D13.2

Efficient Verifiability and Precise Specification of Secure Computation Functionalities

Project number:	609611
Project acronym:	**PRACTICE**
Project title:	Privacy-Preserving Computation in the Cloud
Project Start Date:	1 November, 2013
Duration:	36 months
Programme:	FP7/2007-2013
Deliverable Type:	Report
Reference Number:	ICT-609611 / D13.2 / 1.0
Activity and WP:	Activity 1 / WP 13
Due Date:	April 2016 - M30
Actual Submission Date:	May 2nd 2016
Responsible Organisation:	BIU
Editor:	Manuel Barbosa
Dissemination Level:	PU
Revision:	1

Abstract:	This report is the second deliverable of work package WP13 (*Protocol Specification and Design*) in the PRACTICE project. It describes the results of the activities carried out in Task 13.2, Efficient Verifiability and Precise Specification of Secure Computation Functionalities. The Deliverable is split into parts that correspond to efficient verifiability solutions both relying and not relying in hardware assumptions. The solutions described in the two parts offer different trade-offs in terms of efficiency, trust models and hardware requirements. The precise specification of hardware assumptions is also discussed, and formal methods are applied to the validation of one of the proposed protocols.

Keywords:	Verifiable Multiparty Computation, Hardware-based Attested Computation, Distributed Verifiable Computation

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 609611.

Editor

Manuel Barbosa (INESC)

Contributors (ordered according to beneficiary numbers)

Bernardo Portela, INESC
Berry Schoenmakers, TUE
Niels de Vreede, TUE
Guilaume Scerri, UNIVBRIS
Bogdan Warinschi, UNIVBRIS

Disclaimer

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose subject to any liability which is mandatory due to applicable law. The users use the information at their sole risk and liability.

Executive Summary

Typical mechanisms for secure computation outsourcing used in the real world guarantee confidentiality: the parties involved in the computation are sure their inputs and the results they obtain are secret. However, these protocols do not guarantee efficient verifiability. That is, the parties involved in the protocol may not have the means to independently check that the results of the computation are correct (in an efficient way or without relying on uncomfortably strong trust assumptions) or to prove to third parties that the results are correct. Yet, in many scenarios, such verifiability guarantees for the result of a protocol execution are crucial.

This deliverable presents a series of protocols which advance the state of the art in providing efficient solutions to this problem that were developed by participants in PRACTICE WP13. It is divided into two parts that correspond to two alternative approaches to the problem of verifiability that offer different functionality, trust and efficiency tradeoffs for real-world applications: i. verifiable multiparty computation and ii. verifiable computation from hardware assumptions.

In the first part we discuss two protocols, the first of which (UVCDN) follows the approach of enhancing the built-in proofs of correctness in a existing multiparty computation protocol to be convincing to anyone, not just protocol participants. The second protocol, called Trinocchio, uses the approach of layering a Verifiable Computation (VC) scheme on top of an Multiparty Computation (MPC) protocol. To highlight the efficiency of the latter protocol, we discuss two case studies, including some performance figures. Both protocols achieve universal verifiability and do not rely on special hardware assumptions, so they can be deployed in practical systems.

The second part of this report deals with constructions of protocols for secure outsourced computation that rely on modern trusted hardware technologies, namely technologies such as Intel's new SGX which provide Isolated Execution Environments (IEE) capabilities. We give precise definitions of what IEE-enabled architectures offer, in the style of cryptographic provable security, as to offer a strating point formalizing and proving the security of secure computation protocols. We then design and formally prove the security of two protocols that permit carrying out verifiable computation in two different scenarios: i. secure outsourcing of computation from one computationally weak Client to a powerful but untrusted Worker; and ii. secure function evaluation by multiple (mutually distrusting) Clients relying on an untrusted Worker. In both cases, the designs are natural and formally justify how to leverage the extra guarantees offered by the trusted hardware to obtain solutions to existing problems that offer better efficieny and security trade-offs than previous solutions (in theory for now, but getting practical results for these protocols is ongoing work). We conclude this report with an application of formal methods techniques to the validation of our new approach to the analysis of advanced hardware-based assumptions.

Contents

1 **Introduction** **1**
1.1 Task description 2
1.2 Overview 3
1.3 Structure of the report 5

I **Verifiable Multiparty Computation** **6**

2 **Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems** **7**
2.1 Introduction 7
2.2 Secure Computation from Threshold Cryptography 9
2.2.1 Computation using a Threshold Homomorphic Cryptosystem 9
2.2.2 Proving Correctness of Results 10
2.2.3 Security of the CDN Protocol 11
2.3 Multiparty Non-Interactive Proofs 11
2.3.1 The Fiat-Shamir Heuristic 12
2.3.2 Combined Proofs with the Multiparty Fiat-Shamir Heuristic 13
2.4 Universally Verifiable MPC 14
2.4.1 Security Model for Verifiable MPC 15
2.4.2 Universally Verifiable CDN 16
2.5 Concluding Remarks 19

3 **Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation** **20**
3.1 Introduction 20
3.1.1 Outline 21
3.2 Verifiable Computation from QAPs 21
3.2.1 Modelling Computations as Quadratic Arithmetic Programs 21
3.2.2 Proving Correctness of Computations 22
3.2.3 Making the Proof Zero-Knowledge 23
3.2.4 From Arguments of Knowledge to Verifiable Computation 24
3.3 Security Model for Privacy-Preserving Outsourcing 25
3.4 Distributing the Prover Computation 26
3.4.1 Multiparty Computation using Shamir Secret Sharing 26
3.4.2 The Trinocchio protocol 27
3.4.3 Security of Trinocchio 29
3.5 Handling Mutually Distrusting In- and Outputters 30
3.5.1 Multi-Client Proofs and Keys 30

3.5.2 The Protocol . . . 32
3.5.3 Security of the Trinocchio Protocol . . . 34
3.6 Performance . . . 36
3.6.1 Case Study: Multivariate Polynomial Evaluation . . . 36
3.6.2 Speeding Up Verification by Validation . . . 37
3.7 Discussion and Conclusion . . . 38

II Verifiable Computation From Hardware Assumptions 40

4 Formalizing Hardware Assumptions: Attested Computation 41
4.1 Overview . . . 41
4.2 Software guard extensions . . . 42
4.3 Enclave abstraction . . . 45
4.4 Attested computation . . . 47

5 Secure Outsourced Computation From Attested Computation 54
5.1 Overview . . . 54
5.2 Key exchange for attested computation . . . 55
5.3 Hardware-based secure outsourced computation . . . 61
5.4 Discussion . . . 65

6 Secure Function Evaluation from Attested Computation 66
6.1 Overview . . . 66
6.2 Ideal function evaluation from SGX . . . 67
6.3 Secure Multi-agent Attested Computation (SMAC) . . . 73

7 Formal Specification and Verification 80
7.1 Overview . . . 80
7.2 Formal definition of Slapic . . . 82
7.3 Experimental results . . . 99

8 Conclusion 103

9 List of Abbreviations 104

List of Figures

2.1 Notation in algorithms, protocols, and processes 9

3.1 Ideal-world executions of secure (left) and correct (right) function evaluation. The highlighted text indicates where the two differ. 25

4.1 Process for remote attestation 43
4.2 Attested Computation scenario. 48
4.3 Games defining the correctness (left) and security (right) of an AC scheme. . . . 50
4.4 Games defining minimum leakage of an AC scheme. 51

5.1 Game defining the correctness of an AttKE scheme. 56
5.2 Execution environment for AttKEs. 57
5.3 Details of the AttKE construction. 59
5.4 Game defining the utility of an AttKE scheme when used in the context of attested computation. 61
5.5 Input integrity of a SOC scheme 63
5.6 Input privacy of a SOC scheme 63
5.7 SOC algorithms 64

6.1 Game defining the correctness of our protocol. 68
6.2 Game defining the behavior of Real and Ideal worlds. Fun can only be run once. 69
6.3 Details for running n parallel key exchange protocols (left), and for coding inputs/outputs of an arbitrary functionality (right). 70
6.4 Game defining the one-to-many utility of a AttKE scheme when used in the context of attested computation. 71
6.5 Algorithms defining the protocol. 74
6.6 Security game of SMAC 75
6.7 Labelled parallel composition (labels are assumed pairwise different) 76
6.8 Labelled utility 77
6.9 Algorithms defining the protocol. 78

7.1 Informal example with locations 81
7.2 Informal example with variable locations 82
7.3 Reporting of the one output of prog 83
7.4 Syntax with locations 84
7.5 Reporting of an output 85
7.6 Reporting of an output 86
7.7 E_H definition 86
7.8 attacker deductions rules 87
7.9 Operational semantic 88

7.10 Attested computation implementation 90
7.11 Original and rewritten process 91

Chapter 1

Introduction

Typical mechanisms for secure computation outsourcing used in the real world guarantee confidentiality: the parties involved in the computation are sure their inputs and the results they obtain are secret. However, these protocols do not guarantee efficient verifiability. That is, the parties involved in the protocol may not have the means to independently check that the results of the computation are correct (in an efficient way or without relying on uncomfortably strong trust assumptions) or to prove to third parties that the results are correct. Yet, in many scenarios, such verifiability guarantees for the result of a protocol execution are crucial.

Verifiability comes in a multitude of flavours. In some applications it is crucial for each party involved to be able to obtain guarantees about the correctness of a computation result without relying on trust in other parties. In other scenarios it may also be a requirement that one party proves somehow to another that a result is correct. In particular, this is the case if outsiders who did not participate in the computation are interested in its results. There are several distinct scenarios possible: the set of "outsiders" may be known or unknown prior to the computation, or it may be desirable that anyone can check the result of the computation, "for the common good". A classical example of this latter scenario is e-voting. In addition, one may want that any party that participated in the computation should able to prove to an external authority that they are using the correct result of the computation. For example, medical researchers who publish statistical analysis results on patient data should be able to prove correctness of the published data to an external party. Hence, in such cases, a securely outsourced computation should be verifiable, meaning that a party that was involved in the computation should be able to efficiently check that it was performed correctly, and possibly efficiently transfer this guarantee to third parties.

In light of the above discussion, we distinguish between universal verifiability (also known as public verifiability) where anybody can perform this check and designated verifiability where only a specific set of parties, decided before the computation, can perform the check. We stress that, in its strongest form, the requirement of verifiability goes beyond the trust assumptions that parties in the computation place in each other: the verifier should be sure that, even if all parties involved in the computation may attempt to cheat, the computation was still correctly performed.

Deliverable D12.1 of WP12 (Application Scenarios and their Requirements) identified verifiability as a critical property in many target applications relevant for PRACTICE. This deliverable presents a series of protocols which advance the state of the art in providing efficient solutions to this problem that were developed by participants in PRACTICE WP13 .

1.1 Task description

This report is the second deliverable of work package WP13 (*Protocol Specification and Design*) in the PRACTICE project. The main task of WP13 is the specification and design of new protocols, which are intended to improve the state-of-the-art in secure multi-party computation, in directions that are most relevant to secure computation in the cloud.

We recall the description of this deliverable in Annex I of the PRACTICE project proposal:

Task 13.2 – Efficient verifiability and precise specification of secure computation functionalities

> Efficient (universal) verifiability of secure computations: The verifiability of the result of a secure computation is often restricted to the parties taking part in the computation itself. However, in general, verifiability should also be provided to other parties as well. For instance, if secure linear programming is outsourced by several companies to several service providers, the companies need an efficient way to verify the results produced by the service providers. The research question w.r.t. verifiability is to (a) optimize efficiency and (b) to set the scope for the verification, i.e., who will be convinced. For (a) related notions are PCPs (Probabilistically Checkable Proofs), certificates of primality, etc. For (b) one can consider for example the protocol partners as the scope, but it is more challenging to consider any verifier (parties not taking part in the secure computation) as the scope. The latter case corresponds to potentially 'all dishonest' parties (rather than 'just' a dishonest majority).
>
> Precise specification of secure computation functionalities: The typical approach to defining the security goals of multi-party computation protocols is to specify an ideal functionality, and then declare that a concrete protocol is secure when it "realizes" this functionality. Although this approach is intuitive at the high-level, the path to obtaining meaningful results can be quite challenging and error-prone. This is a natural application area for formal methods techniques. We will extend, adapt, and apply formal methods to the task of specifying and analysing the security guarantees offered by secure computation protocols developed in the PRACTICE project. For example, one critical aspect that can be addressed is to apply formal logics to the validation of the ideal functionalities themselves.

Deliverable D13.2 – Efficient verifiability and precise specification of secure computation functionalities

> This deliverable is the outcome of Tasks 1.3.2. It will provide techniques for generating proofs that enable to verify the outcome of secure computation.

1.2 Overview

Tools such as language/compiler-based application frameworks, databases, application servers, verification tools and deployment tools are being combined within the PRACTICE project into an end-to-end secure computation software stack. These tools are to be used for the implementation and deployment of practical solutions for application scenarios described in D12.2, following the specified functional and security requirements. Many application requirements refer to the need for verifying computational correctness. Typically, verifiability is restricted to the participants running the protocol, however many applications (Platform for Auctions, Tax Fraud Detection, Joint Statistical Analysis Between State Entities, Privacy Preserving Genome-Wide Association Studies Between Biobanks, Privacy Preserving Personal Genome Analyses and Studies and Privacy Preserving Satellite Collision Detection) require for verifiability to be extended for other parties as well. These factors raise two main research goals regarding verifiability: to optimize the efficiency of verification techniques and to extend the verification scope, so that end-users can reduce the level of trust that they deposit in other parties that they interact with.

This report presents four new cryptographic protocols that permit realizing verifiable secure computation. For each protocol, we give a precise specification of the functionality it realizes, as well as a description of the protocol itself. Although we give some intuitive explanation about how the protocols satisfy the required properties, we do not include full security proofs in this report, as these are outside the scope of this deliverable. We note that all of the presented protocols are also presented as independently published papers/technical reports and we refer the interested reader to the full versions of those works instead.[1]

The report is divided into two parts that correspond to two alternative approaches to the problem of verifiability that offer different functionality, trust and efficiency tradeoffs for real-world applications: i. verifiable multiparty computation and ii. verifiable computation from hardware assumptions.

Part I: Verifiable Multiparty Computation In this first part we focus on secure outsourced computing based on secure multiparty computation (MPC). MPC enables multiple parties, each with its own private input, to compute some function of the combined data by engaging in an interactive protocol. Protocols securely realizing MPC should ensure, among other things, that the privacy of the input held by parties is not compromised and that the result each party obtains as output of the protocol is consistent with all inputs, as well as the output the other parties receive.

To securely outsource a computation to a single untrusted worker in a generic model of computation would require the use of so-called fully homomorphic encryption (FHE). At present, FHE can be considered prohibitively inefficient for use in most applications. As a practical solution, it is possible to construct protocols for secure outsourced computing from MPC by outsourcing the computation to not just one, but several independent untrusted workers who are assumed not to collude. When one or more clients wish to outsource some computation, they can do so by distributing their inputs among the workers in such a way that the workers cannot obtain any information about the inputs unless they collude with other workers. The workers then interact with each other, following the MPC protocol to carry out the computation and return

[1] Note that the notation used in the following two chapters is based on their respective source material. As each addresses different concerns, the notation is not fully unified across chapters.

the results to the clients. As mentioned above, the advantage of the MPC based approach over the use of FHE is that it is more efficient and therefore more practically relevant.

In the setting of MPC in which every party communicates interactively with every other party, there is no meaningful way to define security for the situation in which every party is corrupt. However, if we use MPC protocols as a basis for secure outsourced computing, we now have to consider the implications for the clients in case all workers collude to undermine the clients' security. We can directly argue that in this situation it is impossible for a protocol to offer input privacy without resorting to FHE, as such a protocol would necessarily satisfy the required properties of an FHE scheme. However, it is possible to ensure correctness of the result even under the worst possible worker corruption. This can be done by either enhancing the existing mechanisms that are already present in the underlying MPC protocol to ensure the correctness of the result to protocol participants, or by turning to means from the field of verifiable computing, in which this problem is studied independently. By ensuring that the results of a computation cannot be tampered with even if all workers are corrupt, we limit the impact of the worst case scenario.

We will discuss two protocols that have been developed as part of the PRACTICE project to realize secure outsourced computing based on verifiable multiparty computation. The material is based on two papers and we refer the interested reader to the full versions for details. The first protocol [71], which we call UVCDN, is based on a specific MPC protocol due to [26, 29] and follows the approach of enhancing the built-in proofs of correctness to be convincing to anyone, not just protocol participants. Therefore, the resulting protocol achieves universal verifiability. Both the underlying MPC protocols and the enhanced proofs of correctness require the use of cryptographic schemes which satisfy linear homomorphic properties. This is a much weaker requirement that full homomorphism and much more efficient schemes exist than for FHE.

For the second protocol [72], we use the approach of layering a Verifiable Computation (VC) scheme on top of an MPC protocol. Although there is some freedom in the choice of the underlying MPC protocol, we use specific properties of the VC scheme of [62] to make the resulting protocol efficient and achieve the desired security properties. This second protocol, named Trinocchio, ensures that even if all workers are corrupt, the protocol participants can verify the correctness of the results. Making the protocol universally verifiable, i.e., enabling *any* party to verify the correctness is part of ongoing work. To highlight the efficiency of the Trinocchio protocol, we will discuss two case studies, including some performance figures.

Part II: Verifiable Computation From Hardware Assumptions The second part of this report deals with another possibility to base secure outsourced computing on, namely trusted secure hardware. Modern trusted hardware technologies have been examined in the context Task 13.1. The main goal for this approach is to produce solutions that rely on specific hardware developments to meet more demanding levels of performance and security, that would not otherwise be feasible given the current theoretical and technological developments. It stands to reason that this latter approach will allow for greater efficiency, but it requires the use of trusted hardware that is still under active development and not yet commercially available, whereas protocols based on MPC can be implemented on commodity hardware. Furthermore, the use of trusted hardware requires the user to fully trust the designer and manufacturer of the hardware, and it brings with it a significant challenge of precisely modeling and formalizing the exact functionality an guarantees that are offered by highly complex devices that were developed by practically-minded developers in an industrial environment.

This report presents two protocols for secure verifiable computation that take advantage of emerging secure hardware architectures—isolated-execution environments (IEE)—allowing for the execution of arbitrary code within environments completely isolated from the rest of the system. We begin by presenting precise definitions of what IEE-enabled architectures offer, in the style of cryptographic provable security, so that higher-level secure computation protocols can be formalized and proven secure using the guarantees offered by such architectures as formal hardware assumptions. This modelling effort was one of the major contributions of the work published in [8].

We then rely on the above formalization to design and formally prove two secure protocols that permit carrying out verifiable computation in two different scenarios. The first scenario is one where a single Client wishes to outsource computations to an untrusted Worker with full confidentiality and integrity guarantees. The second scenario is one where multiple Clients wish to collaborate to jointly evaluate a computation on their respective (secret) inputs. In both cases, the designs are natural and formally justify how to leverage the extra guarantees offered by the trusted hardware to obtain solutions to existing problems that offer better efficieny and security trade-offs than previous solutions (in theory for now, but getting practical results for these protocols is ongoing work).

A final challenge that we set out to tackle when we initiated the PRACTICE project was to address the problem that the hand-made formalizations and descriptions that cryptographers produce of the functionalities offered by secure computation protocols rapidly become too complex and error-prone. We conclude this report with an application of formal methods techniques to the validation of our new approach to the analysis of advanced hardware-based assumptions. These results are important for two reasons. On the one hand, they show that our approach has been fine-tuned so as to allow rigorous specification and validation—we note that this leaves open the question of whether our abstractions are sound models of the underlying hardware assumptions, which we believe is an important question raised by our work and which will be answered in the coming years by follow up work. On the other hand, these results confirm that today's formal verification technologies are reaching a level of maturity that allow for an increasing interaction between the cryptographic community and the formal methods community that has been evident in many works that have emerged in the last years, and also by parallel work being carried out in other PRACTICE work packages.

1.3 Structure of the report

The structure of this document is as follows. After this introductory chapter, we begin with Part I: Verifiable Multiparty Computation. In Chapter 2 we present a construction of verifiable multiparty computation from threshold homomorphic cryptosystems and, in Chapter 3, we present the Trinocchio system, which provides privacy-preserving outsourcing of computation via distributed verifiable computing. The second part of the deliverable, Part II: Verifiable Computation from Hardware Assumptions then follows. In Chapter 4 we give a formalization of isolated execution environments and introduce attested computation. Then, in Chapter 5, we use this primitive to construct a secure outsourced computation protocol and, in Chapter 6, we use it to achieve secure function evaluation. Finally, we present our work in formally specifying and verifying protocols relying on this new hardware assumption in Chapter 7. We conclude the report with some final remarks in Chapter 8.

Part I

Verifiable Multiparty Computation

Chapter 2

Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems

2.1 Introduction

Multiparty computation (MPC) provides techniques for privacy-friendly outsourcing of computations. Intuitively, MPC aims to provide a cryptographic "black box" which receives private inputs from multiple "input parties"; performs a computation on these inputs; and provides the result to a "result party" (an input party, any third party, or the public). This black box is implemented by distributing the computation between multiple "computation parties", with privacy and correctness being guaranteed in case of passive corruptions (e.g., [17]), active corruption of a minority of computation parties (e.g., [26]), or active corruption of all-but-one computation parties (e.g., [30]).

However, multiparty computation typically does *not* provide any guarantees in case all computation parties are corrupted. That is, the result party has to trust that at least some of the computation parties did their job, and has no way of independently verifying the result. In particular, the result party has no way of proving to an external party that his computation result is indeed correct. *Universally verifiable* multiparty computation addresses these issues by requiring that the correctness of the result can be verified by any party, even if all computation parties are corrupt [32]. It was originally introduced in the context of e-voting [24, 65], but it is relevant whenever MPC is applied in a setting where not all of the parties that provide inputs or obtain outputs are participants in the computation. In particular, apart from contexts like e-voting where "the public" or an external watchdog wants to be sure of correctness, it is also useful in scenarios where (many) different input parties outsource a computation to the cloud and require a correctness guarantee.

Unfortunately, the state-of-the-art on universally verifiable MPC is unsatisfactory. The concept of universally verifiable MPC was first proposed in [32], where it was also suggested that it can be achieved for MPC based on threshold homomorphic cryptosystems. However, [32] does not provide a rigorous security model for universal verifiability or analysis of the proposed construction; and the construction has some technical disadvantages (e.g., a proof size depending on the number of computation parties). The scheme recently proposed in [9] solves part of the problem. Their protocols provide "public auditability", meaning that anybody can verify the

result of a computation, but *only* if that result is public. In particular, it is not possible for a result party to prove just that an encryption of the result is correct, which is important if this result is to be used in a later protocol without being revealed.

In this chapter, we propose a new security model for universally verifiable multiparty computation, and a practical construction achieving it. As in [32], we adapt the well-known actively secure MPC protocols based on threshold homomorphic cryptosystems from [26, 29]. Essentially, these protocols perform computations on encrypted values; security against active adversaries is achieved by letting parties prove correctness of their actions using interactive zero-knowledge proofs. Such interactive proofs only convince parties present at the computation; but making them non-interactive makes them convincing also to external parties. Concretely, the result of a computation is a set of encryptions of the inputs, intermediate values, and outputs of the computation, along with non-interactive zero-knowledge proofs of their correctness. Correctness of the result depends just on the correct set-up of the cryptosystem. Privacy holds under the original conditions of [26], i.e., if under half of the computation parties are corrupted; but as we discuss, this threshold can be raised to $n-1$ at the expense of sacrificing robustness. (Note that when computing with encryptions, we cannot hope to achieve privacy if all computation parties are corrupted: this would essentially require fully homomorphic encryption.)

We improve on [32] in two main ways. First, we provide a security model for universal verifiability (in the random oracle model), and security proofs for our protocols in that model. Second, we propose a new "multiparty" variant of the Fiat-Shamir heuristic to make the zero-knowledge proofs non-interactive, which may be of independent interest. Compared to [32], it eliminates the need for trapdoor commitments. Moreover, it makes the proof size independent of the number of parties performing the computation. We achieve this latter advantage by homomorphically combining contributions from the different parties.

As such, universally verifiable MPC provides a practical alternative to recent (single-party) techniques for verifiable outsourcing. Specifically, many papers on verifiable computation focus on efficient verification, but do not cover privacy [61, 76]. Those works that do provide privacy, achieve this by combining costly primitives, e.g., fully homomorphic encryption with verifiable computation [35]; or functional encryption with garbled circuits [44]. A recent work [4] also considers the possibility of achieving verifiable computation with privacy by distributing the computation; but it does not guarantee correctness if all computation parties are corrupted, nor does it allow third parties to be convinced of this fact. In contrast, our methods guarantee correctness even if all computation parties are corrupted, and even convince other parties than the input party. In particular, any third party can be convinced, and the computation may involve the inputs of multiple mutually distrusting input parties. Moreover, in contrast to the above works, our methods rely on basic cryptographic primitives such as Σ-protocols and the threshold homomorphic Paillier cryptosystem, readily available nowadays in cryptographic libraries like SCAPI [34].

Outline First, we briefly recap the CDN scheme for secure multiparty computation in the presence of active adversaries from [26, 29], instantiated using Paillier encryption (Section 2.2). Then, we show how the proofs in this protocol can be made non-interactive using the Fiat-Shamir heuristic and our new multiparty variant (Section 2.3). Finally, we propose a security model for universally verifiable MPC, and show that CDN with non-interactive proofs is universally verifiable (Section 2.4). We conclude in Section 2.5. We list potentially non-obvious notation in our pseudocode in Figure 2.1.

$a \in_R S$	sample a uniformly at random from S
$\mathsf{send}(v; \mathcal{P}), \mathsf{recv}(\mathcal{P})$	send v to/receive from $\mathcal{P}$ over secure channel
$\mathsf{bcast}(v)$	exchange v over broadcast channel
party $\mathcal{P}$ **do** S	let party $\mathcal{P}$ perform S; other parties do nothing
parties $i \in \mathcal{Q}$ **do** S	let parties $i \in \mathcal{Q}$ perform S in parallel
$\mathcal{H} : \{0,1\}^* \to \{0,1\}^{2l}$	cryptographic hash function (l security parameter)
$F \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R}, \mathcal{V}\}$	global variable: set of parties found to misbehave
$\mathsf{paillierdecode}(x)$	threshold Paillier decoding: $((x-1) \div N)(4\Delta^2)^{-1} \bmod N$
$\mathsf{fsprove}(\Sigma; v; w; aux)$	Fiat-Shamir proof (p. 12): $(a, s) := \Sigma.\mathsf{ann}(v, w)$; $c := \mathcal{H}(v\|\|a\|\|aux); r := \Sigma.\mathsf{res}(v, w, a, s, c); \pi := (a, c, r)$
$\mathsf{fsver}(\Sigma; v; a, c, r; aux)$	verification of Fiat-Shamir Σ-proof (p. 12): $\mathcal{H}(v\|\|a\|\|aux) = c \wedge \Sigma.\mathsf{ver}(v; a; c; r)$

Figure 2.1: Notation in algorithms, protocols, and processes

2.2 Secure Computation from Threshold Cryptography

We review the "CDN protocol" [26] for secure computation in the presence of active adversaries based on a threshold homomorphic cryptosystem. The protocol involves m input parties $i \in \mathcal{I}$, n computation parties $i \in \mathcal{P}$, and a result party $\mathcal{R}$. The aim of the protocol is to compute a function $f(x_1, \ldots, x_m)$ (seen as an arithmetic circuit) on private inputs x_i of the input parties, such that the result party obtains the result.

2.2.1 Computation using a Threshold Homomorphic Cryptosystem

The protocol uses a (t, n)-threshold homomorphic cryptosystem, with $t = \lceil n/2 \rceil$. In such a cryptosystem, anybody can encrypt a plaintext using the public key; add two ciphertexts to obtain a (uniquely determined) encryption of the sum of the corresponding plaintexts; and multiply a ciphertext by a constant to obtain a (uniquely determined) encryption of the product of the plaintext with the constant. Decryption is only possible if at least t out of the n decryption keys are known. A well-known homomorphic cryptosystem is the Paillier cryptosystem [60]: here, the public key is an RSA modulus $N = pq$; $a \in \mathbb{Z}_N$ is encrypted with randomness $r \in \mathbb{Z}_N^*$ as $(1+N)^a r^N \in \mathbb{Z}_{N^2}^*$; and the product of two ciphertexts is an encryption of the sum of the two corresponding plaintexts. (From now on, we suppress moduli for readability.) A threshold variant of this cryptosystem was presented in [28]. The (threshold) decryption procedure is a bit involved; we postpone its discussion until Section 2.2.2. The CDN protocol can also be instantiated with other cryptosystems; but in this chapter, we will focus on the Paillier instantiation.

Computation of $f(x_1, \ldots, x_m)$ is performed in three phases: the input phase, the computation phase, and the output phase. In the input phase, each input party encrypts its input x_i, and broadcasts the encryption X_i. In the computation phase, the function f is evaluated gate-by-gate. Addition and subtraction are performed using the homomorphic property of the encryption scheme. For multiplication of X and Y, each computation party $i \in \mathcal{P}$ chooses a random value d_i, and broadcasts encryptions D_i of d_i and E_i of $d_i \cdot y$. The computation parties then compute $X \cdot D_1 \cdots D_n$, and threshold decrypt it to learn $x + d_1 + \ldots + d_n$. Observe that this allows them to compute an encryption of $(x + d_1 + \ldots + d_n) \cdot y$, and hence, using the E_i,

also an encryption of $x \cdot y$. Finally, in the output phase, when the result of the computation has been computed as encryption X of x, the result party obtains x by broadcasting random encryption D of d and obtaining a threshold decryption $x - d$ of $X \cdot D^{-1}$.

Active security is achieved by letting the parties prove correctness of all information they exchange. Namely, the input parties prove knowledge of their inputs X_i (this prevents parties from choosing inputs depending on other inputs). The computation parties prove knowledge of D_i, and prove that E_i is indeed a correct multiplication of D_i and Y; and they prove the correctness of their contributions to the threshold decryption of $X \cdot D_1 \cdots D_n$ and $X \cdot D^{-1}$. Finally, the result party proves knowledge of D. We now discuss these proofs of correctness and their influence on the security of the overall protocol.

2.2.2 Proving Correctness of Results

The techniques in the CDN protocol for proving correctness are based on Σ-protocols. Recall that a Σ-protocol for a binary relation R is a three-move protocol in which a potentially malicious prover convinces an honest verifier that he knows a *witness* w for *statement* v such that $(v; w) \in R$. First, the prover sends an *announcement* (computed using algorithm $\Sigma.\mathsf{ann}$) to the verifier; the verifier responds with a uniformly random *challenge*; and the prover sends his *response* (computed using algorithm $\Sigma.\mathsf{res}$), which the verifier verifies (using predicate $\Sigma.\mathsf{ver}$). Σ-protocols satisfy the following properties:

Definition 1. *Let $R \subset V \times W$ be a binary relation and $L_R = \{v \in V \mid \exists w \in W : (v; w) \in R\}$ its language. Let Σ be a collection of probabilistic polynomial time algorithms $\Sigma.\mathsf{ann}$, $\Sigma.\mathsf{res}$, $\Sigma.\mathsf{sim}$, $\Sigma.\mathsf{ext}$, and polynomial time predicate $\Sigma.\mathsf{ver}$. Let C be a finite set called the* challenge space. *Then Σ is a* Σ-protocol *for relation R if:*

Completeness *If $(a; s) \leftarrow \Sigma.\mathsf{ann}(v; w)$, $c \in C$, and $r \leftarrow \Sigma.\mathsf{res}(v; w; a; s; c)$, then $\Sigma.\mathsf{ver}(v; a; c; r)$.*

Special soundness *If $v \in V$, $c \neq c'$, $\Sigma.\mathsf{ver}(v; a; c; r)$, and $\Sigma.\mathsf{ver}(v; a; c'; r')$, then $w \leftarrow \Sigma.\mathsf{ext}(v; a; c; c'; r; r')$ satisfies $(v; w) \in R$.*

Special honest-verifier zero-knowledgeness *If $v \in L_R$, $c \in C$, then $(a; r) \leftarrow \Sigma.\mathsf{sim}(v; c)$ has the same probability distribution as $(a; r)$ obtained by $(a; s) \leftarrow \Sigma.\mathsf{ann}(v; w)$, $r \leftarrow \Sigma.\mathsf{res}(v; w; a; s; c)$. If $v \notin L_R$, then $(a; r) \leftarrow \Sigma.\mathsf{sim}(v; c)$ satisfies $\Sigma.\mathsf{ver}(v; a; c; r)$.*

Completeness states that a protocol between an honest prover and verifier succeeds; special soundness states that there exists an extractor $\Sigma.\mathsf{ext}$ that can extract a witness from two conversations with the same announcement; and special honest-verifier zero-knowledgeness states that there exists a simulator $\Sigma.\mathsf{sim}$ that can generate conversations with the same distribution as full protocol runs without knowing the witness. While special honest-verifier zero-knowledgeness demands an identical distribution for the simulation, statistical indistinguishability is sufficient for our purposes; in this case, we speak of a "statistical Σ-protocol". In the remainder, we will need that our Σ-protocols have "non-trivial announcements", in the sense that when $(a; r)$ and $(a'; r')$ are both obtained from $\Sigma.\mathsf{sim}(v; c)$, then with overwhelming probability, $a \neq a'$. (Indeed, this will be the case for all Σ-protocols in this chapter.) This property, which is required by the Fiat-Shamir heuristic [2], essentially follows from the hardness of the relation; see [71] for details.

The CDN protocol uses a sub-protocol in which multiple parties simultaneously provide proofs based on the same challenge, called the "multiparty Σ-protocol". Namely, suppose each party from a set P wants to prove knowledge of a witness for a statement $v_i \in L_R$ with some Σ-protocol. To achieve this, each party in P broadcasts a commitment[1] to its announcement; then, the computation parties jointly generate a challenge; and finally, all parties in P broadcast their response to this challenge, along with an opening of their commitment. The multiparty Σ-protocol is used as a building block in the CDN protocol by constructing a simulator that provides proofs on behalf of honest parties without knowing their witnesses ("zero-knowledgeness"), and extracts witnesses from corrupted parties that give correct proofs ("soundness").

The CDN protocol uses three Σ-protocols proving plaintext knowledge, correct multiplication and correct decryption. These proofs directly correspond with each of the steps in the multiplication protocol that involves communication between parties.

2.2.3 Security of the CDN Protocol

In [26], it is shown that the CDN protocol implements secure function evaluation in Canetti's non-concurrent model [22] if only a minority of computation parties are corrupted. Essentially, this means that in this case, the computation succeeds; the result is correct; and the honest parties' inputs remain private. This conclusion is true assuming honest set-up and security of the Paillier encryption scheme and the trapdoor commitment scheme used. If a majority of computation parties is corrupted, then because threshold $\lceil n/2 \rceil$ is used for the threshold cryptosystem, privacy is broken. As noted [69, 51], this can be remedied by raising the threshold, but in that case, the corrupted parties can make the computation break down at any point by refusing to cooperate. In Section 2.4.1, we present a variant of this model in which we prove the security of our protocols (using random oracles but no trapdoor commitments).

2.3 Multiparty Non-Interactive Proofs

In this section, we show how to produce non-interactive zero-knowledge proofs in a multiparty way. At several points in the above CDN protocol, all parties from a set P prove knowledge of witnesses for certain statements; the computation parties are convinced that those parties that succeed, do indeed know a witness. In CDN, these proofs are interactive; but for universal verifiability, we need non-interactive proofs that convince any third party. The traditional method to make proofs non-interactive is the Fiat-Shamir heuristic; in Section 2.3.1, we outline it, and show that it is problematic in a multiparty setting. In Section 2.3.2, we present a new, "multiparty" Fiat-Shamir heuristic that works in our setting, and has the advantage of achieving smaller proofs by "homomorphically combining" the proofs of individual parties. In the remainder, $C \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R}, \mathcal{V}\}$ denotes the set of corrupted parties; and F denotes the set of parties who failed to provide a correct proof when needed; this only happens for corrupted parties, so $F \subset C$.

[1] A commitment scheme is a cryptographic scheme that let's parties commit to some value such that it is not possible to later change that value, but without revealing the value to other parties. These properties are somewhat similar to those of an encryption scheme, and efficient instantiations of commitment schemes based on encryption exist. One of the typical uses for commitments is, if a protocol calls for multiple parties to communicate information, to prevent parties from choosing their messages based on the messages they've already observed. This is done by first letting all parties commit to their message, and only after each commitment has been published revealing the actual message.

Our results are in the random oracle model [10, 77], an idealised model of hash functions. In this model, evaluations of the hash function $\mathcal{H}$ are modelled as queries to a "random oracle" $\mathcal{O}$ that evaluates a perfectly random function. When simulating an adversary, a simulator can intercept these oracle queries and answer them at will, as long as the answers look random to the adversary. Security in the random oracle model does not generally imply security in the standard model [43], but it is often used because it typically gives simple, efficient protocols, and its use does not seem to lead to security problems in practice [77]. See [71] for a detailed description of our use of random oracles; and Section 2.5 for a discussion of the real-world implications of the particular flavour of random oracles we use.

2.3.1 The Fiat-Shamir Heuristic

The obvious way of making the proofs in the CDN protocol non-interactive, is to apply the Fiat-Shamir heuristic to all individual Σ-protocols. That is, party $i \in P$ produces proof of knowledge π of a witness for statement v as follows[2]:

$$(a; s) := \Sigma.\mathsf{ann}(v; w); c := \mathcal{H}(v||a||aux); r := \Sigma.\mathsf{res}(v; w; a; s; c); \pi := (a; c; r).$$

Let us denote this procedure $\mathsf{fsprove}(\Sigma; v; w; aux)$. A verifier accepts those proofs $\pi = (a; c; r)$ for which $\mathsf{fsver}(\Sigma; v; \pi; aux)$ holds, where $\mathsf{fsver}(\Sigma; v; a, c, r; aux)$ is defined as $\mathcal{H}(v||a||aux) = c \wedge \Sigma.\mathsf{ver}(v; a; c; r)$.

Recall that security proofs require a simulator that simulates proofs of honest parties (zero-knowledgeness) and extracts witnesses of corrupted parties (soundness). In the random oracle model, Fiat-Shamir proofs for honest parties can be simulated by simulating a Σ-protocol conversation (a, c, r) and programming the random oracle so that $\mathcal{H}(v||a||aux) = c$. Witnesses of corrupted parties can be extracted by rewinding the adversary to the point where it made an oracle query for $v||a||aux$ and supplying a different value. However, as the protocol consists of multiple rounds and the adversary is invoked multiple times, it is possible for the adversary to query the oracle in an earlier invocation than it uses the response. In that case, to extract the witness, the adversary should be rewound to the point the query was made. This may in turn require recursive rewinding to extract the witnesses for intermediate rounds. In fact, if Fiat-Shamir proofs take place in R different rounds, then extracting witnesses may increase the running time of the simulator by a factor $O(R!)$. Hence, we can essentially only use the Fiat-Shamir heuristic in a constant number of rounds.

Moreover, in the CDN protocol, applying the Fiat-Shamir heuristic to each individual proof has the disadvantage that the verifier needs to check a number of proofs that depends linearly on the number of computation parties. In particular, for each multiplication gate, the verifier needs to check n proofs of correct multiplication and t proofs of correct decryption. Next, we show that we can avoid both the technical problems and the dependence on the number of computation parties by letting the computation parties collaboratively produce "combined proofs".

[2]Here, aux should contain at least the prover's identity. Otherwise, corrupted parties could replay proofs by honest parties, which breaks the soundness property below because witnesses for these proofs cannot be extracted by rewinding the adversary to the point of the oracle query and reprogramming the random oracle.

2.3.2 Combined Proofs with the Multiparty Fiat-Shamir Heuristic

The crucial observation (e.g., [33, 52]) allowing parties to produce non-interactive zero-knowledge proofs collaboratively is that, for many Σ-protocols, conversations of proofs with the same challenge can be "homomorphically combined". For instance, consider the classical Σ-protocol for proving knowledge of a discrete logarithm due to Schnorr [66]. Suppose we have two Schnorr conversations proving knowledge of $x_1 = \log_g h_1$, $x_2 = \log_g h_2$, i.e., two tuples $(a_1; c; r_1)$ and $(a_2; c; r_2)$ such that $g^{r_1} = a_1(h_1)^c$ and $g^{r_2} = a_2(h_2)^c$. Then $g^{r_1+r_2} = (a_1a_2)(h_1h_2)^c$, so $(a_1a_2; c; r_1 + r_2)$ is a Schnorr conversation proving knowledge of discrete logarithm $x_1 + x_2 = \log_g(h_1h_2)$. For our purposes, we demand that such homomorphisms satisfy two properties. First, when conversations of at least $\lceil n/2 \rceil$ parties are combined, the result is a valid conversation (the requirement of having at least $\lceil n/2 \rceil$ conversations is needed for decryption proofs to ensure that there are enough decryption shares). Second, when fewer than $\lceil n/2 \rceil$ parties are corrupted, the combination of different honest announcements with the same corrupted announcements is likely to lead to a different combined announcement. This helps to eliminate the rewinding problems for Fiat-Shamir discussed above.

