D12.3
Formal Verification Requirements

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 / D12.3 / 1.0
Activity and WP:	Activity 1 / WP 12
Due Date:	August 2015 - M22
Actual Submission Date:	31st August, 2015
Responsible Organisation:	INESC Porto
Editor:	Manuel Barbosa
Dissemination Level:	PU
Revision:	1
Abstract:	This report is the third deliverable of work package WP12 (*Applications Specifications*) in the PRACTICE project. The objective of this report is to document the formal verification requirements that have been identified at M22 by the WP12 team for the high-assurance secure computation framework outlined in D22.2. The context and motivation are provided by the application scenarios identified in earlier stages of WP12, and the presentation of the formally verified secure computation framework is framed by the technical and architectural background established by D21.1 and D21.2.
Keywords:	Formal Verification

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

Editor

Manuel Barbosa (INESC Porto)

Contributors (ordered according to beneficiary numbers)

Niklas Buescher (TUDA)
Peter Nordholt (ALX)
Dan Bogdanov (CYBER)
Roman Jagomägis (CYBER)
Jaak Randmets (CYBER)
José Bacelar Almeida (INESC Porto)
Bernardo Portela (INESC Porto)
Hugo Pacheco (INESC Porto)

Executive Summary

Validating practical solutions of multi-party computation as correct and secure is a non-trivial task. Practical systems rarely provide provide a formal guarantee that executable code displays the required security properties specified using high-level abstractions.

An important question in this context is therefore whether state-of-the-art technology in formal verification has reached a level of maturity that enables constructing implementations for which one can provide end-to-end formal security and correctness guarantees, connecting a high-level specification (e.g., extracted from a theoretical paper) and a low-level implementation that can be used in the real world.

As outlined in Deliverable D22.2, the high-assurance secure computation framework that is being developed in PRACTICE is meant as a proof-of-concept that the above question can indeed be answered in the affirmative.

This report is the third deliverable of work package WP12 (*Applications Specifications*). The objective of this report is to document the formal verification requirements that have been identified at M22 by the WP12 team and their mapping onto the high-assurance secure computation framework specified in D22.2.

The context and motivation are provided by the application scenarios identified in earlier stages of WP12, and the presentation of the formally verified secure computation framework is framed by the technical and architectural background established by D21.1 and D21.2.

Contents

1 Introduction **1**
1.1 Task description . . . 1
1.2 Background . . . 2
1.3 Structure of this document . . . 3

2 Verified secure computation framework **4**
2.1 Overview . . . 4
2.2 Verified secure computation engine . . . 7
2.3 Verified generation of computation descriptions . . . 8
2.4 Verification of high-level specifications . . . 9
2.5 Integration in the PRACTICE architecture . . . 9

3 Application and deployment scenarios **13**
3.1 Overview . . . 13
3.2 Yao's garbled circuits and Yao's SFE protocol . . . 14
3.3 Direct applications of the verified engine . . . 15
3.4 The verified engine as a sub-protocol . . . 17
3.5 Towards more challenging security models . . . 20

4 Formal verification requirements **22**
4.1 Secure computation engine . . . 22
4.2 Computation description generation . . . 32
4.3 High-level specification analysis . . . 36

5 Conclusion **40**

6 List of Abbreviations **41**

List of Figures

2.1 Formal verification landscape 6
2.2 A component view of the architecture. 11

3.1 Twin clouds model: trusted cloud behaves as a proxy between the client and the commodity cloud. 17
3.2 Arithmetic (left) and boolean (right) circuits. 19

4.1 Abstract Two-Party Protocol. 26
4.2 Security of a two-party protocol protocol. 28
4.3 Instantiating Two-Party Protocols into Abstract OT. 29
4.4 Abstract Garbling Scheme. 29
4.5 Security of garbling schemes. 31
4.6 Abstract SFE Construction. 32
4.7 Architecture of the Certified Circuit Generation Tool 35

Chapter 1

Introduction

1.1 Task description

This report is the third deliverable of work package WP12 (*Applications Specifications*) in the PRACTICE project. The objective of this report is to document the formal verification requirements that have been identified at M22 by the WP12 team and their mapping onto the high-assurance secure computation framework specified in D22.2. These requirements arise in the application scenarios identified in earlier stages of WP12. We recall the description of this deliverable in Annex I of the PRACTICE project proposal:

Task 12.3 – Formal verification

> Some of the target applications for the technology developed in PRACTICE will likely require the establishment of strict correctness and security guarantees for critical components, which go beyond those imposed by common practical scenarios. Such challenges can be addressed using formal verification techniques. The need for extra assurance often resides in system components where potential risks justify dedicating additional resources to increasing the level of trust in the specification and development processes. In other scenarios, such as outsourcing and certification, assurance needs to be transferred to a third party. Task 12.3 will analyse the requirements for formal verification in PRACTICE, taking into consideration the trade-off between potential gains and necessary effort. The results of this analysis will be fed into the work on formal methods carried out in WPs 13, 14 and 22.

