, no other user than the ones in the session has access to the session key pointer. By the design of $\mathcal{S}$, the session key is provided by $\mathcal{S}$ to $\mathcal{F}'_{\text{crypto}}$ and $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ is instructed to output session key pointers to (pid_A, $lsid_A$, A) and (pid_B, $lsid_B$, B). Therefore, after the key establishment phase, during the usage of the session key and other cryptographic operations, the real and ideal settings behave identically from the perspective of uncorrupted instances.

(4) Corrupted instances.

Let (pid, $lsid$, r), $r \in \{A, B\}$ be a corrupted instance. The simulator controls the i/o interface of such an instance. This instance is corrupted due to one of the two reasons:

(i) it was explicitly corrupted by the adversary or (ii) the key previously shared with the intended communication partner is corrupted. In case (i), the adversary controls the instance before the key exchange starts or after it finishes. In any of these stages, the adversary gets access only to known keys in $\mathcal{F}_{\text{crypto}}$, which, in fact, do not exist in $\mathcal{F}'_{\text{crypto}}$. Therefore, $\mathcal{S}$ can simulate the same behavior of $\mathcal{F}_{\text{crypto}}$ for this kind of corrupted instance. In case (ii), the simulator also has to simulate unknown keys in $\mathcal{F}_{\text{crypto}}$. However, as we shown during our analysis, an uncorrupted instance actually will never established a session key with a corrupted user, therefore, the unknown keys related to this case will never be inserted in $\mathcal{F}'_{\text{crypto}}$. Hence, $\mathcal{S}$ can also simulate this kind of corrupted instance.

With the analysis of the four possible cases, we conclude the proof of Theorem 5.5.

Now we present Corollary 5.2, that comes first from Theorem 5.3 that guarantee the security analysis of the single-version of protocol *SessionKeyDerivation* we just presented suffices to show the security of the multi-session version of protocol *SessionKeyDerivation*, the Theorem 5.1 and the proof that our extension $\mathcal{P}_{\text{crypto}}$ realizes ou extension $\mathcal{F}_{\text{crypto}}$ (Section 5.5). We use this result to replace $\mathcal{F}_{\text{crypto}}$ by our extension $\mathcal{P}_{\text{crypto}}$, obtaining a real implementation of the *SessionKeyDerivation* protocol (Section 4.3), a universally composable mutual authenticated key exchange protocol.

Corollary 5.2 *Let $\mathcal{P}_{SKD}$ = $!M_A \mid !M_B$ be the IITM modeling of the* SessionKeyDerivation *(Section 4.3) such that M_A and M_B behave as described in Section 5.7.2.2; $\mathcal{F}_{crypto}$ and $\mathcal{P}_{crypto}$ defined as in Section 5.5; $\mathcal{F}^{MA}_{sk\text{-}sig}$ be the functionality for mutually authenticated key exchange with key usability and in-session access to digital signatures, as described in Section 5.6.1; $\mathcal{F}^*$ be a machine such that in any point during the protocol systems communication blocks if the environment cause the commitment problem or does not respect the used-order requirement. Then it holds true that:*

$$\mathcal{F}^* \mid \mathcal{P}_{SKD} \mid \mathcal{P}_{crypto} \leq \mathcal{F}^* \mid \mathcal{F}^{MA}_{sk\text{-}sig} \mid \mathcal{F}_{crypto}.$$

5.7.3 AoT Modular Analysis

In this section, we analyze each stage of an IoT device life-cycle in AoT under the perspective of secure composition of protocols. The first stage, the *Pre-Deployment* (Stage 4.1) takes place at the factory, where all the communication to devices is physically protected, therefore it is considered secure by design. The *Ordering* stage (Stage 4.2), in turn, does not directly involve devices, and yet, it has all of its digital communication protected by a TLS channel, hence, it is also considered secure by design. Therefore, we focus our security analysis on the protocols that compose the *Deployment*, *Functioning*, *Retirement*, and *Inter-DomainOperation* stages (Stages 4.3, 4.4, 4.5, and 4.6, respectively).

5.7.4 Deployment Stage

We start our analysis with the Deployment stage (Stage 4.3). In our method, we decouple the *Deployment* stage specification, identifying the step where the communication between two parties starts with a session key establishment protocol ($\mathcal{P}_{\text{SK}}$ or $\mathcal{P}_{\text{SKD}}$) and the step where the communication end, which means the end of the high-level protocol. Then, we verify if the cryptographic operations executed with the established key during the protocol session are supported by $\mathcal{P}_{\text{SK}}$ or $\mathcal{P}_{\text{SKD}}$. Then, as $\mathcal{P}_{\text{SK}}$ and $\mathcal{P}_{\text{SKD}}$ realize the functionality for mutually authenticated key exchange with key usability and in-session access to digital signatures $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, we can say that the composition of the higher-level module with the session key establishment protocol also realizes $\mathcal{F}^{\text{MA}}_{\text{sk-sig}}$. The first high-level protocol ($\mathcal{P}$) we identify in the Deployment stage (Stage 4.3) comprises the steps 6, 7, and 16. We observe that in step 6, a session key is established using protocol *SessionKey* (Protocol 4.1), i.e., $\mathcal{P}_{\text{SK}}$. We can model it as the composition $\mathcal{P} \mid \mathcal{P}_{\text{SK}}$. Now, we verify that in steps 7 and 16, the cryptographic operations involved are, respectively, a digital signature and a MAC generation, followed by a MAC verification (not shown in the Deployment description). As these operations are all provided by $\mathcal{P}_{\text{SK}}$, we have that $\mathcal{P} \mid \mathcal{P}_{\text{SK}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, i.e., $\mathcal{P} \mid \mathcal{P}_{\text{SK}}$ is a secure universally composable protocol.

```
DEPLOYMENT($device\ D$)
  ...
  6.   $D_{U_r}, H$ :  SESSIONKEY($D_{U_r}, H$)
  7.   $D_{U_r} \rightarrow H$ :  deploy_req, $id_{D,\mathcal{H}}, \mathbb{A}_D, \mathbb{Y}_D, info_D$,
                 ENC($k_{D,H}$)$_{P^{\text{I}}_{H,\mathcal{H}}}$, SIG( $\text{n}_H \mid \text{n}_{D_{U_r}}$)$_{S^{\text{I}}_{D_{U_r},\mathcal{H}}}$
  ...
  16.  $H \rightarrow D_{U_r}$ :  deploy_ack, MAC($\text{n}_H \mid \text{n}_{D_{U_r}}$)$_{k_{D_{U_r},H}}$
```

Now, we identify explicit calls to the high-level protocol *KeyIssue* in steps 11, 12, and 13, and an explicit call to *Binding* in step 15. We start checking the *KeyIssue* protocol.

```
DEPLOYMENT($device\ D$)
  ...
  11.  $D$ :  KEYISSUE($D, H, \mathcal{H}$, I)
  12.  $D$ :  KEYISSUE($D, H, \mathcal{I}$, I)
  13.  $D$ :  KEYISSUE($D, H, \mathcal{H}$, A)
  ...
  15.  $D$ :  BINDING ($D, U_D$)
```

KeyIssue protocol (Protocol 4.4), which we identify as $\mathcal{P}$, in step 2, invokes protocol *SessionKey* (Protocol 4.1) or *SessionKeyDerivation* (Protocol 4.3), i.e., $\mathcal{P}_{\text{SK}}$ or $\mathcal{P}_{\text{SKD}}$, respectively, depending on the case. In both cases, however, the subsequent steps are the same. In step 3 and 4, the established key is used in an authenticated encryption scheme. As this operation is supported by $\mathcal{P}_{\text{SK}}$ and $\mathcal{P}_{\text{SKD}}$, we have that $\mathcal{P} \mid \mathcal{P}_{\text{SK}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}}$ and $\mathcal{P} \mid \mathcal{P}_{\text{SKD}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, i.e., $\mathcal{P} \mid \mathcal{P}_{\text{SK}}$ and $\mathcal{P} \mid \mathcal{P}_{\text{SKD}}$ are secure universally composable protocols.

```
KEYISSUE($device\ D$, $server\ S$, $domain\ \mathcal{Z}$, $cryptosystem$ Y)
1.          $D$ :  if $S^{\mathrm{I}}_{D,\mathcal{S}}$ does not exist:
2.       $D, S$ :  SESSIONKEYDERIVATION($D$, $S$, $k_{D,S}$)
                   otherwise:
2.       $D, S$ :  SESSIONKEY($D$, $S$)

3. $D \rightarrow S$ :  issue_req, AUTH-ENC($\mathrm{n}_S \mid \mathrm{n}_D)_{k_{D,S}}$
4. $S \rightarrow D$ :  $S^{\mathrm{Y}}_{D,\mathcal{Z}}$, issue_ack, AUTH-ENC($\mathrm{n}_S \mid \mathrm{n}_D)_{k_{D,S}}$
```

Binding protocol (Protocol 4.5), which we identify as $\mathcal{P}$, in step 1, invokes protocol *SessionKey* (Protocol 4.1), i.e., $\mathcal{P}_{\text{SK}}$. Then, in step 2, a digital signature is used and, in step 4, the established key is used to generate a MAC. These operations are supported by $\mathcal{P}_{\text{SK}}$. Therefore, we have that $\mathcal{P} \mid \mathcal{P}_{\text{SK}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, i.e., $\mathcal{P} \mid \mathcal{P}_{\text{SK}}$ is a secure universally composable protocol.

```
BINDING($device\ D$, $user\ U$)
1.       $D, C$ :  SESSIONKEY($D$, $C$)
2. $D \rightarrow C$ :  bind_req, $id_{U,\mathcal{C}}$, SIG($\mathrm{n}_C \mid \mathrm{n}_D)_{S^{\mathrm{I}}_{D,\mathcal{C}}}$
3.          $C$ :  binds $U$ to $D$
4. $C \rightarrow D$ :  bind_ack, MAC($\mathrm{n}_C \mid \mathrm{n}_D\ )_{k_{D,C}}$
```

Other steps in the Deployment stage (Stage 4.3) can be supported by $\mathcal{F}_{\text{crypto}}$ as a primitive cryptographic provider. For instance, in step 3, a device generated an ephemeral key to later encrypt it using a public key. Ideal key generation and public key encryption operations are provided as functionalities by $\mathcal{F}_{\text{crypto}}$. Analogously, $\mathcal{F}_{\text{crypto}}$ can also supports a digital signature scheme for the message in step 14. Therefore, all the communication in these steps can be securely composable with $\mathcal{F}_{\text{crypto}}$. The remaining steps (1, 2, 4, and 5), we observe they are related to physically protected communication, which is considered secure by design.

```
DEPLOYMENT($device\ D$)
1.        $U_r \rightarrow D$ :  PHY(pin$_D$)
2.     $D_{U_r} \rightarrow D$ :  PHY($id_{U_r,\mathcal{H}} \mid P^{\mathrm{I}}_{H,\mathcal{H}}$)
3.               $D$ :  generates ephemeral $k_{D,H}$
4.     $D \rightarrow D_{U_r}$ :  PHY($info_D \mid$ ENC($k_{D,H}\ )_{P^{\mathrm{I}}_{H,\mathcal{H}}}$)
5.   $U_r \rightarrow D_{U_r}$ :  PHY($id_{D,\mathcal{H}} \mid U_D \mid \mathbb{A}_D \mid \mathbb{Y}_D$)
14.   $H \Rightarrow \mathbb{G}_H$ :  $\mathbb{Y}_{\mathbb{G}_H}$, $info_{\mathbb{G}_H}$, SIG$_{S^{\mathrm{I}}_{H,\mathcal{H}}}$
```

5.7.5 Functioning Stage

Functioning stage (Stage 4.4), in step 2, invokes the *Long-TermKeyAgreement* protocol (Protocol 4.2) that establishes a long-term pairwise key. In $\mathcal{F}_{\text{crypto}}$, this is modeled as a setup assumption using the command `GetPSK`, which we already considered inside *SessionKeyDerivation* modeling. Then, we model the *Functioning*, which we call $\mathcal{P}$, as a composition with *SessionKeyDerivation* protocol $\mathcal{P}_{\text{SKD}}$ (Protocol 4.3), i.e., $\mathcal{P} \mid \mathcal{F}_{\text{crypto}}$. In the subsequent steps, the established key is used in MAC operations and a digital signature is also performed. All these operations are supported by $\mathcal{P}_{\text{SKD}}$. Therefore, $\mathcal{P} \mid \mathcal{P}_{\text{SKD}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, i.e., $\mathcal{P} \mid \mathcal{P}_{\text{SKD}}$ is a secure universally composable protocol.

```
FUNCTIONING($user\ U$, $device\ A$, $device\ B$, $operation$ op)
1.          $U$ :  uses $A$ to request op over $B$
2.          $A$ :  if $k^{LT}_{A,B}$ does not exist:
         $A, B$ :  LONG-TERMKEYAGREEMENT($A$, $B$)
3.       $A, B$ :  SESSIONKEYDERIVATION($A$, $B$, $k^{LT}_{A,B}$)
4.  $A \to B$ :  op_req, op, $\text{MAC}(\text{n}_B \mid \text{n}_A)_{k_{A,B}}$
5.  $B \to A$ :  op, $\Upsilon_{op}$, $\text{MAC}(\text{n}_B \mid \text{n}_A)_{k_{A,B}}$
6.  $A \to B$ :  op, $\Upsilon_{op}$, $\text{SIG}(\text{n}_B \mid \text{n}_A)_{S^{\text{A}}_{A,\mathcal{H}}}$
7.  $B \to A$ :  op_ack, op, $\text{MAC}(\text{n}_B \mid \text{n}_A)_{k_{A,B}}$
```

Retirement stage (Stage 4.5) and *Reassignment* protocol (Protocol 4.7 are special cases of *Functioning*. However, they explicitly call protocol *Unbinding* (Protocol 4.6), therefore, we have to analyze it.

```
RETIREMENT($user\ U$, $device\ A$, $device\ B$)
  ...
  2.  $B$ :  UNBINDING($B$, $U$)
REASSIGNMENT($user\ U$, $device\ A$, $device\ B$, $user\ V$)
  ...
  4.  $B$ :  UNBINDING($B$, $U$)
```

Unbinding protocol (Protocol 4.6), which we identify as $\mathcal{P}$, in step 1, invokes protocol *SessionKey* (Protocol 4.1), i.e., $\mathcal{P}_{\text{SK}}$. In step 2, a digital signature is used and, in step 4, the established key is used to generate a MAC. These operations are all supported by $\mathcal{P}_{\text{SK}}$. Therefore, we have that $\mathcal{P} \mid \mathcal{P}_{\text{SK}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, i.e., $\mathcal{P} \mid \mathcal{P}_{\text{SK}}$ is a secure universally composable protocol.

```
UNBINDING($device\ D$, $user\ U$)
  1.        $D, C$ :  SESSIONKEY($D$, $C$)
  2.  $D \to C$ :  unbind_req, $id_{U,C}$, SIG($n_C \mid n_D$)$_{S^{\mathrm{I}}_{D,C}}$
  3.           $C$ :  unbinds $U$ to $D$
  4.  $C \to D$ :  unbind_ack,pin$'_B$, MAC($n_C \mid n_D$ )$_{k_{D,C}}$
```

5.7.6 Inter-DomainOperation Stage

Inter-DomainOperation stage (Stage 4.6), which we identify as $\mathcal{P}$, in step 1, invokes protocol *Inter-DomainSessionKey* (Protocol 4.8) which is identical to the *SessionKey* protocol (Protocol 4.2.1), i.e., $\mathcal{P}_{\text{SK}}$. In steps 2 and 3, the established key is used in MAC operations, which are, in turn, supported by $\mathcal{P}_{\text{SK}}$. Therefore, we have that $\mathcal{P} \mid \mathcal{P}_{\text{SK}} \leq \mathcal{F}^{\text{MA}}_{\text{sk-sig}}$, i.e., $\mathcal{P} \mid \mathcal{P}_{\text{SK}}$ is a secure universally composable protocol, which concludes our modular security analysis of AoT.

```
INTER-DOMAINOPERATION($user\ U$, $device\ A$, $device\ B$, $operation$ op)
  5.           $U$ :  uses $A$ to request op over $B$
  6.       $A, B$ :  INTER-DOMAINSESSIONKEY($A$, $B$)
  7.  $A \to B$ :  inter_op_req, op, MAC($n_B \mid n_A$)$_{k_{A,B}}$
  8.  $B \to A$ :  inter_op_ack, op, MAC($n_B \mid n_A$)$_{k_{A,B}}$
```

Chapter 6

Development

We now describe our AoT prototype, its software architecture (Section 6.1), its implementation (Section 6.2), and our proof-of-concept demo (Section 6.2.5).

Our prototype provides authentication and access control using device identities and attributes; isolated Cloud and Home domains, allowing users and manufactures to control devices independently; lightweight requirements, supporting resource-constrained embedded devices; and flexibility, being deployable on different platforms.

6.1 Architecture

Figure 6.1 shows the entities in our architecture. Our architecture comprises one Cloud server (Figure 6.1, label 1) to control the manufacturer's Cloud domain. We assume the Cloud server has on-demand resource allocation (CPU, storage, memory, and bandwidth), typical of cloud environments. Our architecture also comprises multiple Home servers, one for each Home domain (Figure 6.1, label 2). Although Home servers need not scale like Cloud servers, we assume a Home server stays on and has sufficient resources to control hundreds of devices in its domain. Home servers can run on video-game consoles, desktop PCs, network gateways, or Hardware Secure Modules (HSMs).

Each device (Figure 6.1, label 3) connects to its Home domain (dashed line) and also to the manufacturer's Cloud domain (solid line). The devices also communicate to each other in their Home domains (dotted lines). Our architecture considers devices vary wildly (in terms of processing power, available memory, storage capacity, communication technologies, size, and weight) and may have constrained resources.

Entities in our architecture run the software stack shown in Figure 6.2. The implementation of the AoT protocol is the center piece of the software stack. It is built atop a cryptographic library accessed through native method calls, and communicates with the exterior world using the appropriate communication technologies and encoding mechanisms. Most entities also provide interfaces for user interaction (not shown in the software stack).

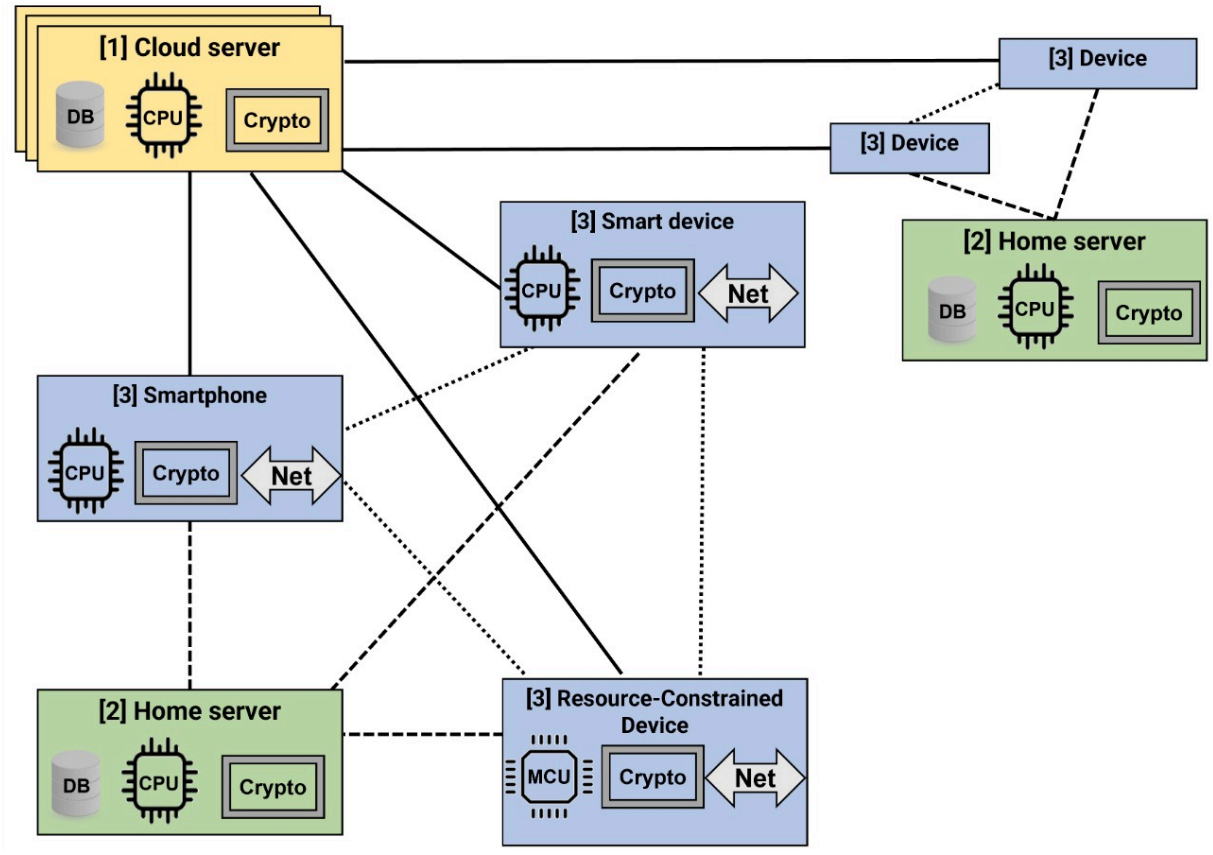


Figure 6.1: Entities comprising the AoT architecture.