Definition 2. *Let Σ be a Σ-protocol for relation $R \subset V \times W$. Let Φ be a collection of partial functions $\Phi.\mathsf{stmt}$, $\Phi.\mathsf{ann}$, and $\Phi.\mathsf{resp}$. We call Φ a* homomorphism *of Σ if:*

Combination *Let c be a challenge; I a set of parties such that $|I| \geq \lceil n/2 \rceil$; and $\{(v_i; a_i; r_i)\}_{i \in I}$ a collection of statements, announcements, and responses. If $\Phi.\mathsf{stmt}(\{v_i\}_{i \in I})$ is defined and $\Sigma.\mathsf{ver}(v_i; a_i; c; r_i)$ holds for all i, then also $\Sigma.\mathsf{ver}(\Phi.\mathsf{stmt}(\{v_i\}_{i \in I}); \Phi.\mathsf{ann}(\{a_i\}_{i \in I}); c; \Phi.\mathsf{resp}(\{r_i\}_{i \in I}))$.*

Randomness *Let c be a challenge; $C \subset I$ sets of parties such that $|C| < \lceil n/2 \rceil \leq |I|$; $\{v_i\}_{i \in I}$ statements s.t. $\Phi.\mathsf{stmt}(\{v_i\}_{i \in I})$ is defined; and $\{a_i\}_{i \in I \cap C}$ announcements. If $(a_i; \cdot), (a'_i; \cdot) \leftarrow \Sigma.\mathsf{sim}(v_i; c)$ $\forall i \in I \setminus C$, then with overwhelming probability, $\Phi.\mathsf{ann}(\{a_i\}_{i \in I}) \neq \Phi.\mathsf{ann}(\{a_i\}_{i \in I \cap C} \cup \{a'_i\}_{i \in I \setminus C})$.*

Given a Σ-protocol with homomorphism Φ, parties holding witnesses $\{w_i\}$ for statements $\{v_i\}$ can together generate a Fiat-Shamir proof $(a; \mathcal{H}(v||a||aux); r)$ of knowledge of a witness for the "combined statement" $v = \Phi.\mathsf{stmt}(\{v_i\})$. Namely, the parties each provide announcement a_i for their own witness; compute $a = \Phi.\mathsf{ann}(\{a_i\})$ and $\mathcal{H}(v||a||aux)$; and provide responses r_i. Taking $r = \Phi.\mathsf{resp}(\{r_i\})$, the combination property from the above definition guarantees that we indeed get a validating proof. However, we cannot simply let the parties broadcast their announcements in turn, because to prove security in that case, the simulator needs to provide the announcements for the honest parties without knowing the announcements of the corrupted parties, hence without being able to program the random oracle on the combined announcement. We solve this by starting with a round in which each party commits to its announcement.

The *multiparty Fiat-Shamir heuristic* (Protocol 1) let parties collaboratively produce Fiat-Shamir proofs based on the above ideas. Apart from the above procedure (lines 8, 9, 10, 13, and 14), the protocol also contains error handling. Namely, we throw out parties that provide incorrect hashes to their announcements (line 11) or incorrect responses (line 15). If we have correct responses for all correctly hashed announcements, then we apply the homomorphism (line 17–18); otherwise, we try again with the remaining parties. If the number of parties drops below $\lceil n/2 \rceil$, the homomorphism can no longer be applied, so we return with an error (line

Protocol 1 MΣ: The Multi-Party Fiat-Shamir Heuristic

1. // **pre**: Σ is a Σ-protocol with homomorphism Φ, P is a set of non-failed
2. // parties ($P \cap F = \emptyset$), $v_P = \{v_i\}_{i \in P}$ statements w/ witnesses $w_P = \{w_i\}_{i \in P}$
3. // **post**: if $|P \setminus F| \geq \lceil n/2 \rceil$, then $v_{P \setminus F}$ is the combined statement
4. // $\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F})$, and $\pi_{P \setminus F}$ is a corresponding Fiat-Shamir proof
5. // **invariant**: $F \subset C$: set of failed parties only includes corrupted parties
6. $(v_{P \setminus F}, \pi_{P \setminus F}) \leftarrow \mathrm{M}\Sigma(\Sigma, \Phi, P, v_P, w_P, aux) :=$
7. **repeat**
8. **parties** $i \in P \setminus F$ **do**
9. $(a_i; s_i) := \Sigma.\mathsf{ann}(v_i; w_i); h_i := \mathcal{H}(a_i || i); \mathsf{bcast}(h_i)$
10. **parties** $i \in P \setminus F$ **do** $\mathsf{bcast}(a_i)$
11. $F' := F; F := F \cup \{i \in P \setminus F \mid h_i \neq \mathcal{H}(a_i || i)\}$
12. **if** $F = F'$ **then** // all parties left provided correct hashes
13. $c := \mathcal{H}(\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F}) || \Phi.\mathsf{ann}(\{a_i\}_{i \in P \setminus F}) || aux)$
14. **parties** $i \in P \setminus F$ **do** $r_i := \Sigma.\mathsf{res}(v_i; w_i; a_i; s_i; c); \mathsf{bcast}(r_i)$
15. $F := F \cup \{i \in P \setminus F \mid \neg \Sigma.\mathsf{ver}(v_i; a_i; c; r_i)\}$
16. **if** $F = F'$ **then** // all parties left provided correct responses
17. **return** $(\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F}),$
18. $(\Phi.\mathsf{ann}(\{a_i\}_{i \in P \setminus F}); c; \Phi.\mathsf{resp}(\{r_i\}_{i \in P \setminus F})))$
19. **until** $|P \setminus F| < \lceil n/2 \rceil$ // until not enough parties left
20. **return** $(\perp, \perp)$

20). Note that, as in the normal Fiat-Shamir heuristic, the announcements do not need to be stored if they can be computed from the challenge and response (as will be the case for the Σ-protocols we consider).

Concerning security, recall that we need a simulator that simulates proofs of honest parties without their witnesses (zero-knowledgeness) and extracts the witnesses of corrupted parties (soundness). In [71], we present such a simulator. Essentially, it "guesses" the announcements of the corrupted parties based on the provided hashes; then simulates the Σ-protocol for the honest parties; and programs the random oracle on the combined announcement. It obtains witnesses for the corrupted parties by rewinding to just before the honest parties provide their announcements: this way, the corrupted parties are forced to use the announcements that they provided the hashes of (hence special soundness can be invoked), whereas the honest parties can provide new simulated announcements by reprogramming the random oracle. The simulator requires that fewer than $\lceil n/2 \rceil$ provers are corrupted so that we can use the randomness property of the Σ-protocol homomorphism (Definition 2). (When more than $\lceil n/2 \rceil$ provers are corrupted, we use an alternative proof strategy that uses witness-extended emulation instead of this simulator.)

2.4 Universally Verifiable MPC

In the previous section, we have shown how to produce non-interactive zero-knowledge proofs in a multiparty way. We now use this observation to obtain universally verifiable MPC. We first define security for universally verifiable MPC; and then obtain universally verifiable MPC by adapting the CDN protocol.

Process 2 $\mathcal{T}_{\text{VSFE}}$: trusted party for verifiable secure function evaluation

1. // compute f on $\{x_i\}_{i\in\mathcal{I}}$ for $\mathcal{R}$ with corrupted parties C; $\mathcal{V}$ learns encryption
2. $\mathcal{T}_{\text{VSFE}}(C, (N, v, v_0, \{v_i\}_{i\in\mathcal{P}})) :=$
3. // input phase
4. **foreach** $i \in \mathcal{I} \setminus C$ **do** $x_i := \mathsf{recv}(\mathcal{I}_i)$ // honest inputs
5. $\{x_i\}_{i\in\mathcal{I}\cap C} := \mathsf{recv}(\mathcal{S})$ // corrupted inputs
6. **if** $|\mathcal{P} \cap C| \geq \lceil n/2 \rceil$ **then** $\mathsf{send}(\{x_i\}_{i\in\mathcal{I}\setminus C}, \mathcal{S})$ // send to corrupted majority
7. // computation phase
8. $r := f(x_1, \ldots, x_m)$
9. // output phase
10. **if** $\mathcal{R} \notin C$ **then** // honest $\mathcal{R}$: adversary learns encryption, may block result
11. $s \in_R \mathbb{Z}_N^*$; $R := (1 + N)^r s^N$; $res := (r, s)$; $\mathsf{send}(R, \mathcal{S})$
12. **if** $|\mathcal{P} \cap C| \geq \lceil n/2 \rceil$ **and** $\mathsf{recv}(\mathcal{S}) = \perp$ **then** $res := \perp$; $R := \perp$
13. $\mathsf{send}(res, \mathcal{R})$
14. **else** // corrupted $\mathcal{R}$: adversary learns output, may block result to $\mathcal{V}$
15. $\mathsf{send}(r, \mathcal{S}); s := \mathsf{recv}(\mathcal{S})$
16. **if** $s = \perp$ **then** $R := \perp$ **else** $R := (1 + N)^r s^N$
17. // proof phase
18. **if** $\mathcal{V} \notin C$ **then** $\mathsf{send}(R, \mathcal{V})$

2.4.1 Security Model for Verifiable MPC

Our security model is an adaptation of the model of [22, 26] to the setting of universal verifiability in the random oracle model. We first explain the general execution model, which is as in [22, 26] but with a random oracle added; we then explain how to model verifiability in this execution model as the behaviour of the ideal-world trusted party. The general execution model compares protocol executions in the real and ideal world.

In the real world, a protocol π between m input parties $i \in \mathcal{I}$, n computation parties $i \in \mathcal{P}$, a result party $\mathcal{R}$ and a verifier $\mathcal{V}$ is executed on an open broadcast network with rushing in the presence of an active static adversary $\mathcal{A}$ corrupting parties $C \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R}, \mathcal{V}\}$. The protocol execution starts by incorruptibly setting up the Paillier threshold cryptosystem, i.e., generating public key $\mathsf{pk} = (N, v, v_0, \{v_i\}_{i\in\mathcal{P}})$ with RSA modulus N and verification values v, v_0, v_i, and secret key shares $\{s_i\}_{i\in\mathcal{P}}$ (see Section 2.2.2). Each input party $i \in \mathcal{I}$ gets input (pk, x_i); each computation party $i \in \mathcal{P}$ gets input (pk, s_i); and the result party $\mathcal{R}$ gets input pk. The adversary gets the inputs $(\mathsf{pk}, \{x_i\}_{i\in\mathcal{I}\cap C}, \{s_i\}_{i\in\mathcal{P}\cap C})$ of the corrupted parties, and has an auxiliary input a. During the protocol, parties can query the random oracle; the oracle answers new queries randomly, and repeated queries consistently. At the end of the protocol, each honest party outputs a value according to the protocol; the corrupted parties output $\perp$; and the adversary outputs a value at will. Define $\text{EXEC}_{\pi,\mathcal{A}}(k, (x_1, \ldots, x_m), C, a)$ to be the random variable, given security parameter k, consisting of the outputs of all parties (including the adversary) and the set $\mathcal{O}$ of oracle queries and responses.

The ideal-world execution similarly involves m input parties $i \in \mathcal{I}$, n computation parties $i \in \mathcal{P}$, result party $\mathcal{R}$, verifier $\mathcal{V}$, and an adversary $\mathcal{S}$ corrupting parties $C \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R}, \mathcal{V}\}$; but now, there is also an incorruptible trusted party $\mathcal{T}$. As before, the execution starts by setting up the keys $(\mathsf{pk}, \{s_i\}_{i\in\mathcal{P}})$ of the Paillier cryptosystem. The input parties receive x_i as input; the trusted party receives a list C of corrupted parties and the public key pk. Then, it runs the code $\mathcal{T}_{\text{VSFE}}$ shown in Process 2, which we explain later. The adversary gets inputs

$(\mathsf{pk}, C, \{x_i\}_{i\in\mathcal{I}\cap C}, \{s_i\}_{i\in\mathcal{P}\cap C})$, and outputs a value at will. In this model, there is no random oracle; instead, the adversary chooses the set $\mathcal{O}$ of oracle queries and responses (typically, those used to simulate a real-world adversary). As in the real-world case, $\text{IDEAL}_{\mathcal{T}_{\text{SFE}},\mathcal{S}}(k,(x_1,\ldots,x_m),C,a)$ is the random variable, given security parameter k, consisting of all parties' outputs and $\mathcal{O}$.

Definition 3. *Protocol* π implements verifiable secure function evaluation in the random oracle model *if, for every probabilistic polynomial time real-world adversary* $\mathcal{A}$*, there exists a probabilistic polynomial time ideal-world adversary* $\mathcal{S}_{\mathcal{A}}$ *such that, for all inputs* $x_1,\ldots,x_m$*; all sets of corrupted parties* C*; and all auxiliary input* a*:* $EXEC_{\pi,\mathcal{A}}(k;x_1,\ldots,x_m;C;a)$ *and* $IDEAL_{\mathcal{T}_{VSFE},\mathcal{S}_{\mathcal{A}}}(k;x_1,\ldots,x_m;C;a)$ *are computationally indistinguishable in security parameter* k*.*

We now discuss the trusted party $\mathcal{T}_{\text{VSFE}}$ for verifiable secure function evaluation. Whenever the computation succeeds, $\mathcal{T}_{\text{VSFE}}$ guarantees that the results are correct. Namely, $\mathcal{T}_{\text{VSFE}}$ sends the result r of the computation and randomness s to $\mathcal{R}$ (line 13), and it sends encryption $(1+N)^r s^N$ of the result with randomness s to $\mathcal{V}$ (line 18); if the computation failed, $\mathcal{R}$ gets $(\perp,\perp)$ and $\mathcal{V}$ gets $\perp$.[3] Whether $\mathcal{T}_{\text{VSFE}}$ guarantees privacy (i.e., only $\mathcal{R}$ can learn the result) and robustness (i.e., the computation does not fail) depends on which parties are corrupted. Privacy and robustness with respect to $\mathcal{R}$ are guaranteed as long as only a minority of computation parties are corrupted. If not, then in line 6, $\mathcal{T}_{\text{VSFE}}$ sends the honest parties' inputs to the adversary; and in line 12, it gives the adversary the option to block the computation by sending $\perp$. Note that the adversary receives the inputs of the honest parties after it provides the inputs of the corrupted parties, so even if privacy is broken, the adversary cannot choose the corrupted parties' inputs based on the honest parties' inputs. For robustness with respect to $\mathcal{V}$, the result party needs to be honest. If not, then in line 15, $\mathcal{T}_{\text{VSFE}}$ gives the adversary the option to block $\mathcal{V}$'s result by sending $\perp$; in any case, it can choose the randomness. (Note that these thresholds are specific to CDN's "honest majority" setting; e.g., other protocols may satisfy privacy if all computation parties except one are corrupted.)

Note that this model does not cover the "universality" aspect of universally verifiable MPC. This is because the security model for secure function evaluation only covers the input/output behaviour of protocols, not the fact that "the verifier can be anybody". Hence, we design universally verifiable protocols by proving that they are verifiable, and then arguing based on the characteristics of the protocol (e.g., the verifier does not have any secret values) that this verifiability is "universal".

2.4.2 Universally Verifiable CDN

We now present the UVCDN protocol (Protocol 3) for universally verifiable secure function evaluation. At a high level, this protocol consists of the input, computation, and multiplication phases of the CDN protocol, with all proofs made non-interactive, followed by a new *proof phase*. As discussed, we can use the normal Fiat-Shamir (FS) heuristic in only a constant number of rounds; and we can use the multiparty FS heuristic only when it gives a "combined

[3] Although we only guarantee computational indistinguishability and the verifier does not know what value is encrypted, this definition *does* guarantee that $\mathcal{V}$ receives the correct result. This is because the ideal-world output of the protocol execution contains $\mathcal{R}$'s r and s and $\mathcal{V}$'s $(1+N)^r s^N$, so a distinguisher between the ideal and real world can check correctness of $\mathcal{V}$'s result. (If s were not in $\mathcal{R}$'s result, this would not be the case, and correctness of $\mathcal{V}$'s result would *not* be guaranteed.) Also, note that although privacy depends on the security of the encryption scheme, correctness *does not rely on any knowledge assumption*.

Protocol 3 UVCDN: universally verifiable CDN

1. // **pre**: $\mathsf{pk}/\{s_i\}_{i\in\mathcal{P}}$ threshold Paillier public/secret keys, $\{x_i\}_{i\in\mathcal{I}}$ function input
2. // **post**: output R according to ideal functionality ITM 2
3. $R \leftarrow \text{UVCDN}(\mathsf{pk}=(N,v,v_0,\{v_i\}_{i\in\mathcal{P}}),\{s_i\}_{i\in\mathcal{P}},\{x_i\}_{i\in\mathcal{I}}) :=$
4. **parties** $i \in \mathcal{I}$ **do** // input phase
5. $r_i \in_R \mathbb{Z}_N^*; X_i := (1+N)^{x_i} r_i^N; \pi_{\mathrm{PK},i} := \mathsf{fsprove}(\Sigma_{\mathrm{PK}}; X_i; x_i, r_i; i)$
6. $h_i := \mathcal{H}(X_i||\pi_{\mathrm{PK},i}||i); \mathsf{bcast}(h_i); \mathsf{bcast}(X_i, \pi_{\mathrm{PK},i})$
7. $F := \{i \in \mathcal{I} \mid h_i \neq \mathcal{H}(X_i||\pi_{\mathrm{PK},i}||i) \vee \neg\mathsf{fsver}(\Sigma_{\mathrm{PK}}; X_i; \pi_{\mathrm{PK},i}; i)\}$
8. **foreach** $i \in F$ **do** $X_i := 1$
9. **foreach** gate **do** // computation phase
10. **if** $\langle$constant gate c with value $v\rangle$ **then** $X_c := (1+N)^v$
11. **if** $\langle$addition gate c with inputs a, $b\rangle$ **then** $X_c := X_a X_b$
12. **if** $\langle$subtraction gate c with inputs a, $b\rangle$ **then** $X_c := X_a X_b^{-1}$
13. **if** $\langle$multiplication gate c with inputs a, $b\rangle$ **then** // [29] multiplication
14. **parties** $i \in \mathcal{P} \setminus F$ **do**
15. $d_i \in_R \mathbb{Z}_N; r_i, t_i \in_R \mathbb{Z}_N^*; D_i := (1+N)^{d_i} r_i^N; E_i := (X_b)^{d_i} t_i^N$
16. $\mathsf{bcast}(D_i, E_i)$
17. $(\cdot, D_c, E_c; \pi_{\mathrm{CM}c}) :=$
18. $\mathrm{M}\Sigma(\Sigma_{\mathrm{CM}}, \Phi_{\mathrm{CM}}, \mathcal{P}\setminus F, \{(X_b, D_i, E_i)\}_{i\in\mathcal{P}\setminus F}, \{(d_i, r_i, t_i)\}_{i\in\mathcal{P}\setminus F})$
19. **if** $|\mathcal{P}\setminus F| < \lceil n/2 \rceil$ **then break**
20. $S_c := X_a \cdot D_c$
21. **parties** $i \in \mathcal{P}\setminus F$ **do** $S_i := (S_c)^{2\Delta s_i}; \mathsf{bcast}(S_i)$
22. $(\cdot, S_{0,c}, \cdot, \cdot; \pi_{\mathrm{CD}c}) :=$
23. $\mathrm{M}\Sigma(\Sigma_{\mathrm{CD}}, \Phi_{\mathrm{CD}}, \mathcal{P}\setminus F, \{(S_c, S_i, v, v_i)\}_{i\in\mathcal{P}\setminus F}, \{\Delta s_i\}_{i\in\mathcal{P}\setminus F})$
24. **if** $|\mathcal{P}\setminus F| < \lceil n/2 \rceil$ **then break**
25. $s := \mathsf{paillierdecode}(S_{0,c}); X_c := (X_b)^s \cdot E_c^{-1}$
26. **if** $|\mathcal{P}\setminus F| < \lceil n/2 \rceil$ **then parties** $i \in \mathcal{I}\cup\mathcal{P}\cup\{\mathcal{R}\}$ **do return** $\perp$
27. **party** $\mathcal{R}$ **do** $d \in_R \mathbb{Z}_N; s \in_R \mathbb{Z}_N^*; D := (1+N)^d s^N$ // output phase
28. **party** $\mathcal{R}$ **do** $\pi_{\mathrm{PK}d} := \mathsf{fsprove}(\Sigma_{\mathrm{PK}}; D; d, s; \mathcal{R}); \mathsf{bcast}(D, \pi_{\mathrm{PK}d})$
29. **if** $\neg\mathsf{fsver}(\Sigma_{\mathrm{PK}}; D; \pi_{\mathrm{PK}d}; \mathcal{R})$ **then parties** $i \in \mathcal{I}\cup\mathcal{P}\cup\{\mathcal{R}\}$ **do return** $\perp$
30. $Y := X_{\text{outgate}} \cdot D^{-1};$ **parties** $i \in \mathcal{P}\setminus F$ **do** $Y_i := Y^{2\Delta s_i}; \mathsf{bcast}(Y_i)$
31. $(\cdot, Y_0, \cdot, \cdot; \pi_{\mathrm{CD}}; y) := \mathrm{M}\Sigma(\Sigma_{\mathrm{CD}}, \Phi_{\mathrm{CD}}, \mathcal{P}\setminus F, \{(Y, Y_i, v, v_i)\}_{i\in\mathcal{P}\setminus F}, \{\Delta s_i\}_{i\in\mathcal{P}\setminus F}, D)$
32. **if** $|\mathcal{P}\setminus F| < \lceil n/2 \rceil$ **then parties** $i \in \mathcal{I}\cup\mathcal{P}\cup\{\mathcal{R}\}$ **do return** $\perp$
33. **party** $\mathcal{R}$ **do**
34. $y := \mathsf{paillierdecode}(Y_0); r := y + d$
35. $\mathsf{send}(\{(D_c, E_c, \Pi_{\mathrm{CM}c}, S_{0,c}, \Pi_{\mathrm{CD}c})\}_{c\in\text{gates}}, (D, \pi_{\mathrm{PK}d}, Y_0, \pi_{\mathrm{CD}y}); \mathcal{V})$ // proof
36. **return** (r, s) // phase
37. **parties** $i \in \mathcal{I}\cup\mathcal{P}$ **do return** $\perp$
38. **party** $\mathcal{V}$ **do** $\pi := \mathsf{recv}(\mathcal{R});$ **return** $\mathsf{vercomp}(\mathsf{pk}, \{X_i\}_{i\in\mathcal{I}}, \pi)$

statement" that makes sense. Hence, we choose to use the FS heuristic for the proofs by the input and result parties, and the multiparty FS heuristic for the proofs by the computation parties.

In more detail, during the *input phase* of the protocol, the input parties provide their inputs (lines 4–8). As in the CDN protocol, each party encrypts its input and compiles a FS proof of knowledge (line 5). In the original CDN protocol, these encryptions and proofs would be broadcast directly; however, if a majority of computation parties are corrupted, then this allows corrupted parties to adapt their inputs based on the inputs of the honest parties. To prevent

Algorithm 4 vercomp: verifier's gate-by-gate verification of the computation

1. // **pre**: pk public key, $\{X_i\}_{i\in\mathcal{I}}$ encryptions, $(\{\Pi_{\text{mul}i}\},\Pi_{\text{result}})$ tuple
2. // **post**: if $(\{\Pi_{\text{mul}i}\},\Pi_{\text{result}})$ proves correctness of Y, $X_o=Y$; otherwise, $X_o=\perp$
3. $X_o \leftarrow \mathsf{vercomp}(\mathsf{pk}=(N,v,v_0,\{v_i\}_{i\in\mathcal{P}}),\{X_i\}_{i\in\mathcal{I}},(\{\Pi_{\text{mul}i}\},\Pi_{\text{result}})):=$
4. // verification of input phase: see lines 6–8 of UVCDN
5. // verification of computation phase
6. **foreach** gate **do**
7. **if** $\langle$constant gate c with value $v\rangle$ **then** $X_c:=(1+N)^v$
8. **if** $\langle$addition gate c with inputs a, $b\rangle$ **then** $X_c:=X_aX_b$
9. **if** $\langle$subtraction gate c with inputs a, $b\rangle$ **then** $X_c:=X_aX_b^{-1}$
10. **if** $\langle$multiplication gate c with inputs a, $b\rangle$ **then**
11. $(D;E;a,c,r;S_0;a',c',r'):=\Pi_{\text{mul}c}$; $S:=X_a\cdot D^{-1}$
12. **if** $\neg\mathsf{fsver}(\Sigma_{\text{CM}};X_b,D,E;a;c;r)$ **then return** $\perp$
13. **if** $\neg\mathsf{fsver}(\Sigma_{\text{CD}};S,S_0,v,v_0;a';c';r')$ **then return** $\perp$
14. $s:=\mathsf{paillierdecode}(S_0)$; $X_c:=(X_b)^sE^{-1}$
15. // verification of output phase
16. $(D;a_{\text{out}},c_{\text{out}},r_{\text{out}};Y_0;a_{\text{dec}},c_{\text{dec}},r_{\text{dec}}):=\Pi_{\text{result}}$
17. **if** $\neg\mathsf{fsver}(\Sigma_{\text{PK}};D;a_{\text{out}},c_{\text{out}},r_{\text{out}};\mathcal{R})$ **then return** $\perp$
18. $Y:=X_{\text{outgate}}\cdot D^{-1}$
19. **if** $\neg\mathsf{fsver}(\Sigma_{\text{CD}};Y,Y_0,v,v_0;a_{\text{dec}},c_{\text{dec}},r_{\text{dec}};D)$ **then return** $\perp$
20. $y:=\mathsf{paillierdecode}(Y_0)$
21. **return** $(1+N)^yD$ // encryption of $y+d=r$

this, we let each party first broadcast a hash of its input and proof; only after all parties have committed to their inputs using this hash are the actual encrypted inputs and proofs revealed (line 6). All parties that provide an incorrect hash or proof have their inputs set to zero (line 7–8).

The remainder of the computation follows the CDN protocol. During the *computation phase*, the function is evaluated gate-by-gate; for multiplication gates, the multiplication protocol from [29] is used, with proofs of correct multiplication and decryption using the multiparty FS heuristic (lines 14–25). During the *output phase*, the result party obtains the result by broadcasting an encryption of a random d and proving knowledge using the normal FS heuristic (lines 27–28); the computation parties decrypt the result plus d, proving correctness using the multiparty FS heuristic (line 31). From this, the result party learns result r (line 34); and it knows the intermediate values from the protocol and the proofs showing they are correct.

Finally, we include a *proof phase* in the UVCDN protocol in which the result party sends these intermediate values and proofs to the verifier (line 35). The verifier runs procedure vercomp (Algorithm 4) to verify the correctness of the computation (line 38). The inputs to this verification procedure are the public key of the Paillier cryptosystem; the encrypted inputs $\{X_i\}_{i\in\mathcal{I}}$ by the input parties; and the proof π by the result party (which consists of proofs for each multiplication gate, and the two proofs from the output phase of the protocol). The verifier checks the proofs for each multiplication gate from the computation phase (lines 6–14); and the proofs from the output phase (lines 16–20), finally obtaining an encryption of the result (line 21). While not specified in vercomp, the verifier does also verify the proofs from the input phase: namely, in lines 7–8 of UVCDN, the verifier receives encrypted inputs and verifies their proofs to determine the encrypted inputs $\{X_i\}_{i\in\mathcal{I}}$ of the computation.

Apart from checking the inputs during the input phase, the verifier does not need to be present for the remainder of the computation until receiving π from $\mathcal{R}$. This is what makes verification

"universal": in practice, we envision that a trusted party publicly announces the Paillier public keys, and the input parties publicly announce their encrypted inputs with associated proofs: then, anybody can use the verification procedure to verify if a given proof π is correct with respect to these inputs. We prove that the UVCDN protocol implements verifiable secure function evaluation in the random oracle model in [71].

2.5 Concluding Remarks

Our security model is specific to the CDN setting in two respects. First, we explicitly model that the verifier receives a Paillier encryption of the result (as opposed to another kind of encryption or commitment). We chose this formulation for concreteness; but our model generalises easily to other representations of the result. Second, it is specific to the setting where a minority of parties may be actively corrupted; but it is possible to change the model to other corruption models. For instance, it is possible to model the setting from [9] where privacy is guaranteed when there is at least one honest computation party (and our protocols can be adapted to that setting). The combination of passively secure multiparty computation with universal verifiability is another interesting possible adaptation.

Our protocols are secure in the random oracle model "without dependent auxiliary input" [77]. This means our security proofs assume that the random oracle has not been used before the protocol starts. Moreover, our simulator can only simulate logarithmically many sequential runs of our protocol due to technical limits. These technical issues reflect the real-life problem that a verifier cannot see if a set of computation parties have just performed a computation, or they have simply replayed an earlier computation transcript. As discussed in [75], both problems can be solved in practice by instantiating the random oracle with a keyed hash function, with every computation using a fresh random key. Note that all existing constructions require the random oracle model; achieving universally verifiable (or publicly auditable) multiparty computation in the standard model is open.

Several interesting variants of our protocol are possible. First, it is easy to achieve publicly auditable multiparty computation [9] by performing a public decryption of the result rather than a private decryption for the result party. Another variant is basic outsourcing of computation, in which the result party does not need to be present at the time of the computation, but afterwards gets a transcript from which it can derive the computation result. Finally, it is possible to achieve universal verifiability using other threshold cryptosystems than Paillier. In particular, while the threshold ElGamal cryptosystem is much more efficient than threshold Paillier, it cannot be used directly with our protocols because it does not have a general decryption operation; but universally verifiable multiparty using ElGamal should still be possible by instead adapting the "conditional gate" variant of the CDN protocol from [68].

Finally, to close the loop, we note that our techniques can also be applied to reduce the cost of verification in universally verifiable voting schemes. Namely, for voting schemes relying on homomorphic tallying, we note that the Σ-proofs for correct decryption of the election result by the respective talliers can be combined into a single Σ-proof of constant size (independent of the number of talliers). Similarly, for voting schemes relying on mix-based tallying, the Σ-proofs for correct decryption of each vote by the respective talliers is reduced to a constant size per vote.

Chapter 3

Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation

3.1 Introduction

Recent cryptographic advances are starting to make verifiable computation more and more practical. The goal of verifiable computation is to allow a client to outsource a computation to a worker and cryptographically verify the result with less effort than performing the computation itself. Based on recent ground-breaking ideas [47, 39], Pinocchio [61] was the first implemented system to achieve this for some realistic computations.

In Pinocchio, the computation is performed on plaintext input and the protocol does not offer clients any privacy from the worker. This feature would enable a client to save time by outsourcing computations, even if the inputs of those computations are so sensitive that it does not want to disclose them to the worker. Also, it would allow verifiable computation to be used in settings where multiple clients do not trust the worker or each other, but still want to perform a joint computation over their respective inputs and be sure of the correctness of the result. By outsourcing a computation to multiple workers, it *is* possible to guarantee privacy (if not all workers are corrupted) and correctness, but existing constructions from the multiparty literature lose the most appealing feature of verifiable computation: namely, that computations can be verified very quickly, even in time independent from the computation size.

In this chapter, we describe Trinocchio, a protocol which allows for outsourcing a computation in a privacy-preserving way to multiple workers, while retaining the fast verification offered by verifiable computation. Trinocchio uses state-of-the-art [61]-style proofs, but distributes the computation of these proofs to, e.g., three workers such that no single worker learns anything about the inputs. The client essentially gets a normal Pinocchio proof, so we keep Pinocchio's correctness guarantees and fast verification. The critical observation is that the almost linear structure of Pinocchio proofs (supporting verification based on bilinear maps) allows us to distribute the computation of Pinocchio proofs such that individual workers perform essentially the same work as a normal Pinocchio prover in the non-distributed setting.

While our Trinocchio protocol ensures correct function evaluation, it only fully protects privacy against semi-honest workers. This is a realistic attacker model; in particular, it means that side

channel attacks on individual workers are ineffective because each individual worker's communication and computation are completely independent from the sensitive inputs. However, even if an adversary should be able to obtain sensitive information, they are unable to manipulate the result thanks to the use of verifiable computation. In this way, our protocol *hedges* against the risk of more powerful adversaries.

3.1.1 Outline

We first recap the Pinocchio protocol for verifiable computation based on quadratic arithmetic programs in Section 3.2. Next, we briefly define the security model for privacy-preserving outsourced computation in Section 3.3. In Section 3.4, we show how Trinocchio distributes the proof computation of Pinocchio in the single-client scenario, and prove security of the construction. We generalise Trinocchio to the setting with multiple, mutually distrusting inputters and outputters in Section 3.5. Finally, we demonstrate the feasibility of Trinocchio in Section 3.6 by analysing its performance in two case studies: computing a multivariate polynomial evaluation and proving optimality of a linear program. We finish with a discussion and conclusions in Section 3.7.

3.2 Verifiable Computation from QAPs

In this section, we discuss the protocol for verifiable computation based on quadratic arithmetic programs from [39, 61].

3.2.1 Modelling Computations as Quadratic Arithmetic Programs

A quadratic arithmetic program, or QAP, is a way of encoding arithmetic circuits, and some more general computations, over a field $\mathbb{F}$ of prime order q. It is given by a collection of polynomials over $\mathbb{F}$.

Definition 4 ([61])**.** *A* QAP *Q over a field $\mathbb{F}$ is a tuple $Q = (\{v_i\}_{i=0}^k, \{w_i\}_{i=0}^k, \{y_i\}_{i=0}^k, t)$, with $v_i, w_i, y_i, t \in \mathbb{F}[x]$ polynomials of degree $\deg v_i, \deg w_i, \deg y_i < \deg t = d$. The polynomial t is called the* target polynomial*. The* size *of the QAP is k; the* degree *is the degree d of t.*

In the remainder, for ease of notation, we adopt the convention that $x_0 = 1$.

Definition 5. *Let $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ be a QAP. A tuple $(x_1, \ldots, x_k)$ is a* solution *of Q if t divides $(\sum_{i=0}^k x_i v_i) \cdot (\sum_{i=0}^k x_i w_i) - (\sum_{i=0}^k x_i y_i) \in \mathbb{F}[x]$.*

In case t splits, i.e., $t = (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_n)$, a QAP can be seen as a collection of rank-1 quadratic equations for $(x_1, \ldots, x_k)$; that is, equations $v \cdot w - y$ with $v, w, y \in \mathbb{F}[x_1, \ldots, x_k]$ of degree at most one. Namely, $(x_1, \ldots, x_k)$ is a solution of Q if t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, which means exactly that, for every α_j, $(\sum_i x_i v_i(\alpha_j)) \cdot (\sum_i x_i w_i(\alpha_j)) - (\sum_i x_i y_i(\alpha_j)) = 0$: that is, each α_j gives a rank-1 quadratic equation in variables $(x_1, \ldots, x_k)$. Conversely, a collection of d such equations (recall $x_0 \equiv 1$)

$$(v_0^j \cdot x_0 + \ldots + v_k^j \cdot x_k) \cdot (w_0^j \cdot x_0 + \ldots + w_k^j \cdot x_k) - (y_0^j \cdot x_0 + \ldots + y_k^j \cdot x_k)$$

can be turned into a QAP by selecting d distinct elements $\alpha_1, \ldots, \alpha_d$ in $\mathbb{F}$, setting target polynomial $t = (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_d)$, and defining v_0 to be the unique polynomial of degree smaller than d for which $v_0(\alpha_j) = v_0^j$, etcetera.
A QAP is said to compute a function $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ if the remaining x_i give a solution exactly if the function is correctly evaluated.

Definition 6 ([61]). *Let $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ be a QAP, and let $f : \mathbb{F}^l \to \mathbb{F}^m$ be a function. We say that Q* computes *f if $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l) \Leftrightarrow \exists\ (x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.*

For any function f given by an arithmetic circuit, we can easily construct a QAP that computes the function f. Indeed, we can describe an arithmetic circuit as a series of rank-1 quadratic equations by letting each multiplication gate become one equation. Apart from circuits containing just addition and multiplication gates, we can also express circuits with some other kinds of gates directly as QAPs. For instance, [61] defines a "split gate" that converts a number a into its k-bit decomposition $a_1, \ldots, a_k$ with equations $a = a_1 + 2 \cdot a_2 + \ldots + 2^{k-1} \cdot a_k$, $a_1 \cdot (1 - a_1) = 0$, ..., $a_k \cdot (1 - a_k) = 0$.

3.2.2 Proving Correctness of Computations

If QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ computes a function f, then it is possible for a prover to prove that $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ by proving knowledge of values $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q, i.e., t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$. [61] gives a construction of a proof system which does exactly this. The proof system assumes discrete logarithm groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ for which the $(4d + 4)$-PDH, d-PKE and $(8d + 8)$-SDH assumptions [61] hold, with d the degree of the QAP. Moreover, the proof is in the common reference string (CRS) model: the CRS consists of an *evaluation key* used to produce the proof, and a *verification key* used to verify it. Both are public, i.e., provers can know the verification key and vice versa.

To prove that t divides $p = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, the prover computes quotient polynomial $h = p/t$ and basically provides evaluations "in the exponent" of h, $(\sum_i x_i v_i)$, $(\sum_i x_i w_i)$, and $(\sum_i x_i y_i)$ in an unknown point s that can be verified using the pairing. More precisely, given generators g_1 of $\mathbb{G}_1$ and g_2 of $\mathbb{G}_2$ (written additively) and polynomial $f \in \mathbb{F}[x]$, let us write $\langle f \rangle_1$ for $g_1 \cdot f(s)$ and $\langle f \rangle_2$ for $g_2 \cdot f(s)$. The evaluation key in the CRS, generated using random $s, \alpha_v, \alpha_w, \alpha_y, \beta, r_v, r_w, r_y = r_v \cdot r_w \in \mathbb{F}$, is:

$$\langle r_v v_i \rangle_1, \langle r_v \alpha_v v_i \rangle_1, \langle r_w w_i \rangle_2, \langle r_w \alpha_w w_i \rangle_1, \langle r_y y_i \rangle_1, \langle r_y \alpha_y y_i \rangle_1,$$
$$\langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1, \langle s^j \rangle_1.$$

where i ranges over $0, l+m+1, l+m+2, \ldots, k$ and j runs from 0 to the degree of t. The proof contains the following elements:

$$\begin{aligned}
\langle V_{\text{mid}} \rangle_1 &= \textstyle\sum_i \langle r_v v_i \rangle_1 \cdot x_i, \quad \langle \alpha_v V_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_v \alpha_v v_i \rangle_1 \cdot x_i, \\
\langle W_{\text{mid}} \rangle_2 &= \textstyle\sum_i \langle r_w w_i \rangle_2 \cdot x_i, \quad \langle \alpha_w W_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_w \alpha_w w_i \rangle_1 \cdot x_i, \\
\langle Y_{\text{mid}} \rangle_1 &= \textstyle\sum_i \langle r_y y_i \rangle_1 \cdot x_i, \quad \langle \alpha_y Y_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_y \alpha_y y_i \rangle_1 \cdot x_i, \\
\langle Z \rangle_1 &= \textstyle\sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1 \cdot x_i, \quad \langle H \rangle_1 = \textstyle\sum_j \langle s^j \rangle_1 \cdot h_j,
\end{aligned} \tag{3.1}$$

where i ranges over $0, l + m + 1, l + m + 2, \ldots, k$, and h_j are the coefficients of polynomial $h = p/t$.

To verify that t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ and hence $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$, a verifier uses the following verification key from the CRS:

$$\langle \alpha_v \rangle_2, \langle \alpha_w \rangle_2, \langle \alpha_y \rangle_2, \langle \beta \rangle_1, \langle \beta \rangle_2, \langle r_v v_i \rangle_1, \langle r_w w_i \rangle_2, \langle r_y y_i \rangle_1, \langle r_y t \rangle_2,$$

where i ranges over $1, 2, \ldots, l+m$[1]. Given the verification key, a proof, and values $x_1, \ldots, x_{l+m}$, the verifier proceeds as follows. First, it checks that

$$\begin{aligned} e(\langle V_{\text{mid}}\rangle_1, \langle \alpha_v \rangle_2) &= e(\langle \alpha_v V_{\text{mid}}\rangle_1, \langle 1\rangle_2); \\ e(\langle \alpha_w\rangle_1, \langle W_{\text{mid}}\rangle_2) &= e(\langle \alpha_w W_{\text{mid}}\rangle_1, \langle 1\rangle_2); \\ e(\langle Y_{\text{mid}}\rangle_1, \langle \alpha_y \rangle_2) &= e(\langle \alpha_y Y_{\text{mid}}\rangle_1, \langle 1\rangle_2): \end{aligned} \tag{3.2}$$

intuitively, under the d-PKE assumption, these checks guarantee that the prover must have constructed $\langle V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, and $\langle Y_{\text{mid}}\rangle_1$ using the elements from the evaluation key. It then checks that

$$e(\langle V_{\text{mid}}\rangle_1 + \langle Y_{\text{mid}}\rangle_1, \langle \beta\rangle_2) \cdot e(\langle \beta\rangle_1, \langle W_{\text{mid}}\rangle_2) = e(\langle Z\rangle_1, \langle 1\rangle_2): \tag{3.3}$$

under the PDH assumption, this guarantees that the same coefficients x_i were used in $\langle V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, and $\langle Y_{\text{mid}}\rangle_1$. Finally, the verifier computes evaluations $\langle V\rangle_1$ of $\sum_{i=0}^{k} x_i v_i$ as $\langle V_{\text{mid}}\rangle_1 + \sum_{i=1}^{l+m} \langle r_v v_i\rangle_1 \cdot x_i$; $\langle W\rangle_2$ of $\sum_{i=0}^{k} x_i w_i$ as $\langle W_{\text{mid}}\rangle_2 + \sum_{i=1}^{l+m} \langle r_w w_i\rangle_2 \cdot x_i$; and $\langle Y\rangle_1$ of $\sum_{i=0}^{k} x_i y_i$ as $\langle Y_{\text{mid}}\rangle_1 + \sum_{i=1}^{l+m} \langle r_y y_i\rangle_1 \cdot x_i$, and verifies that

$$e(\langle V\rangle_1, \langle W\rangle_2) \cdot e(\langle Y\rangle_1, \langle 1\rangle_2)^{-1} = e(\langle H\rangle_1, \langle r_y t\rangle_2): \tag{3.4}$$

under the $(8d+8)$-SDH assumption, this guarantees that, for the polynomial h encoded by $\langle H\rangle_1$, $t \cdot h = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ holds.[2]

Theorem 1 ([39], informal). *Given QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ and values $x_1, \ldots, x_{l+m}$, the above is a non-interactive argument of knowledge of $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.*

3.2.3 Making the Proof Zero-Knowledge

The above proof can be turned into a zero-knowledge proof, that reveals nothing about the values of $(x_{l+m+1}, \ldots, x_k)$ other than that t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ for some h, by performing randomisation. Namely, instead of proving that $t \cdot h = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, we prove that $t \cdot \tilde{h} = (\sum_i x_i v_i + \delta_v \cdot t) \cdot (\sum_i x_i w_i + \delta_w \cdot t) - (\sum_i x_i y_i + \delta_y \cdot t)$ with $\delta_v, \delta_w, \delta_y$ random from $\mathbb{F}$. Precisely, the evaluation key needs to contain additional elements:

$$\langle r_v t\rangle_1, \langle r_v \alpha_v t\rangle_1, \langle r_w t\rangle_2, \langle r_w \alpha_w t\rangle_1, \langle r_y t\rangle_1, \langle r_y \alpha_y t\rangle_1, \langle r_v \beta t\rangle_1, \langle r_w \beta t\rangle_1, \langle r_y \beta t\rangle_1, \langle t\rangle_1.$$

Compared to the original proof, we let

$$\begin{aligned} \langle V'_{\text{mid}}\rangle_1 = \langle V_{\text{mid}}\rangle_1 + \langle r_v t\rangle_1 \cdot \delta_v, \quad & \langle \alpha_v V'_{\text{mid}}\rangle_1 = \langle \alpha_v V'_{\text{mid}}\rangle_1 + \langle r_v \alpha_v t\rangle_1 \cdot \delta_v, \\ \langle W'_{\text{mid}}\rangle_2 = \langle W_{\text{mid}}\rangle_2 + \langle r_w t\rangle_2 \cdot \delta_w, \quad & \langle \alpha_w W'_{\text{mid}}\rangle_1 = \langle \alpha_w W_{\text{mid}}\rangle_1 + \langle r_w \alpha_w t\rangle_1 \cdot \delta_w, \\ \langle Y'_{\text{mid}}\rangle_1 = \langle Y_{\text{mid}}\rangle_1 + \langle r_y t\rangle_1 \cdot \delta_y, \quad & \langle \alpha_y Y'_{\text{mid}}\rangle_1 = \langle \alpha_y Y_{\text{mid}}\rangle_1 + \langle r_y \alpha_y t\rangle_1 \cdot \delta_y, \\ \langle Z'\rangle_1 = \langle Z\rangle_1 + \langle r_v \beta t\rangle_1 \cdot \delta_v + \langle r_w \beta t\rangle_1 & \cdot \delta_w + \langle r_y \beta t\rangle_1 \cdot \delta_y, \langle H'\rangle_1 = \textstyle\sum_j \langle s^j\rangle_1 \cdot \widetilde{h}_j, \end{aligned}$$

with $\widetilde{h}_j$ the coefficients of $h + \delta_v w_0 + \sum_i \delta_v x_i \cdot w_i + \delta_w v_0 + \sum_i \delta_w x_i \cdot v_i + \delta_v \delta_w \cdot t - \delta_y$. Verification remains exactly the same.