Deliverable D12.3 – Formal verification requirements

> This deliverable is the outcome of task T12.3 and contains an analysis of the requirements for formal verification in PRACTICE.

1.2 Background

The overarching goal of PRACTICE is to provide efficient, correct and secure solutions for computations in the cloud. To define objectives, partners from industry observe their customers demands for applications in daily business, while the academic partners provide input for identifying the needs that can be answered by current technology and theoretical developments. During this project, several tools are being devised to support cloud applications using secure computation methods. Tools such as language/compiler-based application frameworks, databases, application servers, verification tools and deployment tools are being combined into an end-to-end secure computation software stack.

Validating practical solutions of multi-party computation as correct and secure is a non-trivial task. Security models imply abstracting real events and require detailed specification of adversary capabilities, which contributes towards models that tend to deviate from what is actually taking place in the real world. Additionally, even if the security analysis is adequate for a given application, the distance between a high-level specification of a protocol and its implementation in software and/or hardware is difficult to bridge when the goal is to provide a formal guarantee that executable code displays the required security properties. Practical systems rarely provide this level of formal assurance, constraining the users into placing trust in the correctness of their implementations.

An important question in this context is whether state-of-the-art technology in formal verification has reached a level of maturity that enables constructing implementations for which one can provide end-to-end formal security and correctness guarantees, connecting a high-level specification (e.g., extracted from a theoretical paper) and a low-level implementation that can be used in the real world. This deliverable describes the formal verification requirements for the high-assurance secure computation framework that is being developed in PRACTICE, and is meant as a proof-of-concept that the above question can indeed be answered in the affirmative. The requirements layed down in this deliverable for the different components of this framework will guide the development, implementation and verification work in the rest of the project.

Background for this deliverable is provided by work in WP12, but also in WP21 and WP22, as reported in the following deliverables:

- D12.1 – *Application scenarios and their requirements*: This deliverable was the outcome of task 12.1 and it identified a broad range of applications and scenarios for secure computation and their basic requirements. (M6)
- D21.1 – *Deployment models and trust analysis for secure computation services and applications*: This deliverable describes how different secure computation technologies can be deployed in information processing systems. Additionally, each description also includes the trust assumptions and relations between the stakeholders of the system. (M12)
- D22.2 – *Tools design document*: This deliverable outlined the design of the tools that will be developed in PRACTICE, including a specification of the tools that will comprise the high-assurance secure computation framework. (M12)
- D12.2 – *Adversary, trust, communication and system models*: This deliverable was the outcome of task 12.2 and it described in a systematic way the technical requirements for deploying secure computation protocols for the application scenarios identified in D12.1, based on the technological overview provided by D21.1. This includes the assumptions on the adversaries, trust, communication, and system models, etc. (M18)

- D21.2 – *Unified architecture for programmable secure computations*: A preliminary version of this deliverable describes the architecture of the PRACTICE software stack. (M18)

The formal verification requirements for the high-assurance secure computation framework that will be documented in this deliverable are a refinement of a subset of those identified for the application scenarios described in the aforementioned WP12 deliverables. We will describe how these requirements can be addressed using the formal verification framework outlined in D22.2, and frame this description within the broader PRACTICE framework deployment models and architecture described in D21.1 and D21.3.

1.3 Structure of this document

The structure for this document is as follows. Chapters 2 and 3 clarify the motivation and scope for the formal verification work that is being carried out in PRACTICE: Chapter 2 will recapitulate the overall objectives and structure of the formally verified secure computation framework sketched in D22.2, recasting it into the context of the architectural work that was subsequently carried out in WP21; Then, Chapter 3 will revisit the application scenarios previously studied in WP12 and examine the extent to which components in the formally verified framework can be deployed in applications that serve such scenarios (here we will refer back to the deployment models discussed in D21.1 and D12.2). Chapter 4 then discusses the critical correctness and security properties that must be guaranteed, providing a detailed description of the formal verification work that will be carried out at each component of the PRACTICE high-assurance secure computation framework. Finally, Chapter 5 wraps up the information presented in this deliverable and points out the directions for future work.

Chapter 2

Verified secure computation framework

In this chapter we recall the description of the formal verification tools that were introduced in D22.2, place them within the broader context of the PRACTICE architecture introduced in D21.2, and lay the ground for discussing the requirements that they should meet, in light of the application scenarios identified previously in WP12.

2.1 Overview

Software verification aims to increase the level of assurance on the functional and non-functional properties displayed by an implementation. Invariably, software verification implies the allocation of additional resources to the software development process, and hence is typically employed in scenarios where this extra effort is justified by a risk analysis of the costs of potential deviations from the original specification. For some application scenarios, this extra degree of assurance is imposed/validated by independent third parties (e.g., certification bodies), and it is therefore critical that software verification techniques enable the demonstration of the obtained guarantees to such independent bodies.