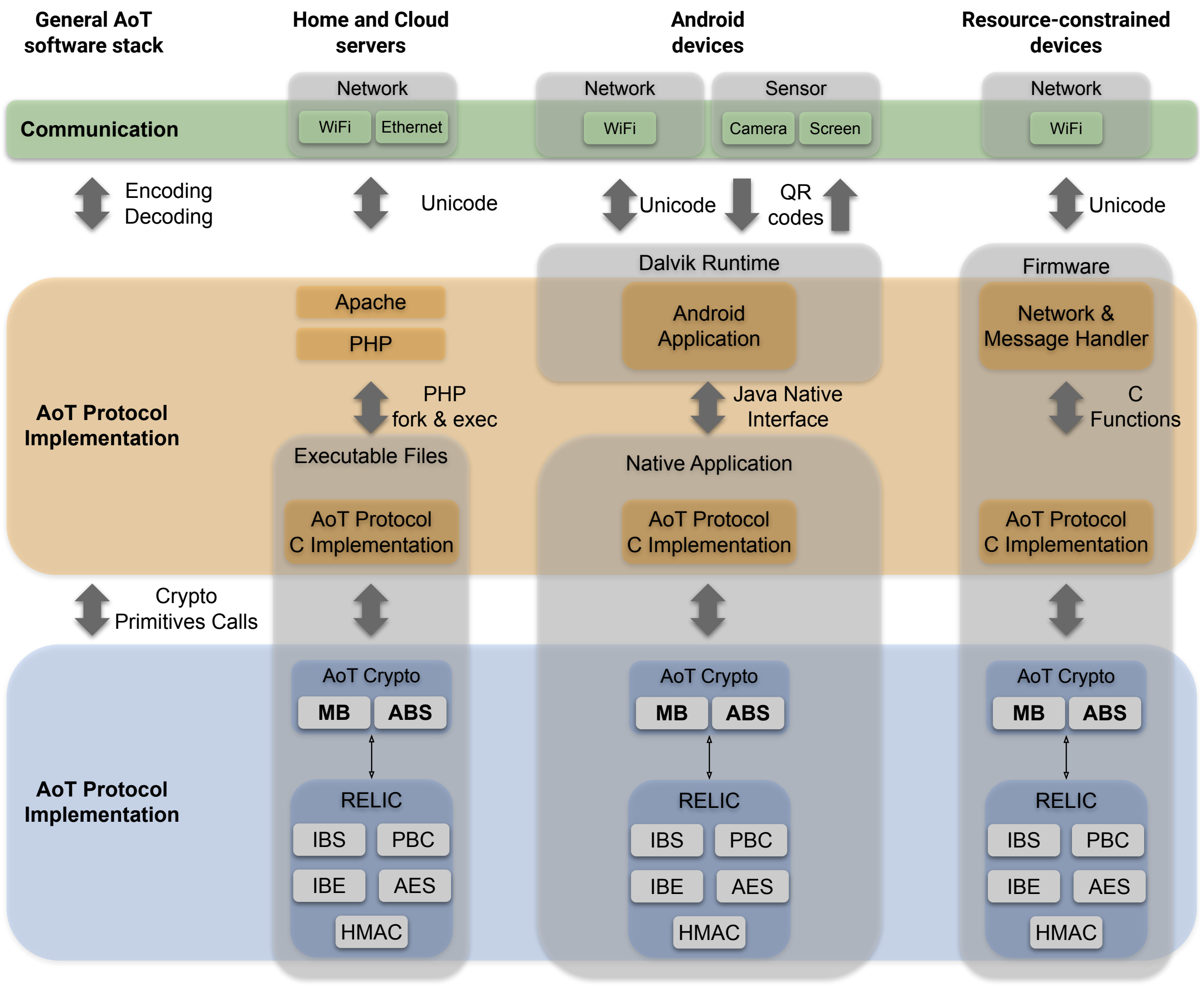


Figure 6.2: AoT entity software stack.

6.2 Implementation

6.2.1 Software stacks

Our architecture encompasses devices running heterogeneous hardware and different operating systems, each with their own specific software stack. We implement Cloud and Home servers on LAMP (Linux, Apache, MySQL, and PHP). We also implement AoT on smartphones as a native Android application. The realization of our software stack on these implementations is shown in Figure 6.2. Our Cloud and Home servers communicate with devices using the HTTP protocol. The Web servers call binaries that implement AoT's protocols from PHP. On Android devices, we implement the user interface using Android's standard interface, and use the Java Native Interface (JNI) to call AoT protocol functions on a native application which, in turn, call the cryptographic primitives from a cross-compiled cryptographic library. On resource-constrained devices, the firmware is formed by a network and message handler module that receives the messages from the network, parses them, and appropriately calls the AoT protocol functions which, analogously to the other cases, call the cryptographic primitives from a cross-compiled cryptographic library. We note that modifications to one layer in the stack do not imply changes in other layers, e.g., we could exchange HTTP for another transport such as HTTPS or CoAP, or use any cryptographic library.

6.2.2 Communication

Our cryptographic code is agnostic to message encoding and transport; any two devices can exchange information using any mutually-supported encoding and protocol. For example, our prototype can exchange data using QR codes, when deploying a fridge in a Home domain, or HTTP, when changing user attributes from a smartphone.

6.2.3 User interfaces

Although entities share the same underlying implementation of the AoT protocol, they run application code and user interfaces implemented using native libraries. For example, our Android applications run in Android's Dalvik virtual machine and use the Java Native Interface (JNI) to call our low-level AoT cryptographic primitives.

6.2.4 Cryptography

All entities in our architecture – Cloud servers, Home servers, and devices – share the same underlying implementation of AoT and use our extended version of the RELIC crypto library [4]. We chose RELIC because it targets resource-constrained devices and efficiently implements several curve-based cryptographic primitives and protocols at different (e.g., 100- and 128-bit) security levels. We have further improved RELIC's performance including new architecture-dependent optimizations by means of carefully crafted assembly for the ARM architecture to implement the base field arithmetic backend for a specific elliptic curve. For instance, we use in the arithmetic backend extensive use of the wide 32-bit multiplier instruction UMLAL for implementing Karatsuba-Comba multiplication, squaring, and Montgomery modular reduction to accelerate primitives used by AoT.

We implement the core of AoT session key establishment and access control mechanisms on top of our extended version of RELIC, which we identify as MB and ABS in Figure 6.2. We need to implement them because differently from the other cryptographic protocols, i.e., IBE, IBS, MAC, and AES, they are not part of RELIC.

6.2.4.1 MB Implementation

We implement our *SessionKey* protocol (Protocol 4.1) which is based on the enhanced [29] identity-based key agreement protocol of [81] (the reason of the name MB) with three algorithms: MB setup, MB key generation, MB public relation generation, and MB session key establishment. Our implementation relies mainly on PBC (explained in Section 2.5), which is parameterized in the library at compile time, which means during the execution of the algorithms, all the PBC parameters, i.e., groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with order r, identity $\mathcal{O}$, and generators G_1 and G_2, respectively, the group $\mathbb{G}_T$ of order r and identity 1, and the admissible *bilinear pairing* $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, the mapping functions $\textsc{Map}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_r^*$ and $\textsc{Map}_2 : \{0,1\}^* \times \{0,1\}^* \times \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_T \rightarrow \{0,1\}^n$ are set. In the AoT context, the MB setup and MB key generation algorithms are run by the server S of domain $\mathcal{Z}$, while MB public relation generation, and MB session key establishment are run by any entity that needs to establish a session key with another entity in the domain $\mathcal{Z}$. The MB setup algorithm (Algorithm 2) sets the public and private parameters of an identity-based cryptosystem. It chooses random a random master secret $s \in \mathbb{Z}_r^*$, and sets the master public key $P^{\mathrm{I}}_{S,\mathcal{Z}}$ of the system. The algorithm outputs s, and $P^{\mathrm{I}}_{S,\mathcal{Z}}$.

Algorithm 2: MB Setup

```
    Output: $s$, $P^{\mathrm{I}}_{S,\mathcal{Z}}$
1   $s \leftarrow$ random from $\mathbb{Z}_r^*$
2   $P^{\mathrm{I}}_{S,\mathcal{Z}} = s \cdot G_1$
```

The MB key generation algorithm (Algorithm 3) generates the keys of the IBC. It takes as input the identity $id_{D,\mathcal{Z}}$ of device D in domain $\mathcal{Z}$, the IBC master public key $P^{\mathrm{I}}_{S,\mathcal{Z}}$, and the IBC master secret s, and outputs the identity private key $S^{\mathrm{I}}_{D,\mathcal{Z}}$ of device D on domain $\mathcal{Z}$.

Algorithm 3: MB Key Generation

```
    Input  : $id_{D,\mathcal{Z}}$, $P^{\mathrm{I}}_{S,\mathcal{Z}}$, $s$
    Output: $S^{\mathrm{I}}_{D,\mathcal{Z}}$
1   $S^{\mathrm{I}}_{D,\mathcal{Z}} = ((\textsc{Map}_1(id_{D,\mathcal{Z}}) + s)^{-1} \cdot G_2)$
```

A device A uses the algorithm MB public relation generation (Algorithm 4) to generate a random scalar associated with the identity of the intended communication device B. Algorithm 4 takes as input the identity $id_{B,\mathcal{Z}}$ of the device B on domain $\mathcal{Z}$, and the IBC master public key $P^{\mathrm{I}}_{S,\mathcal{Z}}$. The algorithm outputs a randomly chosen scalar $\mathsf{x} \in \mathbb{Z}_r^*$ and the public relation $\mathsf{R} = \mathsf{x} \cdot P^{\mathrm{I}}_{B,\mathcal{Z}}$.

Algorithm 4: MB Public Relation Generation

```
    Input  : $id_{B,\mathcal{Z}}$, $P^{\mathrm{I}}_{S,\mathcal{Z}}$
    Output: $\mathsf{x}$, $\mathsf{R}$
1   $P^{\mathrm{I}}_{B,\mathcal{Z}} = (\textsc{Map}(id_{B,\mathcal{Z}}) \cdot G_1 + P^{\mathrm{I}}_{S,\mathcal{Z}})$
2   $\mathsf{x} \leftarrow$ random from $\mathbb{Z}_r^*$
3   $\mathsf{R} = \mathsf{x} \cdot P^{\mathrm{I}}_{B,\mathcal{Z}}$
```

A device A which identity is $id_{A,\mathcal{Z}}$ uses the algorithm MB session key establishment (Algorithm 5) to derive a new identity-based key k. Algorithm 5 takes as input a scalar $\mathsf{x} \in \mathbb{Z}_r^*$, a public relation $\mathsf{R} \in G_1$, the identity of the intended communication device $id_{B,\mathcal{Z}}$, the device identity $id_{A,\mathcal{Z}}$, the requester private key $S^{\mathrm{I}}_{A,\mathcal{Z}}$, the IBC master public key $P^{\mathrm{I}}_{S,\mathcal{Z}}$, and a flag `initiator`, and returns a key k according the Algorithm 5.

Algorithm 5: MB Session Key Establishment

```
   Input  : $\mathsf{x}$, $\mathsf{R}$, $id_{B,\mathcal{Z}}$, $id_{A,\mathcal{Z}}$, $S^{\mathrm{I}}_{A,\mathcal{Z}}$, $P^{\mathrm{I}}_{S,\mathcal{Z}}$, initiator
   Output: $k$
 1 $P^{\mathrm{I}}_{B,\mathcal{Z}} = (\textsc{Map}(id_{B,\mathcal{Z}}) \cdot G_1 + P^{\mathrm{I}}_{S,\mathcal{Z}})$
 2 if initiator then
 3     $k = \textsc{Map}_2(id_{A,\mathcal{Z}}, id_{B,\mathcal{Z}}, \mathsf{x} \cdot P^{\mathrm{I}}_{B,\mathcal{Z}}, \mathsf{R}, \hat{e}(\mathsf{R}, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(\mathsf{x} \cdot G_1, G_2))$
 4 else
 5     $k = \textsc{Map}_2(id_{B,\mathcal{Z}}, id_{A,\mathcal{Z}}, \mathsf{R}, \mathsf{x} \cdot P^{\mathrm{I}}_{B,\mathcal{Z}}, \hat{e}(\mathsf{R}, S^{\mathrm{I}}_{A,\mathcal{Z}})\hat{e}(\mathsf{x} \cdot G_1, G_2))$
```

6.2.4.2 ABS Implementation

Our ABS implementation, core of *Functioning* stage (Stage 4.4), in turn, is based on the evolution of the seminal ABS work of [77, 78], which is one of the most practical constructions of ABS, however, it is criticized of being proved only in the generic group model. Our implementation comprises five algorithms: ABS setup, ABS attribute generation, ABS signature generation (S.ABS), and ABS deterministic (D.ABS) and probabilistic (P.ABS) versions of signature verification.

Analogously to MB, our implementation is based mainly on PBC. Besides, this particular ABS scheme relies on Monotone Spam Programs (MSP), which represent the predicates as monotone boolean functions. Informally, monotone means that the boolean expression contains zero negations (*not* operators). We can work around this limitation by inverting attributes when needed, e.g., creating attributes such as “over 18 years old” and “under 18 years old”. In practice, an MSP is a matrix with dimensions $\ell \times t$, where ℓ denotes the number of attributes and t the number of ‘and’ operators in the boolean expression plus one.

In AoT, the ABS setup and attribute generation algorithms are run by Home servers, while ABS signature generation and ABS signature verification are run by devices in a home domain whenever access control is executed. The ABS setup algorithm (Algorithm 6) sets up the ABS cryptosystem. It takes as input the ABS maximum MSPs width t_{max} and randomly chooses $[H_0, H_{t_{max}}]$ generators of $\mathbb{G}_1$, randomly chooses G and C generators of $\mathbb{G}_2$, randomly chooses $a_0, a, b \in \mathbb{Z}_r^*$ and sets $A_0 = a_0 \cdot H_0$ and $A_i = a \cdot H_i$, $B_i = b \cdot H_i$ for $1 \leq i \leq t_{max}$. The algorithm outputs the ABS master private key as $S^{\mathrm{A}}_{H,\mathcal{H}} = a_0, a, b$ and the ABS master public key as $P^{\mathrm{A}}_{H,\mathcal{H}} = G, C, [H_0, H_{t_{max}}], [A_0, A_{t_{max}}], [B_1, B_{t_{max}}]$.

The ABS attribute generation algorithm (Algorithm 7) generates the keys of each user in the system. It takes as input the set of attributes $\mathbb{A}_D$ of device D, and the ABS master public and private keys, $P^{\mathrm{A}}_{H,\mathcal{H}}$ and $S^{\mathrm{A}}_{H,\mathcal{H}}$, respectively. The algorithm outputs the ABS private key of device D as $S^{\mathrm{A}}_{D,\mathcal{H}} = K_{base}, K_0, \{K_u | u \in \mathbb{A}_D\}$ as detailed in Algorithm 7.

The S.ABS (Algorithm 8) takes as input a message m to be signed, a predicate Υ that should be satisfied, a MSP matrix $M_{\ell \times t}$ that represents Υ, a vector $\vec{u}_{1\times\ell}$ that relates each attribute

Algorithm 6: ABS Setup

```
Input  : $t_{max}$
Output: $P^{\text{A}}_{H,\mathcal{H}} = G, C, [H_0, H_{t_{max}}], [A_0, A_{t_{max}}], [B_1, B_{t_{max}}]$
        $S^{\text{A}}_{H,\mathcal{H}} = a_0, a, b$
1 $H_0 \leftarrow \mathbb{G}_1$ random generator
2 $G, C \leftarrow \mathbb{G}_2$ random generators
3 $a_0, a, b \leftarrow$ random from $\mathbb{Z}_r^*$
4 $A_0 = a_0 \cdot H_0$
5 for $i = 1$ to $t_{max}$ do
6     $H_i \leftarrow \mathbb{G}_1$ random generator
7     $A_i = a \cdot H_i$
8     $B_i = b \cdot H_i$
```

Algorithm 7: ABS Attribute Generation

```
Input  : $P^{\text{A}}_{H,\mathcal{H}}$, $S^{\text{A}}_{H,\mathcal{H}}$,$\mathbb{A}_D$
Output: $S^{\text{A}}_{D,\mathcal{H}} = K_{base}, K_0, \{K_u | u \in \mathbb{A}_D\}$
1 $K_{base} \leftarrow \mathbb{G}_2$ random generator
2 $K_0 = {a_0}^{-1} \cdot K_{base}$
3 foreach $u \in \mathbb{A}_{\text{D}}$ do
4     $K_u = (a + bu)^{-1} \cdot K_{base}$
```

in Υ with a row in $M_{\ell\times t}$, the device D's ABS private key $S^{\text{A}}_{D,\mathcal{H}}$, and the ABS master public key $P^{\text{A}}_{H,\mathcal{H}}$. The algorithm outputs the signature $\sigma = Y, W, [S_1, S_\ell], [P_1, P_t]$ as detailed in Algorithm 8.

Algorithm 8: S.ABS - ABS Signature Generation

```
Input  : $m, \Upsilon, M_{\ell\times t}, \vec{u}_{1\times\ell}, S^{\text{A}}_{D,\mathcal{H}}, P^{\text{A}}_{H,\mathcal{H}}$
Output: $\sigma = Y, W, [S_1, S_\ell], [P_1, P_t]$
1 Computes $\vec{v}M = [1, 0, 0, ..., 0]_{1\times\ell}$
2 $\mu = \mathcal{H}(m||\Upsilon)$
3 $r_0 \leftarrow$ random from $\mathbb{Z}_r^*$
4 $Y = r_0 \cdot K_{base}$
5 $W = r_0 \cdot K_0$
6 for $i = 1$ to $\ell$ do
7     $r_i \leftarrow$ random from $\mathbb{Z}_r^*$
8     $S_i = ((v_i r_0) \cdot K_{u(i)}) + r_i \cdot (C + \mu \cdot G)$
9 for $j = 1$ to $t$ do
10    $X = \mathcal{O}$
11    for $i = 1$ to $\ell$ do
12        $X = X + (M_{ij} r_i) \cdot (A_j + u(i) \cdot B_j)$
13    $P_j = X$
```

Both version of ABS signature verification algorithms, D.ABS (Algorithms 9) and P.ABS (Algorithms 10), take as input the message m, predicate Υ, MSP matrix $M_{\ell\times t}$, vector $\vec{u}_{1\times\ell}$, signature σ, and ABS master public key $P^{\text{A}}_{H,\mathcal{H}}$. The algorithms return true if the signature verifies and false otherwise. P.ABS trades determinism in the signature verification by efficiency. In this case, legitimate signatures are successfully verified with probability 1, while invalid signatures are successfully verified with probability at most $1/q$. In terms of efficiency, the probabilistic algorithm needs $\ell + 2$ computation of pairings compared to $\ell t + t + 2$ performed by the deterministic version.

Algorithm 9: D.ABS - ABS Deterministic Signature Verification

```
   Input  : $m, \Upsilon, M_{\ell \times t}, \vec{u}_{1\times\ell}, \sigma, P^{A}_{H,\mathcal{H}}$
   Output: true or false
1  if $\hat{e}(A_0, W) \neq \hat{e}(H_0, Y)$ return false
2  for $j = 1$ to $t$ do
3      $X = 1$
4      for $i = 1$ to $\ell$ do
5          $X = X\hat{e}(M_{ij} \cdot (A_j + u(i) \cdot B_j), S_i)$
6      if $j = 1$ then
7          if $X \neq \hat{e}(H_1, Y)\hat{e}(P_1, C + \mu \cdot G)$ return false
8      else
9          if $X \neq \hat{e}(P_j, C + \mu \cdot G)$ return false
10 return true
```

Algorithm 10: P.ABS - ABS Probabilistic Signature Verification

```
   Input  : $m, \Upsilon, M_{\ell \times t}, \vec{u}_{1\times\ell}, \sigma, P^{A}_{H,\mathcal{H}}$
   Output: true or false
1  if $\hat{e}(A_0, W) \neq \hat{e}(H_0, Y)$ return false
2  $T = \mathcal{O}$
3  for $j = 1$ to $t$ do
4      $r_j \leftarrow$ random from $\mathbb{Z}_r^*$
5      $T = T + r_j \cdot P_j$
6  $L = 1$
7  for $i = 1$ to $\ell$ do
8      $T = \mathcal{O}$
9      for $j = 1$ to $t$ do
10         $T = T + (M_{ij} r_j) \cdot (A_j + u(i) \cdot B_j)$
11     $L = L\hat{e}(T, S_i)$
12 if $L \neq \hat{e}(r_1 \cdot H_1, Y)\hat{e}(T, C + \mu \cdot G)$ return false
13 return true
```

The computational complexity of ABS, D.ABS, and P.ABS depends on PBC operations, more precisely, on pairings and elliptic curve scalar point multiplication. Although these operations can be executed on resource-constrained devices [92], ABS requires devices to compute a product of pairings (Algorithm 9, line 5 and Algorithm 10, line 11), which increases not only compute time but also memory consumption. As a mitigation to this problem, we optimized RELIC to compute products of pairings simultaneously. Pairing computation can be divided in two phases: the Miller loop consisting of a square-and-multiply algorithm and the final exponentiation. When a product of pairings is computed simultaneously (a multi-pairing operation), squarings in the full extension field and the final exponentiation can be shared for all pairings, keeping a single variable accumulating partial results in the Miller loop [104]. This optimization saves one large $\mathbb{G}_T$ element to be stored and around 50% finite field multiplications per additional computed pairing in the product.

6.2.4.3 AoT Cryptographic Instantiation

Our complete implementation of AoT combines the following cryptographic protocols and algorithms. (i) *McCullagh-Barreto:* an enhanced MB identity-based key agreement protocol [29, 81], the MB algorithms we just described. (ii) *Boneh-Franklin (BF-IBE):* an identity-based encryption scheme, adapted to employ asymmetric pairings [18]. (iii) *Bellare-Namprempre-Nevem (vBNN-IBS):* a pairing-free identity-based signature with short signatures and fast verification [26]. (iv) *Maji-Prabhakaran-Rosulek:* an attribute-based signature [78], the ABS algorithms we just described. (v) *keyed-Hash Message Authentication Code (HMAC):* a specific type of message authentication code (MAC) involving a cryptographic hash function, using SHA384 [59]. (vi) *Advanced Encryption Standard (AES):* the standard for symmetric encryption, using CBC mode and 256 bits key length [32]. From now on, we refer to them as simply MB, IBE, IBS, ABS, HMAC, and AES, respectively.