Theorem 2 ([39], informal). *Given QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ and values $x_1, \ldots, x_{l+m}$, the above is a non-interactive zero-knowledge argument of knowledge of $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.*

[1] In [61], several terms of the verification key includes a value γ; however, a careful look at [61]'s proof reveals that γ is actually not needed. We remove it because it simplifies notation, especially for our multi-client protocols.

[2] We remark that, as shown in [61], a verifier who has generated the evaluation and verification keys, can use the randomness from the generation process to save several of the above pairing checks. We do not consider this optimisation here.

3.2.4 From Arguments of Knowledge to Verifiable Computation

In [61], the above argument of knowledge is used to construct a *public verifiable computation scheme*. In such a scheme, a client outsources the computation of a function f to a worker, obtaining cryptographic guarantees that the result it gets from the worker is correct. It is defined as follows:

Definition 7 ([61])**.** *A* public verifiable computation scheme $\mathcal{VC}$ *consists of three polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{Compute}, \mathsf{Verify})$*:*

- $(EK_f; VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda)$*: a probabilistic* key generation algorithm *that takes as argument a function* $f : \mathbb{F}^l \rightarrow \mathbb{F}^m$ *and a security parameter* λ*, outputting a public evaluation key* EK_f *and a public verification key* VK_f
- $(\vec{y}; \pi) \leftarrow \mathsf{Compute}(EK_f; \vec{x})$*: a probabilistic* worker algorithm *that takes input* $\vec{x} \in \mathbb{F}^l$ *and outputs* $\vec{y} = f(\vec{x}) \in \mathbb{F}^k$ *and a proof* π *of its correctness*
- $\{0, 1\} \leftarrow \mathsf{Verify}(VK_f; \vec{x}; \vec{y}; \pi)$*: a deterministic* verification algorithm *that outputs* 1 *if* $\vec{y} = f(\vec{x})$*, 0 otherwise.*

To outsource the computation of f, a client runs KeyGen and provides EK_f to the worker. When it needs $f(\vec{x})$, it provides $\vec{x}$ to the worker, who runs Compute and provides the result $\vec{y} = f(\vec{x})$ and proof π to the client. The client accepts $\vec{y}$ if Verify succeeds. We require that worker cannot provide incorrect proofs even if it knows VK_f, which makes this verifiable computation scheme "public". In fact, a trusted party could for once and for all perform KeyGen and publish $(\mathrm{EK}_f, \mathrm{VK}_f)$; any client who trusts this party can then use the published VK_f to verify computations. (Trusting this party is needed: the random values used in KeyGen are a trapdoor with which the generator of the keys can produce false proofs.) A public verifiable computation scheme should satisfy *correctness* and *security*. Correctness means that honest workers produce accepting proofs:

Definition 8 ([61])**.** *A public verifiable computation scheme* $\mathcal{VC}$ *is called* correct *if, for all* $f : \mathbb{F}^l \rightarrow \mathbb{F}^m$ *and* $\vec{x} \in \mathbb{F}$*:*

$$\text{if } (EK_f; VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda); (\vec{y}; \pi) \leftarrow \mathsf{Compute}(EK_f; \vec{x}),$$
$$\text{then } \mathsf{Verify}(VK_f; \vec{x}; \vec{y}; \pi) = 1.$$

Security means that corrupt workers cannot convince clients of wrong results:

Definition 9 ([61])**.** *A public verifiable computation scheme* $\mathcal{VC}$ *is called* secure *if, for any* $f : \mathbb{F}^l \rightarrow \mathbb{F}^m$ *and probabilistic polynomial time adversary* $\mathcal{A}$*:*

$$\mathsf{Pr}[\ (EK_f, VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda); (\vec{x}; \vec{y}; \pi) \leftarrow \mathcal{A}(EK_f; VK_f) :$$
$$\vec{y} \neq f(\vec{x}) \wedge \mathsf{Verify}(VK_f; \vec{x}; \vec{y}; \pi) = 1\ ] = \mathsf{negl}(\lambda).$$

Given a QAP Q that computes a function f, the argument of knowledge from Section 3.2.2 directly gives a public verifiable computation scheme known as Pinocchio [61]: KeyGen is the computation of the evaluation and verification keys for Q; Compute computes $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$, $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q, and proof (3.1); and Verify are the checks (3.2–3.4) for this proof.

Theorem 3 (Pinocchio [61], informal)**.** *Let QAP Q be of degree d. Then the above construction is a secure and correct public verifiable computation scheme under the d-PKE,* $(4d + 4)$*-PDH, and* $(8d + 8)$*-SDH assumptions.*

Secure function evaluation:

- Honest parties send inputs x_i to trusted party
- Adversary sends inputs x_i of corrupted parties to trusted party (active adversary may modify them)
- Trusted party computes function $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ (where $y_1 = \ldots = \perp$ if any $x_i = \perp$)
- Trusted party provides outputs y_i for corrupted parties to adversary
- Trusted party provides outputs y_i to honest parties
- Honest parties output received value; corrupted parties output $\perp$; adversary chooses own output

Correct function evaluation:

- Honest parties send inputs x_i to trusted party
- Adversary sends inputs x_i of corrupted parties to trusted party (active adversary may modify them)
- Trusted party computes function $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ (where $y_1 = \ldots = \perp$ if any $x_i = \perp$)
- Trusted party provides all inputs x_i to adversary
- Adversary gives subset of honest parties to trusted party (passive adversary gives all honest parties)
- Trusted party sends outputs y_i to given honest parties, $\perp$ to others
- Honest parties output received value; corrupted parties output $\perp$; adversary chooses own output

Figure 3.1: Ideal-world executions of secure (left) and correct (right) function evaluation. The highlighted text indicates where the two differ.

3.3 Security Model for Privacy-Preserving Outsourcing

In this section, we define security for privacy-preserving outsourcing. Because we have interactive protocols between multiple parties (as opposed to a cryptographic scheme, like verifiable computation above), we define security using the ideal/real-paradigm [22]. In our setting, the parties are several *result parties* that wish to obtain the result of a computation on inputs held by several *input parties*, who are willing to enable the computation, but not to divulge their private input values to anybody else. Therefore, they outsource the computation to several *workers*. (Input and result parties may overlap.) The simplest case is the "single-client scenario" in which one party is the single input/result party.

We consider protocols operating in three phases: an *input phase* involving the input parties and workers; a *computation phase* involving only the workers; and a *result phase* involving the workers and result parties. The work of the input parties and output parties should depend only on the number of other parties and the size of their own in/outputs.

To define security, we will re-use the existing definition framework for secure function evaluation [22]. These definitions not specific to the outsourcing setting; but the outsourcing setting will become apparent when we claim that a protocol, e.g., implements secure function evaluation *if at most X workers are corrupted*. Secure function evaluation is the problem to evaluate $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ with m parties such that the ith party inputs x_i and obtains y_i,

and no party learns anything else. (In outsourcing, result parties have non-empty output, input parties have non-empty inputs, and workers have empty in- and outputs.) A protocol π *securely evaluates function* f if the outputs of the parties and adversary $\mathcal{A}$ in a real-world execution of the protocol can be emulated by the outputs of the parties and an adversary $\mathcal{S}_{\mathcal{A}}$ in an idealised execution, where f is computed by a trusted party that acts as shown in Figure 3.1. Security is guaranteed because the trusted party correctly computes the function. Privacy is guaranteed because the adversary in the idealised execution does not learn anything it should not. Secure evaluation also implies *input independence*, meaning that an input party cannot let its input depend on that of another, e.g., by copying the input of another party; this is guaranteed because the adversary needs to provide the inputs of corrupted parties without seeing the honest inputs. Typically, protocols achieve secure function evaluation for a given, restricted class of adversaries, e.g., adversaries that are passive and only corrupt a certain number of workers. Protocols can require set-up assumptions; these are captured by giving protocol participants access to a set of functions $g_1, \ldots, g_k$ that are always evaluated correctly. In this case, we say that the protocol securely evaluates the function *in the* $(g_1, \ldots, g_k)$*-hybrid model*. For details, see [22].

We only achieve secure function evaluation if not too many workers are corrupted; we still need to formalise that in all other cases, we still guarantee that the function was evaluated correctly. This weaker security guarantee, which we call *correct function evaluation*, captures security and input independence, as above, but not privacy. It is formalised by modifying the ideal-world execution as shown in Figure 3.1. Namely, after evaluating f, the trusted party provides all inputs to the adversary (modelling that the computation may leak the inputs), who, based on these inputs, can decide which honest parties are allowed to see their outputs. Hence, we guarantee that, *if* an honest party gets a result, then it gets the correct result of the computation on independently chosen inputs, but not that the inputs remain hidden, or that it gets a result at all. Note that, in this definition, the adversary has complete control over which result parties see an output and which ones do not.

3.4 Distributing the Prover Computation

In this section, we present the single-client version of our Trinocchio protocol for privacy-preserving outsourcing. In Trinocchio, a client distributes computation of a function $x_2 = f(x_1)$ to n workers (we consider here single-valued input and output, but the generalisation is straightforward). Trinocchio guarantees correct function evaluation (regardless of corruptions) and secure function evaluation (if at most θ workers are passively corrupted, where $n = 2\theta + 1$). Trinocchio in effect distributes the proof computation of Pinocchio; the number of workers to obtain privacy against one semi-honest worker is three, hence its name.

3.4.1 Multiparty Computation using Shamir Secret Sharing

To distribute the Pinocchio computation, Trinocchio employs multiparty computation techniques based on Shamir secret sharing [12]. Recall that in (θ, n) Shamir secret sharing, a party shares a secret s among n parties so that $\theta + 1$ parties are needed to reconstruct s. It does this by taking a random degree-$\leq \theta$ polynomial $p(x) = \alpha_\theta x^\theta + \ldots + \alpha x + s$ with s as constant term and giving $p(i)$ to party i. Since $p(x)$ is of degree at most θ, $p(0)$ is completely independent from any θ shares but can be easily computed from any $\theta + 1$ shares by Lagrange interpolation.

We denote such a sharing as $[\![s]\!]$. Note that Shamir-sharing can also be done "in the exponent", e.g., $[\![\langle a\rangle_1]\!]$ denotes a Shamir sharing of $\langle a\rangle_1 \in \mathbb{G}_1$ from which $\langle a\rangle_1$ can be computed using Lagrange interpolation in $\mathbb{G}_1$.

Shamir secret sharing is linear, i.e., $[\![a+b]\!] = [\![a]\!] + [\![b]\!]$ and $[\![\alpha a]\!] = \alpha[\![a]\!]$ can be computed locally. When computing the product of $[\![a]\!]$ and $[\![b]\!]$, each party i can locally multiply its points $p_a(i)$ and $p_b(i)$ on the random polynomials p_a and p_b. Because the product polynomial has degree at most 2θ, this is a $(2\theta, n)$ sharing, which we write as $[a \cdot b]$ (note that reconstructing the secret requires $n = 2\theta + 1$ parties). Moreover, the distribution of the shares of $[a \cdot b]$ is not independent from the values of a and b, so when revealed, these shares reveal information about a and b. Hence, in multiparty computation, $[a \cdot b]$ is typically converted back into a random (θ, n) sharing $[\![a \cdot b]\!]$ using an interactive protocol due to [40]. Interactive protocols for many other tasks such as comparing two shared value also exist (see, e.g., [32]).

3.4.2 The Trinocchio protocol

We now present the Trinocchio protocol. Trinocchio assumes that Pinocchio's KeyGen has been correctly performed: formally, Trinocchio works in the KeyGen-hybrid model. Furthermore, Trinocchio assumes pairwise private, synchronous communication channels. To obtain $x_2 = f(x_1)$, a client proceeds in four steps:

- The client obtains the verification key, and the workers obtain the evaluation key, using hybrid calls to KeyGen.
- The client secret shares $[\![x_1]\!]$ of its input to the workers.
- The workers use multiparty computation to compute secret-shares $[\![x_2]\!]$ of the output and $[\![\langle V_{\text{mid}}\rangle_1]\!]$, $[\![\langle \alpha_v V_{\text{mid}}\rangle_1]\!]$, $[\![\langle W_{\text{mid}}\rangle_2]\!]$, $[\![\langle \alpha_w W_{\text{mid}}\rangle_1]\!]$, $[\![\langle Y_{\text{mid}}\rangle_1]\!]$, $[\![\langle \alpha_y Y_{\text{mid}}\rangle_1]\!]$, $[\![\langle Z\rangle_1]\!]$, $[\langle H\rangle_1]$ of the Pinocchio proof, as we explain next; and sends these shares to the client.
- The client recombines the shares into $\langle V_{\text{mid}}\rangle_1$, $\langle \alpha_v V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, $\langle \alpha_w W_{\text{mid}}\rangle_1$, $\langle Y_{\text{mid}}\rangle_1$, $\langle \alpha_y Y_{\text{mid}}\rangle_1$, $\langle Z\rangle_1$, $\langle H\rangle_1$ by Lagrange interpolation, and accepts x_2 as computation result if Pinocchio's Verify returns success.

Algorithm 5 shows in detail how the secret-shares of the function output and Pinocchio proof are computed. The first step is to compute function output $x_2 = f(x_1)$ and values $(x_3, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of the QAP (line 4). This is done using normal multiparty computation protocols based on secret sharing. If function f is represented by an arithmetic circuit, then it is evaluated using local addition and scalar multiplication, and the multiplication protocol from [40]. If f is represented by a circuit using more complicated gates, then specific protocols may be used: e.g., the split gate discussed in Section 3.2.1 can be evaluated using multiparty bit decomposition protocols [27, 69]. Any protocol can be used as long as it guarantees privacy, i.e., the view of any θ workers is statistically independent from the values represented by the shares.

The next task is to compute, in secret-shared form, the coefficients of the polynomial $h = ((\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i))/t \in \mathbb{F}[x]$ that we need for proof element $\langle H\rangle_1$. In theory, this computation could be performed by first computing shares of the coefficients of $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, and then dividing by t, which can be done locally using traditional polynomial long division. However, this scales quadratically in the degree of the

Algorithm 5 Trinocchio's $\mathsf{Compute}$ protocol

1: $\triangleright\ \mathcal{S} = \{\alpha_1, \ldots, \alpha_d\}$ denotes the list of roots of the target polynomial of the QAP
2: $\triangleright\ \mathcal{T} = \{\beta_1, \ldots, \beta_d\}$ denotes a list of distinct points different from $\mathcal{S}$
3: **function** $\mathsf{Compute}(\mathsf{EK}_f = \{\langle r_v v_i \rangle_1\}_i, \ldots, \{\langle s^j \rangle_1\}_j; [\![x_1]\!])$
4: $\quad([\![x_2]\!], \ldots, [\![x_k]\!]) \leftarrow f([\![x_1]\!])$
5: $\quad[\![\vec{v}]\!] \leftarrow \{\sum_i v_i(\alpha_j) \cdot [\![x_i]\!]\}_j;\ [\![\vec{V}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\vec{v}]\!]);\ [\![\vec{v}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\vec{V}]\!])$
6: $\quad[\![\vec{w}]\!] \leftarrow \{\sum_i w_i(\alpha_j) \cdot [\![x_i]\!]\}_j;\ [\![\vec{W}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\vec{w}]\!]);\ [\![\vec{w}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\vec{W}]\!])$
7: $\quad[\![\vec{y}]\!] \leftarrow \{\sum_i y_i(\alpha_j) \cdot [\![x_i]\!]\}_j;\ [\![\vec{Y}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\vec{y}]\!]);\ [\![\vec{y}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\vec{Y}]\!])$
8: $\quad[\vec{h}'] \leftarrow \{([\![\vec{v}'_j]\!] \cdot [\![\vec{w}'_j]\!] - [\![\vec{y}'_j]\!])/t(\beta_j)\}_j;\ [\vec{H}] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\vec{h}'])$
9: $\quad[\![\langle V_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_v v_i \rangle_1 \cdot [\![x_i]\!]$
10: $\quad[\![\langle \alpha_v V_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_v \alpha_v v_i \rangle_1 \cdot [\![x_i]\!]$
11: $\quad[\![\langle W_{\text{mid}} \rangle_2]\!] \leftarrow \sum_i \langle r_w w_i \rangle_2 \cdot [\![x_i]\!]$
12: $\quad[\![\langle \alpha_w W_{\text{mid}} \rangle_1]\!] \sum_i \langle r_w \alpha_w w_i \rangle_1 \cdot [\![x_i]\!]$
13: $\quad[\![\langle Y_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_y y_i \rangle_1 \cdot [\![x_i]\!]$
14: $\quad[\![\langle \alpha_y Y_{\text{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_y \alpha_y y_i \rangle_1 \cdot [\![x_i]\!]$
15: $\quad[\![\langle Z \rangle_1]\!] \leftarrow \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1 \cdot [\![x_i]\!]$
16: $\quad[\langle H \rangle_1] = \sum_j \langle s^j \rangle_1 \cdot [\vec{H}_j]$
17: $\quad$**return** $([\![x_2]\!]; [\![\langle V_{\text{mid}} \rangle_1]\!], [\![\langle \alpha_v V_{\text{mid}} \rangle_1]\!], [\![\langle W_{\text{mid}} \rangle_2]\!], [\![\langle \alpha_w W_{\text{mid}} \rangle_1]\!],$
18: $\qquad\qquad [\![\langle Y_{\text{mid}} \rangle_1]\!], [\![\langle \alpha_y Y_{\text{mid}} \rangle_1]\!], [\![\langle Z \rangle_1]\!], [\langle H \rangle_1])$

QAP and hence leads to unacceptable performance. Hence, we take the approach based on fast Fourier transforms (FFTs) from [13], and adapt it to the distributed setting. Given a list $\mathcal{S} = \{\omega_1, \ldots, \omega_d\}$ of distinct points in $\mathbb{F}$, we denote by $\vec{P} = \mathsf{FFT}_{\mathcal{S}}(\vec{p})$ the transformation from coefficients $\vec{p}$ of a polynomial p of degree at most $d-1$ to evaluations $p(\omega_1), \ldots, p(\omega_d)$ in the points in $\mathcal{S}$. We denote by $\vec{p} = \mathsf{FFT}_{\mathcal{S}}^{-1}(\vec{P})$ the inverse transformation, i.e., from evaluations to coefficients. Deferring specifics to later, we mention now that the FFT is a linear transformation that, for some $\mathcal{S}$, can be performed locally on secret shares in $\mathcal{O}(d \cdot \log d)$.

With FFTs available, we can compute the coefficients of h by evaluating h in d distinct points and applying FFT^{-1}. Note that we can efficiently compute evaluations $\vec{v}$ of $v = (\sum_i x_i v_i)$, $\vec{w}$ of $w = (\sum_i x_i w_i)$, and $\vec{y}$ of $y = (\sum_i x_i y_i)$ in the zeros $\{\omega_1, \ldots, \omega_d\}$ of the target polynomial. Namely, the values $v_k(\omega_i)$, $w_k(\omega_i)$, $y_k(\omega_i)$ are simply the coefficients of the quadratic equations represented by the QAP, most of which are zero, so these sums have much fewer than k elements (if this were not the case, then evaluating v, w, and y would take an unacceptable $O(d \cdot k)$). Unfortunately, we cannot use these evaluations directly to obtain evaluations of h, because this requires division by the target polynomial, which is zero in exactly these points ω_i. Hence, after determining $\vec{v}$, $\vec{w}$, and $\vec{y}$, we first use the inverse FFT to determine the coefficients $\vec{V}$, $\vec{W}$, and $\vec{Y}$ of v, w, and y, and then again the FFT to compute the evaluations $\vec{v'}$, $\vec{w'}$, and $\vec{y'}$ of v, w, and y in another set of points $\mathcal{T} = \{\Omega_1, \ldots, \Omega_k\}$ (lines 5–7). Now, we can compute evaluations $\vec{h'}$ of h in $\mathcal{T}$ using $h(\Omega_i) = (v(\Omega_i) \cdot w(\Omega_i) - y(\Omega_i))/t(\Omega_i)$. This requires a multiplication of (θ, n)-secret shares of $v(\Omega_i)$ and $w(\Omega_i)$, hence the result is a $(2\theta, n)$-sharing. Finally, the inverse FFT gives us a $(2\theta, n)$-sharing of the coefficients $\vec{H}$ of h (line 8).

Given secret shares of the values of x_i and coefficients of h, it is straightforward to compute secret shares of the Pinocchio proof. Indeed, $\langle V_{\text{mid}} \rangle_1, \ldots, \langle H \rangle_1$ are all computed as linear combinations of elements in the evaluation key, so shares of these proof elements can be computed locally (lines 9–16), and finally returned by the respective workers (lines 17–18).

Note that, compared to Pinocchio, our client needs to carry out slightly more work. Namely, our client needs to produce secret shares of the inputs and recombine secret shares of the

outputs; and it needs to recombine the Pinocchio proof. However, according to the micro-benchmarks from [61], this overhead is small. For each input and output, Verify includes three exponentiations, whereas $\mathsf{Combine}$ involves four additions and two multiplications; when using [61]'s techniques, this adds at most a 3% overhead. Recombining the Pinocchio proof involves 15 exponentiations at around half the cost of a single pairing. Alternatively, it is possible to let one of the workers perform the Pinocchio recombining step by using the distributed zero-knowledge variant of Pinocchio (Section 3.2.3) and the techniques from Section 3.5. In this case, the only overhead for the client is the secret-sharing of the inputs and zero-knowledge randomness, and recombining the outputs.

Parameters for Efficient FFTs To obtain efficient FFTs, we use the approach of [13]. There, it is noted that the operation $\vec{P} = \mathsf{FFT}_{\mathcal{S}}(\vec{p})$ and its inverse can be efficiently implemented if $\mathcal{S} = \{\omega, \omega^2, \ldots, \omega^d = 1\}$ is a set of powers of a primitive dth root of unity, where d is a power of two. (We can always demand that QAPs have degree $d = 2^k$ for some k by adding dummy equations.) Moreover, [13] presents a pair of groups $\mathbb{G}_1, \mathbb{G}_2$ of order q such that $\mathbb{F}_q$ has a primitive 2^{30}th root of unity (and hence also primitive 2^kth roots of unity for any $k < 30$) as well as an efficiently computable pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$. Finally, [13] remarks that for $\mathcal{T} = \{\eta\omega, \eta\omega^2, \ldots, \eta\omega^d = \eta\}$, operations $\mathsf{FFT}_{\mathcal{T}}^{-1}$ and $\mathsf{FFT}_{\mathcal{T}}^{-1}$ can easily be reduced to $\mathsf{FFT}_{\mathcal{S}}$ and $\mathsf{FFT}_{\mathcal{S}}^{-1}$, respectively. In our implementation, we use exactly these suggested parameters.

3.4.3 Security of Trinocchio

Theorem 4. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the Trinocchio protocol. Assuming the d-PKE, $(4d + 4)$-PDH, and $(8d + 8)$-SDH assumptions:*

- *Trinocchio correctly evaluates f in the* KeyGen*-hybrid model.*
- *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the* KeyGen*-hybrid model.*

The proof of this theorem is easily derived as a special case of the proof for the multi-client Trinocchio protocol later. Here, we present a short sketch.

Sketch. To prove correct function evaluation, we need to show that for every real-world adversary $\mathcal{A}$ interacting with Trinocchio, there is an ideal-world simulator $\mathcal{S}_{\mathcal{A}}$ that interacts with the trusted party for correct function evaluation such that the two executions give indistinguishable results. The only interesting case is when the client is honest and some of the workers are not. In this case, the simulator receives the input of the honest party, and needs to choose whether to provide the output. To this end, the simulator simply simulates a run of the actual protocol with $\mathcal{A}$, until it has finally obtained function output x_2 and the accompanying Trinocchio proof. If the proof verifies, it tells the trusted party to provide the output to the client; otherwise, it tells the trusted party not to. Finally, the simulator outputs whatever $\mathcal{A}$ outputs. Because Trinocchio is secure, except with negligible probability a verifying proof implies that the real-world output of the client (as given by the adversary) matches the ideal-world output of the client (as computed by the trusted party); and by construction, the outputs of $\mathcal{A}$ and $\mathcal{S}_{\mathcal{A}}$ are distributed identically. This proves correct function evaluation.

For secure function evaluation, again the only interesting case is if the client is honest and some of the workers are passively corrupted. In this case, because corruption is only passive, correctness of the multiparty protocol used to compute f and correctness of the Pinocchio proof system used to compute the proof together imply that real-world executions (like ideal-world executions) result in the correct function result and a verifying proof. Hence, we only need to worry about how $\mathcal{S}_{\mathcal{A}}$ can simulate the view of $\mathcal{A}$ on the Trinocchio protocol without knowing the client's input. However, note that the workers only use a multiparty computation to compute f (which we assume can be simulated without knowing the inputs), after which they no longer receive any messages. Hence simulating the multiparty computation for f and receiving any messages that $\mathcal{A}$ sends is sufficient to simulate $\mathcal{A}$. This proves secure function evaluation. □

Privacy against Active Attacks We remark that actually, Trinocchio in some cases provides privacy against corrupted workers as well. Namely, suppose that the protocol used to compute f does not leak any information to corrupted workers in the event of an active attack (even though in this case it may not guarantee correctness). For instance, this is the case for the protocol from [40]: the attacker can manipulate the shares that it sends, which makes the computation return incorrect results; but since the attacker always learns only θ many shares of any value, it does not learn any information. Because the attacker learns no additional information from producing the Pinocchio proof, the overall protocol still leaks no information to the adversary. (And security of Pinocchio ensures the client notices the attacker's manipulation.)

This crucially relies on the workers not learning whether the client accepts the proof: if the workers would learn whether the client obtained a validating proof, then, by manipulating proof construction, they could learn whether a modified version of the tuple $(x_1, \ldots, x_k)$ is a solution of the QAP used, so corrupted workers could learn one chosen bit of information about the inputs (cf. [59]).

3.5 Handling Mutually Distrusting In- and Outputters

We now consider the scenario where there are multiple (possibly overlapping) input and result parties. There are some significant changes between this scenario and the single-client scenario. In particular, we need to extend Pinocchio to allow verification not based on the actual input/output values (indeed, no party sees all of them) but on some kind of representation that does not reveal them. Moreover, we need to use the zero-knowledge variant of Pinocchio (Section 3.2.3), and we need to make sure that input parties choose their inputs independently from each other.

3.5.1 Multi-Client Proofs and Keys

Our multi-client Trinocchio proofs are a generalisation of the zero-knowledge variant of Pinocchio (Section 3.2.3) with modified evaluation and verification keys. Recall that in Pinocchio, the proof terms $\langle V_{\mathrm{mid}} \rangle_1$, $\langle \alpha_v V_{\mathrm{mid}} \rangle_1$, $\langle W_{\mathrm{mid}} \rangle_2$, $\langle \alpha_w W_{\mathrm{mid}} \rangle_1$, $\langle Y_{\mathrm{mid}} \rangle_1$, $\langle \alpha_y Y_{\mathrm{mid}} \rangle_1$, and $\langle Z \rangle_1$ encode circuit values $x_{l+m+1}, \ldots, x_k$; in the zero-knowledge variant, these terms are randomised so that they do not reveal any information about $x_{l+m+1}, \ldots, x_k$. In the multi-client case, additionally, the inputs of all input parties and the outputs of all result parties need to be encoded such that no other party learns any information about them. Therefore, we extend the proof with *blocks* of the above seven terms for each input and result party, which are constructed in the

Algorithm 6 ProofBlock

1: **function** ProofBlock($BK; \vec{x}; \delta_v, \delta_w, \delta_y$)
2: $\langle V\rangle_1 \leftarrow \langle r_v t\rangle_1 \delta_v + \sum_i \langle r_v v_i\rangle_1 x_i$; $\langle V'\rangle_1 \leftarrow \langle r_v \alpha_v t\rangle_1 \delta_v + \sum_i \langle r_v \alpha_v v_i\rangle_1 x_i$
3: $\langle W\rangle_2 \leftarrow \langle r_w t\rangle_2 \delta_w + \sum_i \langle r_w w_i\rangle_2 x_i$; $\langle W'\rangle_1 \leftarrow \langle r_w \alpha_w t\rangle_1 \delta_w + \sum_i \langle r_w \alpha_w w_i\rangle_1 x_i$
4: $\langle Y\rangle_1 \leftarrow \langle r_y t\rangle_1 \delta_y + \sum_i \langle r_y y_i\rangle_1 x_i$; $\langle Y'\rangle_1 \leftarrow \langle r_y \alpha_y t\rangle_1 \delta_y + \sum_i \langle r_y \alpha_y y_i\rangle_1 x_i$
5: $\langle Z\rangle_1 \leftarrow \langle r_v \beta t\rangle_1 \delta_v + \langle r_w \beta t\rangle_1 \delta_w + \langle r_y \beta t\rangle_1 \delta_y + \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i\rangle_1 x_j$
6: **return** $(\langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1)$

Algorithm 7 CheckBlock

1: **function** CheckBlock($BV; \langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1$)
2: **if** $e(\langle V\rangle_1, \langle \alpha_v\rangle_2) = e(\langle V'\rangle_1, \langle 1\rangle_2)$
3: $\wedge e(\langle \alpha_w\rangle_1, \langle W\rangle_2) = e(\langle W'\rangle_1, \langle 1\rangle_2)$
4: $\wedge e(\langle Y\rangle_1, \langle \alpha_y\rangle_2) = e(\langle Y'\rangle_1, \langle 1\rangle_2)$
5: $\wedge e(\langle Z\rangle_1, \langle 1\rangle_2) = e(\langle V\rangle_1 + \langle Y\rangle_1, \langle \beta\rangle_2) e(\langle \beta\rangle_1, \langle W\rangle_2)$ **then**
6: **return** $\top$
7: **else**
8: **return** $\bot$

same way as the seven proof terms above. Although some result parties could share a block of output values, for simplicity we assign each result party its own block in the protocol.

To produce a block containing values $\vec{x}$, a party first samples three random field values δ_v, δ_w, and δ_y and then executes ProofBlock, cf. Algorithm 6. The BK argument to this algorithm is the *block key*; the subset of the evaluation key terms specific to a single proof block. Because each input party should only provide its own input values and should not affect the values contributed by other parties, each proof block must be restricted to a subset of the wires. This is achieved by modifying Pinocchio's key generation such that, instead of a sampling a single value β, one such value, β_j, is sampled for each proof block j and the terms $\langle r_v \beta_j v_i + r_w \beta_j w_i + r_y \beta_j y_i\rangle_1$ are only included for wires indices i belonging to block j. That is, the jth block key is

$$\begin{aligned} BK_j = \{&\langle r_v v_i\rangle_1, \langle r_v \alpha_v v_i\rangle_1, \langle r_w w_i\rangle_2, \langle r_w \alpha_w w_i\rangle_1, \langle r_y y_i\rangle_1, \langle r_y \alpha_y y_i\rangle_1, \\ &\langle r_v \beta_j v_i + r_w \beta_j w_i + r_y \beta_j y_i\rangle_1, \langle r_v \beta_j t\rangle_1, \langle r_w \beta_j t\rangle_1, \langle r_y \beta_j t\rangle_1\}, \end{aligned}$$

with i ranging over the indices of wires in the block. Note that ProofBlock only performs linear operations on its $\vec{x}$, δ_v, δ_w and δ_y inputs. Therefore this algorithm does not have to be modified to compute on secret shares.

A Trinocchio proof in the multi-client setting now consists of one block $\vec{Q}_i = (\langle V_i\rangle_1, \ldots, \langle Z_i\rangle_1)$ for each input and result party, one block $\vec{Q}_{\text{mid}} = (\langle V_{\text{mid}}\rangle_1, \ldots, \langle Z_{\text{mid}}\rangle_1)$ of internal wire values, and Pinocchio's $\langle H\rangle_1$ element. Verification of such a proof consists of checking correctness of each block, and checking correctness of $\langle H\rangle_1$. The validity of a proof block can be verified using CheckBlock, cf. Algorithm 7. Compared to the Pinocchio verification key, our verification key contains "block verification keys" BV_i (i.e., elements $\langle \beta_j\rangle_1$ and $\langle \beta_j\rangle_2$) for each block instead of just $\langle \beta\rangle_1$ and $\langle \beta\rangle_2$. Apart from the relations inspected by CheckBlock, one other relation is needed to verify a Pinocchio proof: the divisibility check of Equation (3.4) (Section 3.2.2). In the protocol, the algorithm that verifies this relation will be called CheckDiv. We denote the modified setup of the evaluation and verification keys by hybrid call MKeyGen.

3.5.2 The Protocol

In this section we present our multi-client Trinocchio protocol in more detail. As before, we assume that each input party provides only a single input and each result party receives only a single output; that is, each block from Section 3.5.1 consists of only one wire. It should be clear from Section 3.5.1 how this can be generalised.

3.5.2.1 Communication Model and Notation

We assume synchronous communication; pairwise secure channels between the input parties and workers; between the workers themselves; and between the workers and result parties. To ensure agreement between the parties about the inputs for the computation, we additionally assume a bulletin board. Through this bulletin board, parties can publish messages which can then be retrieved by any other party. Messages on the bulletin board are authenticated. In our protocol, we denote a party posting a message m as $\mathsf{Post}(m)$. For convenience, we don't explicitly denote a party retrieving information from the bulletin board; instead, we take $\mathsf{Post}(m)$ to mean that any party can now use the value for m.

3.5.2.2 Commitment Scheme

We use a commitment scheme, which allows a party to commit to a certain value, without revealing that value to other parties, but, when at a later time this value is revealed, the other parties can be certain that the revealed value is equal to the original committed to value. Each party has its own public commitment key k and a commitment to a value v using randomness r is denoted $\mathsf{Commit}_k(v; r)$. Because, given explicit randomness, the commitment algorithm is deterministic, the commitment can be opened by simply revealing (v, r). Then any party can verify the commitment by simply recomputing it. To ensure input independence, the commitment scheme must be non-malleable. Each input party will produce one commitment, so each commitment key is used only once.

3.5.2.3 Overview of the Protocol

Our protocol is shown as Algorithm 8. The protocol starts with hybrid calls to obtain the trusted commitment keys and Trinocchio evaluation and verification keys (lines 2–3). The remainder of the protocol consists of the *input phase* (lines 4–16), in which the input parties provide their inputs to the workers; the *computation phase*, in which the workers compute the function and Pinocchio proof (lines 17–31); and the *result phase*, in which the result parties obtain the output from the workers and verify its correctness (lines 32–41).

3.5.2.4 Input Phase

In the input phase, each input party provides its input to the workers. Compared to the single-client case, in which the input party simply provided secret shares of its inputs, we need to take several additional steps. Namely, we need each input party to provide a block for its inputs that other parties can use to verify the proof; and we need to guarantee input independence, namely, that input parties cannot choose their inputs depending on those of others.

To achieve these goals, we proceed as follows. First, each input party computes a block for its input (line 5). Having each input party post its block on the bulletin board would break input independence (in effect, it binds the input parties who provide the blocks first). We circumvent this by letting each input party post a commitment to its block first (line 6). After all commitments have been posted, the input parties post the openings to the commitments, i.e., the blocks and commitment randomness (line 7). (This guarantees input independence because in the security proof, the inputs of the honest parties can still be changed after the corrupted parties provide their inputs.) After this, the validity of the commitments (line 9) and blocks (line 10) are checked; if any input party provided incorrect information, the computation is aborted.

After the input blocks have been posted and checked, the inputs are provided to the workers in the form of $(2\theta, n)$ shares (line 11). The shared information is both input $[x_i]$ and block randomness $[\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}]$: the workers need this latter information to compute the proof's $\langle H \rangle_1$ element. Note that we use $(2\theta, n)$ shares: because $n = 2\theta + 1$, the shares of all workers recombine to a unique value and we do not need to worry about input parties handing out inconsistent shares. The workers check that the shares correspond to the broadcast block by computing additive shares of the block, posting them, and checking if their Shamir recombination (denoted by Combine) matches the value on the bulletin board (lines 13–15). Finally, the $(2\theta, n)$-shares are converted into (θ, n)-shares (each worker (θ, n)-shares its share and applies recombination a la [40]) used for the remainder of the computation (line 16).

3.5.2.5 Computation Phase

In the computation phase, the workers compute function f, and produce a Pinocchio proof that this computation was performed correctly. The computation of f (line 17) and coefficients $\vec{H}'$ of the polynomial $h = (v \cdot w - y)/t$ (lines 18–21) are the same as in the single-client case. To generate the proof block for the internal wires, the workers first generate shared random values $[\![\delta_{v,\mathrm{mid}}]\!], [\![\delta_{w,\mathrm{mid}}]\!], [\![\delta_{y,\mathrm{mid}}]\!]$ (line 22): for instance, by letting each party share a random value or using pseudo-random secret sharing. They then call ProofBlock to produce the block using the shared wires and randomness (line 23). The blocks for the result parties are generated in the same way (lines 24–26). The coefficients of the randomised quotient polynomial $\vec{H}$ are computed from $\vec{H}'$ analogously to the zero-knowledge variant of Pinocchio (Appendix 3.2.3); note that this requires computing overall randomness δ_v, δ_w, δ_y that is the sum of the randomness from all blocks in the proof. This gives $(2\theta, n)$ shares $[\langle H \rangle_1]$ of proof element $\langle H \rangle_1$ (line 30)

Having computed shares of all proof elements, the workers now post these shares on the bulletin board so that everybody can combine them to obtain the full proof. Note that the shares of individual workers might statistically depend on information that we do not want to reveal such as internal circuit wires. To avoid any problems because of this, the workers first re-randomise their proof elements by adding a new random sharing of zero; for instance, obtained by letting each worker share zero or using pseudo-random zero sharing (line 31).

3.5.2.6 Result Phase

In the result phase, the result parties obtain their computation results, and verify them with respect to the information on the bulletin board. First, the result parties obtain secret shares of their output values, and the randomness used in their proof blocks (line 32). Then, they combine the values from the bulletin board into a full multi-client Pinocchio proof (lines 34–36), and verify this proof (lines 37–38). Finally, they recombine their output values (line 39), check if the secret shares of their output values correspond to the posted proof block (line 40), and output the computation result (line 41).

3.5.3 Security of the Trinocchio Protocol

Analogously to the single-client case, we obtain the following result:

Theorem 5. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the multi-client Trinocchio protocol. Assuming the d-PKE, $(4d + 4)$-PDH, and $(8d + 8)$-SDH assumptions:*

- *Trinocchio correctly evaluates f in the* $(\mathsf{ComGen}, \mathsf{MKeyGen})$*-hybrid model.*
- *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the* $(\mathsf{ComGen}, \mathsf{MKeyGen})$*-hybrid model.*

We stress that "at most θ workers are passively corrupted" includes both the case when the adversary is passively corrupted, and corrupts at most θ workers (as well as arbitrarily many input and result parties); and the case when the adversary is actively corrupted, and corrupts no workers (but arbitrarily many input and result parties)

We give a proof sketch of this theorem in the paper [72]. The complete proof is given in the full version of the paper [72]. To prove secure function evaluation, we obtain privacy by simulating the multiparty computation of the proof with respect to the adversary without using honest inputs. To prove correct function evaluation, we run the protocol together with the adversary: if this gives a fake Pinocchio proof, then one of the underlying problems can be broken.

In the single-client case, we remarked that Trinocchio actually provides security against up to θ *actively* corrupted workers. Namely, although θ actively corrupted workers may manipulate the computation of the function and proof, they do not learn any information from this because they do not see the resulting proof that the client gets. In our multi-client protocol, it is less natural to assume that the workers cannot see the resulting proof; and in fact, in our protocol, corrupted workers *do* see the full proof as it is posted on the bulletin board. It should be possible to obtain some privacy guarantees against actively malicious workers (who do not collude with any result parties) by letting the result parties provide proof contributions directly to the result parties instead of posting them on the bulletin board. We leave an analysis for future work.