Depending on the application area, there is a wide range of properties that may justify formal verification. In the context of PRACTICE, we are concerned with security properties of cryptographic software implementations. Although this domain-specific setting has been the subject of considerable attention in the last few years, the complexity of the software stack that is required to enable secure computation constitutes a significant challenge for existing technology and offers various opportunities for advances beyond the state of the art. Figure 2.1 presents an abstract view of this software stack and illustrates the various technology levels that are at play:

- At the lowest level we find the *secure computation engine*. This designation was purposely chosen to encompass powerful virtual machines offering complex operations, such as those offered by Sharemind (see [23]), but also simpler APIs that permit executing low-level descriptions of computations in circuit form (i.e., sequences of simple operations such as Boolean or arithmetic gates), such as those provided by FRESCO. The purpose of the secure computation engine is to take *secure computation descriptions* and to execute them according to a well defined semantics. The formal verification challenge at this level is to formalize such semantics and to guarantee that, for any given secure computation description, the secure computation engine offers the correctness and security guarantees implied by the semantics.

- An intermediate technological level in the secure computation stack is required to bridge the upper and lower layers. In other words, a transformation step must be introduced to convert the high-level specifications into secure computation descriptions that can be executed in the underlying engine. The formal verification challenges at this level are akin to those arising in verified compilation: the goal is to guarantee that the target code generated from the high-level specification preserves the relevant functional and non-functional properties displayed by the latter.

- Finally, at the top level, the developer should be able to use a high-level, possibly domain-specific, language to describe the secure computations that are needed to meet the functional and security requirements of the application. The formal verification challenges at this level include being able to rigorously and naturally specify the relevant requirements, as well as providing support to formally verify that the high-level implementation meets the specified requirements. The solutions to these challenges are somewhat facilitated if the selected high-level language is a general-purpose programming language such as C, for which there exist a wide range of source code formal verification tools. However, the expressiveness of such languages is too limiting when the goal is to describe general multi-party interactions, as they were not designed for this purpose. A set of more interesting and complex challenges arises when the high-level language of choice is tailored for secure multi-party computation, as is the case for example of SecreC (see [30]).

It is important to emphasise that many concrete and alternative solutions exist to provide the functionality required at each of the technological layers described above, offering various trade-offs with respect to efficiency and performance. The PRACTICE project as a whole will be dealing with these technologies in breadth, covering a wide range of cryptographic protocols, software development techniques and application scenarios. Conversely, when it comes to formal verification, only a fraction of these solutions can be covered with the available resources. This leads to two possible strategies when designing a formal verification framework for PRACTICE: i. concentrating on a single layer in the stack, providing a more extensive coverage of the technologies in that layer; or ii. addressing all layers, but limiting the scope of the covered techniques at each layer. As described in Deliverable D22.2, we have opted for the latter option, as we believe that it is more useful to demonstrate the feasibility of providing end-to-end formal guarantees of correctness and security throughout all stages of the development process. In the following subsections, we will recall the general formal verification challenges that arise at each of the technological layers of this secure computation software stack, and conclude the chapter with an analysis of how the artifacts we will produce will be integrated into the overall PRACTICE architecture.

2.2 Verified secure computation engine

Developing verified implementations of secure computation engines is an ambitious goal, as their security hinges on rather sophisticated cryptographic constructions, whose security must itself be verified formally. We identify at least three significant challenges towards achieving this goal, which lie at the frontier of the capabilities offered by existing formal verification tools for cryptography.

The first challenge is to provide support for a broader scope of cryptographic proof techniques. EasyCrypt v0.2 and CryptoVerif (see [23] for a description of both) excel at modelling basic game-playing techniques such as equivalences, failure events, reductions or eager/lazy sampling. However, a number of standard concepts and techniques from provable security, such as hybrid arguments or