6.2.4.4 Parameterization

As previously mentioned, the core of the cryptographic protocols in AoT relies mainly on PBC, which is, in practice, based on elliptic curves, as described in Section 2. Therefore, in order to make AoT practical, it is crucial to choose elliptic curve parameters that allow an efficient implementation of pairing computation without compromising security level.

In AoT we first adopt a curve from Barreto-Naehrig (BN) [9] family, an important class of pairing-friendly elliptic curves [93]. Curves in this family support an efficient optimal Ate pairing construction [116], with a reduction of a quarter on the number of cycles in the Miller loop [34]. This, in turn, makes these curves ideal from an implementation point of view [5]. BN curves have the form $E : y^2 = x^3 + b, b \neq 0$, they are parameterized by an integer $u \in \mathbb{Z}$ and defined over a field $\mathbb{F}_p$ with a prime and group orders $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$

and $r = 36u^4 + 36u^3 + 18u^2 + 6u + 1$, respectively, and embedding degree $k = 12$. We select the BN254 curve [88] $E : y^2 = x^3 + 2$ parameterized by the integer $u = -(2^{62} + 2^{55} + 1)$, which generates a 254-bit prime number curve order. In this instantiation, elements from the groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ take 64, 128, 384 bytes, respectively. BN curves have been affected by improvements of the asymptotic complexity of discrete logarithm in finite fields of small characteristic [58], which reduced the concrete security of BN254 curve that is now considered to be at between 100- and 110-bit security level.

As an alternative for the long-term security, at 128-bit security level we adopt a pairing-friendly curve from the from the Barreto-Lynn-Scott (BLS) family. BLS curves over a field $\mathbb{F}_p$ also have the form $E : y^2 = x^3 + b, b \neq 0$ and support an optimal Ate pairing construction [10]. These curves vary according to different embedding degrees. We select a BLS12 curve, which has embedding degree $k = 12$ and it is parameterized by an integer $u \in \mathbb{Z}$ with $p = \frac{1}{3}(u - 1)^2(u^4 - u^2 + 1) + u$ and $r = u^4 + u^2 + +1$, respectively. In contrast to BN curves, the group order of a BLS curve is not prime but divisible by the large prime parameter r. The pairing is then defined on the r-torsions points. Specifically, we select the BLS12-381 curve [21] $E : y^2 = x^3 + 4$ parameterized by the integer $u = -(2^{63} + 2^{62} + 2^{60} + 2^{57} + 2^{48} + 2^{16})$, which generates a 381-bit prime field number order. In this instantiation, elements from the groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ take 96, 192, 576 bytes, respectively.

At both security levels, variable-base and fixed-base scalar multiplications are implemented using the window NAF and single-table comb methods [71], respectively. Parameters for 100-bit security are enough for more constrained devices and short- medium-term security, while parameters for 128-bit security ensure long-term security, but still lack efficiency in resource constrained devices, as we present in our experiments in Section 7.

6.2.5 Demo

We develop a demo to showcase our AoT prototype. Our demo covers the entire life-cycle of a smart fridge, from pre-deployment to retirement. We demonstrate the functionalities in our architecture executing operations on multiple devices and transferring information using different communication technologies.

6.2.5.1 Scenario Overview

In our demo, we cover the life-cycle of a smart fridge. First, the device is pre-deployed at the factory, which creates it in the Cloud domain. Then, after a user purchases the smart fridge online, she receives a PIN from the vendor, that allows her to put the device in deployment mode. When the user receives the brand new fridge, she deploys it in her home domain. When the user receives the brand new fridge, she deploys it in her home domain. During the deployment process, the user defines the fridge's attributes and access policies to its operations such as defrost, temperature change, access to the internal camera, and retirement. In the user's home domain, there are already deployed a smart TV and a smart motion sensor. The smart TV can request images from other devices and present them on its screen, while the smart sensor can request an operation to another device when it sensor motion. When the users has no intention of using the fridge anymore, she retires the device.

6.2.5.2 Implementation

In our demo we consider the following entities: (i) a cloud server, (ii) an emulated manufacturing device, (iii) a home server, (iv) an emulated smart TV, (v) an emulated smart fridge, (vi) a QRCode reader for the emulated smart fridge, (vii) a smartphone for the administrator of the home domain, and (viii) a smart presence sensor. The cloud and home servers as well as the emulated smart home appliances are implemented in a Raspberry Pi3 with a 64 bits quad-core 1.2GHz processor, 1 GB of RAM and 8 GB storage. We use two Android smartphones LG Nexus5 with a 32 bits ARM quad-core 2.3 GHz processor, 2 GB of RAM and 16 GB storage. One of them plays the roles of manufacturing device and fridge's QRCode reader, while the other acts as the administrator device, which can be used to deploy new devices or to operate any other device in the home domain. Last, the smart sensor is implemented in an Arduino Due with a 32 bits ARM M3 84 MHz processor, 96 KB of RAM and 512 KB storage and plays the smart sensor role.

As mentioned in Section 6.2, all entities in our architecture share the same underlying cryptographic implementation, including the core of our session key establishment [29, 81] and access control mechanism [78], built on top of our extended version of RELIC cryptographic library [4] and accessed through native calls using the first set of parameters described in Section 6.2.4.4. The cloud and home servers as well as the emulated home appliances are based on LAMP and call the binaries from PHP. The Android applications run in Android's Dalvik virtual machine and use the JNI to call our low-level cross-compiled cryptographic functions.

The Arduino microcontroller, in turn, is programmed to call cross-compiled cryptographic functions. The devices exchange information, which is encoded serialized in binary format, using QRCodes during the initial deployment of the fridge and HTTP when changing user attributes from the administrator smartphone or accessing an emulated device operation.

6.2.5.3 Pre-deployment

In our demo, the pre-deployment (Stage 4.1) of the smart fridge if performed by connecting the manufacturing device to the fridge through USB. The smartphone detects the fridge, generate the fridge's cryptographic keys for the Cloud domain, load the keys onto the fridge, and inform the Cloud server about the new fridge.

6.2.5.4 Deployment

The user purchases a brand new smart fridge. The fridge's vendor sends the user a PIN, that will put the device in set up mode (e.g., in a confirmation e-mail after checking out on the vendor's website). Given the new device is bulky, we deploy it in the home domain using the administrator's smartphone as a bridge to the home server. After the user inputs the PIN to put the fridge in deployment mode (Stage 4.3, step 1), the fridge enables its QRCode reader to get the home server's public key, as shown in Figure 6.3.

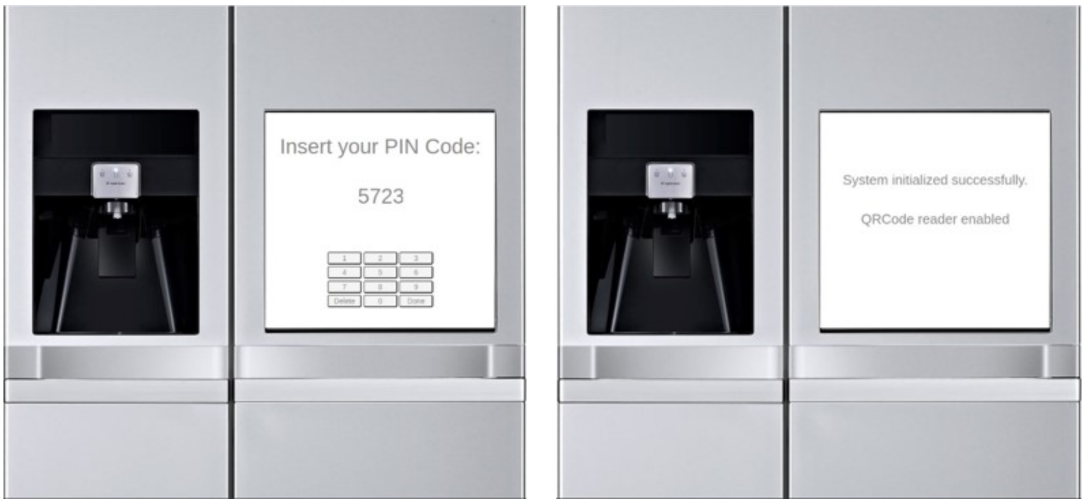


Figure 6.3: Fridge being put in deployment mode (left) and enabling its QRCode reader for home server public key presentation (right).

The user then accesses its Home Manager Android Application and selects the option to deploy a new device, she is instructed to show a QRCode that contains the Home server public key to the fridge's QRCode reader, as shown in Figure 6.4 (Stage 4.3, step 2).

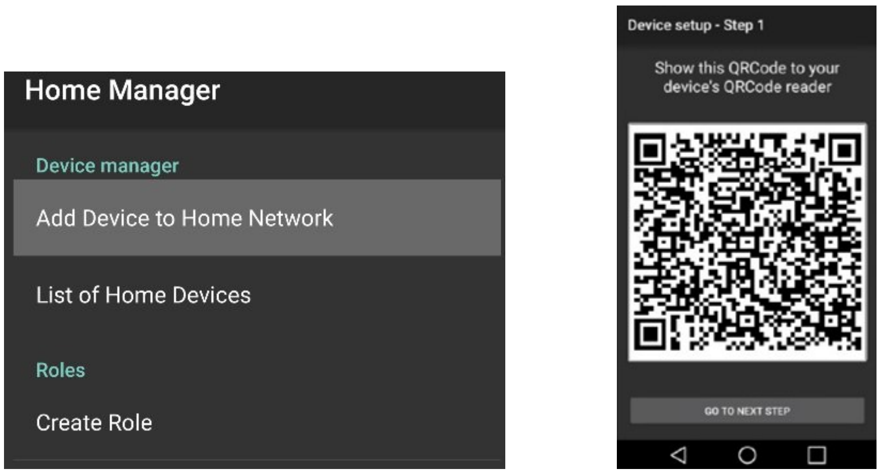
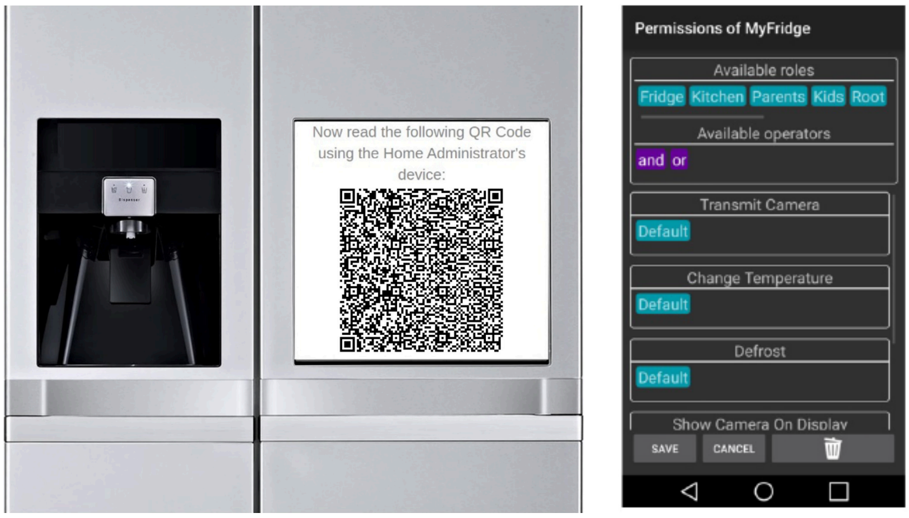


Figure 6.4: The user is instructed to show a QRCode to the device's reader.

Next, the fridge generates an ephemeral pairwise key, encrypts it using the just read home server public key, and shows the ciphertext back to administrator's smartphone as a QR-Code on its built-in display, as shown in Figure 6.5 (left). The administrator reads the fridge's QRCode using her smartphone's camera (Stage 4.3, step 4). Then, the administrator uses the Android application to configure the fridge, choosing its attributes and defining access control policies for each one of available operations (Stage 4.3, step 5), as shown in Figure 6.5 (right).

Upon conclusion, the application sends all the information (including the QRCode ciphertext) to the home server (Stage 4.3, step 7). At this point, the home server can decrypt the data sent by the administrator and, then, it has a pairwise key with the fridge. Hence, it can then generate the home domain keys for the fridge and securely send them to it (Stage 4.3, steps 8-13) and broadcast to the domain information about the new device (Stage 4.3, step 14). Once the fridge joins the home domain, it is able to communicate with all other devices as well as it is able to communicate with the Cloud domain to update its status with the Cloud server (Stage 4.3, step 15), e.g., after using its attributes to receive Internet access through the home's wireless router.


Now read the following QR Code using the Home Administrator's device:

Permissions of MyFridge

Available roles: Fridge Kitchen Parents Kids Root

Available operators: and or

Transmit Camera: Default

Change Temperature: Default

Defrost: Default

Show Camera On Display

Figure 6.5: The fridge displays a ciphertext to the user (left). Then the user sets the fridge's attributes access control policies (right).

6.2.5.5 Functioning

Now that the fridge can be accessed by other devices, the user tries to display the image of the internal camera of her fridge using the smart TV (Stage 4.4, step 4). The fridge then challenges the TV to prove it has the required attributes to access that operation, which is controlled by the policy *"parents OR kitchen"* (Stage 4.4, step 5). However, as the TV only has the *"screen"* attribute, it is not able to appropriate sign the fridge challenge and the access to the fridge's camera is denied (Stage 4.4, step 6), as shown in Figure 6.6 (left). The user then opens the Home Manager Android Application in her smartphone and changes the policy to access the internal camera of the fridge to *"parents OR kitchen OR screen"*. Now, on a new try to access the fridge internal camera, the TV satisfies the access control policy and the operation is executed, as shown in Figure 6.6 (right).

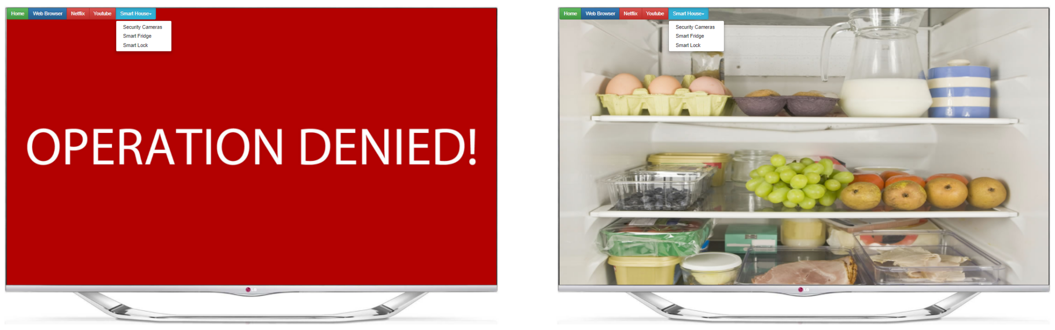


Figure 6.6: An operation requested by the smart TV to the smart fridge is denied (left) and then granted after assigning the appropriate attributes (right).

Last, the user configures her smart motion sensor to request the fridge to show in its built-in screen the content of its internal camera every time someone approaches the fridge's, besides, the user also gives the smart sensor the attribute *"motionsensor"* in her Android application. Now, every time the sensor detects someone near the fridge, it requests the fridge the to display in its built-in screen the image of its internal camera (Stage 4.4, step 4). The fridge then challenges the sensor to prove it has the required attributes to access that operation (Stage 4.4, step 5), which the user sets to be controlled by the policy *"motionsensor OR kitchen"*. As the user assigned such an attribute to the sensor, it is able to appropriate sign the challenge and the operation is executed (Stage 4.4, step 6), as shown in Figure 6.7.

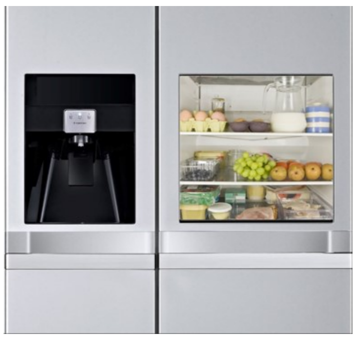
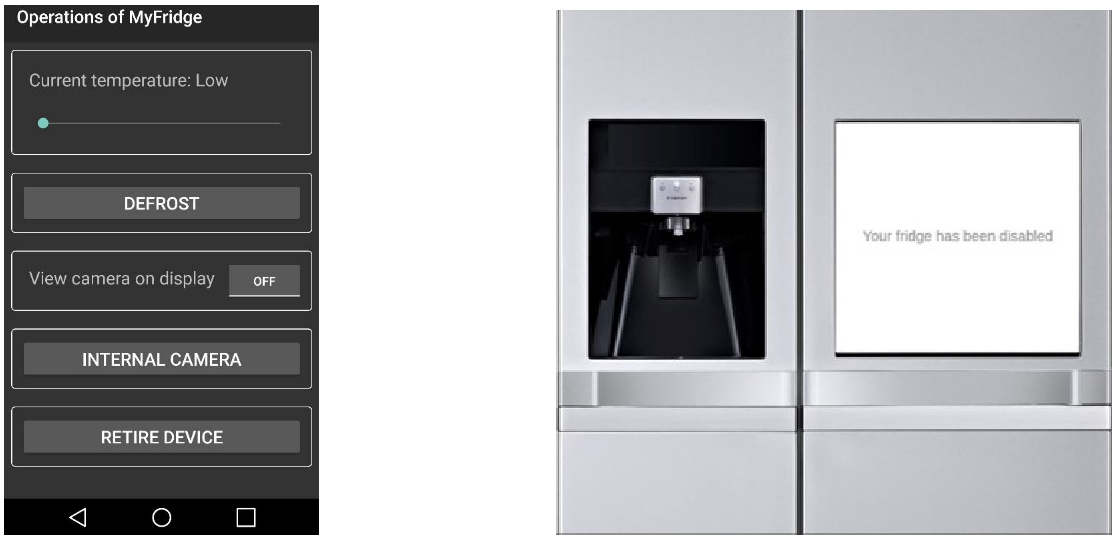


Figure 6.7: Smart motion sensor requests an operation which is successfully executed by the fridge.

6.2.5.6 Retirement

When the life of the fridge comes to an end, the user accesses the Smart Home Controller Android Application in her smartphone and chooses the retirement operation of the fridge (Stage 4.4, step 4). The fridge then challenges the smartphone to prove it has the required attributes to access that operation, which is controlled by the policy *root* (Stage 4.4, step 5). As the user is indeed the Home domain root user and her device has the *root* attribute, the fridge executes the operation (Stage 4.4, step 6) which, in this case, is to unbind itself from its owner on the Cloud domain (Stage 4.5, step 2), erase all of its cryptographic keys (Stage 4.5, step 3) and display a retirement message on its built-in screen (Stage 4.4, step 4), as shown in Figure 6.8 (right).



Current temperature: Low

View camera on display OFF

Your fridge has been disabled

Figure 6.8: The root user requests the smart fridge retirement using her smartphone (left) and the operation is executed (right).

Chapter 7

Evaluation

In this chapter, we present the evaluation of our AoT prototype at both 100- and 128-bit security level. We start with a brief analytical evaluation of computational and communication overhead Section 7.1 to then present our experimental results in Section 7.2. We show that our cryptographic instantiation of AoT supports deployment in a variety of devices, representing a wide range of IoT devices, including resource-constrained embedded devices.

7.1 Analytical Evaluation

In this section, we analyze the computational costs and communication overhead of both parameterizations of our implementation of AoT, described in Section 6.2. We quantify computational costs of AoT as a function of the number of the most expensive operations in its cryptographic primitives. In particular, we consider the number of pairings (denoted ê), elliptic curve scalar multiplications in group $\mathbb{G}_1$ (denoted $\mathrm{p}_{\mathbb{G}_1}$) and group $\mathbb{G}_2$ (denoted $\mathrm{p}_{\mathbb{G}_2}$), exponentiation in group $\mathbb{G}_T$ (denoted $\mathrm{e}_{\mathbb{G}_T}$), and elliptic curve scalar multiplication in $\mathbb{E}(\mathbb{F}_q)$[1] (denoted p). Other operations (symmetric primitives) are executed in negligible time. Table 7.1 summarizes the computational costs results. In the table, ℓ denotes the number of attributes and t the number of 'and' operators in the predicate plus one. ABS is the most expensive cryptographic primitive in AoT. The computational cost of ABS grows with the size of the span program matrix generated from the predicate. It is also the core of our access control mechanism (in the *Functioning* stage), consequently, it is the most frequently used primitive in AoT. In later sections, we focus our analysis on ABS.

We quantify communication overhead as the amount of bytes in a signature, the overhead caused by the encryption technique, or the overhead due to an agreement on parameters to establish a key. We compute communication overhead considering 16-byte nonces, 1-byte labels, and the size of elements in groups $\mathbb{G}_1$ and $\mathbb{G}_2$ described in Section 6.2. Table 7.2 sum-

[1]Recall from Section 2.5 that groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are implemented as subgroups with order r of the elliptic curve $\mathbb{E}(\mathbb{F}_{q^k})$ while group $\mathbb{G}_T$ is implemented using a multiplicative subgroup of the finite extension $\mathbb{F}_{q^k}$.

	Computational Overhead	
Primitive	Sender	Receiver
D.ABS	$2\ell t \mathrm{p}_{\mathbb{G}_1} + (3+2\ell)\mathrm{p}_{\mathbb{G}_2}$	$(t\ell+1)\mathrm{p}_{\mathbb{G}_1} + 2\mathrm{p}_{\mathbb{G}_2} + (\ell t + t + 2)\hat{\mathrm{e}}$
P.ABS	$2\ell t \mathrm{p}_{\mathbb{G}_1} + (3+2\ell)\mathrm{p}_{\mathbb{G}_2}$	$(2\ell t + t + 1)\mathrm{p}_{\mathbb{G}_1} + 1\mathrm{p}_{\mathbb{G}_2} + (\ell + 2)\hat{\mathrm{e}}$
IBS	$1\mathrm{p}$	$3\mathrm{p}$
IBE	$1\mathrm{p}_{\mathbb{G}_1} + 1\mathrm{e}_{\mathbb{G}_T} + 1\hat{\mathrm{e}}$	$1\hat{\mathrm{e}}$
MB	$1\mathrm{p}_{\mathbb{G}_1} + 2\hat{\mathrm{e}}$	$1\mathrm{p}_{\mathbb{G}_1} + 2\hat{\mathrm{e}}$

Table 7.1: Computational and communication overhead for AoT cryptographic primitives.

marizes the results.