Algorithm 8 Trinocchio: n-party verifiable computation

1: $\triangleright$ Input parties $\mathcal{I}$ have x_i, result parties $\mathcal{R}$ output $(x_{l+1},\ldots,x_{l+m}) = f(x_1,\ldots,x_l)$
2: **parties** $i \in \mathcal{I}$ **do** $(k_1,\ldots,k_n) \leftarrow \mathsf{ComGen}()$
3: **parties** $i \in \mathcal{I}\cup\mathcal{W}\cup\mathcal{R}$ **do** $(EK = (\{BK_i\}_i,\ldots), VK = (\{BV_i\}_i,\ldots)) \leftarrow \mathsf{MKeyGen}()$
4: **parties** $i \in \mathcal{I}$ **do** $\triangleright$ input phase
5: $(\delta_{v,i},\delta_{w,i},\delta_{y,i}) \in_{\mathrm{R}} \mathbb{F}^3$; $\vec{Q}_i \leftarrow \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i},\delta_{w,i},\delta_{y,i})$
6: sample commitment randomness ρ_i; $c_i \leftarrow \mathsf{Commit}_{k_i}(\vec{Q}_i;\rho_i)$; $\mathsf{Post}(c_i)$
7: $\mathsf{Post}(\vec{Q}_i,\rho_i)$
8: **for all** $j \in \mathcal{I}\setminus\{i\}$ **do**
9: **if** $c_j \neq \mathsf{Commit}_{k_j}(\vec{Q}_j;\rho_j)$ **then** abort the protocol
10: **if** $\mathsf{CheckBlock}(BV_j;\vec{Q}_j) = \perp$ **then** abort the protocol
11: create $(2\theta,n)$-shares $([x_i],[\delta_{v,i}],[\delta_{w,i}],[\delta_{y,i}])$ and distribute to the workers
12: **parties** $\mathcal{W}$ **do**
13: **for all** $i \in \mathcal{I}$ **do**
14: $[\vec{Q}_i] \leftarrow \mathsf{ProofBlock}(BK_i;[x_i];[\delta_{v,i}],[\delta_{w,i}],[\delta_{y,i}])$; $\mathsf{Post}([\vec{Q}_i])$
15: **if** $\mathsf{Combine}([\vec{Q}_i]) \neq \vec{Q}_i$ **then** abort the protocol
16: convert $(2\theta,n)$ shares $([x_i],[\delta_{v,i}],[\delta_{w,i}],[\delta_{y,i}])$ to (θ,n) shares $([\![x_i]\!],\ldots)$
17: compute $([\![x_{l+1}]\!],\ldots,[\![x_k]\!])$ using MPC $\triangleright$ computation phase
18: $[\![\vec{v}]\!] \leftarrow \{(\sum_i v_i(\omega_j)\cdot[\![x_i]\!]\}_j$; $[\![\vec{V}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\vec{v}]\!])$; $[\![\vec{v}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\vec{V}]\!])$
19: $[\![\vec{w}]\!] \leftarrow \{(\sum_i w_i(\omega_j)\cdot[\![x_i]\!]\}_j$; $[\![\vec{W}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\vec{w}]\!])$; $[\![\vec{w}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\vec{W}]\!])$
20: $[\![\vec{y}]\!] \leftarrow \{(\sum_i y_i(\omega_j)\cdot[\![x_i]\!]\}_j$; $[\![\vec{Y}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\vec{y}]\!])$; $[\![\vec{y}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\vec{Y}]\!])$
21: $[\vec{h}'] \leftarrow \{([\![\vec{v}'_j]\!]\cdot[\![\vec{w}'_j]\!] - [\![\vec{y}'_j]\!])/t(\Omega_j)\}_j$; $[\vec{H}'] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\vec{h}'])$
22: $([\![\delta_{v,\mathrm{mid}}]\!],[\![\delta_{w,\mathrm{mid}}]\!],[\![\delta_{y,\mathrm{mid}}]\!]) \in_{\mathrm{R}} \mathbb{F}^3$
23: $[\![\vec{Q}_{\mathrm{mid}}]\!] \leftarrow \mathsf{ProofBlock}(BK_{\mathrm{mid}};[\![x_{l+m+1}]\!],\ldots,[\![x_k]\!];[\![\delta_{v,\mathrm{mid}}]\!],[\![\delta_{w,\mathrm{mid}}]\!],[\![\delta_{y,\mathrm{mid}}]\!])$
24: **for all** $i \in \mathcal{R}$ **do**
25: $([\![\delta_{v,i}]\!],[\![\delta_{w,i}]\!],[\![\delta_{y,i}]\!]) \in_{\mathrm{R}} \mathbb{F}^3$
26: $[\![\vec{Q}_i]\!] \leftarrow \mathsf{ProofBlock}(BK_i;[\![x_i]\!];[\![\delta_{v,i}]\!],[\![\delta_{w,i}]\!],[\![\delta_{y,i}]\!])$
27: $[\delta_v] \leftarrow [\delta_{v,\mathrm{mid}}] + \sum_{i\in\mathcal{I}\cup\mathcal{R}}[\delta_{v,i}]$
28: $[\delta_w] \leftarrow [\delta_{w,\mathrm{mid}}] + \sum_{i\in\mathcal{I}\cup\mathcal{R}}[\delta_{w,i}]$
29: $[\delta_y] \leftarrow [\delta_{y,\mathrm{mid}}] + \sum_{i\in\mathcal{I}\cup\mathcal{R}}[\delta_{y,i}]$
30: $[\vec{H}] \leftarrow [\vec{H}'] + [\![\delta_v]\!][\![\vec{W}]\!] + [\![\delta_w]\!][\![\vec{V}]\!] + [\![\delta_v]\!][\![\delta_w]\!]\vec{T} - [\![\delta_y]\!]$; $[\langle H\rangle_1] \leftarrow \sum_{j=0}^{d}\langle s^j\rangle_1[\vec{H}_j]$
31: $\mathsf{Post}([\![\vec{Q}_{\mathrm{mid}}]\!] + [\![0]\!])$; $\mathsf{Post}([\langle H\rangle_1] + [0])$; **for all** $i \in \mathcal{R}$ **do** $\mathsf{Post}([\![\vec{Q}_i]\!] + [\![0]\!])$
32: **for all** $i \in \mathcal{R}$ **do** send $([\![x_i]\!],[\![\delta_{v,i}]\!],[\![\delta_{w,i}]\!],[\![\delta_{y,i}]\!])$ to res. party i $\triangleright$ result phase
33: **parties** $i \in \mathcal{R}$ **do**
34: **for all** $j \in \mathcal{R}$ **do** $\vec{Q}_j \leftarrow \mathsf{Combine}([\vec{Q}_j])$
35: $\vec{Q} \leftarrow \mathsf{Combine}([\![\vec{Q}_{\mathrm{mid}}]\!]) + \sum_{j\in\mathcal{I}\cup\mathcal{R}}\vec{Q}_j$
36: $\langle H\rangle_1 \leftarrow \mathsf{Combine}([\langle H\rangle_1])$
37: **if** $\mathsf{CheckBlock}(BV_{\mathrm{mid}};\vec{Q}_{\mathrm{mid}}) = \perp \vee \exists j : \mathsf{CheckBlock}(BV_j;\vec{Q}_j) = \perp \vee$
38: $\mathsf{CheckDiv}(VK;\vec{Q};\langle H\rangle_1) = \perp$ **then** output $\perp$ and abort protocol
39: $(x_i,\delta_{v,i},\delta_{w,i},\delta_{y,i}) \leftarrow \mathsf{Combine}([\![x_i]\!],[\![\delta_{v,i}]\!],[\![\delta_{w,i}]\!],[\![\delta_{y,i}]\!])$
40: **if** $\vec{Q}_i \neq \mathsf{ProofBlock}(BK_i;x_i;\delta_{v,i},\delta_{w,i},\delta_{y,i})$ **then** output $\perp$ and abort protocol
41: output x_i

3.6 Performance

In this section, we show that our approach indeed adds privacy to verifiable computation with little overhead. We demonstrate this in two case studies. First, we take the "MultiVar Poly" application from [61], and show that using Trinocchio, this computation can be outsourced in a private and correct way at essentially the same cost as letting three workers each perform the Pinocchio computation. Second, we show that, using Trinocchio, the performance of "verification by validation" due to [70] can be considerably improved: in particular, we improve the client's performance by several orders of magnitude.

In our experiments, one client outsources the computation to three workers. In particular, we use multiparty computation based on $(1, 3)$ Shamir secret sharing. As discussed in Sections 3.4.3 and 3.5.3, this guarantees privacy against one passively corrupted worker (or, in the single-client case against θ actively corrupted workers when the multiparty computation protocol does not leak any information). We did not implement the multiple client scenario; this would add small overhead for the workers, with verification effort growing linearly in he number of input and result parties but remaining small and independent from the computation size. To simulate a realistic outsourcing scenario, we distribute computations between three Amazon EC2 "m3.medium" instances[3] around the world: one in Oregon, United States; one in Ireland; and one in Tokyo, Japan. Multiparty computation requires secure and private channels: these are implemented using SSL.

3.6.1 Case Study: Multivariate Polynomial Evaluation

In [61], Pinocchio performance numbers are presented showing that, for some applications, Pinocchio verification is faster than native execution. One of these applications, "Multi-Var Poly", is the evaluation of a constant multivariate polynomial on five inputs of degree 8 ("medium") or 10 ("large"). In this case study, we use Trinocchio to add privacy to this outsourcing scenario.

We have made an implementation[4] of Trinocchio's Compute algorithm (Algorithm 5) that is split into two parts. The first part performs the evaluation of the function f (line 4), given as an arithmetic circuit, using the secret sharing implementation of VIFF(We use the arithmetic circuit produced by the Pinocchio compiler, hence f is exactly the same as in [61].) Note that, because f is an arithmetic circuit, this step does not leak any information against actively corrupted workers. Hence, in the single-client outsourcing scenario of Section 3.4, we achieve privacy against one actively corrupted worker. The second part is a completely new implementation of the remainder of Trinocchio using [58]'s implementation of the discrete logarithm groups and pairings from [13].

Table 3.1 shows the performance numbers of running this application in the cloud with Trinocchio. Significantly, evaluating the function f using passively secure multiparty computation (i.e., line 4 of Compute) is more than twenty times cheaper than computing the Pinocchio proof (i.e., lines 5–16 of Comp). Moreover, we see that computing the Pinocchio proof in the distributed setting takes around the same time (per party) as in the non-distributed setting. Indeed, this is what we expect because the computation that takes place is exactly the same as in the non-distributed setting, except that it happens to take place on shares rather than the

[3] Running Intel Xeon E5-2670 v2 Ivy Bridge with 4 GB SSD and 3.75 GiB RAM
[4] Implementation available at http://meilof.home.fmf.nl/

	# mult	Pinoc.	Dist f	Dist π	Trinoc.	Verif.
MultiVar Poly, Medium	203428	2102	96	2092	2187	0.04
MultiVar Poly, Large	571046	6458	275	6427	6702	0.05

Table 3.1: Performance of multivariate polynomial evaluation with Trinocchio: number of multiplications in f; time for single-worker proof; time per party for computing f and proof, and total; and verification time (all times in seconds)

actual values itself. Hence, according to these numbers, the cost of privacy is essentially that the computation is outsourced to three different workers, that each have to perform the same work as one worker in the non-private setting. Finally, as expected, verification time completely vanishes compared to computation time.

Our performance numbers should be interpreted as estimates. Our Pinocchio performance is around 8–9 times worse than in [61]; but on the other hand, we could not use their proprietary elliptic curve and pairing implementations; and we did not spend much time optimising performance. Note that, as expected, our Pinocchio and Trinocchio implementations have approximately the same running time. If Trinocchio would be based on Pinocchio's code base, we would expect the same. Moreover, apart from combining the proofs from different workers, the verification routines of Pinocchio and Trinocchio are exactly the same, so achieving faster verification than native computation as in [61] should be possible with Trinocchio as well. We also note that VIFF is not known for its speed, so replacing VIFF with a different multiparty computation framework should considerably speed up the computation of f.

3.6.2 Speeding Up Verification by Validation

In [70], the idea is proposed to speed up verifiable outsourcing by exploiting the fact that, to see if a solution to a computation is correct, it is often not necessary to consider the whole circuit. Specifically, instead of proving that $\vec{y} = f(\vec{x})$, workers prove that $\phi(\vec{x}, \vec{a}, \vec{y})$ holds for some predicate ϕ and "certificate" $\vec{a}$. [70] proposes to use ElGamal encryptions and zero-knowledge proofs to prove $\phi(\vec{x}, \vec{a}, \vec{y})$. This gives feasible performance, although the overhead compared to just computing f is still quite large. We now show that using Trinocchio both reduces the worker effort and dwarfs the client effort.

Specifically, [70] presents a case study in linear programming. Given a matrix $\vec{A} \in \mathbb{Z}^{m\times n}$ and vectors $\vec{b} \in \mathbb{Z}^m$, $\vec{c} \in \mathbb{Z}^n$, linear programming asks to find vector $\vec{x} \in \mathbb{Z}^n$ and quotient q such that $q > 0$; $\vec{x} \geq 0$; $\vec{A} \cdot \vec{x} \leq q \cdot \vec{b}$, and $(\vec{c} \cdot \vec{x})/q$ is minimal. (In multiparty computation, $\mathbb{Z}$ is embedded into a sufficiently large field $\mathbb{F}$.) To solve this problem, heavy iterative algorithms such as the simplex algorithm are needed; but given the so-called "dual solution" $\vec{p} \in \mathbb{Z}^m$ it is easy to verify that $\vec{x}$ is optimal by checking that $q > 0$; $\vec{p} \cdot \vec{b} = \vec{c} \cdot \vec{x}$; $\vec{A} \cdot \vec{x} \leq q \cdot \vec{b}$; $\vec{x} \geq 0$; $\vec{A} \cdot \vec{p} \leq q \cdot \vec{c}$; and $\vec{p} \leq 0$. This criterion can be formulated as a set of polynomial equations [70], and, in fact, as a QAP, by formulating checks like $q > 0$ in terms of bit decompositions, e.g., $q - 1 = a_0 + a_1 \cdot 2 + a_2 \cdot 2^2 + \ldots$ and $a_0 \cdot (1 - a_0) = 1$, $a_1 \cdot (1 - a_1) = 0$, and so on.

We have adapted the simplex LP solver from [70] to work over the field we need for our discrete logarithm and pairing groups; and then used Trinocchio's Compute to produce the proof that the computed LP solution is optimal. Our performance numbers are shown in Table 3.2. As shown, producing the Pinocchio proof is only a small percentage of the total distributed computation, ranging from 5% to 13% of total computation time. (The percentage decreases with problem size. Asymptotically, the time needed to evaluate f is $O(m \cdot n \cdot (I + l))$, with $m \times n$ the dimensions

LP size	blp	#it	Comp	Cert	Proof	Total	Ver
5x5	31	4	89	4	8	102	0.07
20x20	40	9	289	23	52	364	0.07
48x70	34	25	1080	61	172	1314	0.07
48x70	65	48	2702	119	268	3090	0.07
103x150	61	62	5415	308	713	6436	0.07
288x202	93	176	48781	1257	2479	52516	0.06

Table 3.2: Performance of verifiable linear programming by validation with Trinocchio: bitlength of solution numbers, number of simplex iterations, time for computation, certificate computation, proof, and total; and verification time (all times in seconds)

of the LP, I the number of iterations and l the bitlength needed during the computation; proof time is $O(lmn \cdot \log lmn)$.) Verification is very fast, and in particular much faster than evaluating the simplex algorithm with VIFF's local execution mechanism (which takes 78s on the biggest LP).

Comparing our Trinocchio approach to the ElGamal-based proofs of [70], our proofs are not only much faster to verify, but also faster to produce. For verification, [70] report times that are two-thirds of proof time, where our verification time is almost constant (for our problem sizes, the dominant factor is the computation of the constantly many pairings) and also asymptotically much better, since it only depends linearly on the *sum* of the LP dimensions, and not at all on the bitlength. Concerning proof production, our measured times are comparable (even slightly better), but the circumstances are not. Namely, while [70]'s experiments use machines comparable to ours, they measure local communication whereas we measure communication in the cloud. Since [70]'s proof production requires significant communication, their running time in the cloud should be higher than reported in [70]. (Although we could not verify this latter claim, we did find that running [70]'s LP solver in the cloud is around four times slower than in [70], which is probably for the same reason.) On the other hand, [70]'s proof production has slightly better asymptotics: theirs has $O(lnm)$ running time compared to our $O(lnm \cdot \log lmn)$.

3.7 Discussion and Conclusion

In this chapter, we have presented Trinocchio, a system that adds privacy to the Pinocchio verifiable computation scheme essentially at the cost of replicating the Pinocchio proof production algorithm at three (or more) servers. Trinocchio has the same correctness and security guarantees as Pinocchio; distributing the computation between $2\theta + 1$ workers gives privacy if at most θ of them are corrupted. We have shown in a case study that the overhead is indeed small.

As far as we are aware, our work is the first to deliver efficient verifiable computation (i.e., with cryptographic guarantees of correctness and practical verification times independent of the computation size) with privacy guarantees. Although privacy is only guaranteed if not too many of the workers are corrupt, the use of verifiable computation ensures that the outcome of the protocol cannot be manipulated by the workers. This allows us to hedge against an adversary being more powerful than anticipated in a real world scenario.

Existing verifiable computation constructions in the single-worker setting [38, 44, 35] use very expensive cryptography, while multiple-worker efforts to provide privacy [4] do not guarantee

correctness if all workers are corrupted. In contrast, existing works from the area of multiparty computation [9, 71, 70] deliver privacy and correctness guarantees, but have much less efficient verification.

A major limitation of Pinocchio-based approaches is that they assume trusted set-up of the (function-dependent) evaluation and verification keys. In the single-client setting, the client could perform this set-up itself, but in the multiple-client setting, it is less clear who should do this. In particular, whoever has generated the evaluation and verification keys can use the values used during key generation as a trapdoor to generate proofs of false statements. Even though key generation can likely be distributed using the same techniques we use to distribute proof production, it remains the case that all generating parties together know this trapdoor.

Part II

Verifiable Computation From Hardware Assumptions

Chapter 4

Formalizing Hardware Assumptions: Attested Computation

4.1 Overview

PRACTICE proposes several application scenarios, in which sensitive data is routinely manipulated, typically requiring strong security guarantees against tampering and information leakage. However, satisfying these guarantees while considering platforms that might have vulnerabilities, or that are outright not trustworthy, is a major challenge.

Novel capabilities of modern trusted hardware allow for a promising starting point: *remote attestation* capabilities. These computational platforms are equipped with technology that can guarantee to a remote user various degrees of integrity and isolation to software running within its premises. For example, the Trusted Platform Module (TPM) can provide certified measurements of the state of a platform, and can be used to guarantee integrity of BIOS and boot code right before its execution. More recent technological advancements have expanded the scope and guarantees of trusted hardware, offering the ability to run applications in "clean-slate" isolated execution environments (IEE). These environments are equipped to produce cryptographically authenticated reports to attest their executions.

Another major challenge that arises from this approach is to provide security guarantees that go beyond heuristic arguments. In this context, this document presents solutions considering the methodology of "provable security". The approach provides well-established definitional paradigms for all basic primitives and some of the more used protocols. However, the success of taking this approach to new and more complex scenarios fundamentally hinges on one's ability to tackle with scalability problems, as models and proofs tend to get unwieldy. It may be tempting to assume that, for the analysis of such protocols, designing security models is a simple matter of overlaying/merging the trust model induced by the use of such hardware over well-established security abstractions. Unfortunately, this is not the case.

Two important issues show that neither existent models nor existent techniques are immediately suitable for the analysis of IEE-based protocols. The first in the concept of a *party* which is a key notion in specifying and reasoning about the security of distributed systems. Traditionally, one considers security where there is a PKI and at least some of the parties have associated public keys, and in that sense parties and their cryptographic material are essentially the same. However, in this context users are not expected to maintain long-term keys, which makes long term cryptographic material inadequate as a technical anchor for a party's participation in such

a protocol. The second issue is *composability*. In standard scenarios that involve composing a key-exchange protocol with another functionality, in which the only information passed from the key exchange is the encryption key, then one can design and analyze the two parts separately. However, reliance on the IEE breaks the independence assumption that allows for composability results: the code run by the IEE needs to be loaded at once, or else no isolation guarantees are given by the trusted hardware.

This chapter is dedicated to presenting the state-of-the-art technological advancements with respect to hardware providing IEE-capabilities, and to propose novel models and provably secure protocols capable of producing feasible implementations that rely on these trusted solutions. As such, we begin by describing the technology taken as inspiration: Intel's Software Guard Extensions, and its related work. We then provide a high-level abstraction of IEEs, and present the cryptographic primitive of *attested computation*, the formalization of raw guarantees provided by IEEs with cryptographic functionalities. This abstraction is then used in the following chapters as the basis for the design and security analysis of protocols for secure computation outsourcing and function evaluation. The models and foundations detailed in this chapter are based on those presented in [7], which is a direct result of work developed in the context of Task 13.2 of PRACTICE project.

4.2 Software guard extensions

Intel's Software Guard Extensions (SGX) is a novel instruction set architecture [25] that aims to solve the secure remote computation problem by leveraging trusted hardware in the remote machine. SGX relies on *attestation*, which proves to a user that it is communicating to a specific piece of software running within secure containers under specified isolation conditions, named *enclaves*.

An SGX-enabled processor is equipped to protect the integrity and confidentiality of the computation inside the enclave, by isolating its code and data from the outside environment. This includes the operating system and hypervisor, as well as any hardware devices attached to the system bus.

The proof itself is a cryptographic signature that certifies the hash of the enclave contents (namely the code and memory). This setting does not prevent the remote computer to run any specific software within enclaves, but instead allows for the user to reject any result produced by an enclave whose contents do not match the expected value.

Enclaves

In an SGX-enabled hardware, a subset of memory is reserved as *Processor Reserved Memory* (PRM). The CPU is responsible to protect this memory structure from all external (non-enclave) memory access. The PRM stores the enclave page cache, storing enclave-related information such as code and data. The system software is in charge of assigning these enclave pages to enclaves, and the CPU will make sure that each enclave page corresponds to exactly one enclave. SGX provides a set of special CPU instructions allowing for the management of enclaves, and we now provide a simple description of the ones used for basic enclave operations.

Enclave creation begins with `ECREATE`, establishing the initial environment within the protected range of addresses, and allocating an associated data structure for the enclave on the

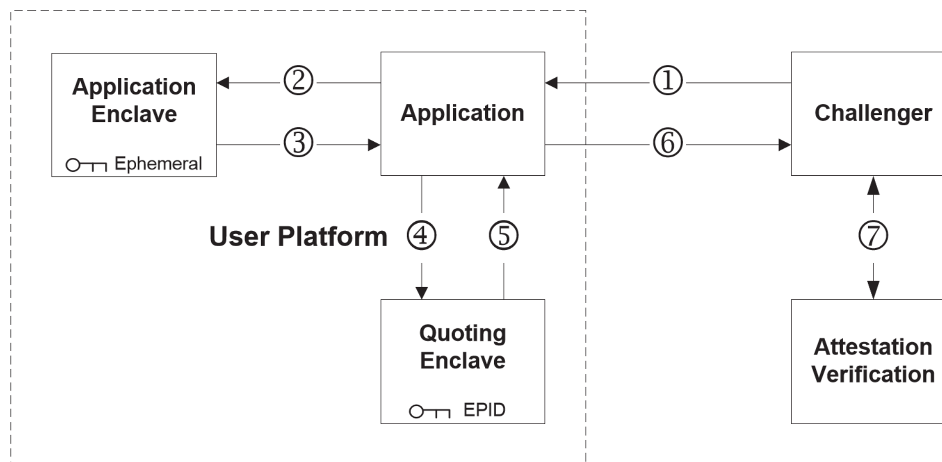


Figure 4.1: Process for remote attestation

PRM. This transits into an initially valid enclave construction. As the enclave is built, the instruction EADD allows the allocation of additional memory pages to the enclave. At each page, EEXTEND is used to measure allocated space. If all added pages have been measured, then the enclave is ready for initialization with EINIT. According to the information within the enclave, this instruction establishes fixed valid entry points for the enclave.

When the enclave is successfully initialized, it enters a locked state, preventing changes on enclave memory pages. From this point onward, the user may enter the enclave under program control with EENTER, specifying a valid address as entry point. Within the program mode, the enclave is restricted from performing any instruction listed on a pre-established set of illegal commands, resulting in an error. From there, it is possible to obtain cryptographic keys with EGETKEY and generate reports for other enclaves with EREPORT. These reports can be used for enclaves to authenticate messages among other enclaves within the same platform.

The enclave is terminated via EEXIT. If something makes the enclave halt (either an expected or unexpected occurrence), the *Asynchronous Enclave Exit* is triggered. This saves the enclave's state using cryptographic techniques, allowing to reenter the enclave using ERESUME. From the moment an enclave is locked, it is possible to execute management commands, such as evictions blocks or loads.

In order to provide inter-platform enclave attestation, SGX-enabled hardware also includes an Enhanced Privacy ID scheme [21] that is used by a special enclave called *quoting enclave* for signing enclave attestations. EPID is a group signature scheme allowing a platform to construct signatures without uniquely identifying the actual platform that has produced it. Only the quoting enclave has access to this EPID key, which is bound to the version of the underlying firmware. The mechanism for *remote attestation* is proposed in [5], can be depicted in Figure 4.1 (from the same paper) and is described as follows:

1. The remote machine establishes communication with an SGX-enabled platform and issues a challenge to validate the machine as running the necessary components inside an enclave, including a nonce for liveness purposes.

2. The application sends to the enclave the identity of the quoting enclave and the remote challenge.

3. The enclave generates a report (using the described instruction EREPORT) comprising the enclave contents and the challenge given to it, and returns it to the application.
4. The application simply forwards the received data structure to the special quoting enclave.
5. The quoting enclave verifies the authenticity of the received report (using EGETKEY to retrieve the associated key) and signs it with its unique EPID key. This produces what is called a *quote*, which is returned to the application.
6. The application forwards the quote to the remote machine.
7. The challenger uses a EPID public key to validate the signature included in the received quote. It can now verify the integrity of the signed data and check the response for the challenge proposed in (1).

The work developed during Task 13.2 PRACTICE is particularly focused on formalizing and proving security guarantees that can be provided by schemes such as this quoting mechanism, that allows for enclaves to prove to external users that they are running according to a specific code and contents. To the best of our knowledge, a modular security analysis of these hardware-based mechanisms has yet to be performed, so this work provides the first efforts towards precisely specifying the idealized models that allow for the implementation of provably secure protocols relying on trusted hardware for remote attestation, such as SGX.

Related work

Some work that looks at provable security for realistic protocols using trusted hardware-based protocol has been developed around approaches using the Trusted Platform Module [19, 74, 20, 37, 36]. However, the functionality and efficiency of protocol offered by TPM makes them more suitable for ensuring integrity of programs right before execution, rather than the run-time guarantees that SGX provides.

Trusted Execution Technology (TXT) [46] is another approach by Intel, using the TPM's software attestation model and auxiliary tamper-resistant chip, but reducing the software inside the secure container to a virtual machine hosted by the CPU's hardware virtualization features. An initialization authenticated code module (SINIT ACM) allows for performing system resets, and enables for software to have exclusive control over computational resources while it is active. However, contrary to SGX, TXT does not implement DRAM encryption, and is vulnerable to physical DRAM tampering (as is the case with similar TPM-based designs).

ARM's TrustZone [3] is a collection of hardware modules employed to partition all system resources between a secure world, hosting secure memory and containers, and a normal world, hosting the standard software stack. TrustZone's CPU core is equipped with two page table base registers, providing separate address translation units for the secure and normal worlds, and the addresses include and additional *secure bit* to establish if the contents belong to the normal or the secure world. Secure containers must also implement a monitor to perform context switches between the two worlds and to handle hardware exceptions (forcing the system to return to the normal world). Similarly to SGX, processes executing in the secure world have unrestricted access to the normal world, so some level of processing between the two is possible. TrustZone's documentation does not specify any mechanism for software attestation, however it describes how to achieve a secure boot by employing a cryptographic hash included in the on-chip polysilicon fuses.

Alternatively, the IBM 4765 secure coprocessor [79] encapsulates an entire computer system, including CPU, caches, DRAM and IO controller within a tamper-resistant environment. In particular, the secure coprocessor destroys the secret stores as soon as it detects a tampering attempt, via an array of sensors. Similarly to SGX, this system securely stores its attestation key in battery-backed memory accessible only to the secure coprocessor, responsible for measuring and loading system software, as well as provide software attestation services for applications loaded within. This hardware has shown to provide good security properties, however these tamper-resistant enclosures tend to be very expensive compared to the expected cost of computer systems [6], which deters practical deployment of solutions in these environments.

4.3 Enclave abstraction

As a first step towards formalizing the security properties enabled by technologies such as SGX, we propose an abstract description of enclaves or, more generally, isolated execution environments. At the high-level, an IEE can be seen as an idealised random access machine running some fixed program P, whose behaviour can only be influenced via a well-specified interface that permits passing inputs to the program, and receiving its outputs. The I/O behaviour of a process running in an IEE is determined by the program it is running, the semantics of the language in which the program is written, and the inputs it receives. This means, in particular, that there is strict isolation between processes running in different IEEs (and any other program running on the machine). Furthermore, the only information that is revealed about a program running within an IEE is contained in its input-output behaviour (which in most hardware systems is simply shared memory between the protected code and the untrusted software outside).

We emphasize that our notion of a machine is intended to be inclusive of any hardware platform that supports some form of isolated execution, rather than focusing on the particular SGX implementation. For this reason, the syntax of this abstraction is minimalistic, so that it can be restricted/extended to capture the specific guarantees awarded by different concrete hardware architectures, such as the ones suggested in 4.2. As an example, our "vanilla" machine supports an arbitrary number of IEEs, where programs can be loaded only once, and where multiple input/output interactions are allowed with the protected code. This is a close match to the SGX/TrustZone functionalities. However, for something like TPM, one could consider a restricted machine where a limited number of IEEs exist, with constrained input/output capabilities, and running specific code (e.g., to provide key storage). Similarly, we consider IEE environments where the underlying hardware is assumed to only keep *benevolent* state, i.e., state that cannot be used to introduce destructive correlations between multiple interactions with an IEE. Again, this closely matches what happens in SGX/Trustzone, but different types of state keeping could be allowed for scenarios where such correlations are not a problem or where they must be dealt with explicitly.

Programs. Implicit throughout the formalization will be a programming language $\mathcal{L}$ in which programs are written. We assume that this language is used by all computational platforms, but we admit IEE-specific system calls giving access to different cryptographic functionalities. These are referred as the *security module* interface. An additional system call `rand` is also assumed to be present in all platforms, giving access to fresh random coins sampled uniformly at random. Language $\mathcal{L}$ is assumed to be deterministic modulo the operation of system calls. As mentioned above, it is important for our results that system calls cannot be used by a program

to store additional implicit state that would escape our control. To this end, we impose that the results of system calls within an IEE can depend only on: i. an initially shared state that is defined when a program is loaded (e.g., the cryptographic parameters of the machine, and the code of the program); ii. the input explicitly passed on that particular call; and iii. fresh random coins. As a consequence of this, we may assume that system calls placed by different parts of a program are identically distributed, assuming that the same input is provided.

A program P must be written as a transition function, mapping bit-strings to bit-strings. Such functions take a current state st and an input i, and they will produce a new output o and an updated state. We will refer to this as an *activation* and express it as $o \leftarrow P[\text{ st }](i)$. Unless otherwise stated, st will be assumed to be initially empty. We impose that every output produced by a program includes a Boolean flag $\mathsf{finished}$ that indicates whether the transition function will accept further input. The transition function may return arbitrary output until it produces an output where $\mathsf{finished} = \mathsf{true}$, at which point it can return no further output or change its state. We extend our notation as $o \leftarrow P[\text{ st };r](i)$ to account for the randomness obtained via the rand system call as extra input r; and as $(o_1, \ldots, o_n) \leftarrow P[\text{ st };r](i_1, \ldots, i_n)$ to represent a sequence of activations. We write $\mathsf{Trace}_{P[\text{ st };r]}(i_1, \ldots, i_n)$ for the corresponding I/O trace $(i_1, o_1, \ldots, i_n, o_n)$.

Program composition. Given two programs P and Q, and a projection function between the internal states of the two programs ϕ, we will refer to the sequential composition of the two programs as $\mathsf{Compose}_\phi\langle P, Q\rangle$. This is defined as a transition function R that has two execution stages, which are signaled in its output via an additional stage bit. In the first stage, every input to R will activate program P. This will proceed until P's last output indicates it has finished (inclusively). The next activation will trigger the start of the second stage, at which point R initialises the state of Q using $\phi(\text{ st }_P)$ before activating it for the first time. Additionally we require that a constant indicating the current stage (termination being counted as a third stage) is appended to any output of a composition. When dealing with such a composed program, we will denote by $\mathsf{ATrace}_{R[\text{ st };r]}(i_1, \ldots, i_n)$ the prefix of the trace that corresponds to the execution of P. Intuitively, this denotes the *attested trace* where only the initial part of the program must be protected via attestation.

Machines. A *machine* $\mathcal{M}$ is an abstract computational device that captures the resources offered by a real world computer or group of computers, whose hardware security functionalities are initialised by a specific manufacturer before being deployed, possibly in different end-users. For example, a machine may represent a single computer produced by a manufacturer, configured with a secret signing key for a public key signature scheme, and whose public key is authenticated via some public key infrastructure, possibly managed by the manufacturer itself. Similarly, a machine may represent a group of computers, each configured with secret signing keys associated with a group signature scheme; again, the public parameters for the group would then be authenticated by some appropriate infrastructure. The provided abstraction is restricted to the simplest case, where standard public key signatures are used, however all results can be easily extended to more complex group management schemes.

We will model machines via a simple external interface, which we see as both the functionality that higher-level cryptographic schemes can rely on when using the machine, and the adversarial interface that will be the basis of our attack models. Loosely speaking, this interface can be thought of as the ideal functionality that captures what is offered by SGX. The interface is as follows:

- $\mathsf{Init}(1^\lambda)$ is the global initialisation procedure which, on input the security parameter, outputs the global parameters prms. This algorithm represents the machine's hardware initialisation procedure, which is out of the user's and the adversary's control. Intuitively, it initialises the internal security module, the internal state of the remote machine and returns any We emphasize that the global parameters of machines are the only pieces of information that are assumed to be authenticated using external mechanisms (such as a PKI) in the entire paper.

- $\mathsf{Load}(P)$ is the IEE initialisation procedure. On input a program/transition function P, the machine produces a fresh handle hdl, creates a new IEE with handle hdl, loads P into the new IEE and returns hdl. The machine interface does not provide direct access to either the internal state of an IEE nor to its randomness input. This means that the only information that is leaked about internal state and randomness input is that revealed (indirectly) via the outputs of the program.

- $\mathsf{Run}(\mathsf{hdl}, i)$ is the process activation procedure. On input a handle hdl and an input i, it will activate process running in isolated execution environment of handle hdl with i as the next input. When the program/transition function produces the next output o, this is returned to the caller.

We define the I/O trace $\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$ of a process hdl running in some machine $\mathcal{M}$ as the tuple $(i_1, o_1, \ldots, i_n, o_n)$ that includes the entire sequence of n inputs/outputs resulting from all invocations of the Run procedure on hdl; $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl})$ is the code (program) running inside the process with handle hdl; $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl})$ represents the coins given to the program by the rand system call; and $\mathsf{State}_{\mathcal{M}}(\mathsf{hdl})$ is the internal state of the program. Finally, we will denote by $\mathcal{A}^{\mathcal{M}}$ the interaction of some algorithm with a machine $\mathcal{M}$, i.e., having access to the Load and Run oracles defined above.

Observe that we use hdl as a convenient identifier for the secure environment executing process P; in some incarnation the handle could be defined as a tuple containing the identity of the machine, some identifier for the secure environment and, say, the hash of the program P. More detailed formalisms are possible. We may consider, for example, different entry/exit points related to P, which is a feature of SGX enclaves. We may also explicitly refine P as a program and some initial associated data.

4.4 Attested computation

We now formalise a cryptographic primitive that builds upon the presented abstraction, and aims to address the remote execution, i.e., outsourcing, of programs as illustrated in Figure 4.2. In this setting, a user running software in a trusted local machine wishes to use an untrusted network to access a pool of remote machines with IEE facilities. The remote machines will be running general-purpose operating systems and other untrusted software. The goal of the user is to run a specific program P within an IEE in one of the remote machines, and to obtain assurance that, not only the program is indeed executing there, but also that it is displaying a particular I/O behaviour. We call this *attested computation*, and introduce it as the cryptographic primitive that formalises the simplest cryptographic application of trusted hardware systems offering IEE functionalities.

Syntax. An *Attested Computation* (AC) scheme is defined by the following algorithms:

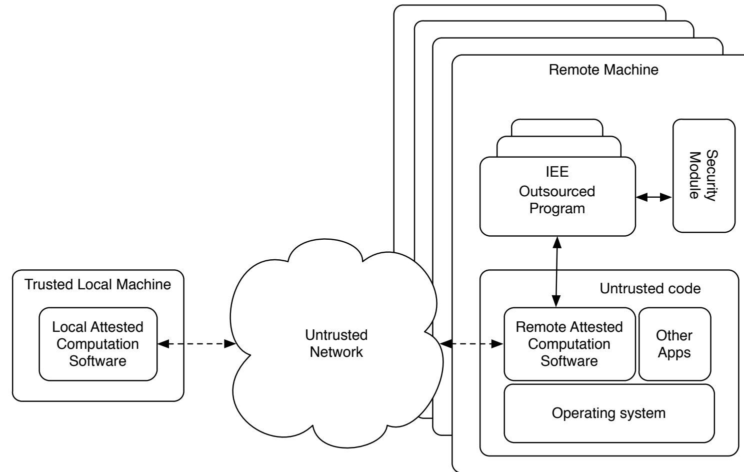


Figure 4.2: Attested Computation scenario.

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$ is the program compilation algorithm. On input global parameters for some machine $\mathcal{M}_R$, and programs P and Q, whose composition under projection function ϕ will be outsourced, it will output program R^*, together with an initial (possibly empty) state st for the verification algorithm. This algorithm is run locally. R^* is the code to be run as an isolated process in the remote machine. Intuitively, P is the initial part of the remote code that requires attestation guarantees, whereas Q is any subsequent code that may be remotely executed (generally leveraging the security guarantees that have been bootstrapped using the initial attested execution).

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ is the attestation algorithm. On input global parameters for $\mathcal{M}_R$, a process handle hdl and an input i, it will use the interface of $\mathcal{M}_R$ to obtain attested output o^*. This algorithm is run remotely, but in an unprotected environment: it is responsible for interacting with the isolated process running R^*, providing it with inputs and recovering the (possibly attested) outputs that should be returned to the local machine.

- $\mathsf{Verify}(\mathsf{prms}, i, o^*, \mathsf{st})$ is the (stateful) output verification algorithm. On input global parameters for $\mathcal{M}_R$, an input i, a (possibly attested) output o^* and some state st, it will produce an output value o and an updated state, or the failure symbol $\perp$. This failure symbol is encoded so as to be distinguishable from a valid output of a program, resulting from a successful verification. This algorithm is run locally on claimed outputs from the Attest algorithm.

In Figure 4.2, the local attested computation software block corresponds to $\mathsf{Compile}$ (one initial usage per program) and Verify (one usage per incoming attested output), whereas the remote attested computation software block corresponds to Attest (one usage per remote program activation, i.e. per I/O transition). The above syntax can be naturally extended to accommodate the simultaneous compilation of multiple input programs and/or the possibility that $\mathsf{Compile}$ may generate multiple output programs. This would allow us to capture, e.g., map/reduce applications such as those described in [73].

Correctness. Intuitively, an AC scheme is correct if, for any given programs P and Q and assuming an honest execution of all components in the scheme, both locally and remotely, the local user is able to accurately reconstruct a view of the I/O sequence that took place in the remote environment. Furthermore, this I/O sequence must be consistent with the semantics of

$\mathsf{Compose}_\phi\langle P;Q\rangle$. In other words, suppose the compiled program is run under handle hdl^* in remote machine $\mathcal{M}_R$, and the local user uses Verify to reconstruct the remote I/O behaviour $(i_1, o_1, \ldots, i_n, o_n)$. Then, if we define $R := \mathsf{Compose}_\phi\langle P;Q\rangle$, we must have

$$\mathsf{Trace}_{R[\text{ st };\mathsf{Coins}_{\mathcal{M}_R}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n) = (i_1, o_n, \ldots, i_n, o_n)$$

The following definition formalizes the notion of a local user *correctly remotely executing program* P using attested computation.

Definition 10 (Correctness)**.** *An Attested Computation scheme* AC *is correct if, for all* λ*, and all adversaries* $\mathcal{A}$*, the experiment in Figure 4.3 (left) always returns* true*.*

The adversary in this correctness experiment definition is choosing inputs, hoping to find a sequence that causes the attestation protocol to behave inconsistently with respect to the semantics of P (when these are made deterministic by hardwiring the same random coins used remotely). We use this approach to defining correctness because it makes explicit what is an honest execution of an attested computation scheme, when compared to the security experiment introduced next.

Structural preservation. Since we are dealing with composed programs, we extend the correctness requirements on attested computation schemes to preserve the structure of the input program (P, ϕ, Q), and to modify only the part of the code that will be attested. Formally, we impose that, given any program P, there exists a (unique) compiled program P^*, such that, for any mapping function ϕ and any program Q, we have that $\mathsf{Compose}_\phi\langle P^*;Q\rangle = \mathsf{Compile}(P, \phi, Q)$.