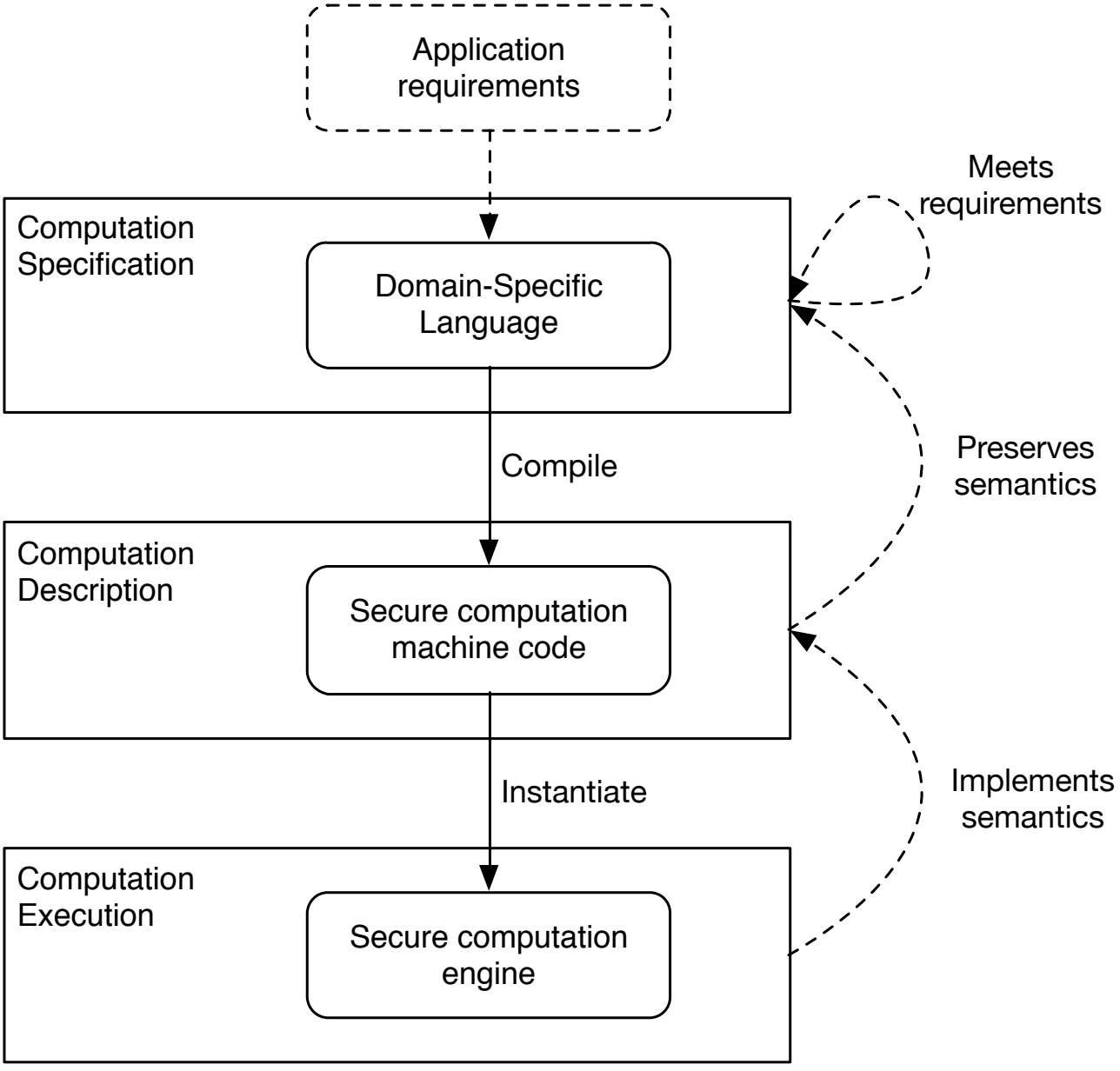

Figure 2.1: Formal verification landscape

rewinding, have thus far resisted formalization in these systems. In addition, comparatively little work has been done in formalising computational security proofs for simulation-based notions of security, which are omnipresent in secure computation protocols.

The second challenge is to ensure that the verification tools permit taming the complexity of secure multi-party computation. CryptoVerif and EasyCrypt v0.2 have primarily been used to verify cryptographic primitives, such as Full-Domain Hash signatures or OAEP, or standalone protocols, such as Kerberos or SSH. While these examples can be intricate to verify, there is a difference of scale with secure computation engines, which typically involve several layers of cryptographic constructions.

The third challenge is to enable a new workflow of specification, validation, and prototyping for building verified implementations of secure computation engines. To ensure that these implementations can have a practical impact, one must develop integrated environments in which cryptographers can specify cryptographic algorithms, perform lightweight validation (e.g., using a type-checking mechanism), formalize security proofs, and generate reasonably efficient implementations. What we envision here is a formal counterpart to existing prototyping systems for cryptography.

In PRACTICE we will adopt the latest version[1] (v1.0) of EasyCrypt, which provides a lightweight module system that is able to reflect the logical design of secure computation engines and the modular structure of their proofs. This version also incorporates a new code generation mechanism to produce reasonably efficient functional implementations of cryptographic algorithms from EasyCrypt descriptions. These additional mechanisms will enable the development of EasyCrypt libraries that permit tackling the above challenges.

Our primary goal will be to provide a fully functional verified implementation of a secure computation engine based on Yao's two-party secure function evaluation (SFE) protocol based on garbled circuits and oblivious transfer. This protocol allows two parties holding private inputs x_1 and x_2, to jointly evaluate any function $f(x_1, x_2)$ described as a Boolean circuit and learn its result, whilst being assured that no additional information about their respective inputs is revealed. This implementation will be added as one of the available secure computations engines offered by the PRACTICE software stack implemented in WP14.

The requirements for such a verified secure computation engine are described in Section 4.1.

2.3 Verified generation of computation descriptions

Most software verification tools and technologies, namely the ones described in the previous section, are deployed at source-code level in some high-level language. The guarantees obtained using these tools are therefore vulnerable to miscompilation: compiler bugs that cause wrong executable code to be silently generated from correct sources. Verified compilation addresses this problem by applying mechanized program verification methods to the compiler itself and proving semantics preservation, i.e., that the generated code behaves as prescribed by the semantics of the source program.