	Communication Overhead (bytes)	
Primitive	100-bit sec. level	128-bit sec. level
D.ABS	$256 + 256\ell + 64t$	$384 + 384\ell + 96t$
P.ABS	$256 + 256\ell + 64t$	$384 + 384\ell + 96t$
IBS	128	160
IBE	96	128
MB	64	96

Table 7.2: Communication overhead for AoT cryptographic primitives considering 100-bit and 128-bit security levels.

Communication overhead for most primitives is constant. In ABS, in turn, communication overhead depends on the size of the span program matrix, which is around a few hundreds of KBs per signed message. We believe this overhead is still small enough to fit in one packet, or few small packets when running over low-power communication networks that employ small frames. Network transmission delay and energy costs induced by AoT are negligible compared to CPU processing time and energy costs [91].

7.2 Experiments

We evaluate both versions of our AoT prototype, at 100- and 128-bit security levels, on different platforms, varying computational resources, representing a wide range of IoT devices. We use a recently launched Android mobile phone, a Google Pixel 6, as a representative of smartphones that could be used in a smart environment supported by AoT. Other powerful entities in a smart environment are represented by a Raspberry Pi3, a low-cost programmable computer. We represent intermediate smart devices with Raspberry Pi1. Last, as our representative of microcontrollers that could be used on low-end appliances supporting AoT, we use an

Arduino Due. Table 7.3 presents the computation resources of each platform. In the next sections, we present the experimental results for the 100- and 128-bit security levels, respectively, starting with the resource-constrained device.

Resouce	**Google Pixel 6**	**Raspberry Pi3**	**Raspberry Pi1**	**Arduino Due**
CPU arch.	arm64-v8a	armv7-a	armv6	armv7-m
Word size	64 bits	64 bits	32 bits	32 bits
Clock	1.8 GHz	1.2 GHz	700 MHz	84 MHz
Cores	8	4	1	1
RAM	12 GB	1 GB	512 MB	96 KB
Storage	256 GB	8 GB	4 GB	512 KB
OS	Android 12	Linux raspberry 4.9.59-v7+	Linux raspberry 5.10.17+	None

Table 7.3: Summary of devices used in experimental evaluation.

7.2.1 100-bit Security Level Results

7.2.1.1 Resource-constrained Platform

We start our evaluation on our representative of resource-constrained devices, the Arduino Due. We evaluate three versions of our prototype at 100-bit security level. The first one is using RELIC without any optimization. In the second one, we implement the base field arithmetic backend for the BN254 curve using crafted assembly code for the ARM architecture, as mentioned in Section 6.2. In the third version of our prototype, we reach the limit of our numbers using the optimized version of RELIC and overclocking the Arduino Due microcontroller.

AoT prototype without optimization. Figure 7.1 shows run times for the cryptographic primitives used in AoT executed on our first prototype. We execute each algorithm to measure 100 run times and plot the quartiles as well as the 5th and 95th percentiles (almost indistinguishable on the Figure). We omit our results for symmetric primitives (AES and HMAC) as they run in less than 5 milliseconds. In the Figure, “Enc”, “Dec”, “Sign”, and “Ver” are abbreviations for encryption, decryption, signature generation, and signature verification, respectively. The abbreviations D.Ver and P.Ver refer to the deterministic (D.ABS) and probabilistic (P.ABS) versions of the ABS signature verification algorithm. In case of IBE, we consider a 32 bytes length message, the size of message that is used in AoT (Stage 4.3 – step 4). In IBS, we use messages with 1KB, covering all cases IBS is used in AoT (Stage 4.3 – steps 7 and 14, and Protocols 4.5 and 4.6, step 2). In ABS, we initially consider predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$, i.e., with two attributes and a single "and" operator. We observe that even without optimization if we consider the context where each primitive is executed in AoT, with the exception of ABS, the cryptographic primitives execute in reasonable time. For instance, IBE encryption is executed at device deployment (Stage 4.3 – step 4), a setup process executed only once. In this context, it might be acceptable to have an embedded device with 1.5s overhead to encrypt the ephemeral key it generates. Analogously, the MB scheme is used to establish session keys for protocols not frequently executed, more specifically for *KeyIssue*, *Binding*, and *Unbinding*, therefore, its 1.8s overhead might also be acceptable in this case. Even so, our optimized prototype significantly reduced these numbers. However, the ABS numbers without optimization can indeed be impractical for some IoT applications, given ABS is the primitive used in each access control on a device operation. In this version, our Due AoT prototype takes around 3.3s to generate the signature on a predicate of the form $\boldsymbol{A} \wedge \boldsymbol{B}$, and 6.5s and 3.8s to verify it using the deterministic and probabilistic algorithms, respectively.

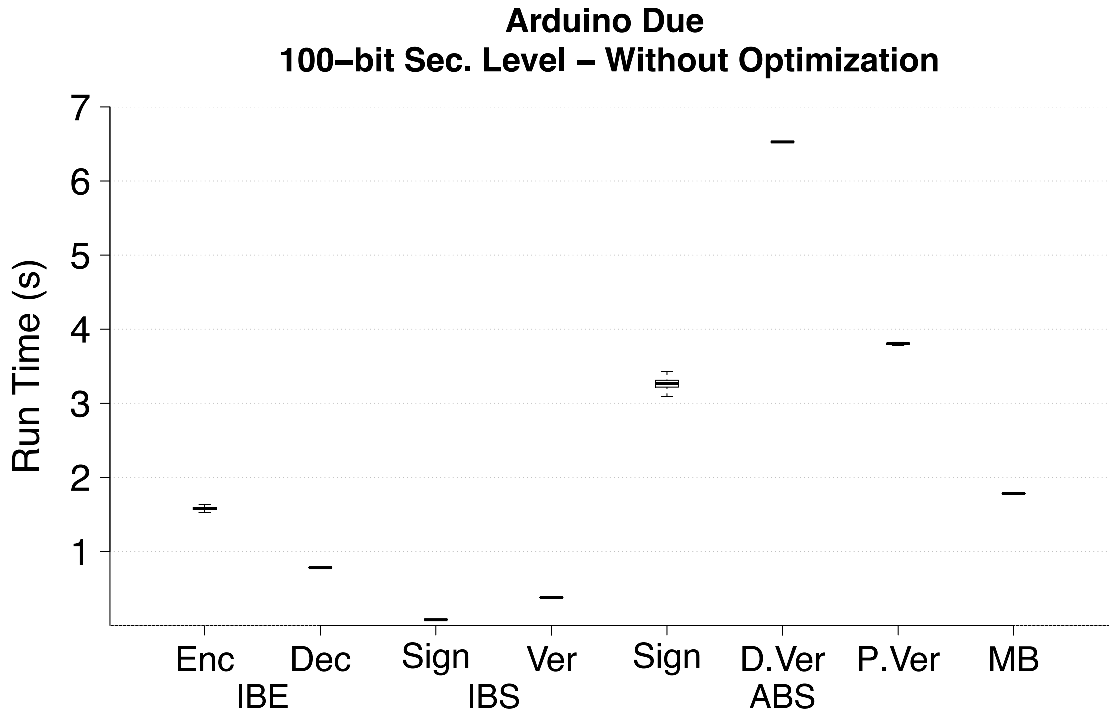


Figure 7.1: Run times for asymmetric cryptographic primitives on the Due at 100-bit security level without optimization. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$.

As discussed in Section 7.1, ABS efficiency is impacted by its most computationally expensive operations, the elliptic curve scalar multiplication in groups $\mathbb{G}_1$ and $\mathbb{G}_2$ and the bilinear pairing ($\hat{e}$). According to Table 7.1, the size of an ABS predicate, i.e., the number ℓ of attributes and the number t of "and" operators plus one, determine the number of expensive operations executed in the ABS signature and verification algorithms. We use those numbers to evaluate how this version of our AoT prototype would scale in a realistic IoT scenario. Table 7.4 shows run times in milliseconds of expensive operations in this first version of our prototype. Figure 7.2 shows analytically estimated run times for ABS signature generation (on the top), deterministic verification (on the middle), and probabilistic verification (on the bottom) considering different predicate structures, where we vary the number of attributes ($1 \leq \ell \leq 10$) and use only "and" operators. Curves in Figure 7.2 are plotted combining the experimental numbers from Table 7.4 and ABS costs from Table 7.1. In the x axis we represent the number of attributes ℓ in each predicate.

Arduino Due **100-bit Sec. Level - No Optimization** **Operation**	**Run Time (ms)**
Scalar point multiplication in $\mathbb{G}_1$	143.0 ± 2.0
Scalar point multiplication in $\mathbb{G}_2$	339.0 ± 5.0
Bilinear pairing ($\hat{e}$)	790.0 ± 1.0

Table 7.4: Expensive operations run times (ms) on our prototype without any optimization on the Arduino Due.

We observe that for complex predicates with ten attributes and nine "and" operators, our AoT prototype would take around 36s to generate a signature, almost two minutes to verify it using the deterministic algorithm and around 40s to verify it using the probabilistic algorithm. We complement the curves in Figure 7.2 with our experimental averaged over 30 runs (coefficients of variation are below 5%, not shown) considering the same predicate structures. By comparing the analytical and experimental results, we observe that the analytical run times significantly overestimate our implementation run times. There are two main reasons for this. First, the signature verification algorithms need the product of pairings, however, we optimized our code (even in the version without assembly code) to compute multi-pairings simultaneously (Section 6.2). Second, and this applies to signature generation as well, the analytical costs consider that the $\ell \times t$ coefficient matrix in a predicate's MSP does not contain any zeros. In practice, we find coefficient matrices are sparse, which significantly reduces the amount of executed operations. For example, the analytical run time for generating a signature on a predicate of the form $A \wedge B \wedge C \wedge D \wedge E$ is around 11.5s. Our implementation, in turn, takes 7.5s (Figure 7.2 on the bottom). To verify such a signature the analytical run times are around 30s and 14s using the deterministic and probabilistic algorithms, respectively, while our implementation takes around 13.5s and 7s.

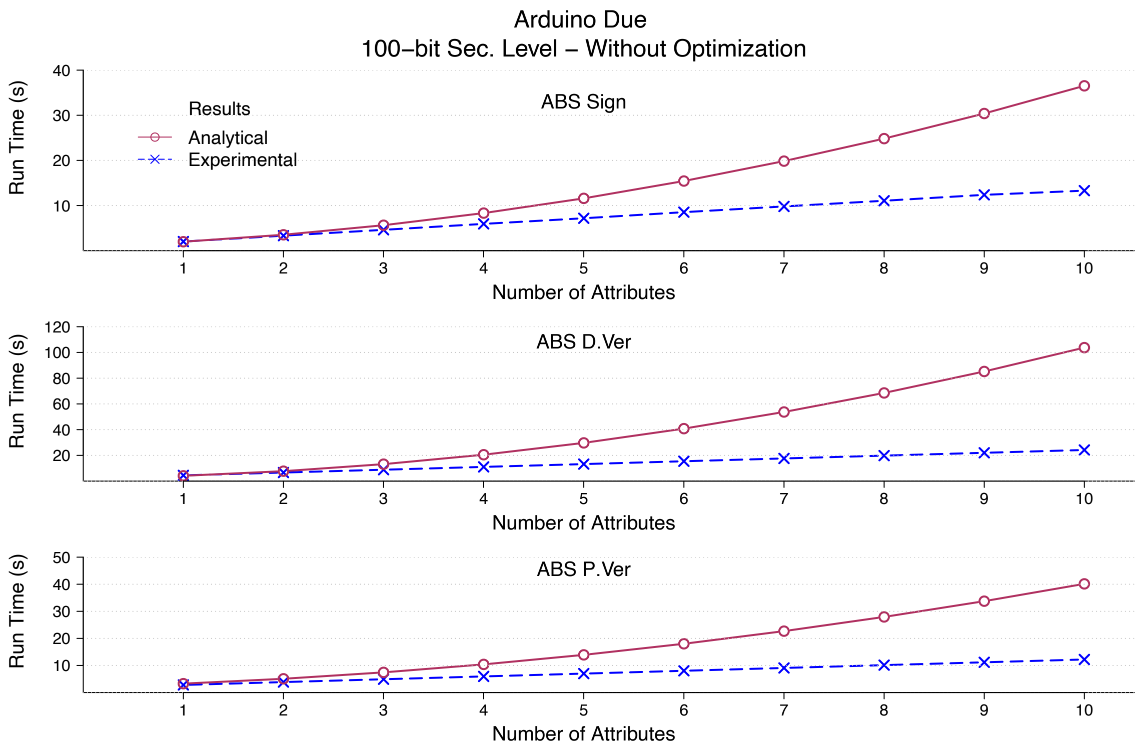


Figure 7.2: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our prototype of AoT without optimization on the Arduino Due. The curves on the Figure are plotted combining Tables 7.1 and 7.4.

We also evaluate the RAM usage of ABS algorithms considering the same variations on the predicate structures we just presented. Figure 7.3 shows our experimental results. We measure maximum memory utilization as the maximum stack size reached by each ABS algorithm during its execution, once RELIC only allocates memory on the stack. In terms of memory consumption, ABS is well-suited for resource-constrained devices as it requires at most 19KB of RAM memory for a predicate with ten attributes and nine "and" operators. The memory usage profile is the same for the three versions of our AoT prototype at 100-bit security level, once our optimization does not target memory consumption but execution time.

Finally, we also evaluate storage requirements for AoT on the Due. Our extended RELIC library plus our AoT implementation takes 146 KB of storage, which fits on the Due while leaving significant storage space for applications. The storage requirement also is not impacted by our optimization as well, all of our three AoT prototypes on the Due have less than 150 KB.

AoT prototype with assembly code optimization. Now we show the results of our optimized version of our AoT prototype. Every result is generated using the same test cases from the previous version of the prototype, i.e., messages length, number of runs, predicate structures, etc. Figure 7.4 shows the run times for the asymmetric cryptographic primitives used in this optimized version of our AoT prototype.

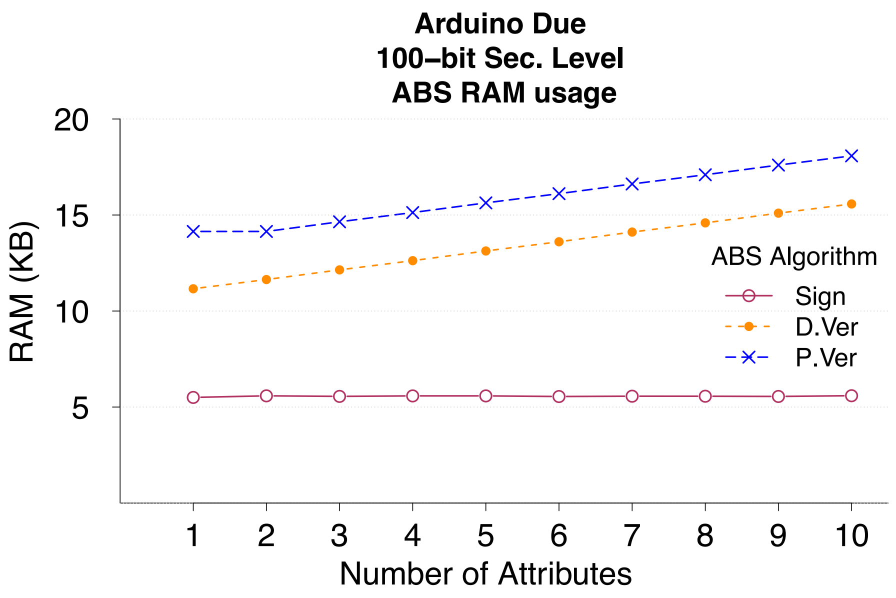


Figure 7.3: Experimental ABS algorithms' RAM requirements at 100-bit security level on the Arduino Due for varying the number of attributes in the predicate and using only "and" operators.

We observe that the optimized prototype considerably improves the performance of all cryptographic primitives used in AoT. IBS signature run times decrease from 76ms to 45ms and from 376ms to 245ms for signature generation and verification, respectively. IBE encryption and decryption run times are reduced from 1.5s to 721ms and from 779ms to 348ms, respectively. The session key establishment using the MB scheme, in turn, is reduced from 1.8s to 840ms. ABS numbers, in turn, become reasonable for a wide range of applications supported by resource-constrained devices in the context of smart environments where a 100-bit security level is enough. With this version of our prototype, an ABS signature on a predicate on the form $A \wedge B$ is reduced from 3.3s to 1.6s compared to the version without optimization. The run time of signature verification algorithms, in turn, are reduced from 6.5s to 3s in the deterministic version and from 3.8 to 1.9s in the probabilistic one, respectively.

We also evaluate how the optimized version of our AoT prototype would scale in a realistic IoT scenario. Table 7.5 shows run times of ABS algorithms' most expensive operations on this optimized version or our prototype. Figure 7.5, in turn, shows the analytically estimated run times combined with our experimental results for the same variation on the number of attributes ($1 \leq \ell \leq 10$) using only "and" operators on the predicate structures we presented before. Analytically estimated run times in Figure 7.2 are plotted combining the experimental numbers from Table 7.5 and ABS costs from Table 7.1. In this case, comparing the optimized prototype with its version without optimization, the experimental run time for generating a signature on a predicate of the form $A \wedge B \wedge C \wedge D \wedge E$ is reduced from 7.5s to 3.6s, while to verify such a signature using the deterministic and probabilistic versions of verification algorithms, the run times are reduced from 13.5s to 6.1s and from 7s to 3.8s, respectively. Last, in this version

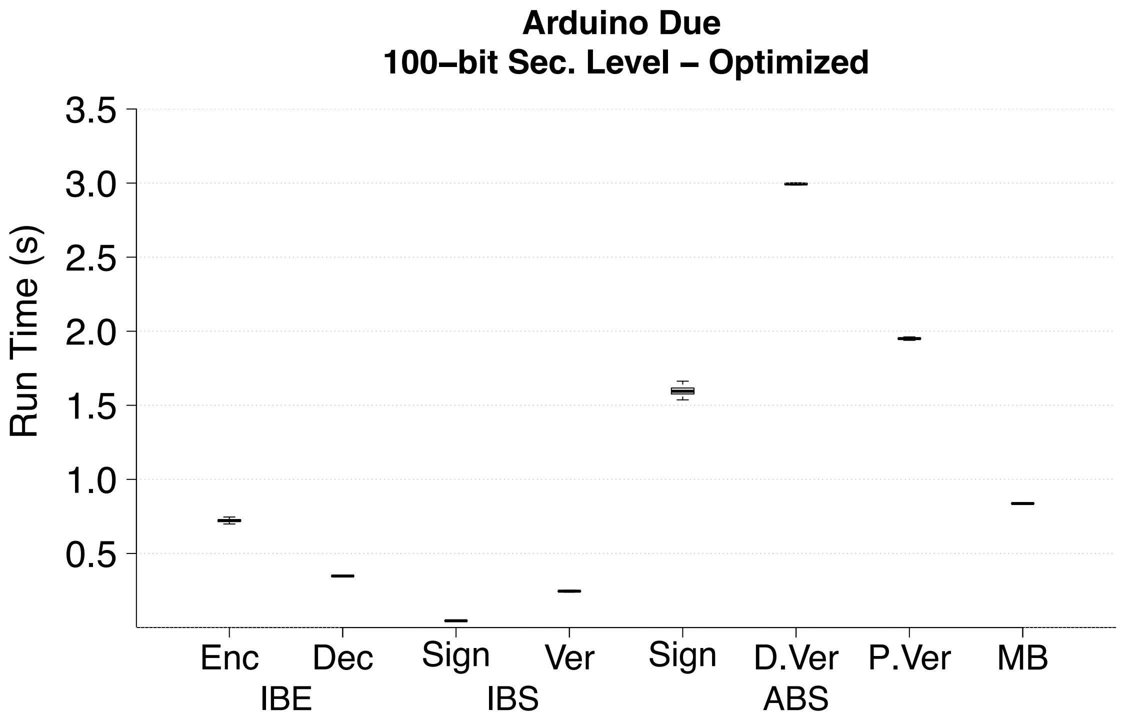


Figure 7.4: Run times for asymmetric cryptographic primitives on the Ardunio Due at 100-bit security level with optimized code. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$.

of our prototype, a complex predicate with ten attributes and nine "and" operators is signed in 7s, while such a signature can be verified in 11s and 7s using the deterministic and probabilistic versions of ABS algorithms, respectively.

Arduino Due
100-bit Sec. Level - Optimized

Operation	**Run Time (ms)**
Scalar point multiplication in $\mathbb{G}_1$	92 ± 2
Scalar point multiplication in $\mathbb{G}_2$	140 ± 6
Bilinear pairing ($\hat{e}$)	353 ± 1

Table 7.5: Expensive operations run times (ms) on our optimized prototype on the Arduino Due.