Security. Security of an attested computation scheme imposes that an adversary with absolute control of the remote machine cannot convince the local user that some arbitrary remote execution of a program P has occurred, when it has not (nothing is said about the subsequent remote execution of program Q). Essentially this considers the possibility of active adversaries managing the computation machine, which is a scenario considered by many application scenarios in PRACTICE (Aeroengine Fleet Management, Platform for Auctions, Platform for Benchmarking, Joint Statistical Analysis Between State Entities, Privacy Preserving Personal Genome Analyses and Studies, Platform for Surveys on Sensitive Data, Location Sharing with Nearby Contacts, Privacy Preserving Satellite Collision Detection, Mobile Data Sharing). Formally, we allow the adversary to freely interact with the remote machine, whilst providing a sequence of (potentially forged) attested outputs. The adversary wins if the local user reconstructs an execution trace without aborting (i.e., all attested outputs must be accepted by the verification algorithm) and one of two conditions occur: i. the execution trace that is validated by Verify is inconsistent with the semantics of P (in which case an adversary would be able to convince the local user of an I/O sequence that could not possibly have occurred!); or ii. there does not exist a remote process hdl^* exhibiting a consistent execution trace (in which case, the adversary would be able to convince the local user that a process running P was executing in the remote machine, when it was not).

Since the adversary is free to interact with the remote machine as it pleases, we can not hope to prevent it from appending arbitrary inputs to the trace of any remote process, while refusing to deliver all of the resulting attested outputs to the local user. This justifies the winning condition in our security game referring to a prefix of the trace in the remote machine, rather than imposing trace equality. Indeed, the definition's essence is to impose that the locally recovered trace and the remote trace share a common prefix ($\sqsubseteq$), which exactly corresponds to the part of the source program's behaviour that should be protected by attestation.

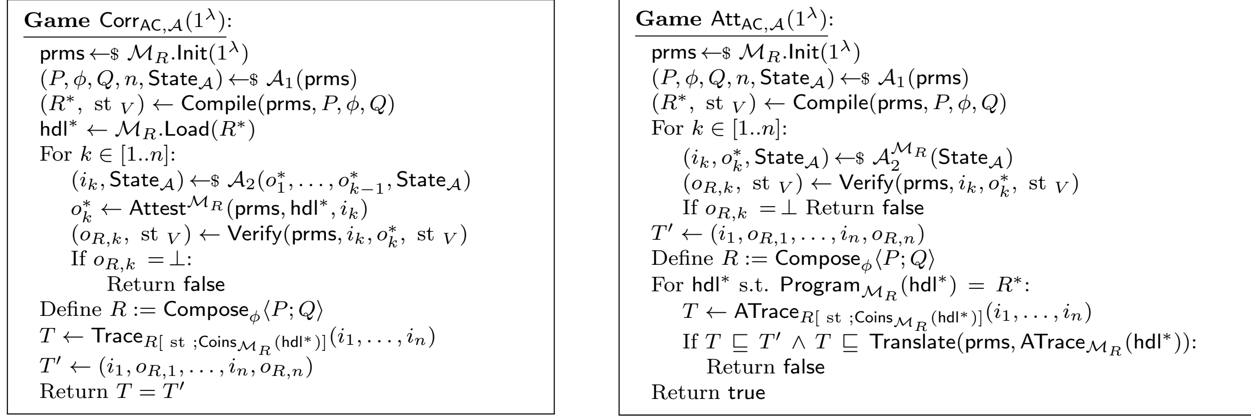


Figure 4.3: Games defining the correctness (left) and security (right) of an AC scheme.

Formally, we need to account for the fact that the actual I/O sequence of the remote program includes more information than that of R, e.g., to allow for the cryptographic enforcement of security guarantees. Our definition is parametrised by a $\mathsf{Translate}$ algorithm that permits formalising this notion of *semantic consistency*. Another way to see $\mathsf{Translate}(\mathsf{prms}, \mathsf{ATrace}_{\mathcal{M}_R}(\mathsf{hdl}^*))$ is as a trace translation procedure associated with a given AC scheme, which maps remote traces into traces at the source level.

Definition 11 (Security). *An attested computation scheme is secure if there exists an efficient deterministic algorithm* $\mathsf{Translate}$ *s.t., for all ppt adversaries* $\mathcal{A}$*, the probability that experiment in Figure 4.3 (right) returns* true *is negligible.*

We note that the adversary loses the game as long as there exists at least one remote process that matches the locally reconstructed trace. This should be interpreted as the guarantee that IEE resources are indeed being allocated in a specific remote machine to run at least one instance of the remote program (note that if the program is deterministic, many instances could exist with exactly the same I/O behaviour, which is *not* seen as a legitimate attack). Furthermore, our definition essentially imposes that the compiled program uses essentially the same randomness as the source program (except of course for randomness that the security module internally uses to provide its cryptographic functionality), as otherwise it will may easy for the adversary to make the (idealized) local trace diverge from the remote. This is a consequence of our modeling approach, but in no way does it limit the applicability of the primitive we are proposing: it just makes it explicit that the transformation that is performed on the code for attestation will typically consist of an instrumentation of the code by applying cryptographic processing to the inputs and outputs it receives.

Minimum leakage. From the discussion above, we gather that an AC scheme should guarantee that the I/O behaviour of the program in the remote machine includes at least the information required to reconstruct an hypothetical local execution of the source program. However, it is important to establish an additional restriction on what AC compilation actually does to a source program, to ensure that we are able to take advantage of this primitive to achieve more ambitious goals, namely to perform attestation of the remote execution of cryptographic code.

The following definition imposes that nothing from the internal state of the source programs (in addition to what is public, i.e. the code and I/O sequence) is leaked in the trace of the compiled program when it is remotely executed.

Game $\mathsf{Leak\text{-}Real}_{\mathsf{AC},\mathcal{A}}(1^\lambda)$:

$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$b \leftarrow\$ \mathcal{A}^{\textsc{Oracles}}(\mathsf{prms})$
Return b

Oracle $\mathsf{Compile}(P, \phi, Q)$:

$(R, \mathrm{st}_V) \leftarrow \mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$
$\mathsf{PrgList} \leftarrow R : \mathsf{PrgList}$
Return R

Oracle $\mathsf{Load}(R)$:

Return $\mathcal{M}_R.\mathsf{Load}(R)$

Oracle $\mathsf{Run}(\mathsf{hdl}, i)$:

Return $\mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i)$

Game $\mathsf{Leak\text{-}Ideal}_{\mathsf{AC},\mathcal{A},\mathsf{S}}(1^\lambda)$:

$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{List} \leftarrow [\,]$
$\mathsf{hdl} \leftarrow 0$
$(\mathsf{prms}, \mathrm{st}_{\mathsf{S}}) \leftarrow\$ \mathsf{S}_1(1^\lambda)$
$b \leftarrow\$ \mathcal{A}^{\textsc{Oracles}}(\mathsf{prms})$
Return b

Oracle $\mathsf{Compile}(P, \phi, Q)$:

$(R, \mathrm{st}_V) \leftarrow \mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$
$\mathsf{PrgList} \leftarrow (P, \phi, Q, R) : \mathsf{PrgList}$
Return R

Oracle $\mathsf{Load}(R)$:

$\mathsf{hdl} \leftarrow \mathsf{hdl} + 1$
$\mathsf{List}[\mathsf{hdl}] \leftarrow (R, \epsilon)$
Return hdl

Oracle $\mathsf{Run}(\mathsf{hdl}, i)$:

$(R, \mathrm{st}\ ) \leftarrow \mathsf{List}[\mathsf{hdl}]$
If $(P, \phi, Q, R) \in \mathsf{PrgList}$:
  $R^* \leftarrow \mathsf{Compose}_\phi\langle P, Q\rangle$
  $o^* \leftarrow\$ R^*[\ \mathrm{st}\ ](i)$
  $(o, \mathrm{st}_{\mathsf{S}}) \leftarrow\$ \mathsf{S}_2(\mathsf{hdl}, P, \phi, Q, R, i, o^*, \mathrm{st}_{\mathsf{S}})$
Else:
  $(o, \mathrm{st}\ , \mathrm{st}_{\mathsf{S}}) \leftarrow\$ \mathsf{S}_3(\mathsf{hdl}, R, i, \mathrm{st}\ , \mathrm{st}_{\mathsf{S}})$
$\mathsf{List}[\mathsf{hdl}] \leftarrow (R, \mathrm{st}\ )$
Return o

Figure 4.4: Games defining minimum leakage of an AC scheme.

Definition 12 (Minimal leakage)**.** *Attested Computation scheme* AC *ensures security with minimal leakage if it is secure according to Definition 11 and there exists a ppt simulator* S *that, for every adversary* $\mathcal{A}$*, the following distributions are identical:*

$$\{\,\mathsf{Leak\text{-}Real}_{\mathsf{AC},\mathcal{A}}(1^\lambda)\,\} \approx \{\,\mathsf{Leak\text{-}Ideal}_{\mathsf{AC},\mathcal{A},\mathsf{S}}(1^\lambda)\,\}$$

where games $\mathsf{Leak\text{-}Real}_{\mathsf{AC},\mathcal{A}}$ *and* $\mathsf{Leak\text{-}Ideal}_{\mathsf{AC},\mathcal{A},\mathsf{S}}$ *are shown in Figure 4.4.*

Notice that we allow the simulator to replace the global parameters of the machine with some value prms for which it can keep some trapdoor information. Intuitively this means that one can construct a perfect simulation of the remote trace by simply appending cryptographic material to the local trace. This property is important when claiming that the security of a cryptographic primitive is preserved when it is run within an attested computation scheme (one can simply reduce the advantage of an adversary attacking the attested trace, to the security of the original scheme using the minimum leakage simulator).

Attested computation using SGX

The remote attestation protocol we will consider is inspired in the SGX architecture described in Section 4.2. The main feature of this system is that the remote machine is equipped with a security module that manages both short-term and long-term cryptographic keys, with which it is capable of producing MACs that enable authenticated communication between various IEEs and digital signatures that can be publicly verified by anyone holding the (long-term) public key for that machine (or group of machines). We first formalise the operation of (a simplified version of) this security module.

Security module. The security module relies on a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ and a MAC scheme $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Ver})$, and it operates as follows:

- When the host machine is initialised, the security module generates a key pair $(\mathsf{pk}, \mathsf{sk})$ using $\Sigma.\mathsf{Gen}$ and a symmetric key key using $\Pi.\mathsf{Gen}$. It also creates a special process running code S^* (see below for a description of S^*) in an IEE with handle 0. The security module then securely stores the key material for future use, and outputs the public key. In this case we will have that the output of $\mathcal{M}.\mathsf{Init}$ will be $\mathsf{prms} = \mathsf{pk}$.

- The operation of IEE with handle 0 will be different from all other IEEs in the machine. Program S^* will permanently reside in this IEE, and it will be the only one with direct access to both sk and key.
- The code of S^* is dedicated to transforming messages authenticated with key into messages signed with sk. On each activation, it expects an input (m, t). It obtains key from the security module and verifies the tag using $\Pi.\mathsf{Ver}(\mathsf{key}, \mathsf{t}, \mathsf{m})$. If the previous operation was successful, it obtains sk from the security module, signs the message using $\sigma \leftarrow_\$ \Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{m})$ and writes σ to the output. Otherwise, it writes $\perp$ in the output.
- The security module exposes a single system call $\mathsf{mac}(\mathsf{m})$ to code running in all other IEEs. On such a request from a process running program P, the security module returns a MAC tag t computed using key over both the code of P and the input message (m).

We note that the operation of the security module allows any process to produce an authenticated message that can be validated by the special process running S^* as coming from within another IEE in the same machine.
We will assume that the message authentication code scheme Π and the signature scheme Σ satisfy the standard notions of correctness and existential unforgeability, and that the machine's public key is authenticated by some external PKI.

Attested Computation scheme. We now define an AC scheme that relies on a remote machine supporting a security module with the above functionality. The operation of the various algorithms is intuitive, except for the fact that basic replay protection using a sequence number does not suffice to bind a remote process to a full trace, since the adversary could then run multiple copies of the same process and *mix and match* outputs from various traces. Instead, the remote process must commit to its entire trace whenever an attested output is produced. Details follow:

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$ will generate a new program $R^* = \mathsf{Compose}_\phi\langle P^*, Q\rangle$ and output it along with the initial state of the verification algorithm $(R^*, [\,], 1)$, where 1 is an indicator of the stage in which remote program R^* is supposed to be executing. Program P^* is instrumented as follows: it keeps a list ios of all the I/O pairs it has previously received and computed, i.e, its own trace; on each activation with input i, P^* first computes $o \leftarrow_\$ P[\text{ st }_P](i)$ and updates the list by adding a new (i, o) pair; it then requests from the security module a MAC of the updated ios. Due to the operation of the security module, this will correspond to a tag t on the tuple (R^*, ios); it finally outputs $(o, \mathsf{t}, R^*, \mathsf{ios})$. We note that we include (R^*, ios) explicitly in the outputs of R^* for clarity of presentation only. This value would be kept in an insecure environment by a stateful Attest program.
- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ invokes $\mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i)$ using the handle and input value it has received. When the process produces an output o, Attest parses it into $(o', \mathsf{t}, R^*, \mathsf{ios})$. It may happen that parsing fails, e.g., if Q is already executing, in which case Attest simply produces o as its own output. Otherwise, it uses $\mathcal{M}_R.\mathsf{Run}(0, (R^*, \mathsf{ios}, \mathsf{t}))$ to convert the tag into a signature σ on the same message. If this conversion fails, then Attest produces the original output o as its own output. Otherwise, it outputs (o', σ).
- $\mathsf{Verify}(\mathsf{prms}, i, o^*, (R^*, \mathsf{ios}, \mathsf{stage}))$ returns o^* if $\mathsf{stage} = 2$. Otherwise, it first parses o^* into (o, σ), appends (i, o) to ios, and verifies the digital signature σ using prms and (R^*, ios)). If parsing or verification fails, Verify outputs $\perp$. If not, then Verify will check if output o indicates that program P^* has finished. If so, it will update stage to value 2. In any case, it terminates outputting o.

This scheme captures the general behaviour of SGX enclaves, and the inter-platform attestation mechanism in particular. The special process of code S^* of handle 0 can be seen as the *quoting enclave*. Computations are attested by having IEE producing MAC tags with code and I/O trace (reports constructed by SGX enclaves, including the challenge input provided), which are directed towards the special process responsible for validating and signing with the key to which only this particular IEE has access. In particular, this can also be an EPID key, such as the one employed in the mechanism presented in 4.2. The local party can now run Verify to check if the produced output matches the expected execution. A more in-depth discussion regarding the correctness and security proofs for this AC scheme can be found in [7].

Chapter 5

Secure Outsourced Computation From Attested Computation

5.1 Overview

We define here a way to securely execute code on a remote platform using IEE backed attested computation as defined in Section 4.4. We aim at providing the same guarantees in terms of integrity and confidentiality as the guarantees that a user would have by running the code on a local trusted machine. In particular, we propose a compilation mechanism that ensures that only the legitimate user of a program executing remotely can submit inputs to it. Additionally we ensure that the I/O trace of the underlying program stays private.

Purely cryptographic solutions to this problem have been proposed using fully homomorphic encryption [41], but they are far from practical. The solution we present here has a very low overhead with respect to the delegated computation and can therefore potentially be used to delegate computationally heavy tasks.

Such primitives for delegating code execution are particularly important in the context of cloud computing. Indeed, local users delegating computationally heavy and potentially critical task to a remote machine have a strong need for both integrity and privacy. Typically in a scenario where an aircraft manufacturer for example wants to run heavy simulations while taking advantage of the flexible could infrastructure, strong guarantees are a prerequisite. The input and output data are immensely valuable trade secrets. Integrity is equally important as a false result of the simulation could cost human lives in the test flights.

In this Chapter, we first propose a bootstrapping procedure that combines a passively secure key exchange with an attested computation scheme, towards enabling the deployment of implementations for secure outsourced computation. The models and foundations detailed in this chapter are based on those presented in [7], which is a direct result of work developed in the context of Task 13.2 of PRACTICE project.

5.2 Key exchange for attested computation

An intermediate step for constructing high-level applications that rely on attested computation is the establishment of a secure communications channel with a process running a particular program inside an IEE in the remote machine. After such a channel has been established, standard cryptographic techniques can be used to ensure (in combination with the isolation provided by IEEs) the integrity and confidentiality of subsequent computations. We finalize this chapter by presenting how attested computation, in combination with a specific flavour of a key exchange protocol, can be seen as a bootstrapping procedure for deploying secure outsourced computation solutions.

We first formalize the precise requirements for a key exchange protocol that can be used in this setting (we call this *authenticated key exchange for attested computation*) and show how a simple transformation can be used to construct such protocols from any passively secure key exchange protocol. Later on we present a utility theorem that precisely describes what it means to use attested computation and a suitable key exchange protocol to establish a secure channel with an arbitrary remote program.

SYNTAX. A *Key Exchange for Attested Computation* (AttKE) protocol is defined by the following pair of algorithms.

- $\mathsf{Setup}(1^\lambda, \mathsf{id})$ is the remote program generation algorithm, which is run on the local machine to initialise a fresh instance of the AttKE protocol under party identifier id. On input the security parameter and id, it will output the code for a program $\mathsf{Rem}_{\mathsf{KE}}$ and the initial state st_L of the $\mathsf{Loc}_{\mathsf{KE}}$ algorithm. This algorithm is run locally.
- $\mathsf{Rem}_{\mathsf{KE}}$ (which is generated dynamically by Setup) is a program that will be run as a part of an IEE process in the remote machine, and it will keep the entire remote state of the key exchange protocol in that protected environment.
- $\mathsf{Loc}_{\mathsf{KE}}(\mathrm{st}_L, \mathsf{m})$ is the algorithm that runs the local end of the AttKE protocol, interacting with $\mathsf{Rem}_{\mathsf{KE}}$. On input its current state and an incoming message m, it will output an updated state and an outgoing message.

When analysing the security of such a protocol we will impose that the $\mathsf{Loc}_{\mathsf{KE}}$ algorithm and all $\mathsf{Rem}_{\mathsf{KE}}$ programs that may be produced by Setup keep in their state the same information that was imposed on general key exchange algorithms. We will refer to the instances of local key exchange executions as $\mathsf{Loc}^s_{\mathsf{KE}}$, for $s \in \mathbb{N}$. The local identity will be implicit in our notation since, in the following discussion we will concentrate our attention on the case where a single local identity id is considered. We do this for the sake of rigour and clarity of presentation: by looking at this simplified case we can present our security models in game-based form, whilst taming the complexity of the resulting games. The extension of these results to the more general case where several local identities are considered is straightforward. On the remote side, the identity of the remote process will actually be generated on the fly by the combined actions of the Setup algorithm and possibly the protocol execution itself, as it may depend for example on the code of the remote program. For this reason we will enumerate over remote instances as $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$ for $i, j \in \mathbb{N}$, and observe that the value of variable oid in this case will be set during the execution of the program itself, rather than passed explicit as an input to one of the algorithms. CORRECTNESS. An AttKE is correct if, after a complete (honest) run between two participants, one local and one remote, and where the remote program is always the one to

```
Game Corr_AttKE(1^λ):
  (st_L, Rem_KE) ←$ Setup(1^λ, id)
  st_R ← ε
  m ←$ Rem_KE[st_R](ε)
  t ← true
  While m ≠ ε:
    If t: (st_L, m) ←$ Loc_KE(st_L, m)
    Else: m ←$ Rem_KE[st_R](m)
    t ← ¬t
  Return st_L.δ = st_R.δ = accept ∧ st_L.key = st_R.key ∧ st_L.sid = st_R.sid ∧
         st_L.pid = st_R.oid ∧ st_L.oid = st_R.pid ∧ st_L.oid = id
```

Figure 5.1: Game defining the correctness of an AttKE scheme.

initiate the communication, both reach the accept state, both derive the same key and session identifier and have matching partner identities. More formally, a protocol $P = \{\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}}\}$ is correct if, for any arbitrary identity id, the experiment in Figure 5.1 always returns true. We note that our definition of correctness imposes that remote programs always operate as initiators and local machines as the responders in the key exchange.

Execution Environment. The specific flavour of key exchange that we will be considering is clarified by the execution environment in Figure 5.2. This follows the standard modelling of active attackers, e.g. [54], when one excludes the possibility of corruption (which we do only for the sake of simplicity). There are, however, two modifications that attend to the fact that AttKE remote programs are designed to be executed under attested computation guarantees. On one hand, the adversary is given the power to create as many remote AttKE programs as it may need, by using the NewLocal oracle, revealing the entire code of the remote AttKE program (and implicitly all of its initial internal state, which is assumed to be empty) to the adversary. This captures the fact that remote AttKE programs will be loaded into IEE execution environments in an otherwise untrusted remote machine, and it implies that remote AttKE programs cannot keep *any* long term secret information. Intuitively, this limitation will be compensated by the attested computation protocol. On the other hand, the adversary is able to freely interact with remote processes, but it is constrained in its interaction with the local machine. Indeed, the SendLocal oracle filters which messages the adversary can deliver to the local machine by checking that these are consistent with at least one remote process that the adversary is interacting with. This captures the fact that AttKE is designed to interact over a partially authenticated channel from the remote machine to the local machine, which will be provided by an attested computation protocol.

Partnering. We will consider the natural extension of the partnering properties introduced for passive key exchange to the AttKE setting. In addition to the syntactic modifications that result from referring to $\mathsf{Loc}^s_{\mathsf{KE}}$ and $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$, we further restrict validity so that partnering is only valid when it occurs between local and remote instances, in which the latter is the initiator. To this end, we will use the following predicate on two instances $\mathsf{Loc}^s_{\mathsf{KE}}$ and $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$ holding $\mathrm{st}^s_L = (\mathrm{st}^s, \delta^s, \rho^s, \mathsf{sid}^s, \mathsf{pid}^s, \mathsf{oid}^s, \mathsf{key}^s)$ and $\mathrm{st}^{i,j}_R = (\mathrm{st}^{i,j}, \delta^{i,j}, \rho^{i,j}, \mathsf{sid}^{i,j}, \mathsf{pid}^{i,j}, \mathsf{oid}^{i,j}, \mathsf{key}^{i,j})$, respectively:

$$\mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \begin{cases} \mathsf{true} \text{ if } \mathsf{sid}^s = \mathsf{sid}^{i,j} \ \wedge \ \delta^s, \delta^{i,j} \in \{\mathsf{derived}, \mathsf{accept}\} \\ \mathsf{false} \text{ otherwise.} \end{cases}$$

The definition of partner is the obvious one, whereas invalid partners now includes an extra possibility.

Game $\mathsf{Att}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda)$:
$\mathsf{InsList} \leftarrow [\,];\ \mathsf{fake} \leftarrow [\,]$
$i \leftarrow 0$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(1^\lambda, \mathsf{id})$
Return $b = b'$

Oracle $\mathsf{NewLoc}()$:
$i \leftarrow i+1;\ T_L^i \leftarrow [\,]$
$(\mathsf{Rem}_{\mathsf{KE}}^i,\ \mathrm{st}_L^i) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$\mathsf{InsList}[i] \leftarrow 0$
Return $\mathsf{Rem}_{\mathsf{KE}}^i$

Oracle $\mathsf{TestLoc}(i)$:
If $\mathrm{st}_L^i.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ return $\mathrm{st}_L^i.\mathsf{key}$
Return $\mathsf{fake}(\mathrm{st}_L^i.\mathsf{key})$

Oracle $\mathsf{SendLoc}(\mathsf{m}, i)$:
If $\nexists j,\ (\mathsf{m} : T_L^i) \sqsubseteq T_R^{i,j}$ return $\perp$
$(\mathsf{m}',\ \mathrm{st}_L^i) \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}^i(\mathrm{st}_L^i, \mathsf{m})$
$T_L^i \leftarrow \mathsf{m}' : \mathsf{m} : T_L^i$
If $\mathrm{st}_L^i.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  If $(\mathrm{st}_L^i.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathrm{st}_L^i.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $(\mathsf{m}',\ \mathrm{st}_L^i.\mathsf{sid},\ \mathrm{st}_L^i.\delta,\ \mathrm{st}_L^i.\mathsf{pid})$

Oracle $\mathsf{RevealLoc}(i)$:
Return $\mathrm{st}_L^i.\mathsf{key}$

Oracle $\mathsf{RevealRem}(i, j)$:
Return $\mathrm{st}_R^{i,j}.\mathsf{key}$

Oracle $\mathsf{NewRem}(i)$:
$\mathsf{InsList}[i] \leftarrow \mathsf{InsList}[i] + 1$
$j \leftarrow \mathsf{InsList}[i]$
$T_R^{i,j} \leftarrow [\,];\ \mathrm{st}_R^{i,j} \leftarrow \epsilon$
Return ϵ

Oracle $\mathsf{TestRem}(i, j)$:
If $\mathrm{st}_R^{i,j}.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ return $\mathrm{st}_R^{i,j}.\mathsf{key}$
Return $\mathsf{fake}(\mathrm{st}_R^{i,j}.\mathsf{key})$

Oracle $\mathsf{SendRem}(\mathsf{m}, i, j)$:
// No restriction
$\mathsf{m}' \leftarrow\$ \mathsf{Rem}_{\mathsf{KE}}[\mathrm{st}_R^{i,j}](\mathsf{m})$
$T_R^{i,j} \leftarrow \mathsf{m}' : \mathsf{m} : T_R^{i,j}$
If $\mathrm{st}_R^{i,j}.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  If $(\mathrm{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathrm{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $(\mathsf{m}',\ \mathrm{st}_R^{i,j}.\mathsf{sid},\ \mathrm{st}_R^{i,j}.\delta,\ \mathrm{st}_R^{i,j}.\mathsf{pid})$

Figure 5.2: Execution environment for AttKEs.

Definition 13 (Partner). *Two instances* $\mathsf{Loc}_{\mathsf{KE}}^s$ *and* $\mathsf{Rem}_{\mathsf{KE}}^{i,j}$ *are partnered if*

$$\mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{true}\,.$$

Definition 14 (Valid Partners). *A protocol* AttKE *ensures valid partners if the bad event* notval *does not occur, where* notval *is defined as one of the following events occurring:*

$$\exists \mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}\ s.t.\ \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{true} \wedge (\,\mathsf{pid}^s \neq \mathsf{oid}^{i,j} \vee \mathsf{oid}^s \neq \mathsf{pid}^{i,j} \vee \mathsf{key}^s \neq \mathsf{key}^{i,j} \vee \rho^s \neq \mathsf{responder} \vee \rho^{i,j} \neq \mathsf{initiator}\,)$$

$$\exists \mathsf{Loc}_{\mathsf{KE}}^r, \mathsf{Loc}_{\mathsf{KE}}^s\ s.t.\ r \neq s \wedge \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^r, \mathsf{Loc}_{\mathsf{KE}}^s) = \mathsf{true}$$

$$\exists \mathsf{Rem}_{\mathsf{KE}}^{i,j}, \mathsf{Rem}_{\mathsf{KE}}^{k,l}\ s.t.\ (i,j) \neq (k,l) \wedge \mathrm{P}(\mathsf{Rem}_{\mathsf{KE}}^{i,j}, \mathsf{Rem}_{\mathsf{KE}}^{k,l})$$

For completeness, we present also the adapted definitions of confirmed and unique partners.

Definition 15 (Confirmed Partners). *A protocol* AttKE *ensures confirmed partners if the bad event* notconf *does not occur, where* notconf *is defined as at least one of the following two events occurring:*

$$\exists \mathsf{Loc}_{\mathsf{KE}}^s\ s.t.\ \delta^s = \mathsf{accept} \wedge \forall \mathsf{Rem}_{\mathsf{KE}}^{i,j}, \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{false}$$

$$\exists \mathsf{Rem}_{\mathsf{KE}}^{i,j}\ s.t.\ \delta^{i,j} = \mathsf{accept} \wedge \forall \mathsf{Loc}_{\mathsf{KE}}^s, \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{false}.$$

Definition 16 (Unique Partners). *A protocol* AttKE *ensures unique partners if the bad event* notuni *does not occur, where* notuni *is defined as at least one of the following two events occurring:*

$$\exists \mathsf{Loc}_{\mathsf{KE}}^s,\ \mathsf{Rem}_{\mathsf{KE}}^{i,j}, \mathsf{Rem}_{\mathsf{KE}}^{i',j'}\ s.t.\ (i,j) \neq (i',j') \wedge \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{true} \wedge \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i',j'}) = \mathsf{true}$$

$$\exists \mathsf{Rem}_{\mathsf{KE}}^{i,j},\ \mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Loc}_{\mathsf{KE}}^{s'}\ s.t.\ s \neq s' \wedge \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{true} \wedge \mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^{s'}, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{true}\,.$$

We will consider that an adversary violates entity authentication if he can get a session to accept, but there is no unique and confirmed valid session in its intended partner.More formally, we wish to verify that none of the bad events notval, $\mathsf{notconf}$, notuni occurs. In the attested computation scenario, it is common to use one-sided authentication where only the local party receives authentication guarantee.

Security. Again, the set of $\mathsf{TestLoc}$ and $\mathsf{TestRem}$ queries must be restricted in order to exclude trivial attacks. An adversary is legitimate if it respects the following freshness criteria:

- For all $\mathsf{TestLoc}(i)$ queries, the following holds:
  1. $\mathsf{RevealLoc}(i)$ was not queried; and
  2. for all $\mathsf{Rem}_{\mathsf{KE}}^{j,k}$ s.t. $\mathrm{P}(\mathsf{Rem}_{\mathsf{KE}}^{j,k}, \mathsf{Loc}_{\mathsf{KE}}^{s}) = \mathsf{true}$, $\mathsf{RevealRem}(j,k)$ was not queried.
- For all $\mathsf{TestRem}(i,j)$ queries, the following holds:
  1. $\mathsf{RevealRem}(i,j)$ was not queried; and
  2. for all $\mathsf{Loc}_{\mathsf{KE}}^{k}$ s.t. $\mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^{k}, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{true}$, $\mathsf{RevealLoc}(i)$ was not queried.

We only consider legitimate adversaries, and say that the winning event guess occurs if $b = b'$ at the end of the experiment. We define AttKE security by requiring both mutual authentication of parties and and key secrecy.

Definition 17 (AttKE security)**.** *An AttKE protocol is secure if, for any ppt adversary in Figure 5.2, and for any local party identifier string* id*:*

1. the adversary violates entity authentication with negligible probability $\Pr[\,\mathsf{notval} \vee \mathsf{notconf} \vee \mathsf{notuni}\,]$*; and*

2. its key secrecy advantage $2 \cdot \Pr[\,\mathsf{guess}\,] - 1$ *is negligible.*

General construction

We now present a construction of an AttKE scheme from any passively secure key exchange protocol, relying additionally on a existentially unforgeable signature scheme. The intuition here is that the attested computation protocol guarantees correct remote execution of a program, but does not ensure uniqueness, i.e., it does not exclude that potentially many replicas of the same key exchange protocol instance could be running in the remote machine. By binding a fresh signature verification key with the identifier for the remote party associated with the key exchange protocol and generating a fresh nonce at the start of every execution, we can remotely execute the key exchange code whilst ensuring one-to-one authentication at the process level. This transformation can be seen as a weaker version of the well-known passive-to-active compilation process by Katz et al. [54], since our target security model is not fully active. We now present the details.

Consider a passively-secure authenticated key exchange protocol Π and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. Our construction splits the execution of Π between the local machine and a remote isolated execution environment: the responder will run locally and the initiator will run remotely within a program $\mathsf{Rem}_{\mathsf{KE}}$.[1] The code of the remote program will have hardwired

[1] Setting the remote machine as the initiator of the protocol is the most common scenario. We considered it for simplicity; the converse can be treated analogously.

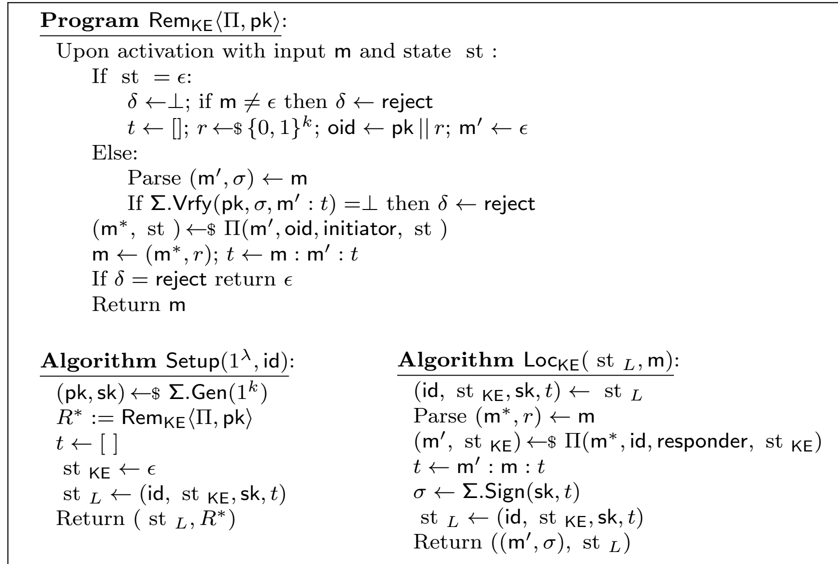


Figure 5.3: Details of the AttKE construction.

into it a unique verification key for the signature scheme. The first activation of $\mathsf{Rem}_{\mathsf{KE}}$ initialises an internal state and computes a nonce, together with the first message in the key exchange protocol. The party identifier string of the remote process will then be defined to comprise the verification key and the nonce. The local part of the protocol signs the full communication trace so far. Subsequent activations of remote program $\mathsf{Rem}_{\mathsf{KE}}$ will simply respond according to the key exchange protocol description, rejecting all inputs that fail signature verification. The details of our construction are shown in Figure 5.3.

- Setup first generates a fresh key pair for the signature scheme and constructs program $\mathsf{Rem}_{\mathsf{KE}}$, parametrised by algorithm Π and verification key pk, as described in Figure 5.3 (top). In this program state variables δ, ρ, key, sid and pid are all shared with Π (this is implicit in the figure). The initial value of st_L will store id, along with the initially empty state for the key exchange $\mathrm{st}_K E$, the signing key for the signature scheme and an initially empty trace t log.

- $\mathsf{Loc}_{\mathsf{KE}}$ takes $(\mathrm{st}_L, \mathsf{m})$ runs $\Pi(\mathsf{m}, \mathsf{id}, \mathsf{responder}, \mathrm{st}_{\mathsf{KE}})$ to compute the next message o, produces signature σ of the entire updated protocol trace, and returns the updated state st_L and message (o, σ).

The following theorem establishes the correctness and security of the generic construction, and the associated full proof can be found in [7].

Theorem 6. *Given a correct passively secure key exchange protocol Π and an existentially unforgeable signature scheme Σ, the generic construction above yields a correct and secure AttKE protocol.*

Key exchange utility

As a final result towards the construction of a bootstrapping procedure for a full-fledged authenticated and private remote attested computation scheme, we will now present a utility theorem that describes precisely the guarantees one obtains when combining an attested computation protocol with an AttKE. Intuitively, this theorem states that attested computation guarantees

that the authentication and secrecy assurance offered by AttKE are retained when we use it to establish session keys with remote IEEs, in the presence of fully active adversaries that control the remote machine, and when the key exchange is composed with arbitrary programs.

Figure 5.4 shows an idealised game where an adversary must distinguish between two remote machines where an AttKE scheme is executed in combination with an AC scheme. Machine $\mathcal{M}_R$ is any standard remote machine that is supported by the attested computation protocol, whereas $\mathcal{M}'_R$ represents a modification of $\mathcal{M}_R$ where one can tweak the operation of $\mathsf{Rem}_{\mathsf{KE}}$ programs. The differences of $\mathcal{M}'_R$ with respect to $\mathcal{M}_R$ are concentrated on the Run interface, which now operates as follows:

- It takes as additional parameters a list fake of pairs of keys and Boolean flag tweak that, when activated, identifies a process that is running an instance of $\mathsf{Rem}_{\mathsf{KE}}$ composed with some program Q. This flag triggers the following modifications with respect to the operations of $\mathcal{M}_R$.
- When it detects that $\mathsf{Rem}_{\mathsf{KE}}$ has transitioned into $\mathsf{derived}$ or accept state, it will check if the derived key exists in list fake. If not, it generates a new random key^*, and $(\mathsf{key}, \mathsf{key}^*)$ is added to the list.
- When it detects that program Q is set to start executing, rather than using the key as an input to ϕ, it uses $\mathsf{fake}(\mathsf{key})$ instead.

The environment presented to the adversary models a standard attested computation interaction, where it is given total control over the remote machine using oracles Load and Run (these oracles will either give access to $\mathcal{M}_R$ or to $\mathcal{M}'_R$, depending on a secret bit b generated in the beginning of the game). The adversary is also able to obtain challenge remote programs using a $\mathsf{NewSession}(Q)$ oracle that uses the attested computation scheme to compile $\mathsf{Rem}_{\mathsf{KE}}$ composed with arbitrary program Q of its choice under a mapping function ϕ_{key} that reveals the relevant parts of the key exchange state (namely the secret key key, the party identifiers oid and pid, the state δ and the session identifier sid). We observe that such arbitrary programs can leak all of the information revealed by ϕ_{key} to the attacker. If the adversary chooses to Load a challenge program, and if $\mathcal{M}'_R$ is being used in the game, then it will be tweaked as described above. Whenever $\mathsf{NewSession}(Q)$ is called, the environment creates a new local session i that the adversary can interact with using a $\mathsf{Send}(i, \mathsf{m})$ oracle. The Send oracle uses the Verify algorithm of the attested computation scheme to validate attested outputs and, if they are accepted, feeds them to the $\mathsf{Loc}_{\mathsf{KE}}$ instance (and also ensures that list fake is updated). Finally, the adversary can explicitly choose to be tested (as opposed to the implicit testing it may trigger using arbitrary programs Q) by calling Test on a local instance. This oracle will either return the true key, if $b = 0$, or the associated random key that is kept in the fake list. As before, we define the winning event guess to occur when $b = b'$ in the end of the game.

The proof of the following theorem can also be found in [7].

Theorem 7 (AttKE utility). *If* AttKE *is correct and secure and the AC protocol is correct, secure and ensures minimum leakage, then for all ppt adversaries in the utility experiment:*

1. the probability that the adversary violates AttKE two-sided entity authentication is negligible; and

2. the key secrecy advantage $2 \cdot \Pr[\,\mathsf{guess}\,] - 1$ *is negligible.*

Game $\mathsf{Att}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda)$:
$\mathsf{prms}_0 \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$ \mathcal{M}'_R.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow []$
$\mathsf{fake} \leftarrow []$
$i \leftarrow 0$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b, \mathsf{id})$
Return $b = b'$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl}_0 \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
$\mathsf{hdl}_1 \leftarrow \mathcal{M}'_R.\mathsf{Load}(R^*)$
Return hdl_b

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$:
$o_0 \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$
$\mathsf{tweak} \leftarrow \mathsf{false}$
If $\mathsf{Program}_{\mathcal{M}'_R}(\mathsf{hdl}) \in \mathsf{PrgList}$ then $\mathsf{flag} \leftarrow \mathsf{true}$
$(o_1, \mathsf{fake}) \leftarrow\$ \mathcal{M}'_R.\mathsf{Run}(\mathsf{hdl}, \mathsf{in}, \mathsf{tweak}, \mathsf{fake})$
Return o_b

Oracle $\mathsf{NewSession}(Q)$:
$i \leftarrow i + 1$
$(\mathsf{Rem}^i_{\mathsf{KE}}, \mathrm{st}^i_{\mathsf{KE}}) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$(R^*_i, \mathrm{st}^i_L) \leftarrow\$ \mathsf{AC.Compile}(\mathsf{prms}_b, \mathsf{Rem}^i_{\mathsf{KE}}, \phi_{\mathsf{key}}, Q)$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{PrgList} \leftarrow R^*_i : \mathsf{PrgList}$
Return R^*_i

Oracle $\mathsf{Send}(\mathsf{m}', i)$:
$(\mathsf{m}, \mathrm{st}^i_L) \leftarrow\$ \mathsf{AC.Verify}(\mathsf{prms}_b, \mathsf{in}^i_{\mathsf{last}}, \mathsf{m}', \mathrm{st}^i_L)$
If $\mathsf{m} = \perp$ then return $\perp$
$(\mathsf{m}^*, \mathrm{st}^i_{\mathsf{KE}}) \leftarrow\$ \mathsf{Loc}^i_{\mathsf{KE}}(\mathrm{st}^i_{\mathsf{KE}}, \mathsf{m})$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{m}^*$
If $\mathrm{st}^i_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \wedge \mathrm{st}^i_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake}$:
  $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
  $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return m^*

Oracle $\mathsf{Test}(i)$:
If $\mathrm{st}^i_{\mathsf{KE}}.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ then return $\mathrm{st}^i_{\mathsf{KE}}.\mathsf{key}$
Return $\mathsf{fake}(\mathrm{st}^i_{\mathsf{KE}}.\mathsf{key})$

Figure 5.4: Game defining the utility of an AttKE scheme when used in the context of attested computation.

5.3 Hardware-based secure outsourced computation

In this section we build on the results in previous sections to design and analyze a protocol for secure outsourced computation. Informally, we require two properties: i) that only the legitimate local user can pass inputs to the outsourced program and ii) that the I/O of the remote program is secret from any observer (even an actively malicious one).

We first give syntax for the protocols that solve this problem, then propose formal definitions for the properties that we outlined above, and conclude with a generic construction that combines a key-exchange for attestation, a scheme for attested computation and an authenticated encryption scheme.

We propose in this a security definition for each one of these properties. We also show that it is easy to satisfy these definitions by building on top of an AttKE and an AC scheme. The main intuition here is using an AttKE over the AC protocol to established a shared authenticated encryption key between the (compiled) remote instance of the program and the local machine. It is then enough to use this shared key to build a secure channel between the remote and the local machine.

SYNTAX. A *Secure Outsourced Computation* scheme (SOC) for a remote machine $\mathcal{M}_R$ is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, \mathsf{id})$ is the program compilation algorithm. On input public parameters prms, a program P and a party identifier id, it outputs a compiled program P^*, together with an initial state st_l for the local side algorithms. We assume that initially $\mathsf{st}_l.\mathsf{accept} = \perp$. Note that unlike the AC compilation algorithm, this algorithm only takes one program as input, as this scheme is intended for providing guarantees for the whole trace and not only for an initial segment.

- $\mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$ is the client side initialization algorithm. On input public parameters prms, o (presumably the last message from the remote machine) and local state st_l, it returns the next message i to be delivered to the remote machine in the bootstrapping step step, together with the updated local state. We assume $\mathsf{BootStrap}$ sets an accept flag to true when the initialization process successfully terminates.