CompCert[2] is an optimizing, multi-pass compiler for a large subset of C that was programmed and proved to be semantics-preserving using the Coq proof assistant. This achievement demonstrated that formal verification of realistic compilers is within reach of today's proof technology, and generates unprecedented confidence in the compilation process and, indirectly, in source-level verification. Moreover, the efficiency of produced code is competitive with respect to state-of-art compilers such

[1] `http://www.easycrypt.info`
[2] `http://compcert.inria.fr`

as GCC. Verified compilation has attracted much attention from the research community, and several projects have adopted/adapted/extended CompCert to meet specific goals.

In PRACTICE we will do the same, taking advantage of the fact that CompCert handles ANSI C inputs, and construct a backend for this compiler that generates computation descriptions as Boolean circuits that can be executed in the verified secure computation engine described in the previous section.

The starting point for the implementation of this backend will be the bit-precise Boolean description generation mechanism of CBMC, described in [25], as well as previous work in incorporating information flow security analysis in the CompCert architecture. Our primary goal will be to offer a trade-off with respect to the tool described in Deliverable 22.2: correctness of the results will be guaranteed, at the cost of supporting a smaller subset of ANSI C and possibly generating less efficient computation descriptions.

The requirements for such a verified computation description generation tool are described in Section 4.2.

2.4 Verification of high-level specifications

As mentioned above, the application developer will use a high-level language to specify the computations that must be evaluated securely. The formal verification tools offered at this level should enable the programmer to express the intended output of the computations, but also the security properties that are imposed by the application (these will typically be expressed as information-flow restrictions). It should then be possible to check that, according to the semantics of the high-level language, the described computation indeed satisfies the application requirements. Furthermore, it is often the case that the underlying secure computation stack supports only a restricted class of the programs that can be expressed using the high-level language. This is particularly important when a general-purpose language such as C is chosen. In this case, it should also be possible to guarantee that the expressed program falls within the class of acceptable computations. In this context, we will consider two high-level languages: ANSI C and the SecreC language.

ANSI C is being considered as the input language for many circuit-based cryptographic primitives (see [25]). As a general purpose language, C is supported by advanced and mature formal verification tools that permit verifying a wide range of functional and non-functional properties (including security properties). In PRACTICE we will explore how some of these tools, such as FRAMA-C[3] and CBMC [25], can be used to tackle the application-specific properties that arise in the project's use cases, namely a set of safety properties that are sufficient to guarantee that we have sound secure computation specifications that can be fed into the computation description generator described in the previous section. More details on these requirements can be found in Section 4.2.

SecreC is a language designed specifically for the secure computation domain (see [30]). It inherits some of the syntactic features of C, but its type system permits explicitly distinguishing secret data from public data, and its semantics captures the concept of multiple parties providing inputs to and obtaining outputs from a computation, what may entail complex data flows between the various parties. At the moment, SecreC is not supported by a formal verification tool that permits specifying and verifying information flow restrictions imposed by an application, expect for the coarse, black and white distinction between public and (non-declassified) private data enforced by its type checker. The inception of such a tool will be considered in PRACTICE, and the requirements for high-level specification verification at this level are described in Section 4.3.

[3] `http://frama-c.com/`

2.5 Integration in the PRACTICE architecture

The successful completion of a verified secure computation engine, together with the formal verification of secure computation specifications and their verified compilation into circuit descriptions that we described earlier in this chapter, will yield the first end-to-end formally verified secure computation software stack. However, this stack is not meant to be used in isolation from the other technologies developed in PRACTICE. In this Section we clarify how these components fit into the wider picture of the PRACTICE architecture.

We recall in Figure 2.2 the development view of the PRACTICE general architecture as presented in the draft version of D21.2 released at M18. This diagram is targeted towards developers needing to implement an instance of the architecture, and hence it is ideal for our purposes. It contains a more in-depth view of the different components and how they interact. The formal verification framework should be seen as providing high-assurance instances of some of the components that are shown in the diagram, more precisely within the DAGGER package: languages for describing computations, a compiler for translating these descriptions into computation specifications, and a protocol suite in which to securely execute these computation descriptions.[4]

Clearly, our work on the verification of high-level computation specifications and verified generation of computation descriptions will yield additional instances of the *Secure Language & Compiler* component. The integration of these components into the remainder of the PRACTICE architecture can be achieved by ensuring that the resulting *Secure Computation Specifications* are represented in a way compatible with the underlying DAGGER instance.

An important observation that emerges from Figure 2.2 is that a complete secure computation application developed using the SPEAR architecture will require a full instance of the DAGGER package, which will be structured around a secure computation engine/virtual machine that is integrated with the rest of the framework via three external interfaces: i. the cloud infrastructure interface, which it will use to obtain computational resources; ii. the secure storage interface, which it will use to keep secure state; and iii. the secure service interface, through which it provides secure computation functionality to the application logic.