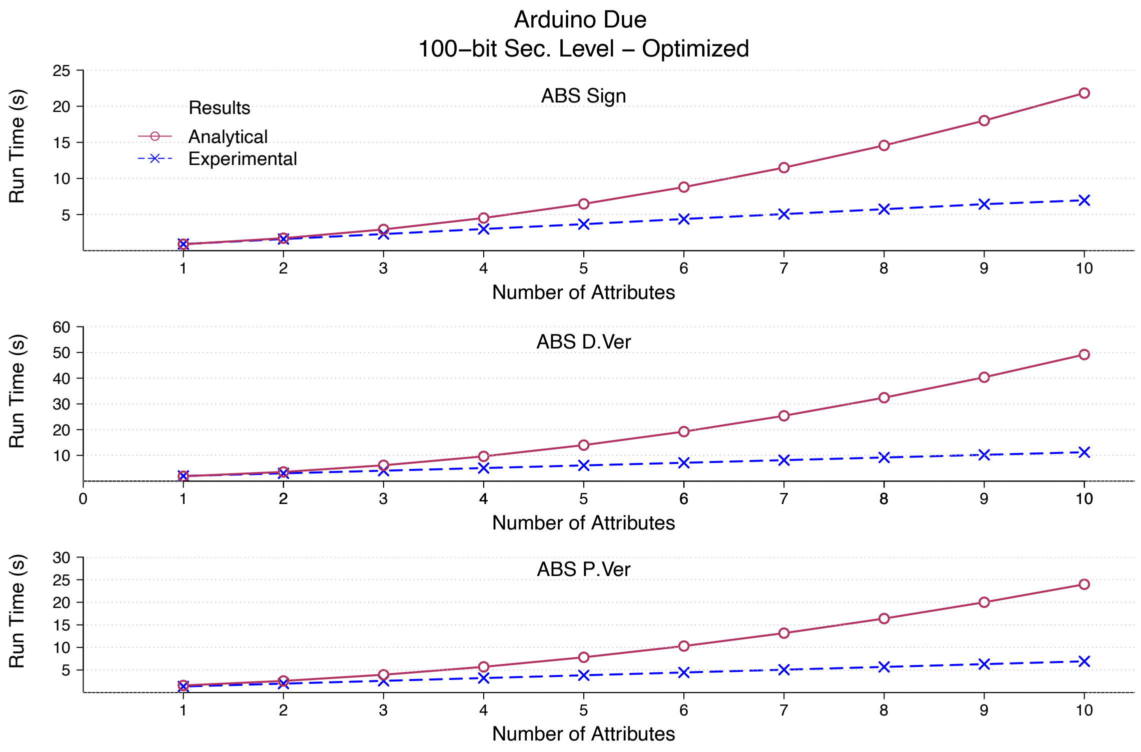


Figure 7.5: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our optimized prototype of AoT on the Arduino Due. The curves on the Figure are plotted combining Tables 7.1 and 7.5.

AoT prototype with assembly code optimization and overclocking the device microcontroller. Trying to improve even more the ABS run times, we show the results we obtain using our optimized version of AoT prototype executed on the Arduino Due with a microcontroller in an overclocking state. The maximum frequency we manage to stabilize the Due microcontroller in a overclocking state is 114 MHz. Considering the microcontroller's original 86 MHz clock frequency, the hypothetical best speedup we can reach with such an overclock rate is 1.36. Figure 7.6 shows the run times for the asymmetric cryptographic primitives used in AoT for an optimized prototype executed on a microcontroller in overclocking state. Focusing specifically on the ABS numbers, this strategy allows us to reduce the ABS signature generation run time from 1.6s to 1.2s, which represents a 1.33 speedup. The ABS signature verification algorithms, in turn, are reduced from 3s to 2.4s in the deterministic version and from 1.9s to 1.5s in the probabilistic one. These reductions represent 1.25 and 1.27 speedups, respectively. As expected, the improvement in the run times is not directly proportional to the clock frequency increase, even so, the results get close to the hypothetical best improvement. With this improvement on ABS run times, the range of applications supported resource-constrained devices in the context of smart environments is amplified. However, in any scenario, this kind of strategy must be applied carefully because it carries several drawbacks such as increased device power consumption, device overheating, and reduced lifespan of hardware components of the device.

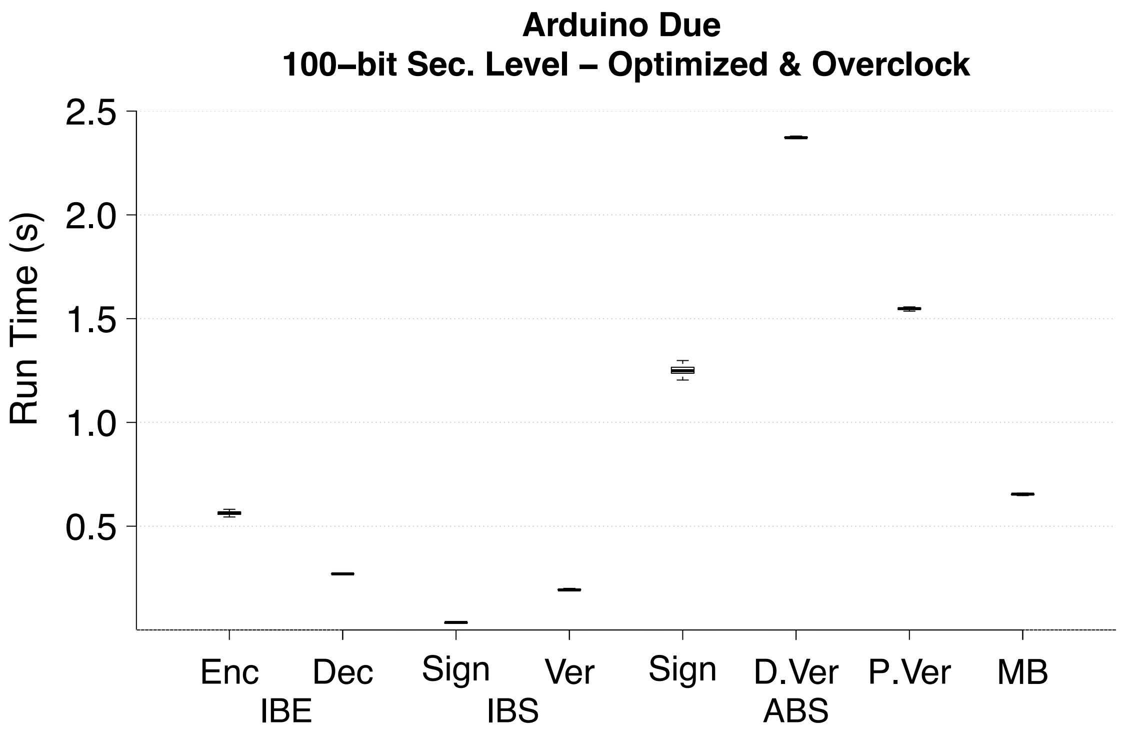


Figure 7.6: Run times for asymmetric cryptographic primitives at 100-bit security level with optimized code and overclocking the Due microcontroller. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$.

Table 7.6 shows run times (in ms) of ABS most expensive operations on this optimized and overclock version or our prototype. Figure 7.7, in turn, shows the analytically estimated run times combined with our experimental results for the same variation on the number of attributes ($1 \leq \ell \leq 10$) and only "and" operators on the predicate structures we presented before. Analytically estimated run times in Figure 7.7 are plotted combining the experimental numbers from Table 7.6 and ABS costs from Table 7.1. In this case, comparing the overclock version with the optimized one, the experimental run times for generating a signature using a predicate of the form $A \wedge B \wedge C \wedge D \wedge E$ is reduced from 3.6s to 2.9s, while to verify such a signature using the deterministic and probabilistic versions of verification algorithms, respectively, the numbers are reduced from 6.1s to 4.7s and from 3.8s to 2.9s. A complex predicate with ten attributes and nine "and" operators is singed in 5.4s and verified in 8.7s and 5.4s using the deterministic and probabilistic versions of ABS algorithms, respectively.

Arduino Due 100-bit Sec. Level - Optimized & Overclock Operation	**Run Time (ms)**
Scalar point multiplication in $\mathbb{G}_1$	69 ± 2
Scalar point multiplication in $\mathbb{G}_2$	107 ± 5
Bilinear pairing ($\hat{e}$)	266 ± 2

Table 7.6: Expensive operations run times (ms) on our optimized prototype on the Arduino Due with an microcontroller in overclocking state.

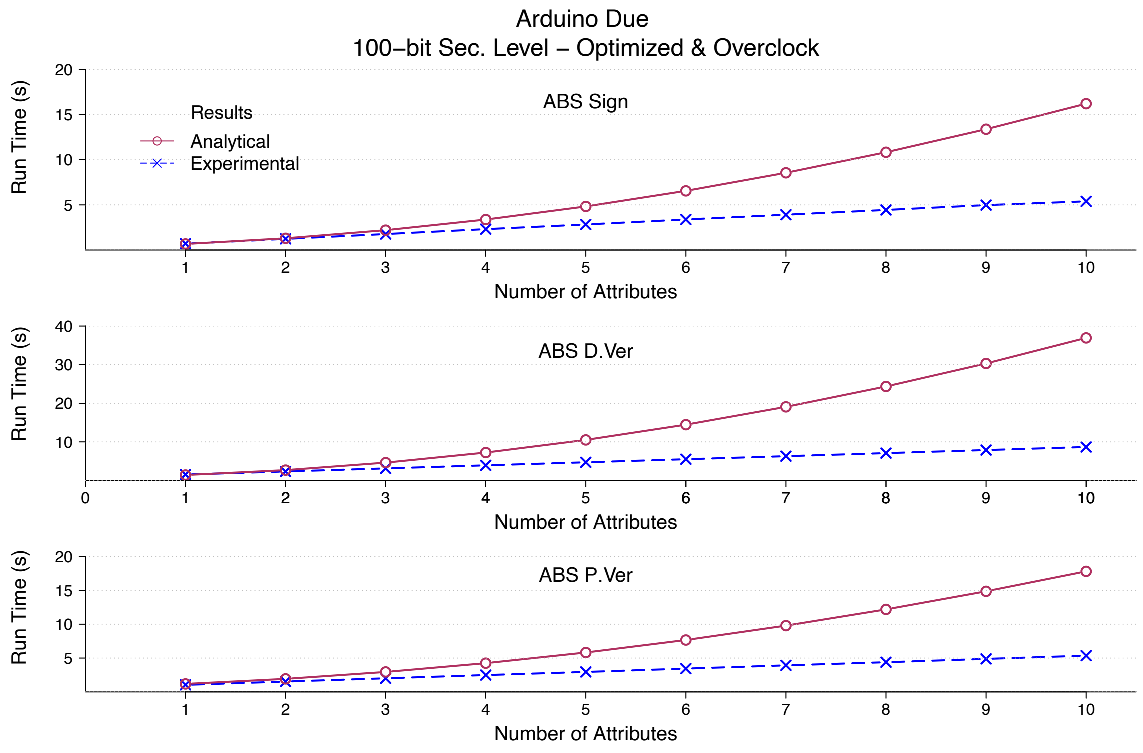


Figure 7.7: Analytical and experimental run times for ABS algorithms varying predicate structures in our optimized prototype of AoT on the Due with microcontroller in overclocking state. The curves on the Figure are plotted combining Tables 7.1 and 7.5.

Strategies to improve ABS performance. Our experiments show that at a 100-bit security level a wide range of applications that depends on resource-constrained devices can indeed be supported by our implementation of AoT. However, it is important to understand what can be done if more performance is needed. To do so, we perform further analysis on our AoT implementation. First, we note the Arduino platform does not provide a proper tool to analyze in detail the execution of a program. Hence we use as an alternative one of our other evaluation platforms, the Raspberry Pi3, which has the CPU architecture with the same instruction set as the Arduino Due. This means our optimized assembly code can be executed on both platforms, which, in turn, allows us to compile and execute the ABS algorithms without optimization and using our optimized code on the Raspberry Pi3 using a profiling software which shows us in detail the speedup obtained with our optimizations. Table 7.7 presents the profiling results of

ABS algorithms considering the the arithmetic backend of curve BN254 base field implemented without optimization and optimized using ARM assembly instructions. In the Table, we show how much the most time consuming arithmetic functions (multiplication and modular reduction on the base field) contribute to the overall execution of ABS algorithms. These numbers are generated using the software *Gprof* [45]. We combine these percentages of contributions with the run times obtained in our experimental evaluation on the Due to estimate the individual run time of such arithmetic functions in the ABS algorithms' execution. Last, we use the estimated run times to estimate the individual speedup obtained with the optimized code.

ABS Algorithm	Function	Without Optimization		Optimized		Speedup
		% Time	Run Time (s)	% Time	Run Time (s)	
Signature Generation	Multiplication	33.65%	1.11	23.81%	0.38	2.91
	Modular Reduction	32.83%	1.08	20.48%	0.33	3.31
	Others	33.52%	1.11	55.71%	0.89	1.24
Deterministic Signature Verification	Multiplication	33.95%	2.21	27.05%	0.81	2.72
	Modular Reduction	25.28%	1.64	16.49%	0.49	3.32
	Others	40.77%	2.65	56.46%	1.69	1.56
Probabilistic Signature Verification	Multiplication	32.83%	1.25	21.00%	0.42	2.97
	Modular Reduction	25.10%	0.95	18.51%	0.37	2.58
	Others	42.07%	1.60	60.49%	1.21	1.32

Table 7.7: Profiling results of ABS algorithms at 100-bit security level using *Gprof* [45] on Raspberry Pi3 combined with experimental run times from the Arduino Due.

We observe that applying Amdahl's law [3] in the numbers from Table 7.7, i.e., if we could hypothetically implement field multiplication and modular reduction functions in negligible time, only the ABS signature generation would perform below 1 second, while the probabilistic and deterministic signature verification would take 1.07s and 1.64s, respectively. This aspect show us that other approaches to improve the performance of an ABS construction such as the one we chose to be part of our prototype must be carefully analyzed. For instance, the usage of an off-the-shelf co-processor that provides only the field multiplication and modular reduction would not be enough to have run times below 1 second. On the other hand, if such a component handles not only these operations but all field arithmetic backend might be a good choice. If other strategies such as the development of dedicated hardware are under consideration, in turn, several other aspects, such as the level of speedup suitable for the application, the power consumption requirements, and prototyping and production costs, must be taken into account. In this case, Application Specific Integrated Circuit (ASIC) provides the best cases for speedup and energy efficiency in a context such as IoT. However, the prototyping and production costs might be too high. As a plausible alternative there are the Field Programmable Gate Array (FPGAs), which offer a flexible environment for prototyping where good speedups for an IoT application can be reached at the same time it has lower costs compared to ASIC.

7.2.1.2 Intermediate Platform

Now we present the evaluation of our prototype on a representative of intermediate platforms that could be part of a smart environment supported by AoT, the Raspberry Pi1. As we use the same underlying implementation of AoT, the memory and storage requirements obtained for the Due prototype show that these numbers are negligible compared to the amount of resources available on our other platforms, including the Raspberry Pi, therefore we focus on evaluating execution time.

First of all, we notice that even being an ARM-based platform, during the development of the optimized code for the ARM architecture, we end up using instructions specifically from the armv7's instruction set which are not available on the instruction set of the Raspberry Pi1 CPU (armv6). This means that our optimized code does not execute directly on the Raspberry Pi1 without proper modification. However, as we will present, the efficiency of AoT prototype without any optimization on the Raspberry Pi1 is enough for most IoT applications, therefore we do not put effort into adapting the original implementation. Every result we present is generated using the same test cases used in the evaluation of AoT on the resource-constrained platform. Figure 7.8 shows the run times for the asymmetric cryptographic primitives used in AoT on the Raspberry Pi1.

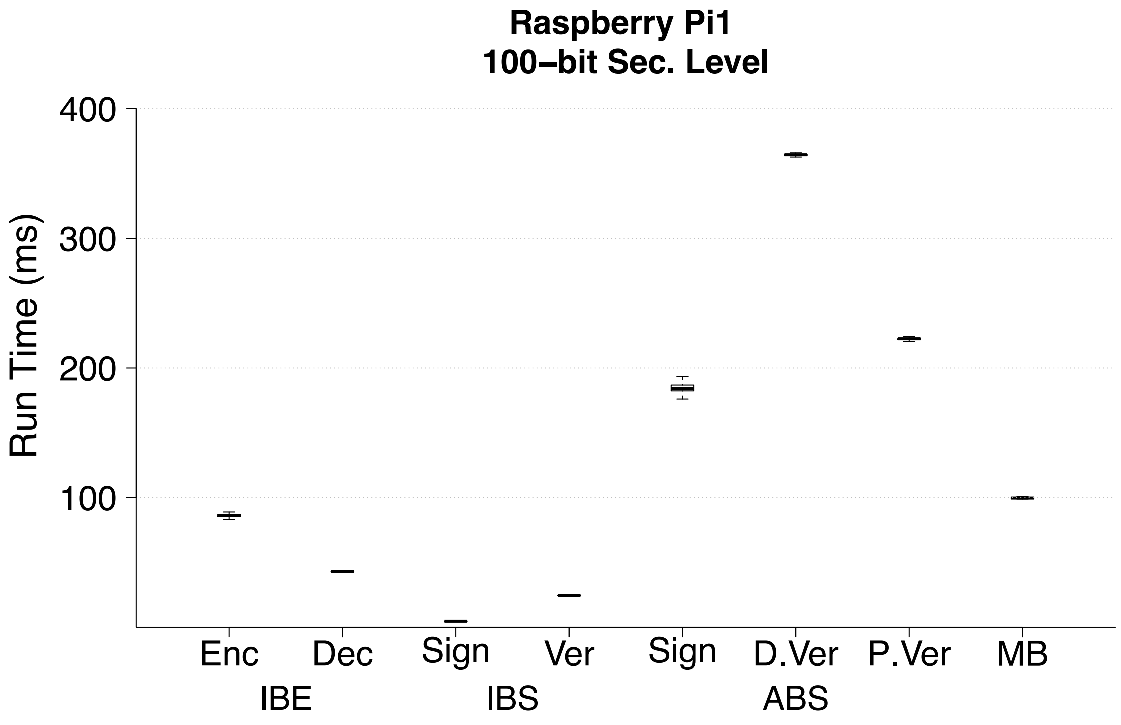


Figure 7.8: Run times for asymmetric cryptographic primitives on the Raspberry Pi1 at 100-bit security level. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $A \wedge B$.

We observe that even without any optimization, all cryptographic primitives used in AoT execute in reasonable time on the Raspberry Pi1. For instance, IBE encryption and decryption, IBS signature generation and verification, and the MB scheme are executed under 100ms. The ABS run times, in spite of being higher than 100ms, are at most 370ms, which makes it suitable for most IoT applications. Specifically, an ABS signature on a predicate of the form $\boldsymbol{A} \wedge \boldsymbol{B}$ takes around 190ms, a probabilistic signature verification takes around 230ms and the most timing consuming deterministic signature verification takes around 370ms.

We also evaluate how our AoT prototype on an intermediate platform would scale in a realistic IoT scenario. Table 7.4 shows run times of ABS algorithms' most expensive operations on our prototype for the Raspberry PI1. Figure 7.9, in turn, shows analytically estimated run times for ABS signature generation (on the top), deterministic verification (on the middle), and probabilistic verification (on the bottom) considering different predicate structures, where we vary the number of attributes ($1 \leq \ell \leq 10$) and use only "and" operator. Curves in Figure 7.9 are plotted combining the experimental numbers from Table 7.8 and ABS costs from Table 7.1. In the x axis we represent the number ℓ of attributes in the predicate.

Raspberry Pi1 **100-bit Sec. Level** **Operation**	**Run Time (ms)**
Scalar point multiplication in $\mathbb{G}_1$	9.1 ± 0.3
Scalar point multiplication in $\mathbb{G}_2$	18.0 ± 2.0
Bilinear pairing ($\hat{e}$)	43.0 ± 0.7

Table 7.8: Expensive operations run times (ms) on the AoT prototype for the Raspberry Pi1.

The analytical curves from Figure 7.9 show that for complex predicates with ten attributes and nine "and" operators, AoT prototype on the Raspberry Pi1 would take around 2.5s to generate a signature, 5.8s to verify it using the deterministic algorithm and around 2.5s to verify it using the probabilistic algorithm. As we did in the evaluation on resource-constrained platform, we complement the Figure 7.9 curves with our experimental averaged over 30 runs (coefficients of variation are below 5%, not shown) considering the same predicate structures. As expected, the experimental results are considerably better. The prototype on the Raspberry Pi1 actually takes around 800ms to sign such a complex predicate, 1.3s to verify it using the deterministic algorithm and around 800ms to verify it using the probabilistic algorithm. Therefore, our AoT implementation is indeed well-suited for intermediate platforms such as the Raspberry Pi1.

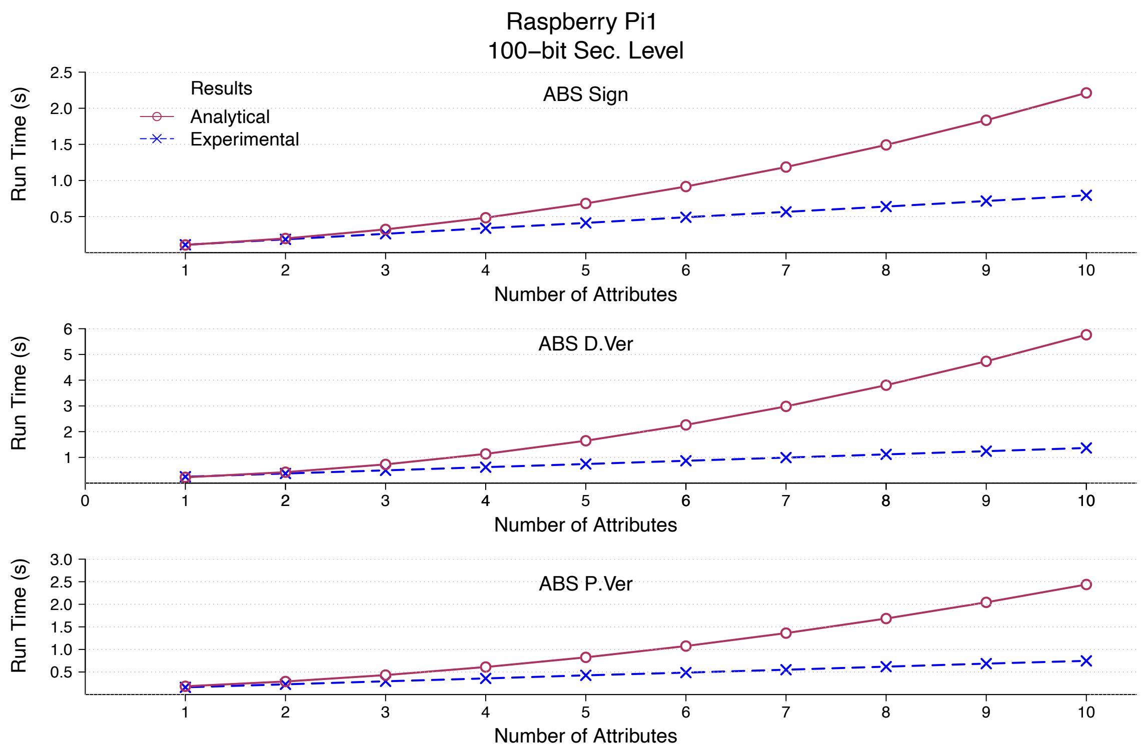


Figure 7.9: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our prototype of AoT on the Raspberry Pi1. The curves on the Figure are plotted combining Tables 7.1 and 7.8.