- $\mathsf{Verify}(\mathsf{prms}, o^*, \mathsf{st}_l)$ is the verification algorithm. It fulfills the same function as the AC verification algorithm. Note that, as all the inputs are provided by the local machine, we do not need to feed it the last input as it can be stored in the state. It is expected to return $\perp$ if $\mathsf{st}_l.\mathsf{accept} \neq \mathsf{true}$.

- $\mathsf{Encode}(\mathsf{prms}, i, \mathsf{st}_l)$ is the encoding algorithm. On input the public parameters, local state and the next intended input for P, it returns the next input i^* for P^* together with the updated local state. It is expected to return $\perp$ if $\mathsf{st}_l.\mathsf{accept} \neq \mathsf{true}$.

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ is, as in an AC scheme, the (untrusted) attestation algorithm.

A party A with identifier id who wants to outsource program P to the remote machine first compiles P with his id, thus obtaining P^* and some secret data st_l. He then loads P^* on the remote machine using some untrusted protocol. As it is, the program P^* is not ready to receive inputs intended for P: an initial bootstrapping phase (until $\mathsf{BootStrap}$ sets the accept flag) is necessary to establish some shared secrets between the IEE in which P^* is executed and A. Then when A wants to send an input to the remote execution, he encodes it using Encode, sends it (using Attest) and verifies the output provided by Attest using Verify.

In this section, for simplicity reasons, we assume that the program P is deterministic. However, as for an AC scheme it would be easy to extend all the definitions to a non-deterministic program.

Input Integrity. While security of attested computation aims at ensuring that a trace was honestly produced on the remote side, it does nothing to restrict the provenance of the inputs received.

We provide a stronger notion named *input integrity* which, intuitively, ensures that if a program is compiled by a party with identifier id, then only that party may use the remote compiled program. We ensure this property by making sure that the local and remote views coincide (up to the last message exchanged, which may not have yet been delivered). The following formula Ψ which relates two input/output traces captures this intuition.

$$\Psi(T, T') := T = T' \ \vee \ \exists o.\big(T = o :: T'\big) \ \exists i.\big(T' = i :: T\big)$$

The formalization that we provide in Figure 5.5 is as follows. The adversary chooses a program P that is compiled with an honest party's id yielding P^* (which is given to the adversary). The adversary is given access to two oracles. A bootstrapping oracle that simply executes $\mathsf{BootStrap}$ honestly; and a send oracle that verifies the last (presumed) output of the remote program and encodes the next input (which is provided by the adversary), while keeping track of the local view of the trace. The goal of the adversary is then create a mismatch between the local and remote view of the trace.

Practically, we let the adversary in our game use a local agent as oracle. This local agent initializes the remote process (using $\mathsf{BootStrap}$) then checks the last output of the remote machine and prepares the next input.

Game $\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$(P^*, \mathsf{st}_l) \leftarrow\$ \mathsf{Compile}(\mathsf{prms}, P, \mathsf{id})$
$\mathsf{tr} \leftarrow []$
Run $\mathcal{A}_2^{\mathcal{O},\mathcal{M}_R}(\mathsf{st}_\mathcal{A}, P^*)$
If $\nexists_{=1}\mathsf{hdl}$ such that
  $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^* \wedge$
  $\mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R(\mathsf{hdl})}) \neq []$
 Return false
$\mathsf{hdl} \leftarrow \mathsf{Program}_{\mathcal{M}_R}^{-1}(P^*)$
$T \leftarrow \mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R(\mathsf{hdl})})$
$T' \leftarrow \mathsf{tr}$
Return $\neg\Psi(T, T')$

Oracle $\mathsf{Send}(o^*, i)$:

$o, \mathsf{st}_l \leftarrow \mathsf{Verify}(\mathsf{prms}, o^*, \mathsf{st}_l)$
If $o = \perp$ Return $\perp$
$i^*, \mathsf{st}_l \leftarrow \mathsf{Encode}(\mathsf{prms}, i, \mathsf{st}_l)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle $\mathsf{BootStrap}(o)$:

If $\mathsf{st}_l.\mathsf{accept}$
 Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Figure 5.5: Input integrity of a SOC scheme

Definition 18 (Input Integrity)**.** *We say that a SOC scheme satisfies* input integrity *if there exists a polynomial time algorithm* Translate *such that for all ppt $\mathcal{A}$ the experiment described in Figure 5.5 returns true with probability negligible in the security parameter.*

Input privacy. We define the privacy of I/O with an indistinguishability game. One important point here is that we chose to restrict the class of programs we consider to *length-uniform* (written lu) programs. A program is length uniform if the length of its outputs depends only on the length of its inputs. Intuitively, this is because the encryption scheme is allowed to leak the length of the messages, which in turn would leak information about the inputs for a non lu program.
The formalization described in Figure 5.6 is as follows. We start by choosing a bit b that will determine whether the adversary will be talking with the left send oracle or the right send oracle (described later). As for input integrity, the adversary then chooses a program P. We compile it for an honest party's identifier and give the resulting P^* to the adversary. The adversary is also given access to the bootstrapping oracle. In addition, he is given access to a left or right send oracle. This oracle, on a request with the last candidate output of the remote machine and two inputs i_0 and i_1, verifies the last candidate output and, depending on the bit b, encodes either i_0 or i_1 and returns the result. The goal of the adversary is to guess the bit b with non-negligible bias from 1/2.

Game $\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:

$b \leftarrow\$ \{0,1\}$
$\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
If $\neg\,\mathrm{lu}(P)$ Return $b' \leftarrow\$ \{0,1\}$
$(P^*, \mathsf{st}_l) \leftarrow\$ \mathsf{Compile}(\mathsf{prms}, P, \mathsf{id})$
$b' \leftarrow \mathcal{A}_2(\mathsf{st}_\mathcal{A}, P^*)^{\mathcal{O},\mathcal{M}_R}$
Return $b = b'$

Oracle $\mathsf{Send}_b(o^*, i_0, i_1)$:

$o, \mathsf{st}_l \leftarrow \mathsf{Verify}(\mathsf{prms}, o^*, \mathsf{st}_l)$
If $|m_0| \neq |m_1|$ Return $\perp$
$i^*, \mathsf{st}_l \leftarrow \mathsf{Encode}(\mathsf{prms}, i_b, \mathsf{st}_l)$
Return i^*

Oracle $\mathsf{BootStrap}(o)$:

If $\mathsf{st}_l.\mathsf{accept}$
 Return $\perp$ // (1 init max)
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Figure 5.6: Input privacy of a SOC scheme

Definition 19. *We say that a SOC scheme satisfies* input privacy *if, for all ppt $\mathcal{A}$, the experiment in Figure 5.6 returns true with probability* 1/2 *up to a negligible function.*

This definition ensures that there exist no two traces (with messages of the same length) played by an honest party over a SOC protocol that are distinguishable for an (active) adversary. This means that no adversary can gain information on the inputs sent out by a local machine using a SOC scheme, besides the length of the messages exchanged, achieving our goal of hiding the honest party's inputs.

Definition 20. *We say that a SOC scheme is* secure *if it satisfies both input privacy and input integrity.*

AN IMPLEMENTATION OF A SECURE SOC SCHEME. Having defined what security we expect from a SOC scheme, we now define a scheme that satisfies these requirements. We base our construction on an AttKE, and an AC scheme. The main idea is using the AttKE to establish a key between the party agent and the IEE, and then communicate with the IEE over the secure channel established with this key.

Formally, let $(\mathsf{Compile}, \mathsf{Attest}, \mathsf{Verify})$ be an AC scheme, $(\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}})$ be an AttKE and (E, D, K) be an authenticated encryption scheme. Figure 5.7 defines a SOC scheme. The most important part is the compilation part, which uses the AC scheme compilation to compile the composition of the $\mathsf{Rem}_{\mathsf{KE}}$ program generated by Setup together with program P running over a secure channel (denoted by $C(P)$). The initial local state is the union of the state provided by the AC compilation and the AttKE setup. The program $C(P)$ simply decrypts the message it receives checks that the sequence number of the message matches its view the passes the decrypted message to P. It then retrieves the output of P, appends the corresponding next sequence number and outputs it. This mechanism ensures that all messages received (resp. sent out) by P^* after the bootstrapping phase have the form $E(i\#m, k)$ where i is the position of the message in the trace, m is the message intended to (resp. produced by) P, and k is the key established by the AttKE.

On the local side, the bootstrapping mechanism simply consists of running the local KE over the AC protocol as already described in the utility definition. Once the key has been established, the local state keeps track of the local view of the sequence number. Verifying an output consists in decrypting it and checking that the sequence number against the local view of it. Encoding an input, is just appending the correct sequence number and encrypting it with the shared key.

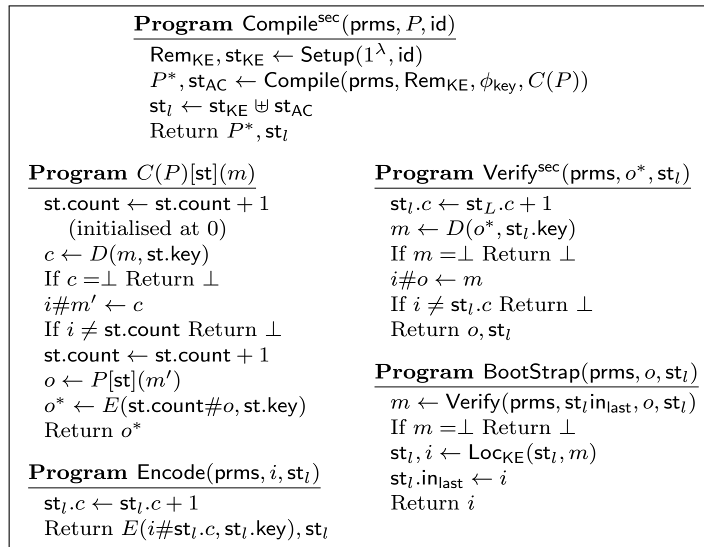


Figure 5.7: SOC algorithms

Theorem 8. *If* $(\mathsf{Compile}, \mathsf{Attest}, \mathsf{Verify})$ *is a correct and secure AC scheme,* $\{\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}}, \mathsf{Rem}_{\mathsf{KE}}\}$ *is a secure AttKE and* (E, D, KG) *is an secure authenticated encryption scheme, then the SOC presented in Figure 5.7 is secure.*

The complete proof can be found in the full version of the paper, we provide here a sketch of proof, following the game hopping paradigm. We first do a game hop that consists in replacing the key established by P^* and the party using the AttKE by a "magically" shared fresh key. The utility property of the AttKE provides us with the fact that we can replace the key shared by the remote machine and the local agent by a freshly generated key. We are left with showing that this key is shared with an IEE which is indeed running P^* and not $\mathsf{Compile}(\mathsf{Rem}_{\mathsf{KE}}, \phi_{\mathsf{key}}, Q)$ for some other Q, this is provided by the security of the AC scheme.

We then prove input integrity by remarking that injecting new messages in the trace would contradict the unforgeability of the authenticated encryption scheme. The sequence number ensures that the messages are delivered in the right order and that replays are impossible.

We remark that the input integrity property ensures that we know that the only meaningful action the adversary can take is to forward messages between the remote and local machines. Taking advantage of that fact we can reduce the input privacy game to the IND-CPA property of the authenticated encryption.

5.4 Discussion

The Secure Outsource Computation protocol we present here is highly independent of the particular architecture considered, therefore providing a generic compiler for such a scheme would be relatively easy. Additionally, besides enabling IEE, it require practically no effort on the side of the cloud provider. On the client side only a relatively simple protocol has to take place.

We also note that besides the simple key exchange taking place at the begining of the computation, the only thing added to the run of the program consists in (highly efficient) symmetric encryptions and tagging messages with sequence numbers. Hence the overhead with respect to running the "naked" program is very low, both in terms of communications and computations. This make our scheme a good candidate for delegating heavy computations to the cloud.

The main caveat of this protocol is its fundamentaly single user nature. Only one client may have access to the remote program. Distributing the key to several clients would not solve this problem as everyone has to know the sequence number. In the next Chapter we explore a way to build primitives with multiple user, namely secure function evaluation, that allow for a one-off evaluation of a computational task on multiple users.

Chapter 6

Secure Function Evaluation from Attested Computation

6.1 Overview

Consider the scenario in which Alice and Bob are two millionaires with respective fortunes A and B. They want to settle over who is richer (i.e. $A > B$?), but none of them want to reveal any additional information about their wealth. This task is typically known as the *millionaires' problem* [78], and is a specific instance of a more general scenario in which a larger number of players wish to collectively compute a function f on secret inputs, without disclosing any additional information. This computational problem for general functions is known as *secure function evaluation*, or *secure multiparty computation*.

The first proposed solutions for these problems were provided by Yao [78] for the two party case, and by Damgård et al. [45] for the multiparty case. Initially this was not considered to be feasible for practical implementation, since these early approaches failed to attain the efficiency requirements for typical usage. However, state-of-the-art research is constantly exploring new ways to increase its feasibility, improving efficiency [49, 50] and usability [42, 48, 67]. This growth in performance is substantiated by the fact that modern SFE implementations are already being employed for solving real-world problems, such as computing Denmark's sugar-beet double auction [16], or predicting satellite collisions [53].

However, asides from solving very specific problems, SFE is still considered to be insufficient for achieving generic feasible real-world deployments. Part of the problem stems from the type of adversary power considered. *Semi-honest* adversaries correctly follow the specified protocol, yet may attempt to learn additional information by analysing the transcript of messages received during the execution; *Malicious* adversaries may behave arbitrarily and are not bound in any way to follow the instructions of the specified protocol. Frameworks for MPC computations such as FairPlayMP [11] and Sharemind [15] have produced interesting practical results (in particular, the previous satellite collision detection example was achieved using Sharemind), but assume a *semi-honest* adversary, which is often insufficient for many real use cases (see security definition of Section 4.4 for application scenarios of PRACTICE in which this is the case). Passively secure protocols can be compiled into actively secure ones by employing zero-knowledge proofs-of-knowledge and verifiable secret sharing schemes, following a construction known as the GMW protocol compiler [57], however these techniques are usually considered inefficient for practical use. Alternative approaches such as SPDZ [31] achieve security against

active adversaries without a large computation overhead, but instead rely on a (possibly heavy) pre-processing stage.[1]

Chapter 5 proposes a mechanism that allows for a local client to securely outsource computation towards a machine that is potentially under (active) adversarial control. This is anchored on the remote machine being equipped with a trusted hardware that enables the usage of isolated execution environments, as described in Chapter 4. This Chapter is dedicated to extending the notions of single user outsourced computation to multiple user outsourced computation, i.e. hardware-based secure function evaluation. In this regard, we consider two approaches. The first one consists of a direct employment of the mechanisms for attested computation used for secure computation outsourcing, but now considering multiple input participants. The second introduces a variant for labeled attested computation, which can then be used for constructing a functionally similar SFE protocol, but requiring a reduced amount of exchanged data.

6.2 Ideal function evaluation from SGX

We want to securely execute a functionality $\mathcal{F}$ defined by algorithms $\{\mathsf{F}, \mathsf{Lin}, \mathsf{Lout}\}$, where F is a function to be evaluated. More precisely $(o_1, \ldots, o_n) \leftarrow \mathsf{F}(i_1, \ldots, i_n)$ is a function with n inputs and n outputs, one for each party; $\mathsf{Lin}(i, k)$ defines the public leakage that can be revealed by a protocol from a given input i by party k; and $\mathsf{Lout}(o_1, \ldots, o_n)$ defines the public leakage associated with the outputs of F.

Execution model. We assume the existence of a machine $\mathcal{M}$ allowing for the usage of isolated execution environments. Following the description in Section 4.3, this machine is first initialized via the Init algorithm, which defines public parameters that one assumes can be independently authenticated by all parties. The machine $\mathcal{M}$ is assumed to be adversarially controlled, but it does include isolated execution environments in which programs can be loaded (via the Load mechanism) and then interactively fed with new inputs (via the Run mechanism) to obtain attested outputs. All the code that is run in $\mathcal{M}$ but outside these execution environments is considered to be adversarially controlled. This adversary controls the interaction with $\mathcal{M}$ and the goal is to guarantee that a set of parties can use the IEE capabilities of $\mathcal{M}$ securely (bar the possibility that $\mathcal{M}$ refuses to allow the protocol to proceed, which would amount to a DoS attack).

Syntax. A protocol π for functionality F with n inputs and n outputs, is a six-tuple composed of five algorithms and an integer $\pi = \{\mathsf{Setup}, \mathsf{Init}, \mathsf{Process}, \mathsf{Output}, \mathsf{Remote}\}$, as follows:

- Setup – This is the party set-up algorithm. Given the security parameter, the public parameters prms for machine $\mathcal{M}$, the party's identifier in $\mathcal{I}$ and the party's participant number in the protocol i (i.e., a number in the range $[1..n]$), it returns the party's initial state (incluing its secret key material) and its public information.
- Init – This is the party protocol initialization algorithm. Given the party's initial state st, the public information of all participants Pub and its private input in, it updates and returns the party's state.

[1]The provided adversarial definitions and secure computation solutions have already been evaluated in detail for WP12 [18] and WP11 [63], respectively.

```
Game Corr_{π,R,A,M}(1^λ, I):
  n ← Length(I)
  prms ←$ M.Init(1^λ)
  For i ∈ [1..n]:
      (st_i, pub_i) ←$ Setup(1^λ, prms, I[i], i)
  Pub ← (pub_1, ..., pub_n)
  (in_1, ..., in_n) ←$ A(1^λ, prms, Pub)
  For i ∈ [1..n]:
      st_i ←$ Init(st_i, Pub, in_i)
  st_R ← ϵ; t ← true
  m' ← ϵ
  For r ∈ [1..R]:
      If t: (st_R, i, m) ←$ Remote^M(st_R, prms, Pub, m')
      Else: (st_i, m') ←$ Process(st_i, m)
      t ← ¬t
  For i ∈ [1..n]:
      out_i ←$ Output(st_i)
  (out'_1, ..., out'_n) ← F(in_1, ..., in_n)
  Return (out_1, ..., out_n) = (out'_1, ..., out'_n)
```

Figure 6.1: Game defining the correctness of our protocol.

- Process – This is the party activation algorithm. Given its internal state st and an input message m, it executes the protocol, updates the internal state and returns output message m'.
- Output – This is the party output retrieval algorithm. Given its interal state st, it simply returns the computed output o.
- Remote – This is the untrusted code that will be run in $\mathcal{M}$ and which will ensure the correctness of the protocol. It states the correct procedure to initialize the necessary IEE in the remote machine, as well as the management of messages exchanged between parties and that IEE. It receives oracle access to $\mathcal{M}$, public parameters prms, its state st_R (initialized as ϵ), the program to compile Pub and an input m', and returns an updated state st_R and the output message m.

Correctness. The following definition formalizes the notion of n users *correctly running a function evaluation protocol* π for F.

Definition 21. *We say π is an R-round correct function evaluation protocol for $\mathcal{F}$ if, for all λ, for all $\mathcal{I}$ and all adversaries $\mathcal{A}$, the experiment in Figure 6.1 always returns* true.

Discussion. First, the public parameters are set by initializing the machine $\mathcal{M}$.Init and running the setup for all n parties Setup. Afterwards, the adversary is given these public parameters, and gets to choose all inputs for the protocol. Then, the protocol is initialized using Init on all n parties, and it is run until it reaches its termination, at which point its outputs are retrieved via Output. The adversary wins the game if it can force the game to produce a set of outputs that wouldn't be obtained by simply running the functionality F with the given inputs.

Security. Our security definition is as follows, and it refers to the games in Figure 6.2.

Definition 22. *We say π is an R-round secure function evaluation protocol for $\mathcal{F}$ if, for any adversary $\mathcal{A} = \{\mathcal{A}_0, \mathcal{A}_1\}$, there exists a simulator $\mathcal{S} = \{\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$ such that the following definition of advantage is a negligible function in the security parameter.*

$$|\Pr[\mathsf{Real}^{\pi,R,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow b = 1] - \Pr[\mathsf{Ideal}^{\mathcal{F},R,\mathcal{A},\mathcal{S}}(1^\lambda) \Rightarrow b = 1]|$$

Game $\mathsf{Real}_{\pi,R,\mathcal{A},\mathcal{M}}(1^\lambda,\mathcal{I})$:
$n \leftarrow \mathsf{Length}(\mathcal{I})$
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(\mathrm{st}_{\mathcal{A}}, k) \leftarrow\$ \mathcal{A}_0(1^\lambda, \mathsf{prms})$
For $i \in [1..k]$:
  $(\mathrm{st}_i, \mathsf{pub}_i) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathsf{prms}, \mathcal{I}[i], i)$
For $i \in [k+1..n]$:
  $(\mathrm{st}_i, \mathsf{pub}_i) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathsf{prms}, \mathcal{I}[i], i)$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_n)$
$b \leftarrow\$ \mathcal{A}_1^{\mathcal{O}}(\mathrm{st}_{\mathcal{A}}, \mathsf{Pub}, \mathrm{st}_{k+1}, \ldots, \mathrm{st}_n)$

Oracle $\mathsf{Load}(P)$:
Return $\mathcal{M}.\mathsf{Load}(P)$

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{m})$:
Return $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, \mathsf{m})$

Oracle $\mathsf{SetInput}(\mathsf{in}, i)$:
If $i \notin [1..k]$ Return $\perp$
$\mathrm{st}_i \leftarrow\$ \mathsf{Init}(\mathrm{st}_i, \mathsf{Pub}, \mathsf{in})$

Oracle $\mathsf{Send}(i, m)$:
If $i \notin [1..k]$ Return $\perp$
$(\mathrm{st}_i, m') \leftarrow\$ \mathsf{Process}(\mathrm{st}_i, m)$
Return m'

Oracle $\mathsf{GetOutput}(i)$:
If $i \notin [1..k]$ Return $\perp$
Return $\mathsf{Output}(\mathrm{st}_i)$

Game $\mathsf{Ideal}_{\mathcal{F},R,\mathcal{A},\mathcal{S}}(1^\lambda,\mathcal{I})$:
$n \leftarrow \mathsf{Length}(\mathcal{I})$
$(\mathrm{st}, \mathsf{prms}) \leftarrow\$ \mathcal{S}_0(1^\lambda)$
$(\mathrm{st}_{\mathcal{A}}, k, \mathsf{pub}_{k+1}, \ldots, \mathsf{pub}_n) \leftarrow\$ \mathcal{A}_0(1^\lambda, \mathsf{prms})$
For $i \in [1..k]$:
  $(\mathrm{st}, \mathsf{pub}_i) \leftarrow\$ \mathcal{S}_1(1^\lambda, \mathrm{st}, \mathcal{I}[i], i)$
For $i \in [k+1..n]$:
  $(\mathrm{st}_i, \mathsf{pub}_i) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathsf{prms}, \mathcal{I}[i], i)$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_n)$
$b \leftarrow\$ \mathcal{A}_1^{\mathcal{O}}(\mathrm{st}_{\mathcal{A}}, \mathsf{Pub}, \mathrm{st}_{k+1}, \ldots, \mathrm{st}_n)$

Oracle $\mathsf{Load}(P)$:
Return $\mathcal{S}_2(\mathrm{st}, \mathsf{Pub}, P)$

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{m})$:
Return $\mathcal{S}_3^{\mathsf{Fun}}(\mathrm{st}, \mathsf{Pub}, \mathsf{hdl}, \mathsf{m})$

Oracle $\mathsf{SetInput}(\mathsf{in}, i)$:
If $i \notin [1..k]$ Return $\perp$
$l \leftarrow \mathsf{Lin}(\mathsf{in}, i)$
$\mathrm{st} \leftarrow\$ \mathcal{S}_4(\mathrm{st}, l, i)$

Oracle $\mathsf{Send}(i, m)$:
Return $\mathcal{S}_5^{\mathsf{Fun}}(\mathrm{st}, \mathsf{Pub}, i, \mathsf{m})$

Oracle $\mathsf{GetOutput}(i)$:
If $i \notin [1..k]$ Return $\perp$
$c \leftarrow\$ \mathcal{S}_6(\mathrm{st}, i)$
Return $(\mathsf{out}_i * c)$

Oracle $\mathsf{Fun}(\mathsf{in}_{k+1}, \ldots, \mathsf{in}_n)$:
$\mathsf{out}_1, \ldots, \mathsf{out}_n \leftarrow\$ \mathsf{F}(\mathsf{in}_1, \ldots, \mathsf{in}_n)$
$l_{\mathsf{out}} \leftarrow \mathsf{Lout}(\mathsf{in}_1, \ldots, \mathsf{in}_n, \mathsf{out}_1, \ldots, \mathsf{out}_n)$
Return $(\mathsf{out}_{k+1}, \ldots, \mathsf{out}_n, l_{\mathsf{out}})$

Figure 6.2: Game defining the behavior of Real and Ideal worlds. Fun can only be run once.

Discussion. We take the ideal versus real world approach. Our definition is a simplification of UC definitions such as [23] in which we assume active adversaries, static corruptions and a fixed number of participants n.

Real world considers a machine $\mathcal{M} = \{\mathsf{Init}, \mathsf{Load}, \mathsf{Run}\}$ following the description in Section 4.3. It starts by setting the parameters using $\mathcal{M}.\mathsf{Init}$ and letting the adversary select how many parties it wants to corrupt, allowing it to set the associated public parameters. Afterwards, the remaining honest parties are initialized via the Setup procedure. Finally, the adversary is to interact with the system via oracles SetInput, GetOutput, Load, Run and Send. SetInput and GetOutput are used to set the inputs and read outputs of the honest parties, Load and Run allow interaction with the machine $\mathcal{M}$, and Send enables the adversary to deliver messages to honest parties. When the interaction is complete, the adversary will attempt to guess the world it is executing in, by outputting a bit b.

In the ideal world, a simulator will mimic the behavior of the parties and the machine. First the simulator sets the machine's parameters and the adversary sets the public parameters of the parties it wants to corrupt. Afterwards, the simulator initializes the parameters of the remaining honest parties. Finally, the adversary is to interact with the system via oracles SetInput, GetOutput and the simulator. SetInput and GetOutput are used to set inputs and get outputs without providing the simulator with more information than what is allowed by Lin, while the simulator must emulate the behavior of Load, Run and Send for the honest parties. To do so, the simulator can run the idealized functionality via Fun, providing inputs from corrupt

parties and getting the associated outputs and honest input leakage. Similar to the real world, the adversary will finish the interaction by outputting a bit b regarding its guess.

As building blocs for our protocol, we introduce $\mathsf{ParComp}$ and Box, described in Figure 6.3. $\mathsf{ParComp}$ can be used to perform parallel execution of several instances of key exchange for attested computation (see Section 5.2). This program construction takes n programs $\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}$ and produces a program that is built to expect every input to consist in an array of n inputs: it parses the received input, runs every instance $\mathsf{Rem}^i_{\mathsf{KE}}$ with the i-th input and constructs an output array with the outputs of the executions. Box is used to enable input/output security for some desired functionality. This program construction takes some generic function F and an encryption scheme Λ and produces a program that is built to decrypt every input using Λ and some key within its internal state $\mathsf{st.key}$, as well as encrypt every output using the same scheme and key.

Program $\mathsf{ParComp}\langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle(\mathsf{m}, \mathsf{st})$:	**Program** $\mathsf{Box}\langle \mathsf{F}, \Lambda \rangle(\mathsf{m}_1, \ldots, \mathsf{m}_n, \mathsf{st})$:

If $\mathsf{st} = \epsilon$:
    For $i \in [1..n]$: $\mathsf{st}[i] \leftarrow \epsilon$
For $i \in [1..n]$:
    $o_i \leftarrow\$ \mathsf{Rem}^i_{\mathsf{KE}}(\mathsf{m}[i], \mathsf{st}[i])$
$\mathsf{out} \leftarrow o_1, \ldots, o_n$
Return out
	For $i \in [1..n]$:
	    $\mathsf{inp}_i \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st.key}, \mathsf{m}_i)$
	$(\mathsf{out}_1, \ldots, \mathsf{out}_n) \leftarrow \mathrm{F}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n)$
	For $i \in [1..n]$:
	    $o_i \leftarrow \Lambda.\mathsf{Enc}(\mathsf{st.key}, \mathsf{out}_i)$
	Return $(o_1, \ldots, o_n)$

Figure 6.3: Details for running n parallel key exchange protocols (left), and for coding inputs/outputs of an arbitrary functionality (right).

One-to-many utility

We now present a one-to-many utility theorem to precisely describe the guarantees obtained by combining an attested computation protocol with several instances of attested key exchange running in parallel. This theorem validates that, if the authentication and secrecy assurances offered by AttKE are retained when we use it to establish keys with a remote IEE, then this is also the case for multiple clients establishing keys with a remote IEE. This setting considers k honest participants, and the presence of a fully active adversary, in control of the machine and the remaining $n - k$ participants.

Figure 6.4 shows an idealized experiment similar to the approach taken for the AttKE Utility theorem. The adversary is challenged to distinguish between two remote machines, where several AttKE schemes are being combined and executed with an AC scheme. Machine $\mathcal{M}_R$ represents the standard remote machine supported by the AC protocol, while machine $\mathcal{M}'_R$ is a modification of $\mathcal{M}_R$ in which the operation of the first k (honest) $\mathsf{Rem}_{\mathsf{KE}}$ programs is tweaked. The differences between these machines are mostly concentrated on the Run interface. Run takes, as additional parameters, a list of key pairs fake, a Boolean flag tweak and the number of honest participants k. The activation of this flag identifies an IEE running the selected parallel execution of $\mathsf{Rem}_{\mathsf{KE}}$ composed with program Q, and triggers the following modifications with respect to the operations in $\mathcal{M}_R$:

- When it detects that any of the first k executions of $\mathsf{Rem}_{\mathsf{KE}}$ has transitioned into the $\mathsf{derived}$ or accept state, it records the derived key and checks if it exists in the provided list fake. If that is not the case, it generates a new random key^* and adds $(\mathsf{key}, \mathsf{key}^*)$ to the list.

Game $\mathsf{Att}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda, \mathcal{I}, k, \phi_{\mathsf{key}}, \mathsf{F})$:
$\mathsf{prms}_0 \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$ \mathcal{M}'_R.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{fake} \leftarrow [\,]$
$i \leftarrow 0$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b)$
Return $b = b'$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl}_0 \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
$\mathsf{hdl}_1 \leftarrow \mathcal{M}'_R.\mathsf{Load}(R^*)$
Return hdl_b

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$:
$o_0 \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$
$\mathsf{tweak} \leftarrow \mathsf{false}$
If $\mathsf{Program}_{\mathcal{M}'_R}(\mathsf{hdl}) \in \mathsf{PrgList}$ then $\mathsf{tweak} \leftarrow \mathsf{true}$
$(o_1, \mathsf{fake}) \leftarrow\$ \mathcal{M}'_R.\mathsf{Run}(\mathsf{hdl}, \mathsf{in}, \mathsf{tweak}, \mathsf{fake}, k)$
Return o_b

Oracle $\mathsf{NewSession}()$:
$i \leftarrow i + 1$
For $j \in [1..n]$:
  $(\mathsf{st}^{ij}_{\mathsf{KE}}, \mathsf{Rem}^j_{\mathsf{KE}}) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathcal{I}[j])$
  $\mathsf{in}^{ij}_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{RemComp} := \mathsf{ParComp}\langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle$
$Q := \mathsf{Box}\langle \mathsf{F}, \Lambda \rangle$
For $j \in [1..n]$:
  $(R^*_i, \mathsf{st}^{ij}_L) \leftarrow\$ \mathsf{AC.Compile}(\mathsf{prms}_b, \mathsf{RemComp}, \phi_{\mathsf{key}}, Q)$
$\mathsf{PrgList} \leftarrow R^*_i : \mathsf{PrgList}$
Return $(R^*_i, \mathsf{st}^{i(k+1)}_{\mathsf{KE}}, \ldots, \mathsf{st}^{in}_{\mathsf{KE}})$

Oracle $\mathsf{Send}(\mathsf{m}', i, j)$:
$(\mathsf{ins}, \mathsf{outs}^*) \leftarrow \mathsf{m}'$
$(\mathsf{outs}, \mathsf{st}^{ij}_L) \leftarrow \mathsf{AC.Verify}(\mathsf{prms}, \mathsf{ins}, \mathsf{m}', \mathsf{st}^{ij}_L)$
If $\mathsf{outs} = \perp$ Return $\perp$
If $\mathsf{in}^{ij}_{\mathsf{last}} \neq \mathsf{ins}[j]$ Return $\perp$
$(\mathsf{m}^*, \mathsf{st}^{ij}_{\mathsf{KE}}) \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}^{ij}_{\mathsf{KE}}, \mathsf{outs}[j])$
$\mathsf{in}^{ij}_{\mathsf{last}} \leftarrow \mathsf{m}^*$
If $\mathsf{st}^{ij}_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \wedge \mathsf{st}^{ij}_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake}$:
  $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
  $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return m^*

Oracle $\mathsf{Test}(i)$:
$\mathsf{Keys} \leftarrow [\,]$
For $j \in [1..k]$:
  If $\mathsf{st}^{ij}_{\mathsf{KE}}.\delta \neq \mathsf{accept}$ return $\perp$
  If $b = 0$ then $\mathsf{Keys} \leftarrow \mathsf{st}^{ij}_{\mathsf{KE}}.\mathsf{key} : \mathsf{Keys}$
  Else $\mathsf{Keys} \leftarrow \mathsf{fake}(\mathsf{st}^{ij}_{\mathsf{KE}}.\mathsf{key}) : \mathsf{Keys}$
Return Keys

Figure 6.4: Game defining the one-to-many utility of a AttKE scheme when used in the context of attested computation.

- When it detects that program Q is executing for the first time, rather than using all keys as input to ϕ_{key}, it replaces the first k as the respective $\mathsf{fake}(\mathsf{key})$ (from $k+1$ to n, the behavior is similar to $\mathcal{M}_R$).

The provided environment models the adversary power according to standard attested computation interactions, where it is given access to the oracles Load and Run, matching to $\mathcal{M}_R$ or $\mathcal{M}'_R$, depending on the sampled secret bit b. The adversary can also initialize challenge programs by running $\mathsf{NewSession}()$, which initializes all n executions, composes the n KE programs and composes them with program Q, having the set of exchanged keys mapped to its initial state via ϕ_{key}. Note that, additionally to the adversary having access to the machine, here we extend its power to control $n-k$ participants, by having $\mathsf{NewSession}$ provide the internal states of corrupt parties $k+1, \ldots, n$. Upon calling $\mathsf{NewSession}$, the environment creates new session i, with which the adversary can interact with in behalf of participant j by using $\mathsf{Send}(\mathsf{m}', i, j)$. The Send oracle mimics the behavior of a local participant by using the Verify algorithm of the AC scheme to validate if the output was correctly attested, and if the j-th input (the input provided by the local participant) is consistent with what was provided. If both of these conditions are met, the result is fed to the $\mathsf{Loc}_{\mathsf{KE}}$ instance. Finally, the adversary can challenge an explicit session i to test, by calling $\mathsf{Test}(i)$. This oracle will return either the honest participant (the initial k) set of true keys, if $b = 0$, or the associated random keys kept in the fake list. We define the winning event guess to occur when $b = b'$ in the end of the game.

Theorem 9 (One-to-many AttKE utility)**.** *If the AttKE is correct and secure, and the AC protocol is correct, secure and ensures minimal leakage, then for all ppt adversaries in the utility*

experiment: the probability that the adversary violates the AttKE two-sided entity authentication is negligible; and the key secrecy advantage $2 \cdot \Pr[\mathsf{guess}] - 1$ *is negligible.*

Protocol

We now present a construction of a protocol that relies on attested computation (given the one-to-many utility theorem) to bootstrap secure multiparty computation. The main property of this construction is that messages are bundled between all local clients and the remote IEE executing the protocol. In particular,

- All n local participants receive the same message for every computation round, containing the n responses. Every participant is aware of its position within the protocol, and uses that information to extract its corresponding message.
- The remote untrusted code is responsible for collecting every individual message. As it receives all n inputs, it concatenates them and feeds it to the IEE executing the protocol.

Note that, since the remote code handling the inputs is considered untrusted, the set of algorithms running locally must ensure that the adversary is unable to tamper with the execution trace).

Generic construction. Our protocol is composed of five algorithms, following the syntax in Section 6.2: $\{\mathsf{Setup}, \mathsf{Init}, \mathsf{Process}, \mathsf{Output}, \mathsf{Remote}\}$. These algorithms are expressed in Figure 6.5 and will now be described in brief.

$\mathsf{Setup}(1^\lambda, \mathsf{prms}, \mathsf{id}, \mathsf{pos})$ starts participant of identifier id and position pos in the protocol, by generating its key exchange program $\mathsf{Rem}_{\mathsf{KE}}$ and initializing its state st. The local state is now ready to execute Init ($\mathsf{stage} \leftarrow 1$).

$\mathsf{Init}(\mathsf{st}, \mathsf{Pub}, \mathsf{in})$ composes the set of key exchange programs in Pub, prepares the code F to be securely evaluated, and compiles it to R^*. The internal state st is then updated with the computed verification sate st_V and with the provided input for the computation in. The local state is now ready to execute $\mathsf{Process}$ ($\mathsf{stage} \leftarrow 2$).

$\mathsf{Process}(\mathsf{m}, \mathsf{st})$, while the key exchange is executing ($\mathsf{stage} = 2$), verifies the attestation for the input received and if the included input of its position matches the last input provided ($\mathsf{in}_{\mathsf{last}}$), generates the next response for the key exchange and updates its state. After completing the key exchange ($\mathsf{stage} = 3$), $\mathsf{Process}$ extracts input in from its state, encrypts it with the exchanged key key to produce an encrypted output. Finally, after sending its input ($\mathsf{stage} = 4$), $\mathsf{Process}$ expects an output encrypted using the same shared key key, decrypting it and storing the result in out. From this point onwards, $\mathsf{Process}$ will not execute ($\mathsf{stage} = 5$).

$\mathsf{Output}(\mathsf{st})$ will simply return the value stored in variable out. This means that, before $\mathsf{Process}$ reaching its completion, the output will be $\perp$, otherwise it will be the result of securely evaluating F.

$\mathsf{Remote}(\mathsf{st}_R, \mathsf{prms}, \mathsf{Pub}, \mathsf{m})$ begins by composing and compiling the code, similarly to what is performed in Init, and loading the result into $\mathcal{M}$. Remote also keeps track of the messages produced by $\mathcal{M}$ and received from all local participants in $\mathsf{in}[\mathsf{pos}]$, as well as the next message to produce p. p defines the position of the participant that will receive the next message, so the behavior highly depends on the value of p:

- $p = 1$ means that a message must be provided for the first participant. This triggers Remote to collect the input received into $\mathsf{in}[n]$ (if the next participant is the first, then this is either the first message of the protocol, or the received message came from the last participant, i.e. n), run the IEE via $\mathsf{AC.Attest}$ for the collected inputs in (ϵ for the first execution) and update the record of the last inputs provided lin. The received output will be stored in out, the internal state st_R is updated and the message out is returned alongside the last input given to the IEE (lin), signaled to be delivered to participant p.
- $p \in [2, \ldots, n]$ stores the received input in $\mathsf{in}[p-1]$ (if the next response goes to p, then the received message was from $p-1$), updates its state and returns the previously computed $(\mathsf{out}, \mathsf{lin})$ flagged for participant p.

Contrary to the secure outsourced computation protocol proposed in Chapter 5, the local users don't have access to all inputs, and therefore cannot verify its trace in the same way. However, they do not need to verify the full trace. Indeed, every participant is interested in ensuring two properties regarding the computation: i) that the composed program within the IEE produced the provided I/O ($\mathsf{AC.Verify}$), and ii) that the provided I/O matches the participant output in its previous execution ($\mathsf{in}_{\mathsf{last}}$). As such, the fact that the untrusted code is responsible for providing the reference inputs is not problematic, as inaccurate data will result in either failing the AC check (if the I/O was not produced by the IEE) or failing the last input check (if the IEE was run with a tampered participant input). The adversary can, in fact, trick honest participant i by running the IEE with fake inputs from other participants, but doing this does not break the key exchange security. This is demonstrated by the one-to-many utility theorem in Section 6.2.

6.3 Secure Multi-agent Attested Computation (SMAC)

Observe that the previously described protocol relies on exchanging messages with length depending on the number of participants in the protocol. The reason for this is that the verification mechanism requires the full execution trace, which encompasses inputs from all participants.

Here we describe a variant of attested computation named *secure multi-agent attested computation* (SMAC), intended to provide guarantees to each agent with respect to the part of the trace corresponding to its label. The intuition here is that each participant does not need guarantees over the legitimacy of the entire trace, but rather of the part of the trace with respect to the inputs he (in particular) has provided. In this context, the user is not interested in validating the key exchange between the IEE and other users, and by not having to do so, we reduce the amount of data exchanged by the resulting protocol.