The real-world use of the formally verified secure computation engine introduced in this Chapter therefore requires that we integrate it into (at least) one instance of the DAGGER package. A closer look at the component diagram in Figure 2.2 permits identifying exactly the interfaces that will need to be implemented in order to achieve this integration: i. the interface that permits the secure computation engine to load a verified secure computation specification and ii. the Protocol API that allows the secure computation engine to load and run the verified protocol suite. Our goal, as will be detailed next, is to implement such interfaces for at least one of the instances of the DAGGER package that will be developed in PRACTICE, namely FRESCO as we will detail below.

We observe that the low-level protocol layer in the PRACTICE architecture will be refined within WP14, and specified in D14.1 (M24). The end goal is that if one implements a protocol following the refined architecture, it should be portable between the various DAGGER instances following the PRACTICE architecture. The integration of the formal verified protocol suite will take place within WP14, and it will follow this refined architecture.

[4]This is indicated in the diagram by presenting formal verification as an activity that will target the secure computation engine and specification levels, as well as the secure language and compiler component.

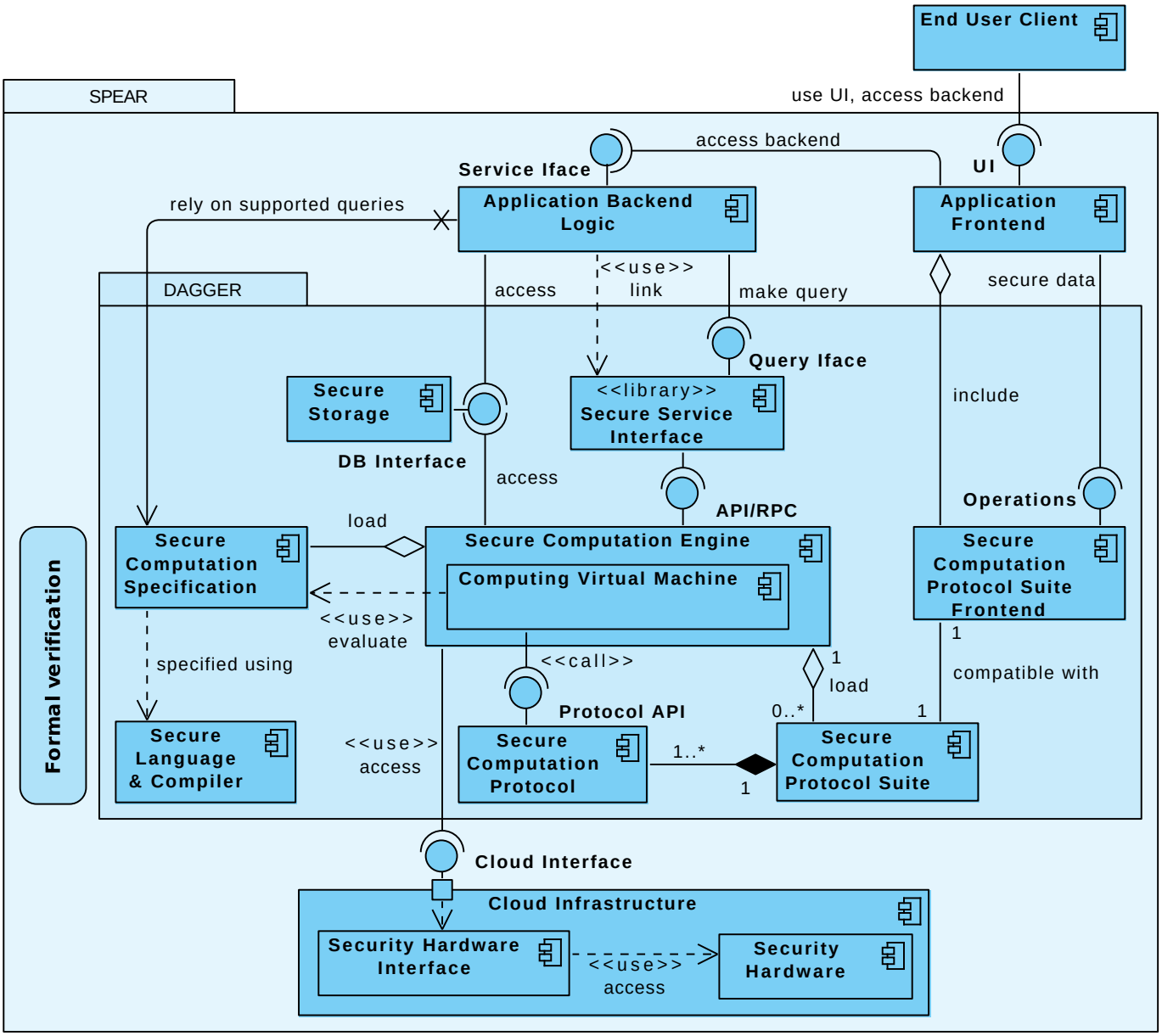

Figure 2.2: A component view of the architecture.

Integration in FRESCO

The FRESCO framework is intended for two different groups of developers, namely the application and protocol developers.