7.2.1.3 Powerful Platforms

Now we present the evaluation of our AoT prototype on the representatives of powerful platforms, the Raspberry Pi3 and the Google Pixel 6. Theoretically, both devices have CPUs (armv8 and armv7, respectively) compatible with our prototype optimized for the ARM platform with assembly code based on the armv7 instruction set. However, we only use our optimized prototype on the Raspberry Pi3 due to the many problems we face during the library building using the Android native development kit tool chain for the smartphone. Nevertheless, AoT imposes negligible overhead on the Google Pixel 6, even without any optimization. Every result we present is generated using the same test cases used in the evaluation of AoT on other platforms.

Figure 7.10 shows the run times of asymmetric cryptographic primitives used in AoT on both powerful platforms (Raspberry Pi3 results on the left and Google Pixel 6 results on the right). These devices have processors significantly more powerful than the Arduino Due and the Raspberry Pi1, which results in significantly higher performance. All cryptographic primitives used in AoT execute in less than 100ms on the Raspberry Pi3 and in less than 25ms on the Google Pixel 6.

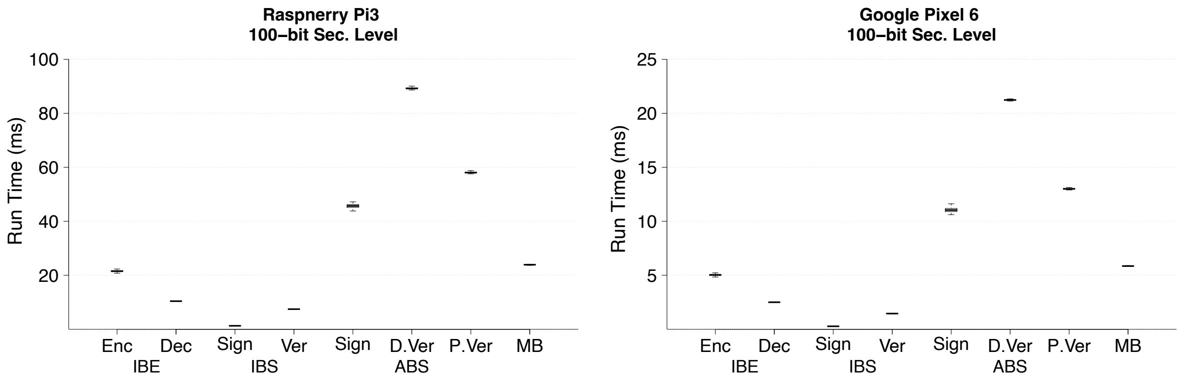


Figure 7.10: Run times for asymmetric cryptographic primitives in AoT on the Raspberry Pi3 (left) and Google Pixel 6 (right) at 100-bit security level. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$.

We also evaluate how our AoT prototype on the powerful platforms would scale in realistic IoT scenarios. Table 7.9 shows run times of ABS algorithms' most expensive operations on our prototype for the Raspberry Pi3 and Google Pixel 6. Figures 7.11 and 7.12 show analytically estimated run times for ABS signature generation (on the top), deterministic verification (on the middle), and probabilistic verification (on the bottom) on the Raspberry Pi3 and Google Pixel 6, respectively, considering the different predicate structures varying the number of attributes ($1 \leq \ell \leq 10$) and use only "and" operator. Curves in the Figures are plotted combining the experimental numbers from Table 7.9 and ABS costs from Table 7.1. In the x axis we represent the number ℓ of attributes in the predicate.

	100-bit Sec. Level	
	Raspberry Pi3	**Google Pixel 6**
Operation	**Run Time (ms)**	**Run Time (ms)**
Scalar point multiplication in $\mathbb{G}_1$	2.73 ± 0.05	0.54 ± 0.01
Scalar point multiplication in $\mathbb{G}_2$	3.70 ± 0.20	1.05 ± 0.05
Bilinear pairing ($\hat{e}$)	10.40 ± 0.20	2.49 ± 0.01

Table 7.9: Expensive operations run times (ms) on the AoT prototype at 100-bit security level for the Raspberry Pi3 and the Google Pixel 6.

The analytical curves in Figures 7.11 and 7.12 show that for a complex predicate with ten attributes and nine "and" operators our AoT prototype on the Raspberry Pi3 and on the Google Pixel 6 would take, respectively, around 640ms and 140ms to generate a signature, 1.5s and 340ms to verify it using the deterministic algorithm and around 710ms and 150ms to verify it using the probabilistic algorithm. As we did in the evaluation on other platforms, we complement the curves on the Figures with our experimental averaged over 30 runs (coefficients of variation are below 5%, not shown) considering the same predicate structures. As expected, the experimental results are even better. The AoT prototype on Raspberry Pi3 actually takes around 200ms to sign such a complex predicate, 340ms to verify it using the deterministic algorithm and around 210ms to verify it using the probabilistic algorithm. The Google Pix 6, in turn, actually takes 48ms to sign this predicate, 80ms to verify it using the deterministic algorithm and around 45ms to verify it using the probabilistic algorithm. Therefore, experimental results show that AoT imposes negligible overhead on powerful devices.

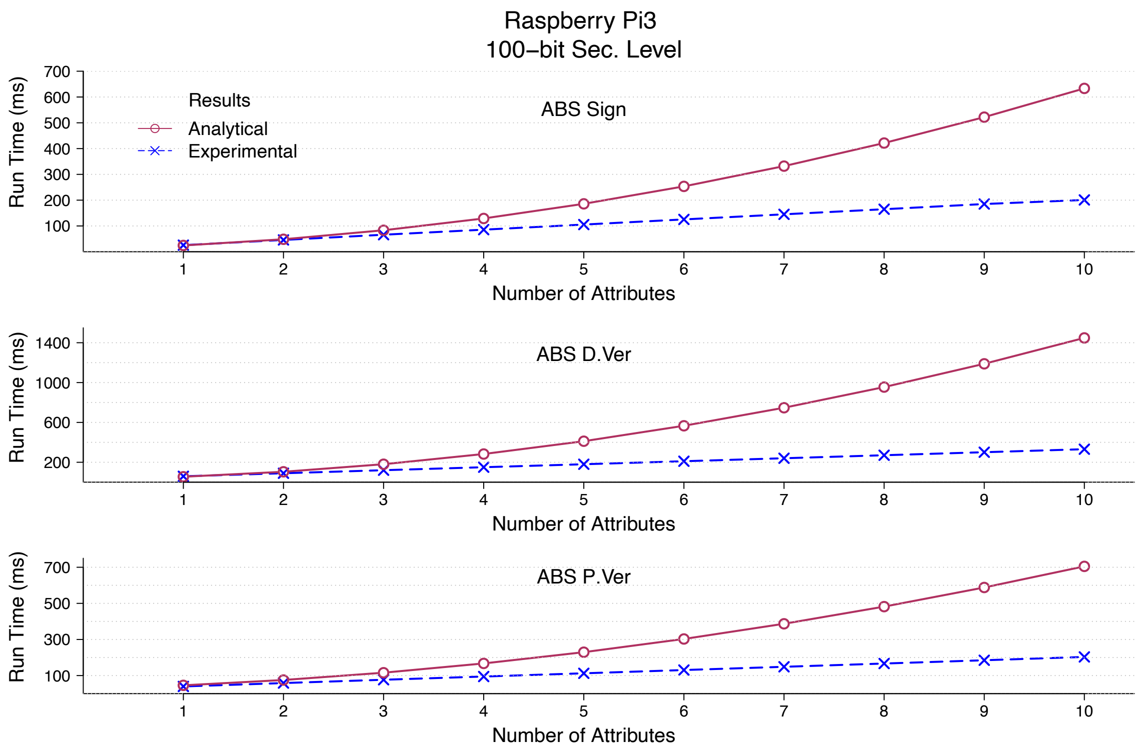


Figure 7.11: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our prototype of AoT on the Raspberry Pi3. The curves on the Figure are plotted combining Tables 7.1 and 7.9.

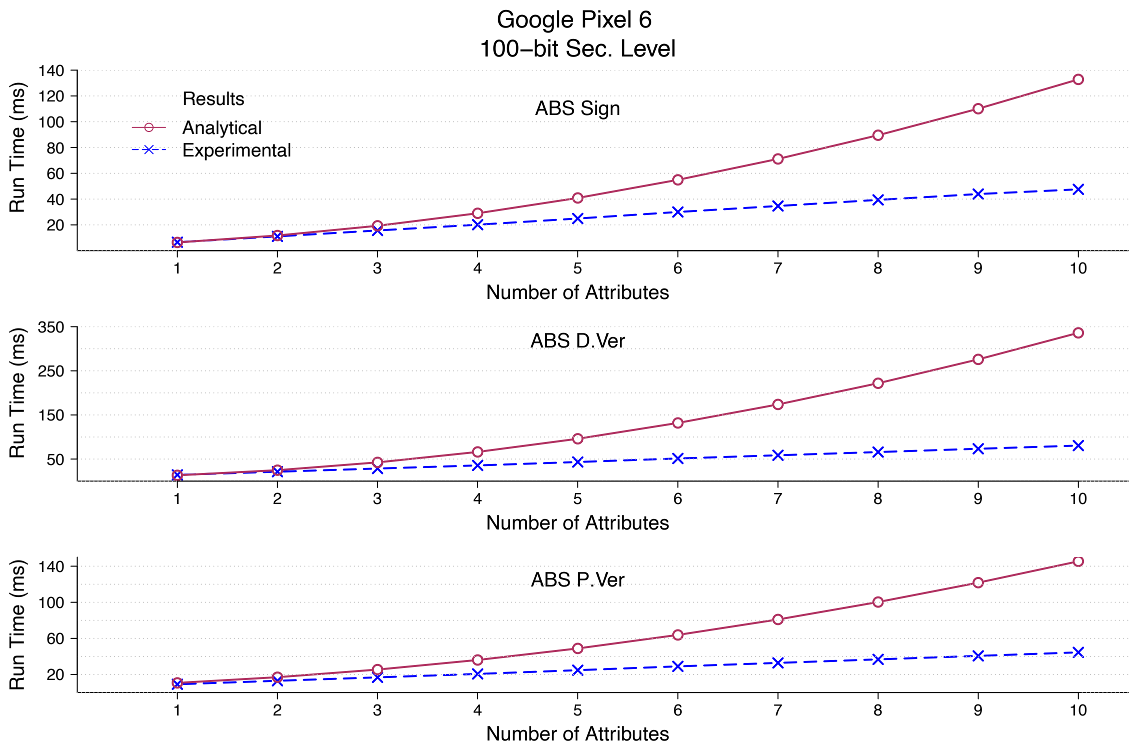


Figure 7.12: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our prototype of AoT on the Google Pixel 6. The curves on the Figure are plotted combining Tables 7.1 and 7.9.

7.2.2 128-bit Security Level Results

7.2.2.1 Resource-constrained Platform

We start the evaluation of our AoT prototype at 128-bit security level on the Arduino Due. Differently from the 100-bit security level, there is no version of RELIC with the base field arithmetic of curve BLS12-381 implemented in assembly code for the ARM architecture. Therefore, we generate the experimental results using an evaluation prototype that does not include any optimization. Then, we estimate the run times of a hypothetical optimized prototype using the speedup we obtained on the 100-bit security level experiments. Every result we present is generated using the same test cases used in the evaluation of AoT prototype on other platforms.

Before discussing the run time aspects of this version of our prototype, we evaluate the usage of RAM for ABS algorithms and the storage requirement of our AoT prototype at 128-

bit security level. Starting with RAM, we measure maximum memory utilization during the execution of each ABS algorithm varying the number of attributes ($1 \leq \ell \leq 10$) and using only "and" operators on test predicates. Figure 7.13 shows our experimental results. In terms of memory consumption, the ABS implementation on our AoT prototype at 128-bit security level, in spite of requiring twice as much memory as the 100-bit security level version of our prototype, is well-suited for resource-constrained devices as it requires at most 38KB of RAM memory for a complex predicate with ten attributes and nine "and" operators. The storage requirements for AoT on the Due, in turn, do not differ from the other versions of our prototype. The RELIC library plus our AoT implementation takes 147 KB of storage, which fits on the Due while leaving significant storage space for applications. RAM and storage requirements are not affected by the optimizations we envision in the subsequent analysis, remaining the same for a hypothetical optimized prototype.

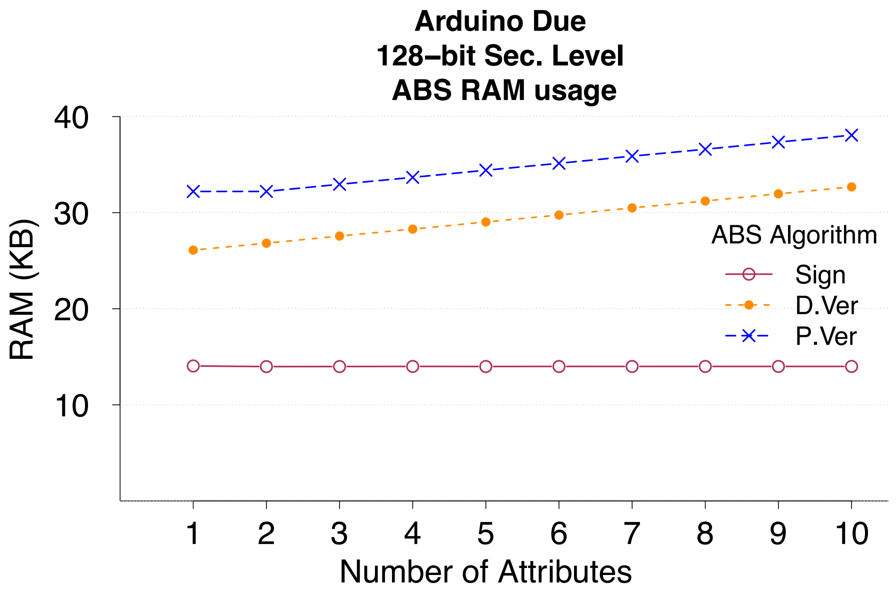


Figure 7.13: Experimental ABS algorithms' RAM requirements at 128-bit security level on the Arduino Due for varying the number of attributes in the predicate and using only "and" operators.

Figure 7.14 shows the run times for the cryptographic primitives used in AoT. As expected, using a curve with a larger base field order (381-bit in this case versus 254-bit in the case of the BN curve) and without optimization, the run time results are high on the Due. Our 128-bit security level prototype on the Due takes around 3.7s and 1.9s to perform IBE encryption and decryption, respectively, 4.4s to establish keys using the MB scheme, and 142ms and 707ms to generate and verify an IBS signature. Even with high run times, as these primitives are not frequently executed, they are acceptable for resource-constrained devices executing AoT at 128-bit security level. However, the ABS run times for this version of our prototype on the Due, as ABS algorithms are executed in each access control of a device operation, fulfill the timing requirements of a narrow range of IoT applications. In this version, our AoT prototype on the

Due takes around 6s to generate a signature for a predicate of the form $A \wedge B$, and 15s and 8s to verify it using the deterministic and probabilistic algorithms, respectively.

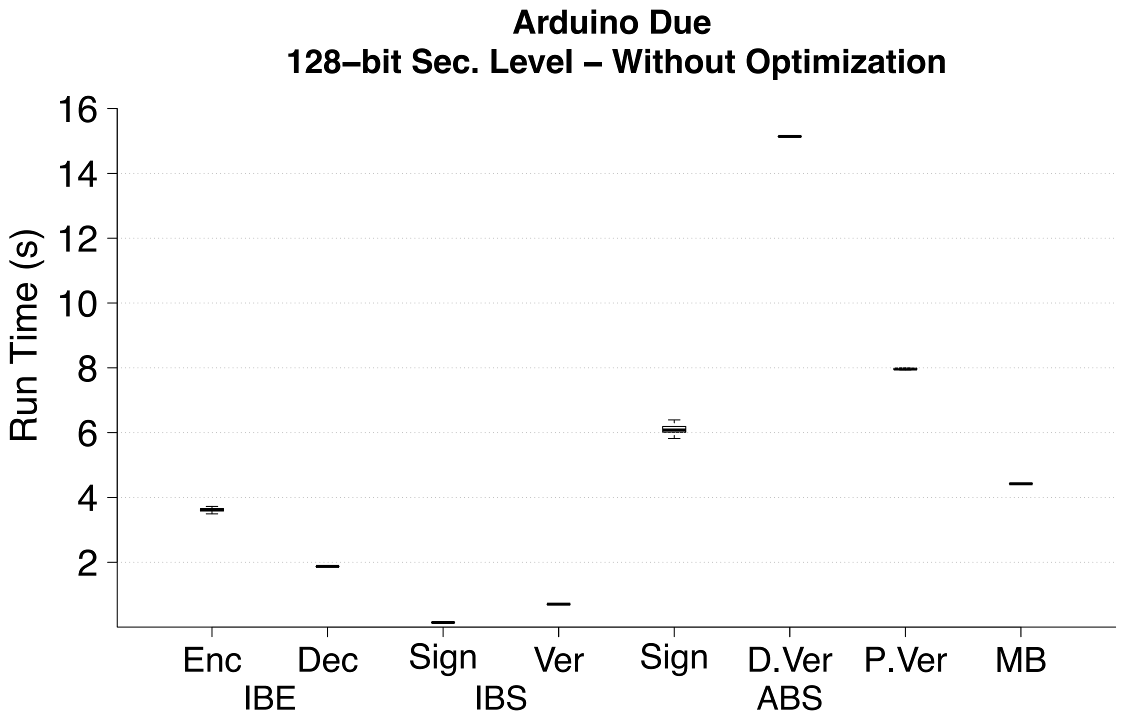


Figure 7.14: Run times for asymmetric cryptographic primitives on the Due at 128-bit security level without optimization. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$.

As there is no optimization for curve BLS12-381 on RELIC, we estimate the ABS run times for a prototype if such an option was available based on the results obtained in the profiling analysis of prototype versions without optimization and with assembly code for curve BN254 (Section 7.2.1). First of all, we use the Raspberry Pi3 as an auxiliary platform to profile our prototype at 128-bit security level without optimization. We use the contribution percentages of multiplication, modular reduction on the base field, and other functions on the overall run times of ABS algorithms obtained from the profiling on the Raspberry Pi3 combined with the experimental run times for ABS algorithms on the Due to estimate the run times of each arithmetic function on the Due. Then, we use the optimization speedup we obtained with the assembly implementation in the 100-bit prototype evaluation to estimate the ABS algorithms' run times on a hypothetically optimized version of RELIC for curve BLS12-381. Table 7.10 presents all these numbers. We estimate that our AoT prototype using an optimized version of RELIC for curve BLS12-381 on the Due would take around 2.7s to generate an ABS signature for a predicate of the form $A \wedge B$, and 6.5s and 4s to verify such a signature using the deterministic and probabilistic ABS algorithms, respectively.

Analogously to the strategy adopted for our 100-bit prototype, another effort to reduce the ABS run times would be to overclock the Arduino Due microcontroller. We also estimate the

ABS Algorithm	Function	Without Optimization % Time	Without Optimization Run Time (s)	Optimization Speedup	Estimated Run Time (s)	ABS Estimated Run Time (s)
Signature Generation	Multiplication	37.31%	2.28	2.91	0.78	
	Modular Reduction	38.08%	2.32	3.31	0.70	2.69
	Others	24.61%	1.50	1.24	1.21	
Deterministic Signature Verification	Multiplication	40.16%	6.04	2.72	2.22	
	Modular Reduction	28.69%	4.31	3.32	1.30	6.52
	Others	31.15%	4.68	1.56	3.00	
Probabilistic Signature Verification	Multiplication	33.72%	2.69	2.97	0.90	
	Modular Reduction	32.56%	2.59	2.58	1.01	3.94
	Others	33.72%	2.69	1.32	2.03	

Table 7.10: Profiling results of ABS algorithms at 128-bit security level using *Gprof* [45] on Raspberry Pi3 combined with experimental run times on the Due and speedup obtained on the 100-bit security level evaluation.

results of such a scenario by applying the speedup obtained by overclocking the microcontroller in the 100-bit security level experimental evaluation to the run times we just presented. We estimate that our AoT prototype using optimized assembly code at 128-bit security level on the Arudino Due with its microcontroller in an overclocking state at 114MHz would take around 2s to generate an ABS signature on a predicate of the form $A \wedge B$, 5.2s to verify such a signature using the deterministic ABS algorithm, and 3.1s to verify the signature using the probabilistic ABS algorithm. Therefore, IoT applications that can afford such performance overheads on the resource-constrained devices as a trade-off for 128-bit security level can leverage such an instantiation of AoT. Otherwise, alternative schemes to speedup computation, as discussed in Section 7.2.1, should be considered.