Syntax. A labelled reactive program takes as input a state st, a label l intended to represent the sender of the received message, and a message i. It outputs an updated state, and an

```
Algorithm Setup(1^λ, prms, id, pos):
  (st_L, Rem_KE) ← Setup_KE(1^λ, id)
  st_V ← ε; in_last ← ε; in ← ε; out ← ε
  stage ← 1
  st ← (prms, st_L, id, pos, st_V, in_last, in, out, stage)
  Return (st, Rem_KE)

Algorithm Process(m, st):
  (prms, st_L, id, pos, st_V, in_last, in, out, stage) ← st
  (id, st_KE, sk, t) ← st_L
  If stage = 2:
     (ins, outs*) ← m
     (outs', st_V) ← AC.Verify(prms, ins, m, st_V)
     If outs' =⊥ Return ⊥
     (ins, outs) ← outs'
     If in_last ≠ ins[pos] Return ⊥
     (o, st_L) ←$ Loc_KE(st_L, outs[pos])
     in_last ← o
     If (st_KE.δ) ∈ {derived, accept}
        stage ← 3
  Else If stage = 3:
     o ←$ Λ.Enc(st_KE.key, in)
     stage ← 4
  Else If stage = 4:
     (ins, outs) = m
     out ←$ Λ.Dec(st_KE.key, outs)
     stage ← 5
     o ← ε
  Else:
     Return ⊥
  st ← (prms, st_L, id, pos, st_V, in_last, in, out, stage)
  Return (st, o)

Algorithm Output(st):
  (prms, st_L, id, pos, st_V, in_last, in, out, stage) ← st
  Return out

Algorithm Init(st, Pub, in):
  (prms, st_L, id, pos, st_V, in_last, in, out, stage) ← st
  If (stage ≠ 1):
     Return ⊥
  (Rem^1_KE, ..., Rem^n_KE) ← Pub
  RemComp := ParComp⟨Rem^1_KE, ..., Rem^n_KE⟩
  Q := Box⟨F, Λ⟩
  (R*, st_V) ← AC.Compile(prms, RemComp, φ_key, Q)
  stage ← 2
  st ← (prms, st_L, id, pos, st_V, in_last, in, out, stage)
  Return st

Algorithm Remote^M(st_R, prms, Pub, m):
  n ← Length(Pub)
  If st_R = ε:
     (Rem^1_KE, ..., Rem^n_KE) ← Pub
     RemComp := ParComp⟨Rem^1_KE, ..., Rem^n_KE⟩
     Q := Box⟨F, Λ⟩
     (R*, st_V) ← AC.Compile(prms, RemComp, φ_key, Q)
     hdl ← M.Load(R*)
     For i ∈ [1..n]:
        in[i] ← ε
     p = 1
  Else:
     (hdl, in, lin, out, p) ← st_R
  If p = 1:
     in[n] ← m
     out ←$ AC.Attest^M(prms, hdl, in)
     lin ← in
  Else in[p − 1] ← m
  If (p = n) p' = 1
  Else p' ← p + 1
  st_R ← (hdl, in, lin, out, p')
  Return (st_R, p, (lin, out))
```

Figure 6.5: Algorithms defining the protocol.

output message o. We assume that the program always answers to its sender (a possibly empty message if no answer is needed).

$$P[\mathsf{st}](l, i) \rightarrow \mathsf{st}', o$$

In all the following we will assume that input and output messages are clearly identified. Given a label l, we write $\mathsf{Trace}^{l}_{P[\mathsf{st};r]}((i_1, l_1), \ldots, (i_n, l_n))$ for the subset of the I/O trace of P with random coins r, initial state st, on inputs $(i_1, l_1), \ldots, (i_n, l_n)$ that is labeled with l. The syntax of a SMAC scheme is rather similar to the one of AC provided in Section 4.4, and is as follows.

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q, L)$ where L is the set of label for which attestation is expected. Returns the outsourced program R^* together with the initial state of the verification algorithms st_l for every label $l \in L$.
- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i, l)$ behaves as previously described.
- $\mathsf{Verify}[\mathsf{st}_l](\mathsf{prms}, i, o^*)$ is the verification algorithm, which given the last input labeled l and the attested answer o^* to that input checks that the output is valid and produces the (decoded) output o.

Security. For security, intuitively we need to ensure that

- there is a procedure that extracts the trace from an IEE running R^* and translates it properly into the corresponding trace of R, and
- ensure that every verified labeled interaction is contained in the extraction of such a trace.

More precisely, we let the adversary choose a program and compile it into R^*. We then let the adversary interact with the remote machine, and then choose a label l. His goal is to produce a trace that passes the verification tests for l but is not included in a trace of a IEE running R^*. This is formalised in Figure 6.6.

Game $\mathsf{Att}_{\mathsf{SMAC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \phi, Q, L, n, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$(R^*, (\text{ st }_l)_{l \in L}) \leftarrow \mathsf{Compile}(\mathsf{prms}, P, \phi, Q, L)$
For $k \in [1..n]$:
  $(i_k, l_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_2^{\mathcal{M}_R}(\mathsf{st}_\mathcal{A})$
  $(o_{R,k}, \mathsf{st}_l) \leftarrow \mathsf{Verify}[\mathsf{st}_l](\mathsf{prms}, i_k, o_k^*)$
  If $o_{R,k} = \perp$ Then $V_l \leftarrow \mathsf{false}$
$l \leftarrow \mathcal{A}_3(\mathsf{st}_\mathcal{A})$
If $l \notin L$ Return false
If $V_l = \mathsf{false}$ Return false
$T_l \leftarrow (i_j, o_{R,j})_{\substack{j \leq n \\ l_j = l}}$
Define $R := \mathsf{Compose}_\phi\langle P; Q\rangle$
For hdl^* s.t. $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}^*) = R^*$:
  $((i'_1, l'_1), o'_1, \ldots, (i'_m, l'_m), o'_m) \leftarrow \mathsf{Translate}(\mathsf{ATrace}_{\mathcal{M}_R}(\mathsf{hdl}^*))$
  If there exists $r \leq m$ such that $T'' = \mathsf{ATrace}_{P[\text{ st };\mathsf{Coins}_{\mathcal{M}_R}(\mathsf{hdl}^*)]}((i'_1, l'_1), \ldots, (i'_r, l'_r))$ and $T_l \sqsubseteq T''_{|l}$:
    Return false
Return true

Figure 6.6: Security game of SMAC

Definition 23 (SMAC security)**.** *We say that a SMAC scheme is secure if there exists an algorithm* $\mathsf{Extract}$ *such that the experiment in Figure 6.6 returns* true *with only negligible probability.*

We define correctness of a SMAC scheme as expected. Minimal leakage is defined as for an AC scheme (ignoring the labels).

A SMAC scheme. The intuition behind the scheme is to do attestation independently for every label. The reasoning is also very similar to the proposed instantiation of the attested computation scheme.

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q, L)$ generates a new program $R^* = \mathsf{Compose}_\phi(P^*, Q)$ and outputs it together with the initial state of the verifier for each label $(l, R^*, [], 1)$. Program P^* keeps a list ios_l of I/O for each label in $l \in L$. On request (i, l), P^*
  - checks $l \in L$, if not returns $P[\mathsf{st}](i, l)$
  - computes $o = P[\mathsf{st}](i, l)$, adds (i, o) to ios_l, requests a MAC of (l, ios_l) from the security module getting a tag t on $(R^*, (l, \mathsf{ios}_l))$ and return $(o, t, R^*, \mathsf{ios}_l)$.
- Attest transmits the query to the IEE and then transforms the tag (if any) into a signature and returns it together with the output of P.
- $\mathsf{Verify}[\mathsf{st}_l](\mathsf{prms}, i, o^*)$ returns o^* if $\mathsf{stage} = 2$. Otherwise parses $o^* = (o, \sigma)$, appends (i, o) to the local ios_l list and checks that σ is indeed a valid signature of $(R^*, (l, \mathsf{ios}_l))$.

Theorem 10. *The SMAC protocol presented above is a secure SMAC protocol ensuring minimal leakage as long as the signature and mac scheme are unforgeable.*

Proof sketch. This proof is similar to the proof of security of an AC scheme. The proof of minimal leakage is exactly the same, ignoring the labels.

For security, we first use unforgeability of the MAC scheme to ensure that every signed message was a properly tagged message. We then use unforgeability of the signature to ensure that every message accepted by the verification algorithm was indeed produced by an IEE running the relevant program. The result then follows directly from the checks performed on ios_l. □

Protocol

As building blocs for our protocol, we introduce ParComp and Box, described in Figure 6.7. ParComp can be used to perform parallel execution of several instances of key exchange for attested computation [8]. Box is used to enable input/output security for some desired functionality. This program construction takes some generic function F and an encryption scheme Λ and produces a program that is built to decrypt every input using Λ and some key within its internal state, as well as encrypt every output using the same scheme and key. Note that we treat all the inputs at once as the output of the function can only be computed when all of the inputs have been received. The task of buffering the inputs is left to the untrusted I/O part of the remote machine.

Program $\mathsf{ParComp}\langle P_{l_1}, l_1 \dots, P_{l_n}, l_n\rangle[\mathsf{st}](i, l)$

- If $\exists j \leq n. l = l_j$:
  - Return $P_l[\mathsf{st}.l](i)$ Else: Return $\perp$

Program $\mathsf{Box}\langle \mathsf{F}, \Lambda, l_1, \dots, l_n\rangle[\mathsf{st}](\mathsf{m}, l)$:

- If $l \notin \{l_1, \dots, l_n\}$: Return $\perp$
- If $\mathsf{st}.L = \{l_1, \dots, l_n\}$: Return $\mathsf{st.out}_i$
- If $l \in \mathsf{st}.L$: Return none
- $\mathsf{st}.L \leftarrow \mathsf{st}.L \cup \{l\}$
- $\mathsf{st.inp}_l \leftarrow \Lambda.\mathcal{D}(\mathsf{st}.l.\mathsf{key}, \mathsf{m})$
- If $\mathsf{st}.L \neq \{l_1, \dots, l_n\}$: Return ok
- $(\mathsf{out}_1, \dots, \mathsf{out}_n) \leftarrow \mathsf{F}(\mathsf{inp}_1, \dots, \mathsf{inp}_n)$
- For $i \in [1..n]$:
  - $\mathsf{st.out}_{l_i} \leftarrow \Lambda.\mathcal{E}(\mathsf{st}.l_i.\mathsf{key}, \mathsf{out}_i)$
- Return ok

Figure 6.7: Labelled parallel composition (labels are assumed pairwise different)

Labelled Utility. We define utility almost as in the AC case, except that the key exchange can be composed in parallel with anything else. The proof follows the same lines. This security experiment intuitively states that the adversary can not distinguish between the derived key and a random value after a key exchange has been performed between an honest party and a remote machine running the key exchange in parallel with other programs in an IEE.

In the experiment in Figure 6.8 the adversary has to distinguish between an ideal machine and a real world machine where and AttKE is run in parallel with other programs in the first phase of a SMAC-compiled protocol. The machine $\mathcal{M}_r$ represents the remote machine expected by the SMAC protocol and the machine $\mathcal{M}'_r$ is a modification of machine $\mathcal{M}_r$ in which the key derived by the key-exchanges is magically replaced by a fresh key. In order to maintain consistency between the tested keys and the keys used in $\mathcal{M}'_r$, $\mathcal{M}'_r.\mathsf{Run}$ takes two additional parameters, a list fake of pairs of keys and a flag tweak and a label l. If the flag is activated, the following modifications in the behaviour of $\mathcal{M}_r$ occur in $\mathcal{M}^*_r$:

- When it detects the execution of the key exchange $\mathsf{Rem}_{\mathsf{KE}}$ labeled l in the initial parallel composition has reached the derived or accept state, it record the derived key. If there is no association (key, _) in fake it generates a fresh key key* and appends (key, key*) to fake.
- When it detects the first execution of the non-attested part of the program, Q it performs $\mathsf{st}.l.\mathsf{key} \leftarrow \mathsf{fake}(\mathsf{key})$ before applying ϕ_{key}.

The oracles provided to the adversary provide access to the remote machine. Additionally the adversary can create new sessions of the key exchange using the NewSession oracle, where the remote key exchange is composed in parallel (with label l_0) with programs $P_1, \ldots, P_n$, and compiled for SMAC followed with Q. It can also make the local part of the key exchange progress using the Send oracle, provided that the message passes the SMAC verification step for the relevant label. Finally the adversary can challenge a session by executing the Test oracle, which return either the real key of a fake key according to b (provided that the key exchange has reached a derived or accept state).

Game $\mathsf{Att}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda, \mathsf{id})$:

$\mathsf{prms}_0 \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$ \mathcal{M}'_R.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow []$
$\mathsf{fake} \leftarrow []$
$i \leftarrow 0$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b)$
Return $b = b'$

Oracle $\mathsf{Load}(R^*)$:

$\mathsf{hdl}_0 \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
$\mathsf{hdl}_1 \leftarrow \mathcal{M}'_R.\mathsf{Load}(R^*)$
Return hdl_b

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$:

$o_0 \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$
$\mathsf{tweak} \leftarrow \mathsf{false}$
If $\mathsf{Program}_{\mathcal{M}'_R}(\mathsf{hdl}), l \in \mathsf{PrgList}$ then $\mathsf{tweak} \leftarrow \mathsf{true}$
$(o_1, \mathsf{fake}) \leftarrow\$ \mathcal{M}'_R.\mathsf{Run}(\mathsf{hdl}, \mathsf{in}, \mathsf{tweak}, l, \mathsf{fake})$
Return o_b

Oracle $\mathsf{NewSession}(P_1, l_1, \phi_1, \ldots, P_n, l_n, \phi_n, l_0, Q, L)$:

If $\exists i, j.i \neq j.l_i = l_j$: Return $\perp$
If $l \notin L$: Return $\perp$
$i \leftarrow i + 1$
$(\mathsf{st}^i_{\mathsf{KE}}, \mathsf{Rem}^i_{\mathsf{KE}}) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{RemComp} := \mathsf{ParComp}\langle \mathsf{Rem}^i_{\mathsf{KE}}, l_0, P_1, l_1, \ldots, P_n, l_n \rangle$
$\phi := \phi^{l_0}_{\mathsf{key}} | \phi^{l_1}_1 | \ldots | \phi^{l_n}_n$
$(R^*_i, \mathsf{st}_{l_0}, \ldots, \mathsf{st}_{l_n}) \leftarrow\$ \mathsf{SMAC.Compile}(\mathsf{prms}_b, \mathsf{RemComp}, \phi, Q)$
$\mathsf{st}^i_L \leftarrow \mathsf{st}^i_{l_0}$
$\mathsf{PrgList} \leftarrow R^*_i : \mathsf{PrgList}$
Return R^*_i

Oracle $\mathsf{Send}(\mathsf{m}', i)$:

$(i, o^*) \leftarrow \mathsf{m}'$ $\mathsf{m} \leftarrow \mathsf{SMAC.Verify}[\mathsf{st}^i_L](\mathsf{prms}, \mathsf{in}_{\mathsf{last}}, \mathsf{m}')$
If $\mathsf{m} = \perp$: Return $\perp$
If $\mathsf{in}^i_{\mathsf{last}} \neq i$ Return $\perp$
$(\mathsf{m}^*, \mathsf{st}^i_{\mathsf{KE}}) \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}^i_{\mathsf{KE}}, m)$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{m}^*$
If $\mathsf{st}^i_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \wedge \mathsf{st}^i_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake}$:
  $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
  $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return m^*

Oracle $\mathsf{Test}(i)$:

If $\mathsf{st}^i_{\mathsf{KE}}.\delta \neq \mathsf{accept}$: Return $\perp$
If $b = 0$: Return $\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key}$
Else: Return $\mathsf{fake}(\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key})$

Figure 6.8: Labelled utility

Theorem 11 (Distributed AttKE utility). *If the AttKE is correct and secure, and the SMAC protocol is correct, secure and ensures minimal leakage, then for all ppt adversaries in the labelled utility experiment: the probability that the adversary violates the AttKE two-sided entity authentication is negligible; and the key secrecy advantage $2 \cdot \Pr[\mathsf{guess}] - 1$ is negligible.*

PROTOCOL DESCRIPTION.

We describe here a secure SFE protocol based on a SMAC protocol and an AttKE. Note that this description is entirely generic in the SMAC and AttKE protocols, and its security relies only on the security of both these protocols.

Intuitively, each party starts by executing an attested key exchange with the an IEE, deriving guarantees from it from the labelled utility theorem. Then each party transmits its input and receives its output on a secure channel established using this key.

The protocol defined in Figure 6.9 behaves as follows:

- $\mathsf{Setup}(1^\lambda, \mathsf{prms}, \mathsf{id}, pos)$ sets up an AttKE, publishes the remote part and remembers the local verifying state.
- $\mathsf{Init}(\mathsf{st}, \mathsf{Pub}, \mathsf{in})$ compiles using SMAC the parallel composition of the remote key exchange of each participant (with label its position), followed by the functionality running over secure channel established using the derived keys.
- Remote simply executes the compiled code on a remote machine together with the untrusted part of attestation: SMAC.Attest.
- $\mathsf{Process}(m, \mathsf{st})$ executes the local key exchange, and once the key exchange has derived the key send its input and receives its output from the remote machine on the secure channel obtained from the key.
- $\mathsf{Output}(\mathsf{st})$ simply returns the output received from the remote machine.

```
Algorithm Setup(1^λ, prms, id, pos):
  (st_L, Rem_KE) ← Setup_KE(1^λ, id)
  st_V ← ε; in_last ← ε; in ← ε; out ← ε
  stage ← 1
  st ← (prms, st_L, id, pos, st_V, in_last, in, out, stage)
  Return (st, Rem_KE)

Algorithm Process(m, st):
  (prms, st_L, id, pos, st_V, in_last, in, out, stage) ← st
  (id, st_KE, sk, t) ← st_L
  If stage = 2:
     (ins, outs*) ← m
     (outs', st_V) ← SMAC.Verify(prms, ins, m, st_V)
     If outs' =⊥ Return ⊥
     (ins, outs) ← outs'
     If in_last ≠ ins[pos] Return ⊥
     (o, st_L) ←$ Loc_KE(st_L, outs[pos])
     in_last ← o
     If (st_KE.δ) ∈ {derived, accept}
        stage ← 3
  Else If stage = 3:
     o ←$ Λ.E(st_KE.key, in)
     stage ← 4
  Else If stage = 4 and m ≠ none:
     (ins, outs) = m
     out ←$ Λ.D(st_KE.key, outs)
     stage ← 5
     o ← ε
  Else:
     Return ⊥
  st ← (prms, st_L, id, pos, st_V, in_last, in, out, stage)
  Return (st, (pos, o))

Algorithm Init(st, Pub, in):
  (prms, st_L, id, pos, st_V, in_last, in, out, stage) ← st
  If (stage ≠ 1):
     Return ⊥
  (Rem^1_KE, ..., Rem^n_KE) ← Pub
  RemComp := ParComp⟨Rem^1_KE, 1, ..., Rem^n_KE, n⟩
  Q := Box⟨F, Λ, 1, ..., n⟩
  (R*, st_V) ← SMAC.Compile(prms, RemComp, φ_key, Q, [[1, n]])
  stage ← 2
  st ← (prms, st_L, id, pos, st_V, in_last, in, out, stage)
  Return st

Algorithm Remote^M(st_R, prms, Pub, m):
  n ← Length(Pub)
  If st_R = ε:
     (Rem^1_KE, ..., Rem^n_KE) ← Pub
     RemComp := ParComp⟨Rem^1_KE, 1, ..., Rem^n_KE, n⟩
     Q := Box⟨F, Λ, 1, ..., n⟩
     (R*, st_V) ← SMAC.Compile(prms, RemComp, φ_key, Q, [[1, n]])
     hdl ← M.Load(R*)
     st_R ← hdl
  out ←$ SMAC.Attest^M(prms, hdl, in)
  Return (st_R, out)

Algorithm Output(st):
  (prms, st_L, id, pos, st_V, in_last, in, out, stage) ← st
  Return out
```

Figure 6.9: Algorithms defining the protocol.

In this protocol, each participant obtains the guarantee that its input is secretly transmitted from the labelled utility theorem and the structure of the Box construction. The participant does not have to check properties of the rest of the trace as the key exchanges performed by other honest participants are also checked. The SMAC guarantees then ensure that the input

is processed according to the expected functionality. This protocol is much more lightweight in terms of communication than the one presented in Section 6.2, as the communication for each participant is constant (instead of linear) in the number of participants. The computational overhead is also very small, due to the efficiency of AttKE.

Chapter 7

Formal Specification and Verification

7.1 Overview

In line with the goals of this workpackage, we have considered automatic verification of the type of protocols that we have developed. Automated verification of security protocols has been a widely successful field, yielding tools like ProVerif [14] and Sapic [55]. These tools consider protocols specified at a high abstraction language (e.g. as applied pi-calculus processes) [1, 64] and for which the attacker capabilities are represented as deduction rules. It turns out that while both of these tools have been widely successful in proving a wide range of protocols, their underlying specification language is not expressive enough to model IEEs. Important aspects like location (i.e. a means to specify *where* protocols are executed) and statefulness (needed to model the state maintained by IEEs between invocations) are not part of the basic syntax. We propose ways to mitigate this problem. Specifically, in this chapter we describe an extension of the Sapic calculus that encompasses what is needed to model IEEs, and then build a semantics-preserving encoding of this extension into the base calculus. We demonstrate the usefulness of our extensions through several experimental results obtained using our tool, namely an semi-automated proof of security of our AC scheme, the utility property and our Secure Outsourced Computation scheme.

The state of the art

A major aspect regarding computation in IEEs is their inherently stateful nature. The main tool able to handle stateful extensions of the applied pi-calculus is Sapic[55], which does not provide a decision procedure, but is rather a largely automatic proof assistant. It has been successfully used for proving various protocols based on the TPM hardware module [56]. The following work is based on this tool.

The key capability of an IEE is its capability of reporting on its execution. The main challenge here is, therefore, introducing a dependency between the code executed in an IEE and the semantics of this reporting function. Indeed it is a non-trivial problem to model a module that can produce a signed hash of the code. We solve this problem by introducing locations.

Adding Locations

Attaching locations to processes is an efficient way to identify different processes and specify the fact that they are running in different places, or IEEs. Intuitively, the location of a process is a public value that represents the hash of the code running at this location. Provided that we enforce that there is a one to one mapping from processes to locations, we have the part of the IEE that act as a black-box and have a public identity. One location correspond to one IEE and one process, and all this can be publicly known.

A location is not any more just a place where a process can run, it is now a place where only a specific process can run. The location identifier can then abstract the hash of the process used in the real world. Then, we can naturally use this location to report on messages computed by the remote process running in an IEE. We define our calculus such that only particular processes can report on messages with a trusted location. In this model, under the assumptions that every process attached to a trusted location has its code publicly known, when a verifier checks the signature he obtains both the knowledge that it was computed inside the IEE X by the program Y.

A process should not be able to produce a report that does not correspond to its location, so we must not give direct access to the reporting scheme. We want a process to be able to simply call a function with a term, and this function will act as a black box and produce a report on the term according to the location of the caller. This is actually close to what happens in an IEE in the real world where a process would report a message via a function call to the TM.

Figure 7.1 shows what we would like to be able to do in an informal setting. We again have two processes, a local verifier V and a remote process P. We consider that the name 'Provider' is known as a trusted public location and that the verifier knows what is the code of the remote process. We also give a function report(x) to the remote process running in an IEE that will magically produce a report on x according to the location of the remote process.

```
P :
  let x = report(m) in
    out(x)
  ________________________________
V :
  in(c);
    if chkreport(c) = 'Provider' then
      Ok
    else Fail
```

Figure 7.1: Informal example with locations

This example quite close to what we want. We now have a simple way for a process to prove its identity to a verifier. The difficult part is to actually define the semantic of this black box reporting function. We want locations to be public, as it is intuitively the case in the real world. Then, if locations are public, we cannot use a classical signature scheme as it would not make any sense to sign something with a public key. Our first intuition was to use secret-keys paired with the locations, but if this could actually be the case at low-level, we wanted to propose a top-level language easy to use and understand as the previous example. Moreover, as

several implementations of IEEs exist, we needed to stay general and not stick to one low-level modelisation.

Then, our semantic needs to define a set of "trusted locations" that denote the set of program we know and trust. It is also be necessary to define carefully the rules for the attacker, since he needs to be able to report anything he wants from an untrusted location, but he should be unable to produce reports corresponding to trusted locations.

Finally, locations may not necessarily be fixed. As we said, what we call location are not just locations, but also characterize the code running at this location. The problem is that characterizing the code of a location may requires some parameters because the code itself could depend on said parameters. The idea would be for a user to be able to create an IEE that will only communicate with him. We show a basic example where we just pass a session identifier to the remote process in order to be able to distinguish the different runs.

```
Provider(sid):
  in(sid);
  let x = report(m) in
    out(x)
  ________________________________
Verifier:
  new sid; out(sid);
  in(c);
  if chkreport(c) = 'Provider(sid)' then
    Ok
  else Fail
```

Figure 7.2: Informal example with variable locations

7.2 Formal definition of Slapic

We move on to the definition of our calculus: Slapic, a stateful and localized applied pi-calculus. This is an extension of Sapic that handles locations and reporting, in such a way that we can define a translation from Slapic to Sapicb and use the automated reasoning capabilities of Sapic. Many classical definitions regarding pi-calculus, such as terms, substitutions or facts, are ommitted here.

Locations

We first define locations in order to be able to attach locations to protocols. We will simply use the syntax $P@l_P$ to specify that P runs at location l_P. We allow locations to be bound to any process or sub-process, the idea being that a location set to one process will be inherited by all its sub-processes, unless they have another location specifically attached. In the following example, P has location l_P but Q has location l_T.

$$(new\ t; out(t); (P@l_P|Q))@l_T$$

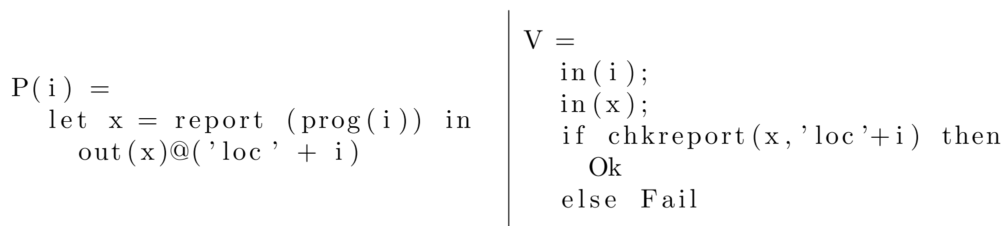
This could model the process at locations T that ask for the start of the process P.

Finally, we need to make a distinction between the set of locations that we assume to be secure, and the set of locations that is insecure and that the attacker could use. Intuitively, trusted locations correspond to code we know. To keep flexibility and allow the user to define its own set of trusted (or attested) locations, we will define the set of trusted, or known locations with a predicate. Then, the predicate can simply be used to capture a static set of locations like $X\ is\ trusted \Leftrightarrow X =' loc'$ or it can capture a more complex behavior such as $X\ is\ trusted \Leftrightarrow \exists z X = ('loc', z))$.

Definition 24. *Let $\mathcal{L} \subset T$ be the set of attested locations terms that will be attached to processes such that for a specified predicate HLoc:*

$$x \in \mathcal{L} \Leftrightarrow HLoc(x)$$

Our definitions is made to be able to handle ordered location , as shown in figure 7.3. Here, we would specify $HLoc(l) \Leftrightarrow' loc'\ is\ prefix\ of\ l$. Then, every location starting with loc would be trusted, and thus be considered honest. As an example, in Figure 7.3, we show how this mechanism could be used to model a program that reports on exactly one predefined output of a program with a verifier that gets the parameter from an outside source and checks the validity of the output.



```
P(i) =
  let x = report (prog(i)) in
    out(x)@('loc' + i)
```

```
V =
  in(i);
  in(x);
  if chkreport(x,'loc'+i) then
    Ok
  else Fail
```

Figure 7.3: Reporting of the one output of prog

Syntax

We define here the syntax of our calculus Slapic. We start from the syntax of the Sapic calculus, and add an operator that allows a process to report on a message according to its location. The syntax with the new rule and the location is shown in figure 7.4, with the additions highlighted in red. M and N shows the constructions of terms, and P and Q describe the localized processes.

Our two additions from Sapic are $P@l_P$ that allows to specify the location of a process and let $x = \text{report}(y)$ in P that allow report y with the location of the action.

Furthermore, we assume there is a function $\text{chkreport}(x, l)$ that represents checking that message x has been reported at location l.

We consider that processes are constructed such that if a process has an unspecified location, it will inherit the on of its parent. If there is no parent with a location, we use the default location 'l_X' for process X.

$\langle M, N \rangle$ ::= $x, y, z \in \mathcal{V}$		
$\mid$ (M)		*basic term*
$\mid$ $p \in PN$		*public name*
$\mid$ $n \in FN$		*fresh name*
$\mid$ $f(M_1, \cdots, M_n)$	(where $f \in \Sigma^n$)	*function application*
$\langle P, Q \rangle$ ::=		
0		*null process*
$\mid$ (P)		*plain process*
$\mid$ $P \mid Q$		*parallelisation*
$\mid$ $!P$		*replication*
$\mid$ $\nu n;\ P$		*binding of a fresh name*
$\mid$ $\text{out}(M, N);\ P$		*output of N on channel M*
$\mid$ $\text{in}(M, N);\ P$		*input*
$\mid$ $\text{if } M = N \text{ then } P \text{ else } Q$		*conditionnal*
$\mid$ $\text{event } F;\ P$		*event*
$\mid$ $\text{insert } M, N;\ P$		*set value of state M to N*
$\mid$ $\text{delete } M;\ P$		*delete state M*
$\mid$ $\text{lookup } M \text{ as } v \text{ in } P \text{ else } Q$		*read the state*
$\mid$ $\text{lock } M;\ P$		*lock a state*
$\mid$ $\text{unlock } M;\ P$		*unlock a state*
$\mid$ $[L] -[A]\rightarrow [R];\ P$	$(L, R, A \in \mathcal{F}^*)$	*a MSR rule*
$\mid$ $(P)@M$		*localised process*
$\mid$ $\text{let } x = \text{report}(y) \text{ in } P$		*reporting according to location*

Figure 7.4: Syntax with locations

Operational semantic

The operational semantic rewrites report(y) into report(l_P, y) (we're overloading the report symbol to represent both a function and an action) with l_P being the location of the process containing the report action. Then, with a chkreport function, a verifier should be able to verify the origin of a term. First, we need to define the context in which processes are run, then define an equationnal theory with a reporting scheme. Then, we define the attacker's power carefully so that it cannot produce a report with a trusted location but can still manipulate all the inputs and outputs. Finally, we define the operational semantic that rewrites the report action.

To give an intuition of how we expect our semantic to behave, let us consider the example of Figure 7.5. This process should run as follows:

- P runs report
- x takes the value report(l,'hello')
- v receives x=report(l,'hello')
- chkreport(report(l,'hello'),l) succeeds
- v return Ok

```
let p=
  (
    let x = report('hello') in
      out(x)
  )@l

let v =
  in(x);
  if chkreport(x,l) != fail then
    event Ok
  else
    event Fail

(p|v)
```

Figure 7.5: Reporting of an output

Of course, the attacker could pass for exemple report('myloc','hello) to the verifier, who would then raise the event Fail.

A more complex example is in Figure 7.6. Now v starts by creating a fresh value and send it to P who then makes the signing at the location ('loc'+i). Then, x would have the value sign('loc'+i,'hello') and v will return Ok only for a message coming from this instance of p. Here, p and v now run in parallel with an unbounded number of session. This means for example that an attacker can launch an instance of P with an i of its choice.

Context. In order to specify the operational semantic, we must first formally define the context in which the processes are ran because it is slightly different than in Sapic. Basically, the context contains variables populated by the processes at runtime. The context must contain for example a list of defined bound name, there is a also in the context a dictionary mapping a state to its value. Formally, we simply reuse the definition of Sapic, but with localized processes, i.e processes built on our new syntax.

Definition 25. *A localized configuration process is a 6-tuple* $(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L}k)$*, defined as a configuration process verifying :*

- $\mathcal{E} \subseteq FN$ *is the set of fresh names generated by the processes.*
- $\mathcal{S} : \mathcal{M} \to \mathcal{M}$ *is the set of states.*
- $\mathcal{S}^{MS} \in \mathcal{G}^{\#}$ *is the set of facts introduced by the syntax rule.*
- $\mathcal{P}$ *is a multi-set of localized process, where every set correspond to a process running in parallel in regard to the others.*
- σ *is the closed substitution that represent the knowledge exposed to the attacker.*
- $\mathcal{L}k \subseteq \mathcal{M}$ *is the set of current locks on states.*

```
let p=
  in(i);
  (
    let x = report('hello') in
      out(x)
  )@('loc'+i)

let v =
  new i; out(i); in(x);
  if chkreport(x,'loc'+i) != fail then
    event Ok
  else
    event Fail

!(p|v)
```

Figure 7.6: Reporting of an output

The part differing from classical pi-calculus is the fact that we store the processes in a multiset, following the choice of Sapic. Intuitively, this multiset represents the collection of processes running in parallel.

Equationnal theory. We now define the equationnal theory in figure 7.7 with a reporting scheme that allows someone to obtain the content of a report if it is reported with a known location. We use the classic definitions of an equationnal theory as a subterm convergent term rewriting system, where every equation is a a rule mapping a term to one of its subterm and $=$ is the symmetric, reflexive and transitive closure of $\rightarrow$.

$$\Sigma_H = \{\text{true}, \text{false}, \text{pair}, \text{fst}, \text{snd}, \text{report}, \text{chkreport}\}$$
$$\begin{aligned} \text{fst}(\text{pair}(x,y)) &= x \\ \text{snd}(\text{pair}(x,y)) &= y \\ \text{chkreport}(x, \text{report}(x,y)) &= y \end{aligned}$$

Figure 7.7: E_H definition

Attacker capabilities. The attacker controls all the communication between the processes and can manipulate terms. To define how he can manipulate them, we define attacker deductions rules. They are slightly different from Sapic in order to give the attacker the power to report on terms with unattested locations without letting him report with attested locations. The predicate Hloc that defines the set of trusted locations is therefore needed inside the attacker deduction rules and is then a parameter of the attacker capabilities. We show the rules in figure 7.8, showing the restriction on the DAppl and the Report rule.

For example, let us consider the case where $\text{HLoc}(l) \leftrightarrow l =' loc'$, the attacker can then obtain the term $\text{report}('cloc','hello')$:

$$\dfrac{\dfrac{'cloc' \in PN \quad 'cloc' \notin \emptyset}{\emptyset.\emptyset \vdash' cloc'}\,Name \qquad 'cloc' \neq' loc' \qquad \dfrac{'hello' \in PN \quad 'hello' \notin \emptyset}{\emptyset.\emptyset \vdash' hello'}\,Name}{\emptyset.\emptyset \vdash \text{report}('cloc','hello')}\,Report$$

$$\frac{a \in FN \cup PN \quad a \notin \tilde{n}}{\nu\tilde{n}.\sigma \vdash a}\,Name \qquad \frac{\nu\tilde{n}.\sigma \vdash t \quad t =_E t'}{\nu\tilde{n}.\sigma \vdash t'}\,DEq$$

$$\frac{x \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash x\sigma}\,DFrame \qquad \frac{\nu\tilde{n}.\sigma \vdash t_1 \;\cdots\; \nu\tilde{n}.\sigma \vdash t_n \quad f \in \Sigma^k/\{\text{report}\}}{\nu\tilde{n}.\sigma \vdash f(t_1, \cdots, t_n)}\,DAppl$$

$$\frac{\nu\tilde{n}.\sigma \vdash loc \quad \neg\,\text{HLoc}(loc) \quad \nu\tilde{n}.\sigma \vdash y}{\nu\tilde{n}.\sigma \vdash \text{report}(loc, y)}\,Report$$

Figure 7.8: attacker deductions rules

However, we can see that as 'loc' = 'loc', the attacker can not apply the Report rule for 'loc'. Also, he cannot use DAppl with $f = \text{report}$.

$$\emptyset.\emptyset \nvdash \text{report}('loc','\,hello')$$

Of course, if a process has an action $out(\text{report}('loc','\,hello'))$, then for example $\{^{\text{report}('loc','hello')}/_x\} \in \sigma$ and the attacker can deduce the reported term :

$$\frac{x \in \mathbf{D}(\sigma)}{\emptyset.\sigma \vdash x\{^{\text{report}('loc','hello')}/_x\} = \text{report}('loc','\,hello')}\,DFrame$$

We have the expected behaviour, a term reported with a trusted location can only be obtained if it was produced by a process executing at this location and the attacker can report on terms, but only with untrusted locations.

Semantic. Finally, we can define the operational semantic in Figure 7.9. We extend extend the Sapic semantics with the reporting operation shown at the end. Note that Sapic is embedded in our new semantic, hence our calculus is strictly more expressive.

The configuration processes can be difficult to read at first, but focusing on the multiset of processes, it is reasonably simple simple. If we have $P|Q$ inside a set, the action Par will simply spawn two processes in the multiset, P and Q. A replication will allow to spawn a new process in a separated set while keeping the replication available. The derivation bellow is a basic example, where we only write the needed part of the context for clarity.

$$\{(P|!Q)\} \rightarrow_{Par} \{P, !Q\} \rightarrow_{Rep} \{P, !Q, Q\}$$

The four rules concerning the communications are A-out,P-out,A-in and P-in. The A is for attacker and the P is for process, the intuition being that A-out correspond to an output made by the attacker and A-in to an input given by the attacker, while P-out is an output made by a process and P-in an input received from another process. A-out can only be made if the attacker can deduce the term he is producing. P-out must be made on a channel known by the attacker and will add to the output term to the attacker knowledge. However, two processes can communicate on a possibly hidden channel with P-in. For filtering inputs, we allow pattern matching, meaning that we only accept inputs of the specified form N. We have :

$$\{(in('chan', x)); out('chan', x)\} \xrightarrow{K('chan','hello')}_{A-in} \{out('chan','\,hello')\}$$

Classical Operations :

End	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{0\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L}k)$
Par	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(P \mid Q)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P, Q\}, \mathcal{R}, \mathcal{I}', \sigma, \mathcal{L}k)$
Rep	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{!P\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{!P\} \cup^{\#} \{P\}, \mathcal{I}, \sigma, \mathcal{L}k)$
New	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\nu a;\ P)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E} \cup \{a'\}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(P \left\{^{a'}/_{a}\right\})\}, \sigma, \mathcal{L}k)$ with a' fresh
$A - out$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L}k)$	$\xrightarrow{K(M)}$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L}k)$ if $\nu\mathcal{E}.\sigma \vdash M$
$P - out$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{out}(M, N);\ P)\}, \sigma, \mathcal{L}k)$	$\xrightarrow{K(M)}$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma \cup \{^{N}/_{x}\}, \mathcal{L}k)$ if x is fresh and $\nu\mathcal{E}.\sigma \vdash M$
$A - in$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{in}(M, N);\ P)\}, \sigma, \mathcal{L}k)$	$\xrightarrow{K(\langle M, N\tau \rangle)}$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\tau\}, \sigma, \mathcal{L}k)$ if $\exists \tau. \tau$ is grounding for N and $\nu\mathcal{E}.\sigma \vdash M$, $\nu\mathcal{E}.\sigma \vdash N\tau$
$P - in$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{out}(M, N); P),\ (\text{in}(M', N'); Q)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P, Q\tau\}, \sigma, \mathcal{L}k)$ if $M =_E M'$, $\exists \tau. N =_E N'\tau$ and τ grounding for N'
If	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{if } M = N \text{ then } P \text{ else } Q)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}k)$ if $M =_E N$
Else	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{if } M = N \text{ then } P \text{ else } Q)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{Q\}, \sigma, \mathcal{L}k)$ if $M \neq_E N$
Evt	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{event}(F); P)\}, \sigma, \mathcal{L}k)$	$\xrightarrow{F}$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}k)$

States operations :

Ins	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{insert } M, N;\ P)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}[M \mapsto N], \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}k)$
Del	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{delete } M; P)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}[M \mapsto \bot], \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}k)$
Rd	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\{^{V}/_{x}\}\}, \sigma, \mathcal{L}k)$ if $\mathcal{S}(N) =_E V$ is defined and $N =_E M$
Rdelse	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{Q\}, \sigma, \mathcal{L}k)$ if $\mathcal{S}(N)$ is undefined for all $N =_E M$
Lck	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{lock } M; P)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P_{id}\}, \sigma, \mathcal{L}k \cup \{M\})$ if $M \notin_E \mathcal{L}k$
Ulck	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{unlock } M; P)\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}k \setminus \{M' \mid M' =_E M\})$
Fact	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(l\text{–}[a]\!\rightarrow\! r;\ P)\}, \sigma, \mathcal{L}k)$	$\xrightarrow{a'}$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS} \setminus^{\#} lfacts(l') \cup^{\#} r', \mathcal{P} \cup^{\#} \{P\tau\}, \sigma, \mathcal{L}k)$ and τ grounding $l\text{–}[a]\!\rightarrow r$, $l'\text{–}[a']\!\rightarrow\! r' =_E (l\text{–}[a]\!\rightarrow r)\tau$ $lfacts(l') \subseteq^{\#} \mathcal{S}^{MS}$, $pfacts(l') \subseteq \mathcal{S}^{MS}$

Reporting operation :

Sign	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{(\text{let } x = \text{report}(y) \text{ in } P)@locP\}, \sigma, \mathcal{L}k)$	$\rightarrow$	$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\{^{\text{report}(locP, y)}/_{x}\}\}, \sigma, \mathcal{L}k)$

Figure 7.9: Operational semantic

If and Else works as expected, the reduction depending on whether the condition is valid or not.

To keep the semantic readable, we did not specify the propagation of the locations but it is encapsulated into every rules and it behave naturally. Par and Rep keeps the locations stables:

$$\{(P@'A'|(!Q@'B'))\} \rightarrow_{Par} \{P@'A', (!Q@'B')\} \rightarrow_{Rep} \{P@'A', (!Q@'B'), Q@'B'\}$$

Any sequential rules make the sub-process inherit the locations :

$$\{((in('chan', x)); out('chan', x))@'A'\} \xrightarrow{K('chan','hello')}_{A-in} \{out('chan',' hello')@'A'\}$$

If there is a new location specified, it overwrites the older one:

$$\{((in('chan', x)); out('chan', x)@'B')@'A'\} \xrightarrow{K('chan','hello')}_{A-in} \{out('chan',' hello')@'B'\}$$

$$\{(P@'A'|Q@'B')@'C'\} \rightarrow_{Par} \{P@'A', Q@'B'\}$$

Finally, a rule applying a substitution to a process, for example A-in with $P\tau$ also applies the substitution to the location:

$$\{(in('chan', x); out('chan', x))@('A' + x)\} \xrightarrow{K('chan','hello')}_{A-in} \{out('chan',' hello')@'Ahello'\}$$

The last rule is the our new report operation. When we encounter (let $x = \text{report}(y)$ in $P)@l_P$, we rewrite it to $P\{^{\text{report}(l_P,y)}/_x\}$, meaning we replace in P every occurence of x by the message y reported at the location of the process process, here l_P. Intuitively, this correspond to a function call to a TM, an SGX for example.

$$\begin{aligned}&\{\text{let } x = \text{report}('hello') \text{ in } out('chan', x)@'loc'\}, \sigma\\ &\quad \rightarrow_{Report} \{out(\text{report}('loc',' hello'))@'loc'\}, \sigma\\ &\quad \xrightarrow{K(\text{report}('loc','hello'))}_{P-out} \{\}, \sigma \cup \{^{\text{report}('loc','hello')}/_x\}\end{aligned}$$

We make sure that only a process located at $'loc'$ can produce terms of the form report$('loc', x)$. Reported terms can not be computed directly in the processes as we would loose the intuitive meaning of the function call to a TM, but instead only report according to the current location.