A *protocol developer* will have to implement an interface for basic multi-party computation operations such as inputting and outputting values, as well as a few basic numeric or binary operations such as *add*, *multiply*, *and* and *xor*. Building on these simple operations, FRESCO provides generic versions of more advanced operations such as *comparison*, *equality* or even *linear program solvers*. These advanced operations will inherit various properties such as performance and security assumptions from the basic protocol. If a protocol has nice properties allowing for optimizing advanced operations such as sorting or similar, the generic FRESCO operation can be substituted by a protocol-specific operation.

The *application programmer* is offered a library of (secure) operations, operating on a combination of secret or open primitive types such as integers or booleans. These are implemented as a set

of Java objects, namely SInt, SBool, OInt and OBool, representing secret and publicly observable integers and booleans. These objects are used in conjunction with a number of Java operations, to construct a circuit representing the computation the application is intended to perform. In addition to constructing the application circuit itself, the application programmer also chooses a runtime (i.e. set of protocols) that should be used for executing the application. The choice of runtime is, however, independent of the circuit, e.g., the same 2-party FRESCO application circuit can be executed using both a Yao-based and a shared secret based implementation.

During evaluation of the application circuit, the advanced operations are broken down into a series of basic operations supported by the selected runtime, e.g., *add* and *mult* in the case of numeric protocols and *and* and *xor* in the case of binary protocols. These basic operations can then be executed using the runtime.

Network communication does not have to be reimplemented for each runtime protocol suite. FRESCO offers a generic interface for sending and receiving byte arrays. Each runtime just takes an instance of this interface and uses it for communication with the other parties in whatever way the protocol dictates. However, for convenience, the runtime is likely to provide some wrapper around this interface to convert values of a type natural to the runtime to byte arrays and the other way around.

The formally verified secure computation engine that will be developed in PRACTICE can be integrated into FRESCO in two different ways. The first option is to implement a verified FRESCO run-time protocol suite for evaluating Boolean circuits by simply encapsulating the verified secure computation engine around an appropriate wrapper. In this case, and as was mentioned above, the verified circuit compiler should be instrumented to generate FRESCO-compatible representations of the produced Boolean circuits. Another, more modular approach, would be to use the verified secure computation framework to produce verified implementations of specific operations that could be seen as higher level *gates* by existing FRESCO runtimes. In both cases, the underlying FRESCO communications infrastructure would be used to implement the necessary interaction.

Chapter 3

Application and deployment scenarios

3.1 Overview

In this chapter, we will revisit the different application scenarios that have been identified in earlier stages of WP12, and the PRACTICE architecture that is being developed in WP21. Our goal is to highlight the possible usages that the formally verified secure computation framework can have and, drawing from these usages, feed the specification of formal verification requirements that will be presented in the next chapter. For this reason, we will not discuss how our work relates to protocols that are not yet practical, such as those based on fully homomorphic encryption, nor will we cover specialized protocols that are only applicable to a very small domain of problems.

Since the higher-level components of our formally verified framwork are totally generic, in the sense that they simply allow developers to reason about and produce arbitrary specifications following a given representation, the set of application scenarios that we can address will be essentially that which can be realized using the underlying verified secure computation engine.

In this direction, we will first consider scenarios in which our implementation of secure function evaluation based on Yao's protocol can be used directly. We will consider multi-party computation approaches that either prioritize performance (as is the case of the framework of Huang et al. [26]), or deployment usability (as is the case with the Twin Clouds architecture [13]). We can then evaluate the guarantees offered by the resulting solutions from the perspective of the PRACTICE application scenarios.

Subsequently, we consider modern efficient secure function evaluation protocols that can be obtained by combining several basic techniques, such as garbled circuits and (partially) homomorphic encryption, and employ them for settings in which they can excel. We take a look at these approaches and detail how we could apply our verified implementation of secure function evaluation as a sub-protocol in such frameworks. Finally, we evaluate generic approaches to boost the security of our verified protocol to deal with stronger adversarial models. We note that the latter use cases for our protocol will result in a only partially verified implementation, but they significantly expand the range of application scenarios in which it can be used.

3.2 Yao's garbled circuits and Yao's SFE protocol

Excellent descriptions of Yao's contributions can be found in [40, 8]. Yao's idea of garbling the circuit computing f consists informally of: i. expressing such a circuit as a set of truth tables (one for each gate) and meta information describing the wiring between gates; ii. replacing the actual boolean values in the truth tables with random cryptographic keys, called *labels*; and iii. translating the wiring relations using a system of *locks*: truth tables are encrypted one label at a time so that, for each possible combination of the input wires, the corresponding labels are used as encryption keys that lock the label for the correct boolean value at the output of that gate. Then, given a garbled circuit for f and a set of labels representing (unknown) values for the input wires encoding x_1 and x_2, one can obliviously evaluate the circuit by sequentially computing one gate after another: given the labels of the input wires to a gate, only one entry in the corresponding truth table will be decryptable, revealing the label of the output wire. The output of the circuit will comprise the labels at the output wires of the output gates.