7.2.2.2 Intermediate Platform

Now we present the evaluation of our prototype at 128-bit security level on the Raspberry Pi1. As we use the same underlying implementation of AoT, the memory and storage requirements obtained for the Due prototype show that these numbers are negligible compared to the amount of resources available on our other platforms, including the Raspberry Pi, therefore we focus on evaluating execution time. Every result we present is generated using the same test cases used in the evaluation of AoT on other test platforms. Figure 7.15 shows the run times for the asymmetric cryptographic primitives used in AoT on the Raspberry Pi1.

We observe that even without any optimization, all cryptographic primitives used in AoT execute in reasonable time on the Raspberry Pi1. For instance, IBE encryption and decryption, IBS signature generation and verification, and the MB scheme are executed under 250ms. The ABS run times are also suitable for most IoT applications. Specifically, an ABS signature on a predicate of the form $\boldsymbol{A} \wedge \boldsymbol{B}$ takes around 320ms, a probabilistic signature verification takes

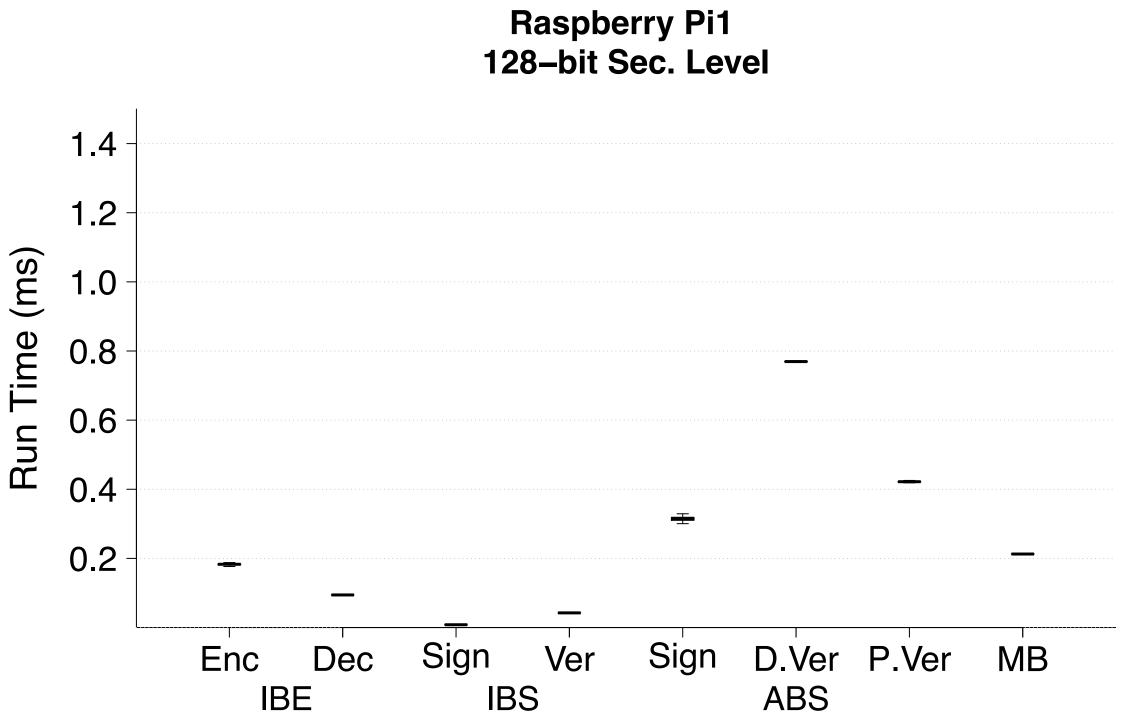


Figure 7.15: Run times for asymmetric cryptographic primitives on the Raspberry Pi1 at 128-bit security level. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$.

around 420ms and the most timing consuming deterministic signature verification takes around 770ms.

Table 7.11 shows run times of ABS algorithms' most expensive operations for the Raspberry PI1 to allow us to evaluate how our AoT prototype at 128-bit security level executing on an intermediate platform would scale in a realistic IoT scenario. Figure 7.16 shows analytically estimated run times for ABS signature generation (on the top), deterministic verification (on the middle), and probabilistic verification (on the bottom) considering different predicate structures, where we vary the number of attributes ($1 \leq \ell \leq 10$) and use only "and" operator. Curves in Figure 7.16 are plotted combining the experimental numbers from Table 7.11 and ABS costs from Table 7.1. In the x axis we represent the number ℓ of attributes in the predicate.

Raspberry Pi1
128-bit Sec. Level

Operation	**Run Time (ms)**
Scalar point multiplication in $\mathbb{G}_1$	15.6 ± 0.3
Scalar point multiplication in $\mathbb{G}_2$	30.0 ± 1.0
Bilinear pairing ($\hat{e}$)	93.9 ± 0.4

Table 7.11: Expensive operations run times (ms) on the AoT prototype at 128-bit security level for the Raspberry Pi1.

The analytical curves from Figure 7.16 show that for complex predicates with ten attributes and nine “and” operators, AoT prototype on the Raspberry Pi1 would take around 4s to generate a signature, 12s to verify it using the deterministic algorithm, and around 4.5s to verify it using the probabilistic algorithm. As we did in the evaluation on our other test platforms, we complement the Figure 7.16 curves with our experimental averaged over 30 runs (coefficients of variation are below 5%, not shown) considering the same predicate structures. As expected, the experimental results are considerably better. The prototype on the Raspberry Pi1 actually takes around 1.3s to sign such a complex predicate, 2.7s to verify it using the deterministic algorithm, and around 1.2s to verify it using the probabilistic algorithm. In terms of scalability, the numbers for complex predicates are a bit high for intermediate platforms on this version of our AoT prototype. However, as we do not implement any optimization on this version, there is much room for improvement, as discussed for resource-constrained devices.

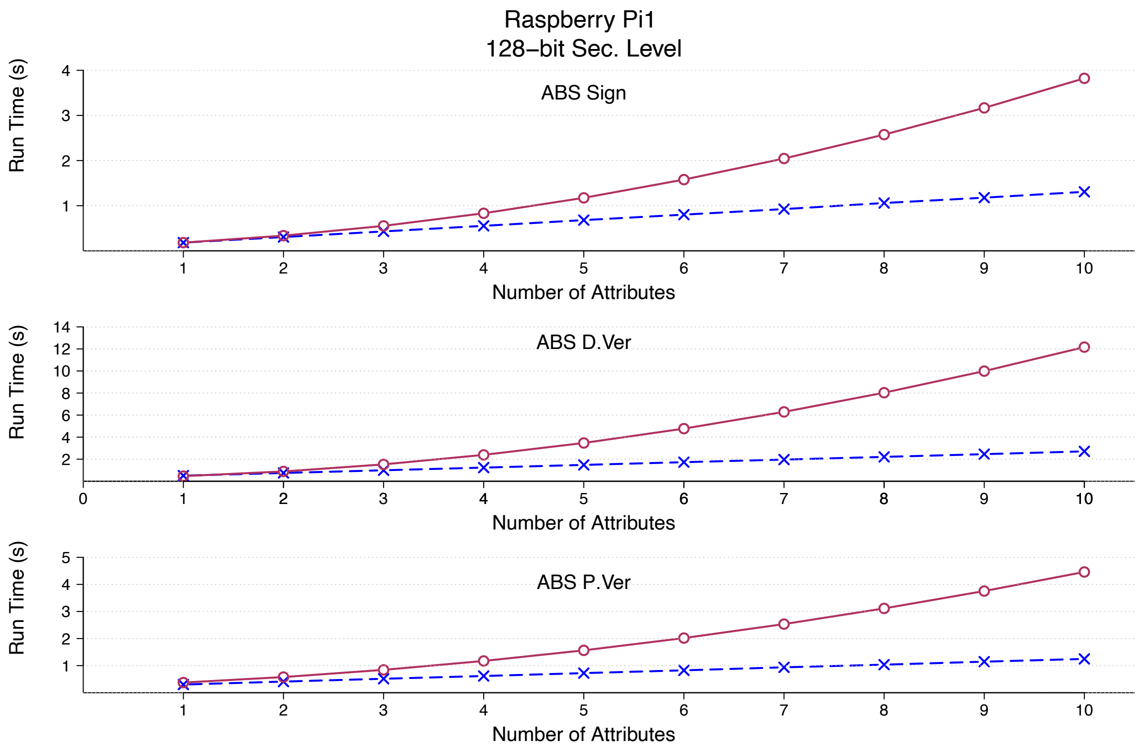


Figure 7.16: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our prototype of AoT on the Raspberry Pi1. The curves on the Figure are plotted combining Tables 7.1 and 7.8.

7.2.2.3 Powerful Platforms

Now we present the evaluation of our AoT prototype at 128-bit security level on the Raspberry Pi3 and the Google Pixel 6. Every result we present is generated using the same test cases used in the evaluation of AoT on other platforms.

Figure 7.17 shows the run times of asymmetric cryptographic primitives used in AoT on both powerful platforms (Raspberry Pi3 results on the left and Google Pixel 6 results on the right). All cryptographic primitives used in AoT execute in less than 300ms on the Raspberry Pi3 and in less than 70ms on the Google Pixel 6.

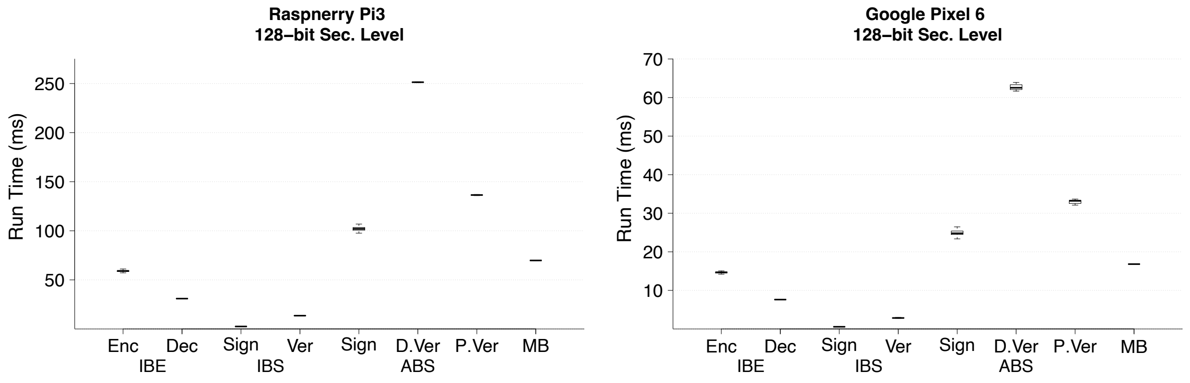


Figure 7.17: Run times for asymmetric cryptographic primitives in AoT on the Raspberry Pi3 (left) and Google Pixel 6 (right) at 128-bit security level. Run times for IBE and IBS use 32 bytes and 1KB messages, respectively, and ABS uses predicates of the form $\boldsymbol{A} \wedge \boldsymbol{B}$.

Table 7.12 shows run times of ABS algorithms' most expensive operations on our prototype for the Raspberry Pi3 and Google Pixel 6. We use these numbers to evaluate how our AoT prototype at 128-bit secutity level would scale in realistic IoT scenarios when execited on powerful platforms. Figures 7.18 and 7.19 show analytically estimated run times for ABS signature generation (on the top), deterministic verification (on the middle), and probabilistic verification (on the bottom) on the Raspberry Pi3 and Google Pixel 6, respectively, considering the different predicate structures varying the number of attributes ($1 \leq \ell \leq 10$) and use only "and" operator. Curves in the Figures are plotted combining the experimental numbers from Table 7.12 and ABS costs from Table 7.1. In the x axis we represent the number ℓ of attributes in the predicate.

	128-bit Sec. Level	
Operation	**Raspberry Pi3 Run Time (ms)**	**Google Pixel 6 Run Time (ms)**
Scalar point multiplication in $\mathbb{G}_1$	4.9 ± 0.1	1.07 ± 0.03
Scalar point multiplication in $\mathbb{G}_2$	9.8 ± 0.4	2.50 ± 0.10
Bilinear pairing ($\hat{e}$)	30.8 ± 0.10	7.70 ± 0.10

Table 7.12: Expensive operations run times (ms) on the AoT prototype at 128-bit security level for the Raspberry Pi3 and the Google Pixel 6.

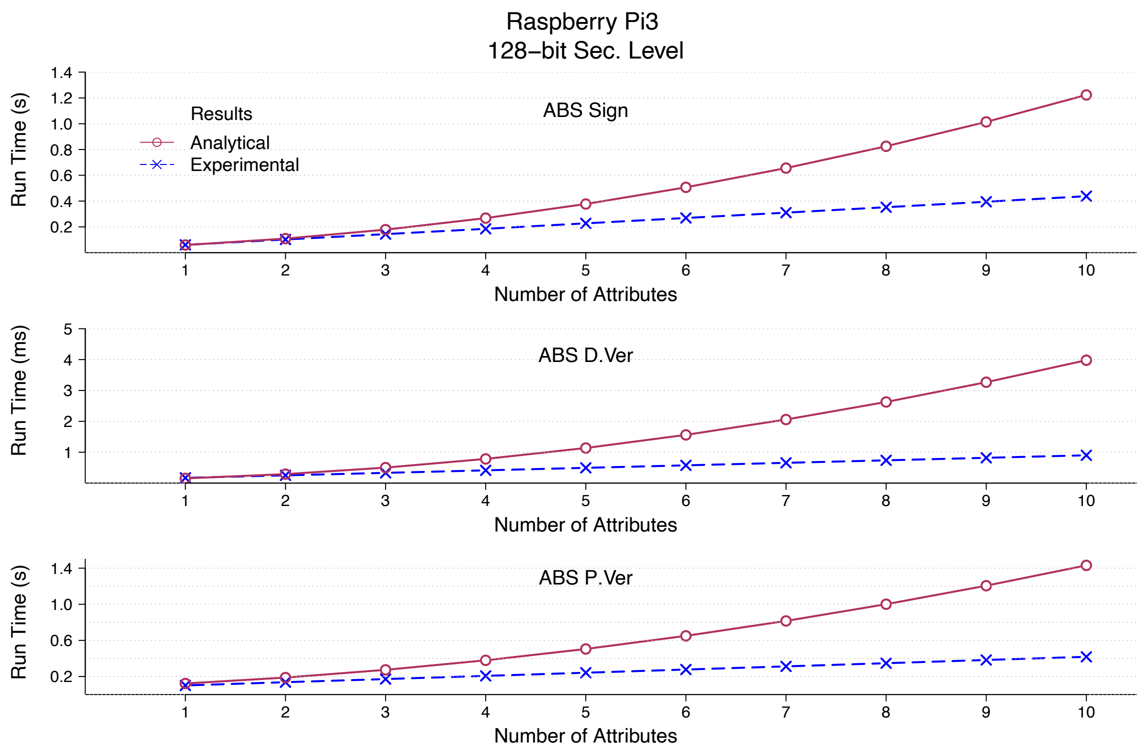


Figure 7.18: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our prototype of AoT at 128-bit security level on the Raspberry Pi3. The curves on the Figure are plotted combining Tables 7.1 and 7.12.

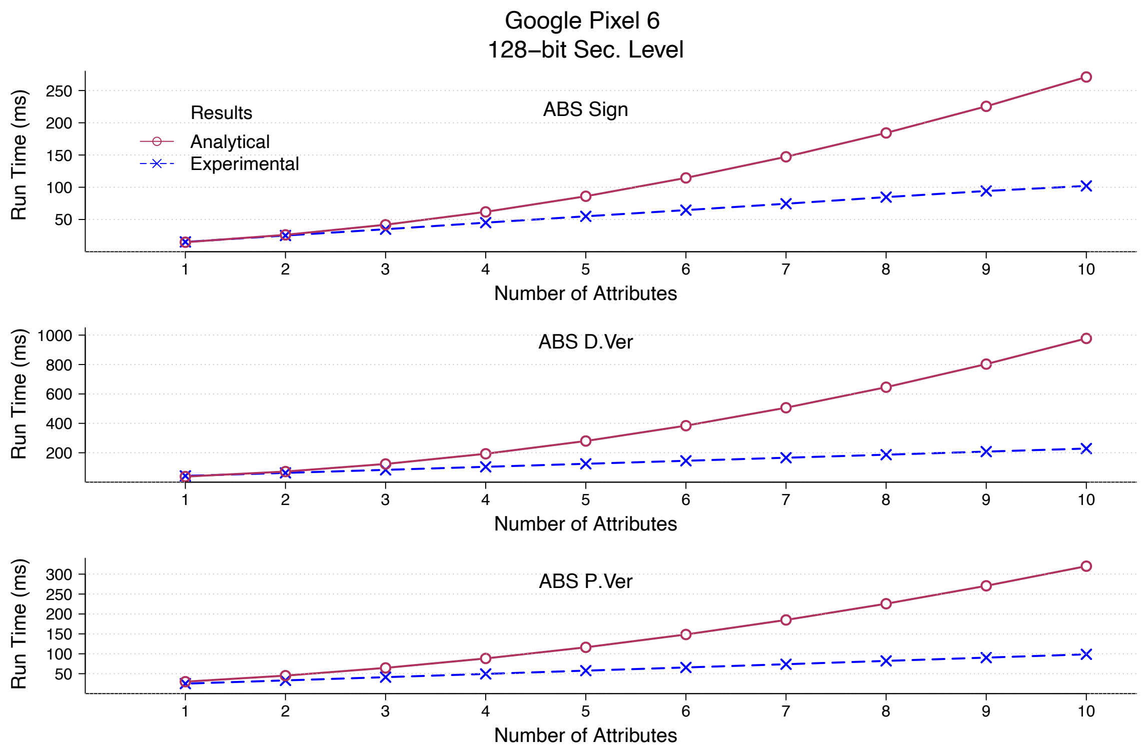


Figure 7.19: Analytical and experimental run times for ABS algorithms (signature generation on the top, deterministic signature verification on the middle, and probabilistic signature verification on the bottom) varying predicate structures in our prototype of AoT at 128-bit security level on the Google Pixel 6. The curves on the Figure are plotted combining Tables 7.1 and 7.12.

The analytical curves in Figures 7.18 and 7.19 show that for a complex predicate with ten attributes and nine “and” operators our AoT prototype on the Raspberry Pi3 and on the Google Pixel 6 would take, respectively, around 1.2s and 270ms to generate a signature, 4s and 1s to verify it using the deterministic algorithm and around 1.5s and 320ms to verify it using the probabilistic algorithm. As we did in the evaluation on other platforms, we complement the curves on the Figures with our experimental averaged over 30 runs (coefficients of variation are below 5%, not shown) considering the same predicate structures. As expected, the experimental results are considerably better. The AoT prototype at 128-bit security level on Raspberry Pi3 actually takes around 450ms to sign such a complex predicate, 900ms to verify it using the deterministic algorithm and around 420ms to verify it using the probabilistic algorithm. The prototype executed on the Google Pix 6, in turn, actually takes 100ms to sign this predicate, 230ms to verify it using the deterministic algorithm and around 100ms to verify it using the probabilistic algorithm. Therefore, experimental results show that AoT at 128-bit security level also imposes negligible overhead on powerful devices.

Chapter 8

Conclusion and Future Work

In IoT, questions as how to enable authentication and fine-grained access control remain unanswered. Authentication schemes for IoT which target resource-constrained devices, typically base their access control mechanism solely on authentication. Alternatives for fine-grained access control for resource-constrained devices in IoT, in turn, usually delegate the access control decision to an external trusted entity, which impacts user experience in cases of instability or unavailability on the authorization service. Besides, questions regarding the portability and mobility of "things" arise. Portability and mobility are typical of IoT and they reinforce the call for interoperation between local and guest devices. Last, there is a lack of options for authentication and access control solutions that cover the entire IoT device life-cycle, i.e., from device manufacturing to decommissioning.

In this work we proposed AoT, a holistic authentication and fine-grained access control solution for the entire IoT device life-cycle, namely: *pre-deployment*, *ordering*, *deployment*, *functioning*, and *retirement*. Besides, in AoT, all device-to-device authentication and access control processes' decision do not have any delegation to third parties, the solution allows device ownership reassignment, interoperation between devices from different local domains of trust, and contemplates separated device-to-manufacturer and device-to-device trust relationships during the IoT device life-cycle.

AoT protocols relies on IBC to distribute keys and authenticate devices as well as ABC to cryptographically enforce a fine-grained ABAC model. Our insight is to tackle the well-known key escrow problem of IBC is designing a two-domain architecture to manage the separated manufacturer-to-device and local device-to-device trust relationships.

We designed AoT as a composition of cryptographic protocols and primitives, and presented its modeling and security analysis under the Universal Composability paradigm. In this context, we extended the functionality for cryptographic primitives from [62] to support a new set of primitives which, in turn, allows the analysis of the identity-based authenticated key agreement protocols categorized into the same family of protocols proposed by [81].

We implemented an AoT prototype, at different security levels, on different platforms, varying computational resources, representing a wide range of IoT devices. We used a recently launched Android mobile phone, a Google Pixel 6, as a representative of smartphones that could be used in a smart environment supported by AoT. Other powerful entities in a smart environment were represented by a Raspberry Pi3. We represented intermediate smart devices with Raspberry Pi1. Last, as our representative of microcontrollers that could be used on low-end appliances supporting AoT, we used an Arduino Due. We used our prototype to quantify CPU, memory, storage, and communication overheads imposed by AoT protocols. Our results showed that memory and storage requirements of AoT are, at both 100- and 128-bit security levels, well-suited for all types of devices analyzed. In terms of performance, at 100-bit security-level, the AoT performance ranges from affordable on resource-constrained devices like the Arduino Due to efficient on intermediate devices like the Rapsberry Pi1, and negligible on powerful devices like the Raspberry Pi3 and on smartphones like the Google Pixel 6. The same apply if we observe results at 128-bit security level for intermediate and powerful devices. In case of resource-constrained devices, however, the run times might be enough for a limited number of applications, and alternatives for speeding up the ABS computation might be considered.