In this semantic, the attacker cannot produce terms reported with a trusted locations, the process at the trusted location is unalterable and can obtain through a black-box a term reported with its location, and is the only one who can do so. A verifier can easily check the location of a reported terms as locations are public. We now propose a first case study in order to argue the usability and the interest of this model.

Example: defining AC

We use slapic to model an attested computation protocol similar to the one shown in Chapter 4. The basic idea is that every computation will be reported by the IEE, thus allowing the user to verify the computations.

A provider P who, with a given primitive prog() that models the program that someone wants to run remotely, receives inputs and then computes and reports on the corresponding reuslts. In parallel there is a verifier V who, given the a series of inputs and outputs of the provider will be able to check that they have actually been produced by the provider. We model here a program whose outputs can depend on all previous inputs, hence prog will take 2 arguments,

```
        HLoc(l) <=> l = lP

let p@lP=
  [StoreP(oip)] --> []; in(ip);
  let x = report(oip,ip,prog(ip,oip)) in
    out(prog(ip,oip),x);
    [] --> [StoreP(ip,oip)]


let v =
  [StoreV(oip)] --> []; in(ip); in(o,ios);
  if  (oip,ip,o) = chkreport(lP,signedios) then
    [] --> [StoreV(ip,oip)])
  else event Fail;

let startpv =
    [] --> [StoreP(init)]; [] --> [StoreV(init)]

new init;
 ( (!startpv) || (!p) || (!v)  )
```

Figure 7.10: Attested computation implementation

the last input received and the list of all the previous inputs. Therefore, both the provider and the verifier will need to keep in their state the list of all inputs.
As we want the protocol accept an arbitrary number of inputs, we use a token to implement the iteration of a process. The idea is that thanks to the rule Fact, we can from nothing create a token at the initialization step. Indeed, Fact allows us to interact directly with the multiset rewriting rules and let us create new facts or use them as hypothesis. So, at the initialization, we create a new fact Token(state). Then P will at the start of its execution delete this fact, thus obtaining the knowledge state. At the end of its run P recreates the fact Token(state), where state is the previous state enriched with the new input received by P. Then, if we allow an infinite number of P to run in parallel, as there is only one token, we know there is only one running at a time. Moreover, every new running iteration acquire the knowledge of the previous one thanks to the state.
The resulting protocol modeled using Slapic is described in figure 7.10. We first define P, V and finally the main protocol that does the initialization of the tokens and runs the previous one.

Property. Now that we have the model, we need to express its desired property. Intuitively, the attested computation property states that if V accepts a sequence of input and outputs then this sequence is included the one of P. Indeed, it means that if V successfully accepts an input, it was computed by P. We construct V so that once an input is checked it raises an event containing this input, meanwhile P raises an event for any output it produces. We want to ensure that for any event raised by V, there is a matching previous event on P side.
If this holds for any trace and the attacker choosing all the inputs of the processes, we have the symbolic equivalent of the attested computation property defined in Chapter 4.
We remark that here, we do not always have unicity, several IEEs at location l_P could have made the computations. However, we will know that at least one IEE made all the computations corresponding to the outputs verified by V.

Definition 26. *An AC protocol is a protocol is defined by two iterating processes V and P. P sequentially accepts an input, produces an output, raising just before the output the event Poutput(ios) where ios is the list of all previous inputs and outputs of P. V sequentially accepts two inputs and either terminates, never replicating again or raises the event Voutput(ios) where ios is the list made of all the inputs received by V.*

We define the attested computation property as a trace property, based on the fact that we want every Voutput to be preceded by a Poutput. Intuitively it boils down to the fact that when V verifies a couple made of the input received by P and of the output made by V, there was a P that did receive previously this input and made this output.

Definition 27. *An AC protocol provides attested computation if for any trace t:*

$$\forall\ t_1\ t_2\ ios,\ \ t = t_1.Voutput(ios).t_2 \Rightarrow Poutput(ios) \in t_1$$

Encoding in Sapic

We want to encode protocols defined in Slapic into Sapic in order to take advantage of the Sapic tool. We need the encoded protocols to conserve their properties and behave the same way. However Sapix has a different syntax, different attacker deductions rules and a different semantic. Thus, we will need to find a way to encode the locations and the reporting scheme in Sapic.

Processes. The first step is that is encoding some operations of the semantic. So, while the operational semantic of Slapic uses (let $x = \text{report}(y)$ in P) a substitution for x, we define a rewriting on the processes that deletes this operation and rewrites in the rest of the process every occurrence of x by report$(locP, y)$. Intuitively, the reporting actions made at runtime in Slapic are executed before the run in Sapic by the rewriting. We can then run the process in a pi-calculus where the Report operation does not exist.

Definition 28. *The rewriting function rw is defined as:*

$$\begin{aligned}
rw((let\ x = \text{report}(y)\ \ in\ P)@l_P) &= rw(P\{^{\text{report}(l_P,y)}/_x)\}) \\
rw(a;P) &= a; rw(P) \\
rw(if\ M = N\ then\ P\ else\ Q) &= (if\ M = N\ then\ rw(P)\ else\ rw(Q)) \\
rw(P \| Q) &= rw(P) \| rw(Q) \\
rw(!P) &= !rw(P)
\end{aligned}$$

We provide an example of the application of the rewriting to a simple provider process P in figure 7.11.

`in(i);`
`let x= report(prog(i)) in`
`out(x)@lP`
	`in(i);`
	`out(report(lP,prog(i)))`

Figure 7.11: Original and rewritten process

As we are going to run the translated process within the set of deduction rules of Sapic, an attacker could easily forge any report$(locP, y)$. Therefore, we need an other way to report terms according to locations in an unforgeable way.

To block the attacker from creating forged report, we first create a secret key unknown from the attacker called sk_{loc}. Then, when in Slapic we had a term reported with a location, we will have in Sapic a location and term signed with sk_{loc}. If the process P@l_p signed 'hello', we perform the following rewriting.

$$\text{report}(l_p, 'hello') \rightarrow \text{sign}(l_p, 'hello', sk_{loc})$$

Then, as the attacker does not know sk_{loc}, he may not forge $\text{sign}(l_p, 'hello', sk_{loc})$. Intuitively, we replace a limitation of the attacker that was in the deduction rules by a limitation due to a secret key. This is inspired from actual implementation of IEEs. Technically, we ensure that sk_{loc} is secret from the attacker by using a fresh name, and we control exactly where it is used. Of course, if we modify the report function, we need to rewrite accordingly a chkreport function.

In order to do this rewriting, we simply define a rewriting system that rewrite every report and chkreport in Slapic to a sign and chksign with the appropriate key.

Definition 29. *Let us consider a rewriting on terms from E_H to E_G defined with :*

$$\text{report}(x, y) \to \text{sign}(x, y, sk_{loc})$$

$$\text{chkreport}(x, y) \to \text{chksign}(x, pk(sk_{loc}), y)$$

Then, define for any term $\rho(T)$ the normal form of T as usual.

We naturally extend ρ to sets of terms,substitutions, processes and configuration by replacing every terms that they contain by its normal form.

What is missing from this translation is the ability, for the attacker, to report with untrusted locations. Indeed as sk_{loc} is secret, the attacker cannot sign anything with it, however we need to emulate the Report attacker rule. In order to do so, we create an helper process which upon receiving a location and a term, signs the pair if the location is not in the set of attested locations, for example locP. The helper process can easily be described with :

$$Hlp = in(H, (x, y)); \text{if } (x \notin \mathcal{L}) \text{ then } out(H, sign(x, y, sk_{loc}))$$

We easily see that this mimics the Report attacker deduction rule of Slapic :

$$\frac{\nu\tilde{n}.\sigma \vdash loc \quad loc \notin \mathcal{L} \quad \nu\tilde{n}.\sigma \vdash y}{\nu\tilde{n}.\sigma \vdash \text{report}(loc, y)}\ \textit{Report}$$

Indeed, with the Report rule, the attacker can obtain $\text{report}(loc, y)$ if he can deduce loc and y and if loc is not an attested location. The helper can be called by the attacker with inputs x and y if the attacker can deduce them, and the helper outputs the signed message only if x is not an attested location. In the following example, the attacker signs a term with its custom location 'myloc' :

$$Hlp \xrightarrow{K(H,('myloc','hello'))}_{A-in} \rightarrow_{If} \xrightarrow{K(sign('myloc','hello',sk_{loc}))}_{P-out}$$

Finally, we can define the translation of a configuration process in Slapic to one is Sapic. We must remember to add sk_{loc} to the set of bounded terms, and we also, apply rw and ρ to the processes, and also apply ρ to the set of substitutions and states in order to be able to translate any configuration.

Definition 30. *Considering a localized configuration process $M = (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L})$, we define its translation into $\tilde{M} = (\mathcal{E}_{\tilde{M}}, \mathcal{S}_{\tilde{M}}, \mathcal{S}^{MS}_{\tilde{M}}, \mathcal{P}_{\tilde{M}}, \sigma_{\tilde{M}}, \mathcal{L}_{\tilde{M}})$.*

$$\textit{Finally}: \tilde{M} = (\mathcal{E}_M \cup sk_{loc}, \rho(\mathcal{S}), \mathcal{S}^{MS}, \rho(rw(\mathcal{P})) \cup^{\#} !Hlp, \rho(\sigma), \mathcal{L})$$

$$\textit{with } Hlp = in(H, (x, y)); \textit{if } (x \notin \mathcal{L}) \textit{ then } out(H, sign(x, y, sk_{loc})), H \in FN/(\mathcal{E}_M \cup sk_{loc})$$

Traces. With the previous translation, Slapic protocols can be encoded into Sapic. However, the trace properties in Slapic do not immediately have a meaning in Sapic. The first basic example is that if the property uses the report function, we need to transform it into the sign function with sk_{loc}. The second problem is that in Slapic, a trace could us the Report attacker rule to create a signed term while in Slapic the attacker would use the helper. This create differences between the traces of Sapic and Slapic because when running the helper the attacker adds to the trace inputs and outputs made on the helper channel :

$$Hlp \xrightarrow{K(H,(l,'hello'))}_{A-in} \rightarrow_{If} \xrightarrow{K(sign(l,'hello',sk_{loc}))}_{P-out}$$

Those K() events added to the traces should not influence the validity of the properties we are checking so we first need to take them off the trace before checking the validity of a property. Basically, we delete from the trace of a Slapic process every communication made on the channel H, which is the channel only used by the helper.

Definition 31. *For all trace τ of a configuration process M we define the projected trace without the action of the helper,*

$$\pi(\tau) = \tau/\{K(H)\} \cup \{K(H,N)|\ N\ \in T_{\Sigma_G}\}$$

Now that we have a translation of traces, we need to translate trace properties in order to check them. Intuitively, as we are just rewriting terms in a way that preserves equality, we just have to change the domain of the property and check it is true on the translated trace. However, this only works in one direction. Indeed, if any terms produced in Slapic as a translation in Sapic, there can be term in Sapic that does not come from a translation, an example is any $sign(x, y, z)$ where $z \neq sk_{loc}$. The attacker can produce those terms in Sapic just with a DAppl but those terms do not have any preimage by the translation.

This is not a major hurdle as we can define the set of terms that have no preimage by ρ, terms that we will consider as not well-formed. Those terms are not relevant when we consider the validity of an attack, because a well formed term does not actually exist in Slapic, so we can say that a property of Slapic is falsified in Sapic only by attacks where every terms are well formed.

We then define the property of well formedness of terms:

Definition 32.

$$WF(T) \Leftrightarrow T \in Im(\rho)$$

We naturally extend this definition to traces where every terms are well-formed.

Intuitively, a well formed trace is a trace of Sapic that can be obtained by translating a Slapic process.

Then, when we consider a property ϕ in Slapic the only pertinent attacks in Sapic on this property are the well-formed one. So, a not well-formed attack should not falsify a property. Also, if T is well-formed, we can naturally consider ρ^{-1} of the trace. This lead us to this definition of the translation on properties:

Definition 33. *For every property ϕ and every trace T of Slapic:*

$$\tilde{\phi}(T) \Leftrightarrow \neg WF(T) \vee \phi(\pi(\rho^{-1}(T)))$$

In the end, a translated property is true for every trace not well-formed, which means that only well formed attack invalidate the property.

Correctness

Now, we need to prove that our translation is correct. The goal is to show that if there is an attack in Slapic, there is an attack on the translation in Sapic. To do so, we will need to show that the attacker has the same deduction power in both world. If the attacker can obtain exactly the same terms in both the original and the translated process, an attack on the original should happen in the translated. We will then construct the Sapic trace from the Slapic trace, showing that every action can be executed in both settings.

Our translation works on several levels and on both processes and terms. Processes depends on terms, either when doing an "If M=N Then P Else Q" where we test $M =_{E_H} N$ or when doing an action like In(M,N) with the attacker input N' where we need the existence of a substitution τ such that $N =_{E_H} N'\tau$. So if we want our translated processes to run the same way as the original, the first thing to do is to be sure that we maintain the equality of terms between our translations.

We recall that in our case an equationnal theory is a subterm convergent term rewriting system, where every equation is a a rule mapping a term to one of its subterm and = is the symmetric, reflexive and transitive closure of $\to$. Moreover, as it is convergent, we have that $(M =_{E_H} N) \Leftrightarrow (!\exists U,\ M \to^* U \ and \ N \to^* U)$. So, we are first going to prove that our translation conserve the rewriting to normal form and then obtain the equality.

Lemma 1.

$$\forall M, N \ \in T_{\Sigma_H}, M \to^*_{E_H} N \Leftrightarrow \rho(M) \to^*_{E_G} \rho(N)$$

Sketch of proof: The basic idea is to consider the case in which the reduction is of length one. If we have a T of the the form $T = checksign(x, sign(x, y))$, then $T \to_{E_H} y$. We have then that

$$\rho(T) = checksign(\rho(x), pk(sk_{loc}), sign(\rho(x), \rho(y), sk_{loc}))$$

and we obtain $\rho(T) \to_{E_G} \rho(y)$. This works for the others rules of E_H so the lemma works for one reduction.

Then, as ρ preserve the sub-terms of a term, we can prove the result by induction on the length of the reduction and use the previous consideration to obtain the induction step.

■

Now we obtain our desired property, the stability of the equality :

Corollary 1.

$$\forall M, N \ \in T_{\Sigma_H}, M =_{E_H} N \Leftrightarrow \rho(M) =_{E_G} \rho(N)$$

Proof. With the previous lemma and the convergence of both equationnal theory, the result is instantaneous.

□

The next step is to prove the stability with respect to the substitution, so that for example on an input in Sapic, the equality with the application of the substitution is conserved in Slapic.

Lemma 2.

$$\forall N, N' \ \in T_{\Sigma_G}, \exists \sigma M =_{E_H} N\sigma \Leftrightarrow \rho(M) =_{E_G} \rho(N)\rho(\sigma)$$

Sketch of proof: With the corollary 1, the equality in E_H is equivalent to the equality in E_G for the translated terms. Then, we just have to prove that the application of ρ and σ can be done in any order :

$$\rho(N)\rho(\sigma) = \rho(N\sigma)$$

■

We now know that the equality between terms is stable with the translation, so we can take a further look at the processes. We want to show that a trace of a process in Slapic has some matching trace in Sapic. When we look at the operational semantic, we see that for the processes to run in a similar fashion before and after the translation, we need the attacker to be able to deduce the same set of terms. This is intuitively a very important thing, if the attacker can instantly deduce something in the Slapic processes, he must be able to do it in Sapic. This was the reason for implementing the helper. However, when in Slapic we can just do Report, in Sapic we need to run the helper and we might even need to run it several time. By running the helper, we obtain processes that are almost exactly the original, but that has a greater expressiveness than the original. Based on this difference, we are going to define a pre-order on the processes to be able to consider all the processes in which we ran the helper, but that all started from the same process.

For example, let us consider M such that (in abridged notation) $M = \{\mathcal{P}, !Hlp\}, \sigma$. Then, from M, the attacker could run the helper with either a name l1 and 'hello', or a different name l2 and 'hello'. We would obtain M1 and M2 :

$$M1 = \{\mathcal{P}, !Hlp\}, \sigma \cup \{^{sign(l1,'hello',sk_{loc})}/_{x}\}$$

$$M2 = \{\mathcal{P}, !Hlp\}, \sigma \cup \{^{sign(l2,'hello',sk_{loc})}/_{x}\}$$

Intuitively, M1 and M2 are not actually more expressive than M. However, they are in a sense greater than M, and we can characterize the set of configuration that came from M considering all the greater configurations.

Definition 34. *Let $\trianglelefteq$ be the pre-order such that for all configurations M and M', $M \trianglelefteq M'$ if:*

- $\sigma_M \subset \sigma_{M'}$
- $\forall \{^{y}/_{x}\} \in \sigma_{M'}/\sigma_M, \nu\mathcal{E}_{M'}.\sigma_{M'}/\{^{y}/_{x}\} \vdash y$
- $\forall \{^{y}/_{x}\} \in \sigma_{M'}/\sigma_M, \exists z, loc, loc \notin \mathcal{L}oc, y = sign(loc, z, sk_{loc})$

Considering again the previous example we have :

$$M \trianglelefteq M1 \text{ and } M \trianglelefteq M2$$

$$max(M1, M2) = \{\mathcal{P}, !Hlp\}, \sigma \cup \{^{sign(l2,'hello',sk_{loc})}/_{x}\} \cup \{^{sign(l1,'hello',sk_{loc})}/_{x}\}$$

Then, with this pre-order, all the configuration reachable from M by using only the helper can be obtained just by considering all the configurations greater than M.

Definition 35. *$\mathcal{C}(M)$ is the set of upper bounds of M with respect to $\trianglelefteq$.*

If M is a translated process, $\mathcal{C}(M)$ is the set of all the configuration we can reach only by running the helper.

We can then prove the first lemma which states that the translation is as expressive as the original (modulo ρ) .

Lemma 3. *For all M localized configuration processes :*

$$\nu\mathcal{E}_M.\sigma_M \vdash T \Leftrightarrow \exists N \in \mathcal{C}(\tilde{M}), \nu\mathcal{E}_N.\sigma_N \vdash \rho(T)$$

Sketch of proof: In both senses, the idea is to rewrite the proof, on one side deleting the SIGNloc and using the helper and DFrame on the other side, or deleting the DFrames on one side to use SIGNlocs on the other.

For the direct sense, we show that from the proof of a term T we can construct a proof of $\rho(T)$. We do it by induction on the height of the proof and showing that the last rule application can be also obtain in Slapic. For most rules application, we can almost apply them instantly. The main problem is Report that does not exist in Sapic. We have then to run the helper to add the signed term to the frame and then we can use DFrame to replace the Report.

For the indirect way, it is more complex. As in the previous case, we add $\sigma_M \subset sigma_N$. Before, it was useful because every Dframe application was instantly true but in this sense it means that many DFrames must be replaced by a Report. And worst, a DFrame could need to use several time the SIGNloc rule if the attacker signed a signature.

Finally, we first need to do an induction on the size of σ_N/σ_M to prove that any terms obtained through the helper and a DFrame can be obtained with Reports. Then, inside the induction step, as we can replace every DFrame, we can prove the result thanks to an induction on the height of the proof.

■

We now show that in the Slapic universe any element of the equivalence class can be obtained with a null projected trace, i.e just by using the helper.

Lemma 4. *For all M localized configuration processes :*

$$\forall N_1, N_2 \in \mathcal{C}(\tilde{M}), N_1 \trianglelefteq N_2, N_1 \overset{\tau}{\Rightarrow} N_2 \textit{ with } \pi(\tau) = \epsilon$$

Sketch of proof: Intuitively, if a configuration process is greater than an other, it means we can just run the helper several time in the smaller one to reach the greater one. We just need to take care of the fact that a signed term may contain an other, so we do a proof on the number of substitutions in N_2 and not in N_1. We then just prove that we can use the helper to construct a specific term and apply the induction hypothesis. ■

Theorem 12. *For all M, M' localized configuration processes :*

$$\forall N \in \mathcal{C}(\tilde{M}), M \overset{t}{\Rightarrow} M' \Rightarrow \exists N' \in \mathcal{C}(\tilde{M'}) N \overset{t'}{\Rightarrow} N' \textit{ with } \pi(t') = \rho(t)$$

Sketch of proof: We prove the result by induction on the length of the trace. Then, we just have to prove that the final step can be made. The semantic rule Sign does not exist any more but the rewriting has been done by the translation. Finally, with the previous results, we have that any term deduced by the attacker on the left can be obtained on the right, so most rules can be instantly applied. ■

Corollary 2.

$$if\ (M \stackrel{\tau}{\Rightarrow})\ then\ (\tilde{M} \stackrel{\tau'}{\Rightarrow}), \pi(\tau') = \rho(\tau)$$

Proof. Direct consequence of theorem 5.1 as $\tilde{M} \in \mathcal{C}(\tilde{M})$. □

Finally, we can have a look at the properties. What we want is that if there is a trace that falsifies the property in Slapic, there is a trace that falsifies the translation in Sapic.

Theorem 13. *For any localized configuration process M and property ϕ:*

$$\exists t, (M \stackrel{t}{\Rightarrow}) \wedge \neg\phi(t) \Rightarrow \exists t', (\tilde{M} \stackrel{t'}{\Rightarrow}) \wedge \neg\tilde{\phi}(t')$$

Sketch of proof: With the previous corollary, we can obtain the trace t'. Then, we consider the t' that is the shortest, because it is the one closest from t and we are then sure that the attacker has not built some term signed with an other key than sk_{loc}. Then we have $\pi(t') = \rho(t)$ and t' is well-formed, so if $\phi(t)$ is falsified $\tilde{\phi}(t')$ will also be falsified.

■

Finally, we obtained the correctness, so if a protocol in Slapic has an attack, we know that the translation has an attack. So if a property is verified in Sapic, it is verified in Slapic. In conclusion, if Tamarin proves that a property of the translation is true, we will know that the property is true in the Slapic protocol. We can now prove that our translation does not create false attack, i.e if Tamarin says that a translated property is falsified then the original property is also falsified.

Completeness

We now want to prove the converse, i.e if there is a trace that falsifies a translated property in a translated process, there is a trace that falsifies the property in the process.

As a translated configuration could have a trace that corresponds to a partial run of the helper, we first introduce the notion of normal form of a translated configuration, which corresponds to the completion of any partial helper run present in the multi-set process.

Definition 36. *The completion trace comp(M) of a process is such that for any process of the form $\{if\ (x \notin \mathcal{L})\ then\ out(H, sign(x, y, sk_{loc}))\} \in \mathcal{P}_M$, the corresponding $\Rightarrow_{If}\Rightarrow_{A-out}\in t$.*

With comp(M), we can run all partially used helper. We now just have to delete the unused ones that may have appeared with Rep and we obtain a process with only a clean helper $!Hlp$.

Definition 37. *We consider M' such that: $M \xrightarrow{comp(M)} M'$. The normal form nf(M) is then equal to M' except for the processes :*

$$\mathcal{P}_{nf(M)} = P_{M'}/\{Hlp\}$$

To be usable, the normalization needs to preserve the availability of any action not depending on the helper, hence the following lemma.

Lemma 5. *For any M, N* $\in \mathcal{C}(\tilde{M})$*, N' :*

$$\forall a, \pi(a) \neq \epsilon, (N \Rightarrow_a N') \Rightarrow (nf(N) \Rightarrow_a nf(N'))$$

Sketch of proof: As a is not an action of the helper, we show that completing all the helpers does not affect a, and a does not affect the completion trace. Intuitively, the normalization and the action a commute. ■

We can now show that if there is a well formed trace for a translated process, there is a corresponding trace for the original process.

Theorem 14. *For all M, N' configuration process :*

$$\forall N \in \mathcal{C}(\tilde{M}), N \overset{t}{\Rightarrow} N' \wedge WF(t) \Rightarrow \exists M', nf(N') \in \mathcal{C}(\tilde{M'}) \text{ and } M \xrightarrow{\pi(\rho^{-1}(t))} M'$$

Sketch of proof: Intuitively, if the trace is well-formed, it contains only terms that can be translated to Slapic. Then, we proceed as for the correctness, reasoning on the length of the trace and proving that we can do one step. The proof is not however as simple because several small details must be taken care of. For example, if we consider an action that can do N, for example executing a $out(x)$. Then, this action must exist in M, but it could be after a sign action ,$let\ z = \text{report}('h'); out(x)$, so we cannot just run it.

Being careful, we can however reproduce every action and obtain the desired result. ■

Corollary 3.

$$if\ (\tilde{M} \overset{t}{\Rightarrow}) \wedge WF(t)\ then\ M \xrightarrow{\pi(\rho^{-1}(t))}$$

Proof. Direct consequence of theorem 8.1 as $\tilde{M} \in \mathcal{C}(\tilde{M})$. □

Finally, we say that if there is a trace that falsifies the property in Sapic, there is a trace that falsifies the translation in Slapic.

Theorem 15. *For any localized configuration process M and property* ϕ*:*

$$\exists t, (\tilde{M} \overset{t}{\Rightarrow}) \wedge \neg\tilde{\phi}(t) \Rightarrow \exists t', (M \overset{t'}{\Rightarrow}) \wedge \neg\phi(t'),$$

Sketch of proof: If $\tilde{\phi}$ is falsified, it means that it is falsified by a well-formed trace. Then, the corresponding trace directly falsify ϕ ■

In conclusion, we now have a translation such that the validity of a property in the translated process is equivalent to the validity of the property in the original process. Therefore, we can now prove security properties in Slapic using Tamarin.

7.3 Experimental results

Now that we have a translation from Slapic to Sapic, we can use it to automatically prove security properties of protocols following the steps :

- Model a protocol and security properties in slapic
- Apply our translation to obtain a Sapic theory
- Use the Sapic tool to convert into MSR
- Launch the Tamarin prover on the MSR

After translating protocol 7.10, we can now ask Tamarin to prove the AC trace property in the Tamarin language:

```
lemma attested-computation:
    "All #t1 h . Voutput(h)@t1 ==> (Ex #t2 . Poutput(h)@t2 & t2<t1)"
```

The syntax of the trace properties is common to Slapic and Tamarin, # being use to declare time variables and @ to say that an event occurred at some precise time. By default, lemmas are proved for all possible traces of a process unless if ”exists-trace” is specified at the start of the lemma.

Tamarin returns that it is verified for all traces, thus proving that the protocol provides attested computation. As mentioned before, this is a basic form of attested computation, we now want to obtain more powerful properties like secrecy of the messages and unicity of the computation in the spirit of the SOC properties from Chapter 5.

Attested key exchange

To achieve secrecy which yields unicity, the idea is to encrypt all the inputs and outputs of the IEE with a shared secret key. We then need to use a key exchange. We could of course use NSL or any other protocol, but with the IEEs capabilities and the locations, we can start from a basic key exchange as in SOC:

$$\begin{array}{ccc} Bob & \xrightarrow{pk} & Alice \\ Bob & \xleftarrow{aenc(pk,key)} & Alice \end{array}$$

pk and *key* are fresh keys generated respectively by Alice and Bob. This protocol is trivially not a secure key exchange, however, if we encapsulate it into a secure attested computation protocol, we obtain a valid key exchange.

Basically, we consider that a verifier wants to share a secret key with an IEE. Then, the verifier remotely starts an IEE that contains *pk*. The IEE generate a fresh key, encrypt it with the verifier public key and then sign it with its location. Then, the verifier can check the signing and decode the shared key.

Here, we define a special set of trusted location. Indeed, we need to be able to differentiate all the IEEs that have different keys, so the location of an IEE will actually contain the public key of the verifier. We will give the location (’loc’,pk) to an IEE containing the public key pk, and we say that every location (’loc’,_) is a trusted location.

We finally obtain the protocol bellow :

```
hloc(loc) <=> Ex z. loc = <'loc',z>


// first the provider
let p=
    in(pk(skV)); // receives a public key
    !( // and initiate any number of IEE with the according location

        new shared_k; //generate the fresh key
        event SessionP(pk(skV),shared_k);
        let x = sign_loc aenc(shared_k,pk(skV)) in
            out(<aenc(shared_k,pk(skV)),x>);

     )@<'loc',pk(skV)>


// Run part of the NSL on the verifier side.
let v =
    new skV;
    event HonestP(pk(skV));
    out(pk(skV)); // public key sent

    in(<aenc(shared_k,pk(skV)),signed>); //reception of the signed key

    if  aenc(shared_k,pk(skV)) = checksign(<'loc',pk(skV)>,pk(skloc),signed) then
        event SessionV(pk(skV),shared_k); //the session is established



new init; ( (!p) || (!v) )
```

We can then prove the classical security properties of a key exchange. We want secrecy of the key obtained and the fact that if V accept a session there is a P that accepted it too. Both of the following lemmas were verified by Sapic.

```
lemma sessions:
      "All pka k #t1 . SessionV(pka,k)@t1 ==> Ex #t2. SessionP(pka,k)@t2 & t2<t1"

lemma secrecy[reuse]:
        "not (Ex pka k #t1 #t2 . SessionV(pka,k)@t1 & K(k)@t2)"
```

We now have a simple and valid key exchange.

Secure Outsourced Computation

Using the previous key exchange, the verifier and the producer can communicate with a channel encrypted using the shared key, thus obtaining a Secure Outsourced Computation protocol. This protocol is given below:

```
hloc(loc) <=> Ex z. loc = <'loc',z>


// first the provider
let p=
    in(pk(skV)); // receives a public key
    !( // and initiate any number of IEE with the according location

        new shared_k; //generate the fresh key
        event SessionP(pk(skV),shared_k);
        let x = sign_loc aenc(shared_k,pk(skV)) in
            out(<aenc(shared_k,pk(skV)),x>);
            [] --> [StoreP(init,shared_k)];
            !(   // start the computation part
            [StoreP(old_i,shared_key)] --> [];
            in(senc(ip,shared_key));  // for any encoded input
            event Poutput(senc(prog(ip,old_i), shared_key));
            out(senc(prog(ip,old_i), shared_key)); // compute the result and send it
            [] --> [StoreP(list(ip,old_i), shared_key)]
            )
     )@<'loc',pk(skV)>


let v =
    new skV;
    event HonestP(pk(skV));
    out(pk(skV)); // public key sent

    in(<aenc(shared_k,pk(skV)),signed>); //reception of the signed key

    if  aenc(shared_k,pk(skV)) = checksign(<'loc',pk(skV)>,pk(skloc),signed) then
        (
        event SessionV(pk(skV),shared_k); //the session is established
        [] --> [StoreV(init,shared_k)];
        !( //and we start sending input to the provider
            [StoreV(old_i,shared_key)] --> [];
            new ip;
            event Input(senc(ip,shared_key));
            out(senc(ip,shared_key));
            in(senc(prog(ip,old_i), shared_key));
            event Voutput(senc(prog(ip,old_i), shared_key));
            [] --> [StoreV(list(ip,old_i), shared_key)]
```

```
        )
        )


new init; ( (!p) || (!v) )
```

The lemmas corresponding to the secret attested computation were then proved by Tamarin.

```
lemma secrecy_computed:
        "not (
                Ex pka ip k oldi #t1 #t2 #t3 .
                 SessionV(pka,k)@t1
                & Input(senc(ip,k))@t2
                & K(prog(ip,oldi))@t3

        )"
lemma attested_computation[reuse]:
"All #t1 h . Voutput(h)@t1 ==> (Ex #t2 . Poutput(h)@t2 & t2<t1)"
```

Finally, combining attested computation with a simple protocol, we obtained a secure attested computation protocol.

Chapter 8

Conclusion

In this report we address the problem of verifiability in secure computation, i.e., to design secure computation protocols where parties can efficiently and independently check that the results they obtain from the computation are correct and/or prove to third parties that this is the case.

We have considered two classes of solutions. In the first part of the deliverable we have looked at two protocols that go beyond the state-of-the-art in extending secure multiparty computation techniques with verifiability guarantees. These solutions provide better performance and extend the range of functionalities that can be computed in practice. The proposed protocols provide *universal verifiability*, which means that correctness guarantees can be transferred to third parties—a key property in applications such as electronic voting and electronic cash. Furthermore, these protocols do not rely on special hardware assumptions, and can therefore be deployed today in real-world systems.

In the second part of the deliverable we explore a novel range of solutions that leverage emerging computational platforms that offer inbuilt software isolation and attestation guarantees. These new secure hardware architectures are not yet available to the general public but their general design is public. We have formalized the security guarantees that this new type of hardware aims to provide, designed protocols whose verifiability strongly relies on these designs and have shown how one can reason formally about their security.

While an experimental efficiency comparison between the two different approaches is currently not possible as actual hardware is not yet available, we note that these new protocols offer potentially significantly faster alternatives to secure outsourcing of computation and secure functional evaluation than the present day, software-only solutions. Furthermore, we have validated our specifications using formal verification techniques.

These results are essential stepping stones to ensure that the verifiable secure computation protocols that emerge in the future offer a comparable degree of security when compared to present day solutions, and to exactly pinpoint the trust/performance trade-offs that they entail.

Chapter 9

List of Abbreviations

FHE	Fully Homomorphic Encryption
MPC	Multi-party Computation
OT	Oblivious Transfer
UV	Universal Verifiability
VC	Verifiable Computation

Bibliography

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *SIGPLAN Not.*, 36(3):104–115, January 2001.

[2] Michel Abdalla, Jee Hea An, Mihir Bellare, and Chanathip Namprempre. From Identification to Signatures Via the Fiat-Shamir Transform: Necessary and Sufficient Conditions for Security and Forward-Security. *IEEE Transactions on Information Theory*, 54(8):3631–3646, 2008.

[3] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[4] Prabhanjan Ananth, Nishanth Chandran, Vipul Goyal, Bhavana Kanukurthi, and Rafail Ostrovsky. Achieving Privacy in Verifiable Computation with Multiple Servers - Without FHE and without Pre-processing. In *Proceedings of PKC*, 2014.

[5] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[6] Ross Anderson. Security engineering: A guide to building dependable distributed systems. 2001.

[7] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to sgx.

[8] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Hardware-based attested computation: Foundations, protocols, and proofs. In *2016 IEEE European Symposium on Security and Privacy, EuroSP 2016, Saarbrücken, Germany, March 21-24, 2016*, 2016.

[9] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly Auditable Secure Multi-Party Computation. In *Proceedings of SCN*, 2014.

[10] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of CCS '93*, pages 62–73. ACM, 1993.

[11] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266. ACM, 2008.

[12] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In *Proceedings of STOC*, 1988.

[13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Proceedings of CRYPTO*. 2013.

[14] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[15] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, 2013.

[16] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[17] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure Multiparty Computation Goes Live. In *Proceedings of FC '09*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[18] Ferdinand Brasser, Hiva Mahmoodi, Ahmad-Reza Sadeghi, Agnes Kiss, Michael Stausholm, Cem Kazan, Sander Siim, Manuel Barbosa, Bernardo Portela, Meilof Veeningen, Neils de Vreede, Antonio Zilli, and Stelvio Cimato. PRACTICE Deliverable D12.2: adversary, trust, communication and system models, 2015. Available from `http://www.practice-project.eu`.

[19] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 132–145. ACM, 2004.

[20] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In *Trusted Computing - Challenges and Applications, First International Conference on Trusted Computing and Trust in Information Technologies, Trust 2008, Villach, Austria, March 11-12, 2008, Proceedings*, volume 4968 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2008.

[21] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2*, 1(1):3–33, 2011.

[22] Ran Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, 13:2000, 1998.

[23] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 494–503. ACM, 2002.

[24] J. Cohen and M. Fischer. A Robust and Verifiable Cryptographically Secure Election Scheme. In *Proceedings of FOCS '85*, pages 372–382. IEEE, 1985.

[25] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org.

[26] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In *Proc. EUROCRYPT*. 2001.

[27] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. 3rd Theory of Cryptography Conference (TCC 2006)*, volume 3876 of *lncs*, pages 285–304, Berlin, 2006. Springer-Verlag.

[28] Ivan Damgård and Mads Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Proceedings of PKC '01*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[29] Ivan Damgård and Jesper Buus Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Proceedings of CRYPTO '03*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[30] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Proceedings of CRYPTO '12*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[31] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012.

[32] Sebastiaan de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming*. PhD thesis, Eindhoven University of Technology, 2012.

[33] Yvo Desmedt. Threshold cryptosystems. In *Proceedings of AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1993.

[34] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: The Secure Computation Application Programming Interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.

[35] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently Verifiable Computation on Encrypted Data. In *Proceedings of CCS*, 2014.

[36] Aurélien Francillon, Quan Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014.

[37] He Ge and Stephen R. Tate. A direct anonymous attestation scheme for embedded devices. In *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings*, volume 4450 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2007.

[38] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proceedings of CRYPTO*, 2010.

[39] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Proceedings of EUROCRYPT*. 2013.

[40] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In *Proceedings of PODC*, 1998.

[41] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[42] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 555–564. ACM, 2013.

[43] Shafi Goldwasser and Yael Tauman Kalai. On the (In)security of the Fiat-Shamir Paradigm. In *Proceedings of FOCS '03*, pages 102–113. IEEE Computer Society, 2003.

[44] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of STOC*, 2013.

[45] Shafi Goldwasser, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with an honest majority. In *Proc. of the Nienteenth Annual ACM STOC*, volume 87, pages 218–229, 1987.

[46] David Grawrock. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[47] Jens Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *Proceedings of ASIACRYPT*, 2010.

[48] Wilko Henecka, Ahmad-Reza Sadeghi, Thomas Schneider, Immo Wehrenberg, et al. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462. ACM, 2010.

[49] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[50] Yan Huang, Chih-hao Shen, David Evans, Jonathan Katz, and Abhi Shelat. Efficient secure computation with garbled circuits. In *Information Systems Security*, pages 28–48. Springer, 2011.

[51] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure Arithmetic Computation with No Honest Majority. In *Proceedings of TCC '09*, volume 5444 of *Lecture Notes in Computer Science*, pages 294–314. Springer, 2009.

[52] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 797–808, 2012.

[53] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 14(6):531–548, 2015.

[54] Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2003.

[55] Steve Kremer and Künnemann Robert. Automated analysis of security protocols with global state. Research report, March 2014.

[56] Robert Künnemann. Automated backward analysis of pkcs#11 v2.20. In Riccardo Focardi and Andrew C. Myers, editors, *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9036 of *Lecture Notes in Computer Science*, pages 219–238. Springer, 2015.

[57] Yehuda Lindell. *Composition of Secure Multi-Party Protocols: A Comprehensive Study*, volume 2815. Springer Science & Business Media, 2003.

[58] Shigeo Mitsunari. A Fast Implementation of the Optimal Ate Pairing over BN curve on Intel Haswell Processor. Cryptology ePrint Archive, Report 2013/362, 2013. `http://eprint.iacr.org/`.

[59] Payman Mohassel and Matthew K. Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Proceedings of PKC*, 2006.

[60] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[61] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of S&P*, 2013.

[62] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society, 2013.

[63] Benny Pinkas, Claudio Orlandi, Bogdan Warinschi, Dan Bogdanov, Thomas Schneider, Meilof Veeningen Michael Zohner, and Niels de Vreede. PRACTICE Deliverable D11.1: a theoretical evaluation of the existing secure computation solutions, 2013. Available from `http://www.practice-project.eu`.

[64] Mark D. Ryan and Ben Smyth. Applied pi calculus. 2011.

[65] K. Sako and J. Kilian. Receipt-Free Mix-Type Voting Scheme—A Practical Solution to the Implementation of a Voting Booth. In *Proceedings of EUROCRYPT '95*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer, 1995.

[66] Claus-Peter Schnorr. Efficient Identification and Signatures for Smart Cards. In *Proceedings of CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 1989.

[67] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Advances in Cryptology—CRYPTO'99*, pages 148–164. Springer, 1999.

[68] Berry Schoenmakers and Pim Tuyls. Practical Two-Party Computation Based on the Conditional Gate. In *Proceedings of ASIACRYPT '04*, volume 3329 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2004.

[69] Berry Schoenmakers and Pim Tuyls. Efficient Binary Conversion for Paillier Encrypted Values. In *Proceedings of EUROCRYPT*, 2006.

[70] Berry Schoenmakers and Meilof Veeningen. Guaranteeing Correctness in Privacy-Friendly Outsourcing by Certificate Validation. Cryptology ePrint Archive, Report 2015/339, 2015. `http://eprint.iacr.org/`.

[71] Berry Schoenmakers and Meilof Veeningen. Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems. In *Proceedings of ACNS*, 2015. `http://eprint.iacr.org/2015/058`.

[72] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-Friendly Outsourcing by Distributed Verifiable Computation. Cryptology ePrint Archive, Report 2015/480, 2015. `http://eprint.iacr.org/`.

[73] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54. IEEE Computer Society, 2015.

[74] Ben Smyth, Mark Ryan, and Liqun Chen. Direct anonymous attestation (DAA): ensuring privacy with corrupt administrators. In *Security and Privacy in Ad-hoc and Sensor Networks, 4th European Workshop, ESAS 2007, Cambridge, UK, July 2-3, 2007, Proceedings*, volume 4572 of *Lecture Notes in Computer Science*, pages 218–231. Springer, 2007.

[75] Dominique Unruh. Random Oracles and Auxiliary Input. In *Proceedings of CRYPTO '07*, volume 4622 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2007.

[76] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near-practicality. *Electronic Colloquium on Computational Complexity*, 20:165, 2013.

[77] Hoeteck Wee. Zero Knowledge in the Random Oracle Model, Revisited. In *Proceedings of ASIACRYPT '09*, volume 5912 of *Lecture Notes in Computer Science*, pages 417–434. Springer, 2009.

[78] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

[79] Bennet Yee. *Using secure coprocessors.* PhD thesis, Citeseer, 1994.