In future work, we plan to enhance the following aspects of AoT.

- Key establishment strategy will be extended to consider an identity-based key agreement protocol without escrow, which will make it provide also perfect forward secrecy.
- Propose a new key establishment protocol also for the *Functioning* stage, such that it will be resistant to key-compromise impersonation.
- Our extension on the $\mathcal{F}_{\text{crypto}}$ functionality will be once more extended to support the analysis of a wider set of identity-based key agreement protocols under the Universal Composability paradigm;
- We will research a new possibility for an ABS scheme which has the following requirements:
  - lighter than the ABS scheme used in our prototype;
  - analyzed in a more robust security model than the scheme used in our prototype;
  - practical to be implemented;
  - allows AoT inter-domain access control.

Bibliography

[1] Hezam Akram Abdul-Ghani, Dimitri Konstantas, and Mohammed Mahyoub. A comprehensive iot attacks survey based on a building-blocked reference model. *International Journal of Advanced Computer Science and Applications*, 2018.

[2] Asma Alshehri and Ravi Sandhu. Access control models for virtual object communication in cloud-enabled iot. In *IEEE international conference on information reuse and integration (IRI)*, 2017.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *SJCC'67*, 1967.

[4] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`, 2008.

[5] Diego F Aranha, Paulo SLM Barreto, Patrick Longa, and Jefferson E Ricardini. The realm of the pairings. In *International Conference on Selected Areas in Cryptography*, 2013.

[6] Kevin Ashton. That 'Internet of Things' Thing. *RFiD Journal*, 2009.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 2010.

[8] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–16. Springer, 2014.

[9] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography (SAC)*, 2005.

[10] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *International conference on security in communication networks*, pages 257–267. Springer, 2002.

[11] Karyn Benson, Hovav Shacham, and Brent Waters. The k-bdh assumption family: Bilinear map cryptography from progressively weaker assumptions. In *Cryptographers' Track at the RSA Conference*, pages 310–325. Springer, 2013.

[12] Florent Bersani and Hannes Tschofenig. The eap-psk protocol: A pre-shared key extensible authentication protocol (eap) method. Technical report, RFC 4764, January, 2007.

[13] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy Attribute-based Encryption. In *Symposium on Security and Privacy (S&P)*, 2007.

[14] Matt Bishop. *Computer security: art and science*, volume 200. Addison-Wesley, 2012.

[15] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. Identity-based Encryption with Efficient Revocation. In *Conference on Computer and Communications Security (CCS)*, 2008.

[16] D. Boneh, B. Lynn, and H. Schacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[17] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International conference on the theory and applications of cryptographic techniques*, pages 56–73. Springer, 2004.

[18] Dan Boneh and Matthew Franklin. Identity-based encryption from the weil pairing. *SIAM Journal on Computing*, 32(3):586–615, 2003.

[19] Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 2001.

[20] Dan Boneh, Craig Gentry, and Michael Hamburg. Space-efficient identity based encryption without pairings. In *Symposium on Foundations of Computer Science (FOCS)*, 2007.

[21] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *Cryptology ePrint Archive*, 2017.

[22] Seyit A Camtepe and Blent Yener. Combinatorial Design of Key Distribution Mechanisms for Wireless Sensor Networks. In *European Symposium on Research in Computer Security (ESORICS)*, 2004.

[23] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[24] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *International conference on the theory and applications of cryptographic techniques*, pages 453–474. Springer, 2001.

[25] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 337–351. Springer, 2002.

[26] Xuefei Cao, Weidong Kou, Lanjun Dang, and Bin Zhao. IMBAS: Identity-based Multi-user Broadcast Authentication in Wireless Sensor Networks. *Computer Communications*, 31(4):659 – 667, 2008.

[27] S. Capkun, L. Buttyan, and J. P. Hubaux. Self-Organized Public-Key Management for Mobile Ad Hoc Networks. *Transactions on Mobile Computing*, 2(1):17, 2003.

[28] Sanjit Chatterjee and Palash Sarkar. *Identity-Based Encryption*. Springer Publishing Company, Incorporated, 2011.

[29] Zhaohui Cheng and Liqun Chen. On security proof of mccullagh-barreto's key agreement protocol and its variants. *International Journal of Security and Networks*, 2007.

[30] Simone Cirani, Marco Picone, Pietro Gonizzi, Luca Veltri, and Gianluigi Ferrari. Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios. *IEEE sensors journal*, 2014.

[31] Clifford Cocks. An Identity Based Encryption Scheme Based on Quadratic Residues. In *International Conference on Cryptography and Coding (IMACC)*, 2001.

[32] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[33] Maissa Dammak, Omar Rafik Merad Boudia, Mohamed Ayoub Messous, Sidi Mohammed Senouci, and Christophe Gransart. Token-based lightweight authentication to secure iot networks. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2019.

[34] Augusto Jun Devegili, Michael Scott, and Ricardo Dahab. Implementing cryptographic pairings over barreto-naehrig curves. *Pairing*, 4575:197–207, 2007.

[35] Sheng Ding, Jin Cao, Chen Li, Kai Fan, and Hui Li. A novel attribute-based access control scheme using blockchain for iot. *IEEE Access*, 2019.

[36] Marlon C Domenech, Azzedine Boukerche, and Michelle S Wangham. An authentication and authorization infrastructure for the web of things. In *Proceedings of the 12th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, 2016.

[37] W. Du, J. Deng, Y. S. Han, P. K. Varshney, J. Katz, and A. Khalili. A Pairwise Key Predistribution Scheme for Wireless Sensor Networks. *Transactions on Information and System Security (TISSEC)*, 2005.

[38] Federico Fernández, Álvaro Alonso, Lourdes Marco, and Joaquín Salvachúa. A model to enable application-scoped access control as a service for iot using oauth 2.0. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017.

[39] Paul Fremantle, Benjamin Aziz, Jacek Kopeckỳ, and Philip Scott. Federated identity and access management for the internet of things. In *2014 International Workshop on Secure Internet of Things*, 2014.

[40] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

[41] Steven D Galbraith, Kenneth G Paterson, and Nigel P Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 2008.

[42] Conrado PL Gouvêa, Leonardo B Oliveira, and Julio López. Efficient software implementation of public-key cryptography on sensor networks using the msp430x microcontroller. *Journal of Cryptographic Engineering*, 2(1):19–29, 2012.

[43] Conrado Porto Lopes Gouvêa and Julio López. Software implementation of pairing-based cryptography on sensor networks using the msp430 microcontroller. In *Indocrypt*, pages 248–262. Springer, 2009.

[44] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In *Conference on Computer and Communications Security (CCS)*, 2006.

[45] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN'82*, 1982.

[46] Jorge Guajardo, Rainer Blümel, Uwe Krieger, and Christof Paar. Efficient implementation of elliptic curve cryptosystems on the ti msp430x33x family of microcontrollers. In *International Workshop on Public Key Cryptography*, pages 365–382. Springer, 2001.

[47] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2004.

[48] Mohamed Tahar Hammi, Badis Hammi, Patrick Bellot, and Ahmed Serhrouchni. Bubbles of trust: A decentralized blockchain-based authentication system for iot. *Computers & Security*, 2018.

[49] Dennis Hofheinz and Victor Shoup. Gnuc: A new universal composability framework. *Journal of Cryptology*, 28(3):423–508, 2015.

[50] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication*, 800:162, 2013.

[51] Yih-Chun Hu, Adrian Perrig, and David B. Johnson. Ariadne: a Secure On-demand Routing Protocol for Ad Hoc Networks. In *International Conference on Mobile Computing and Networking (MobiCom)*, 2002.

[52] René Hummen, Jan H Ziegeldorf, Hossein Shafagh, Shahid Raza, and Klaus Wehrle. Towards viable certificate-based authentication for the internet of things. In *Proceedings of the 2nd ACM workshop on Hot topics on wireless network security and privacy*, 2013.

[53] Antoine Joux. The Weil and Tate Pairings as Building Blocks for Public Key Cryptosystems. In *International Symposium on Algorithmic Number Theory (ANTS)*, 2002.

[54] Antoine Joux. A One Round Protocol for Tripartite Diffie-Hellman. *J. Cryptology*, 17(4):263–276, 2004.

[55] Ahammad Karim and Muhammad Abdullah Adnan. An openid based authentication service mechanisms for internet of things. In *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, 2019.

[56] Umair Khalid, Muhammad Asim, Thar Baker, Patrick CK Hung, Muhammad Adnan Tariq, and Laura Rafferty. A decentralized lightweight blockchain-based authentication mechanism for iot systems. *Cluster Computing*, 2020.

[57] Wazir Zada Khan, Mohammed Y Aalsalem, and Muhammad Khurram Khan. Five acts of consumer behavior: A potential security and privacy threat to internet of things. In *IEEE international conference on consumer electronics (ICCE'18)*, 2018.

[58] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Annual international cryptology conference*, pages 543–571. Springer, 2016.

[59] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication, 1997.

[60] Saru Kumari, Marimuthu Karuppiah, Ashok Kumar Das, Xiong Li, Fan Wu, and Neeraj Kumar. A secure authentication scheme based on elliptic curve cryptography for iot and cloud servers. *The Journal of Supercomputing*, 2018.

[61] Ralf Kusters. Simulation-based security with inexhaustible interactive turing machines. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 12–pp. IEEE, 2006.

[62] Ralf Küsters and Daniel Rausch. A framework for universally composable diffie-hellman key exchange. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 881–900. IEEE, 2017.

[63] Ralf Küsters and Max Tuengerthal. Joint state theorems for public-key encryption and digital signature functionalities with local computation. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 270–284. IEEE, 2008.

[64] Ralf Küsters and Max Tuengerthal. Universally composable symmetric encryption. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 293–307. IEEE, 2009.

[65] Ralf Küsters and Max Tuengerthal. Composition theorems without pre-established session identifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 41–50, 2011.

[66] Ralf Küsters and Max Tuengerthal. Ideal key derivation and encryption in simulation-based security. In *Cryptographers' Track at the RSA Conference*, pages 161–179. Springer, 2011.

[67] Ralf Küsters and Max Tuengerthal. The iitm model: a simple and expressive model for universal composability. 2013.

[68] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. The iitm model: a simple and expressive model for universal composability. *Journal of Cryptology*, pages 1–124, 2020.

[69] Mikko Kärkkäinen, Jan Holmström, Kary Främling, and Karlos Artto. Intelligent products—a step towards a more effective project delivery chain. *Computers in Industry*, 2003. Advanced Web Technologies for Industrial Applications.

[70] Songping Li, Quan Yuan, and Jin Li. Towards security two-part authenticated key agreement protocols. *Cryptology ePrint Archive*, 2005.

[71] Chae Hoon Lim and Pil Joong Lee. More Flexible Exponentiation with Precomputation. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1994.

[72] Chao Lin, Debiao He, Xinyi Huang, Kim-Kwang Raymond Choo, and Athanasios V Vasilakos. Bsein: A blockchain-based secure mutual authentication with fine-grained access control system for industry 4.0. *Journal of network and computer applications*, 2018.

[73] Donggang Liu, Peng Ning, and Rongfang Li. Establishing Pairwise Keys in Distributed Sensor Networks. *Transactions on Information and System Security (TISSEC)*, 2005.

[74] Jing Liu, Yang Xiao, and CL Philip Chen. Authentication and access control in the internet of things. In *2012 32nd International Conference on Distributed Computing Systems Workshops*, 2012.

[75] Mark Luk, Adrian Perrig, and Bram Whillock. Seven Cardinal Properties of Sensor Network Broadcast Authentication. In *Workshop on Security of Ad Hoc and Sensor Networks (SASN)*, 2006.

[76] Roben Castagna Lunardi, Regio Antonio Michelin, Charles Varlei Neu, and Avelino Francisco Zorzo. Distributed access control on iot ledger-based architecture. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, 2018.

[77] Hemanta K Maji, Manoj Prabhakaran, and Mike Rosulek. Attribute-Based Signatures: Achieving Attribute-Privacy and Collusion-Resistance. *IACR Cryptology ePrint Archive*, 2008:328, 2008.

[78] Hemanta K Maji, Manoj Prabhakaran, and Mike Rosulek. Attribute-based signatures. In *Cryptographers' track at the RSA conference*, pages 376–392. Springer, 2011.

[79] David J. Malan, Matt Welsh, and Michael D. Smith. A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography. In *Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, 2004.

[80] Tobias Markmann, Thomas C Schmidt, and Matthias Wählisch. Federated End-to-End Authentication for the Constrained Internet of Things Using IBC and ECC. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[81] Noel McCullagh and Paulo S. L. M. Barreto. A New Two-party Identity-based Authenticated Key Agreement. In *International Conference on Topics in Cryptology (CT-RSA)*, 2005.

[82] A. Menezes, T. Okamoto, and St Vanstone. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. *Transactions on Information Theory*, 39(5):1639–1646, 1993.

[83] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[84] Silvio Micali, Oded Goldreich, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, pages 218–229, 1987.

[85] V. Mora-Afonso, P. Caballero-Gil, and J. Molina-Gil. Strong Authentication on Smart Wireless Devices. In *International Conference on Future Generation Communication Technology (FGCT)*, 2013.

[86] Mohammed Nafi, Samia Bouzefrane, and Mawloud Omar. Matrix-based key management scheme for iot networks. *Ad Hoc Networks*, 2020.

[87] National Institute of Standards and Technology. *Recommended Elliptic Curves for Federal Government Use*. 1999.

[88] Yasuyuki Nogami, Masataka Akane, Yumi Sakemi, Hidehiro Katou, and Yoshitaka Morikawa. Integer variable chi-based ate pairing. *Pairing*, 5209:178–191, 2008.

[89] Leonardo B. Oliveira, Diego Aranha, Eduardo Morais, Felipe Daguano, Julio López, and Ricardo Dahab. TinyTate: Computing the tate pairing in resource-constrained nodes. In *6th IEEE International Symposium on Network Computing and Applications (NCA'07)*, 2007.

[90] Leonardo B. Oliveira, Ricardo Dahab, Julio Lopez, Felipe Daguano, and Antonio A. F. Loureiro. Identity-based encryption for sensor networks. In *5th IEEE Int'l Conference on Pervasive Computing and Communications Workshops (PERCOMW '07)*, 2007.

[91] Leonardo B. Oliveira, Aman Kansal, Bodhi Priyantha, Michel Goraczko, and Feng Zhao. Secure-TWS: Authenticating Node to Multi-user Communication in Shared Sensor Networks. In *International Conference on Information Processing in Sensor Networks (IPSN)*, 2009.

[92] Leonardo B. Oliveira, Michael Scott, Julio Lopez, and Ricardo Dahab. TinyPBC: Pairings for Authenticated Identity-Based Non-Interactive Key Distribution in Sensor Networks. In *International Conference on Networked Sensing Systems (INSS)*, 2008.

[93] Geovandro CCF Pereira, Marcos A Simplício, Michael Naehrig, and Paulo SLM Barreto. A family of implementation-friendly bn elliptic curves. *Journal of Systems and Software*, 84(8):1319–1326, 2011.

[94] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, 8(5):521–534, 2002.

[95] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 184–200. IEEE, 2000.

[96] R. Di Pietro, L. V. Mancini, and A. Mei. Random Key-Assignment for Secure Wireless Sensor Networks. In *Workshop on Security of Ad Hoc and Sensor Networks (SASN)*, pages 62–71, 2003.

[97] Otto Julio Ahlert Pinno, Andre Ricardo Abed Gregio, and Luis CE De Bona. Controlchain: Blockchain as a central enabler for access control authorizations in the iot. In *IEEE Global Communications Conference GLOBECOM'17*, 2017.

[98] Pawani Porambage, Corinna Schmitt, Pardeep Kumar, Andrei Gurtov, and Mika Ylianttila. Two-phase authentication protocol for wireless sensor networks in distributed iot applications. In *2014 IEEE Wireless Communications and Networking Conference (WCNC)*, 2014.

[99] Shahid Raza, Ludwig Seitz, Denis Sitenkov, and Göran Selander. S3k: Scalable security with symmetric keys—dtls key establishment for the internet of things. *IEEE Transactions on Automation Science and Engineering*, 2016.

[100] Diego Rivera, Luis Cruz-Piris, German Lopez-Civera, Enrique de la Hoz, and Ivan Marsa-Maestre. Applying an unified access control for iot-based intelligent agent systems. In *2015 IEEE 8th international conference on service-oriented computing and applications (SOCA)*. IEEE, 2015.

[101] Amit Sahai and Brent Waters. Fuzzy Identity-based Encryption. In *International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2005.

[102] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems Based on Pairing. In *Symposium on Cryptography and Information Security (SCIS)*, 2000.

[103] Maria LBA Santos, Jéssica C Carneiro, Antônio MR Franco, Fernando A Teixeira, Marco AA Henriques, and Leonardo B Oliveira. Flat: Federated lightweight authentication for the internet of things. *Ad Hoc Networks*, 2020.

[104] Michael Scott. Computing the Tate Pairing. In *International Conference on Topics in Cryptology (CT-RSA)*, 2005.

[105] Ludwig Seitz, Göran Selander, and Christian Gehrmann. Authorization framework for the internet-of-things. In *2013 IEEE 14th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, 2013.

[106] Adi Shamir. Identity-based Cryptosystems and Signature Schemes. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1984.

[107] Marcos A Simplicio Jr, Bruno T De Oliveira, Cintia B Margi, Paulo SLM Barreto, Tereza CMB Carvalho, and Mats Näslund. Survey and comparison of message authentication solutions on wireless sensor networks. *Ad Hoc Networks*, 2013.

[108] Marcos A Simplicio Jr, Marcos VM Silva, Renan CA Alves, and Tiago KC Shibata. Lightweight and escrow-less authenticated key agreement for the internet of things. *Computer Communications*, 2017.

[109] Douglas Stinson. *Cryptography: Theory and Practice*. CRC/C&H, 2002.

[110] Hui Suo, Jiafu Wan, Caifeng Zou, and Jianqi Liu. Security in the internet of things: a review. In *international conference on computer science and electronics engineering*, 2012.

[111] Piotr Szczechowiak, Anton Kargl, Michael Scott, and Martin Collier. On the Application of Pairing Based Cryptography to Wireless Sensor Networks. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2009.

[112] Piotr Szczechowiak, Leonardo B Oliveira, Michael Scott, Martin Collier, and Ricardo Dahab. Nanoecc: Testing the limits of elliptic curve cryptography in sensor networks. In *European conference on Wireless Sensor Networks*, 2008.

[113] Jenny Torres, Michele Nogueira, and Guy Pujolle. Identity-Based Cryptography: Applications, Vulnerabilities and Future Directions. In *IT Policy and Ethics: Concepts, Methodologies, Tools, and Applications*, pages 430–450. IGI Global, 2013.

[114] Thomas Unterluggauer and Erich Wenger. Efficient pairings and ecc for embedded systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 298–315. Springer, 2014.

[115] Lakshmi Venkatraman and Dharma P. Agrawal. A novel authentication scheme for ad hoc networks. In *Wireless Communications and Networking Conference (WCNC)*, 2002.

[116] Frederik Vercauteren. Optimal Pairings. *Transactions on Information Theory*, 56(1):455–461, 2010.

[117] Huai-Xi Wang, Yan Zhu, Rong-Quan Feng, and Stephen S Yau. Attribute-Based Signature with Policy-and-Endorsement Mechanism. *Journal of Computer Science and Technology*, 25(6):1293–1304, 2010.

[118] Ronald J. Watro, Derrick Kong, Sue fen Cuti, Charles Gardiner, Charles Lynn, and Peter Kruus. TinyPK: Securing Sensor Networks with Public Key Technology. In *Workshop on Security of Ad Hoc and Sensor Networks (SASN)*, 2004.

[119] Mark Weiser. The computer for the 21 st century. *Scientific american*, 1991.

[120] Wei Xi, Chen Qian, Jinsong Han, Kun Zhao, Sheng Zhong, Xiang-Yang Li, and Jizhong Zhao. Instant and robust authentication and key agreement among mobile devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[121] Guohong Xie. Cryptanalysis of noel mccullagh and paulo slm barreto's two-party identity-based key agreement. *Cryptology EPrint Archive*, 2004.

[122] Attila Altay Yavuz. ETA: Efficient and Tiny and Authentication for Heterogeneous Wireless Systems. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[123] Ning Ye, Yan Zhu, Ru-chuan Wang, Reza Malekian, and Qiao-min Lin. An efficient authentication and access control scheme for perception layer of internet of things. 2014.

[124] Narges Yousefnezhad, Avleen Malhi, and Kary Främling. Security in product lifecycle of iot devices: A survey. *Journal of Network and Computer Applications*, 2020.

[125] Eric Yuan and Jin Tong. Attributed Based Access Control (ABAC) for Web Services. In *International Conference on Web Services (ICWS)*, 2005.

[126] Fangguo Zhang, Reihaneh Safavi-Naini, and Willy Susilo. An efficient signature scheme from bilinear pairings and its applications. In *International Workshop on Public Key Cryptography*. Springer, 2004.

[127] Yongguang Zhang and Wenke Lee. Intrusion Detection in Wireless Ad-hoc Networks. In *International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[128] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong-Kuan Chen, and Shiuhpyng Shieh. Iot security: ongoing challenges and research opportunities. In *2014 IEEE 7th international conference on service-oriented computing and applications*, 2014.

[129] Lidong Zhou and Zygmunt J. Haas. Securing Ad Hoc Networks. *IEEE Network*, 13(6):24–30, 1999.

[130] Sencun Zhu, Sanjeev Setia, and Sushil Jajodia. LEAP: Efficient Security Mechanisms for Large-scale Distributed Sensor Networks. In *Conference on Computer and Communications Security (CCS)*, 2003.