To build a secure function evaluation protocol between two semi-honest[1] parties, one can use Yao's garbled circuits as follows. Bob (holding x_2) garbles the circuit and provides this to Alice (holding x_1) along with: i. the label assignment for the input wires corresponding to x_2, and ii. all the information required to decode the boolean values of the output wires. In order for Alice to be able to evaluate the circuit, she should be able to obtain the correct label assignment for x_1. Obviously, Alice cannot reveal x_1 to Bob, as this would totally destroy the goal of secure function evaluation. Furthermore, Bob cannot reveal information that would allow Alice to encode anything other than x_1, since this would reveal more than $f(x_1, x_2)$. To solve this problem, Yao proposed the use of an *oblivious transfer* (OT) protocol. This is a (lower-level) secure function evaluation protocol for a very simple functionality that allows Alice to obtain the labels that encode x_1 from Bob, without revealing anything about x_1 and learning nothing more than the labels she requires.[2] The protocol is completed with Alice evaluating the circuit, recovering the output, and providing the output value back to Bob.[3] The combined security of the garbled circuit technique and the OT protocol guarantee that f can be securely evaluated in this manner.

This classical setting assumes the existence of only two parties (Alice and Bob), playing the roles of input, computation and output parties. This suffices for some application scenarios, as is with PRACTICE's Privacy Preserving Genome-Wide Association Studies Between Biobanks and Location Sharing with Nearby Contacts use cases [17]. However, this protocol can also be extended for multi-party applications via secret sharing techniques such as those referred in [7], where an arbitrary number of input players provide their confidential inputs and an arbitrary number of output players receive the outputs of the secure computation. For this, each input player secret-shares its inputs among the two computation servers (which are assumed not to collude). Then, the two computation servers run the secure computation protocol on the input shares, during which they learn no intermediate information. Finally, they send the output shares to the output players, who can reconstruct the outputs.

As mentioned in the beginning of this chapter, we are interested in integrating a verified implementation of Yao's secure function evaluation protocol as a part of the PRACTICE architecture. This can be employed as a secure computation protocol (recall Figure 2.2), to be used either exclusively, or as

[1] Semi-honest parties refers to participants of a multi-party computation protocol that, despite not being necessarily honest, will not deviate from the protocol behavior even in the case of adversarial corruption.

[2] Luckily, efficient OT protocols exist that can be used for this specific purpose, thereby eliminating what could otherwise be a circular dependency.

[3] This is a simplified view of Yao's protocol. It suffices because we are dealing with semi-honest adversaries assumed to follow the protocol. Stronger adversarial models will be considered later on.

a sub-protocol of some broader secure computation protocol suite. In order to understand the reach of Yao's protocol, what follows is a descriptive analysis of frameworks that take on this approach to perform secure computations. Additionally, this gives us the opportunity to verify the usefulness of this implementation with respect to the security requirements of application scenarios associated with the PRACTICE project.

3.3 Direct applications of the verified engine

We now describe two relevant research contributions, namely the Huang et al. [26] framework and the Twin Clouds [13] architecture, that enable our direct approach. This discussion will expand on their strengths and limitations, taking into consideration performance, security model and broadness of scope, given the proposed use cases from the PRACTICE project. The verified secure computation framework that will be developed in PRACTICE will implement the same secure computation technology, and therefore will display similar characteristics.

The Huang et al. two-party computation framework [26] is a tool developed by Might Be Evil: Privacy-Preserving Computing[4] for secure two-party computation. The tool produces protocols for computing any function f, starting from a boolean representation of f. This approach was motivated by the perspective that generic secure computation is underestimated, aiming to salvage Yao's garbled circuits protocol for many applications where they are capable of outperforming specific-purpose protocols. Associated contributions encompass an improvement on efficiency and scalability of garbled circuit execution and a framework with flexibility for circuit optimizations. This framework assumes the semi-honest adversarial model, but if an oblivious transfer protocol with security against malicious adversaries is used, the implementations can provide security properties against malicious evaluators and generators (the interested reader is referred to the original paper for the details).

The two players start the protocol by instantiating the circuit structure, which is known to both of them. Afterwards, garbled gates are transmitted by the generator to the evaluator as they are produced, and these must be systematically associated with the corresponding circuit gates. The evaluator automatically triggers the evaluation execution when all necessary inputs are ready, and immediately discards the gates after completing the evaluation. Since the order of generation and evaluation does not depend on the private inputs, no synchrony assumptions on the communication channels are made.

The major strengths of this approach are centered in the enhancement of garbled circuit implementation. Instead of fully generating the circuits before transmission, the circuit generation is pipelined, meaning that the gates are sent as soon as they are generated. Gate evaluation is also performed as soon as possible, and results in the deletion of the gate from memory. Since there is no need to wait for the entire circuit to be generated, and since the gates are erased as soon as they are evaluated, this approach reduces the memory space occupied by the circuits, improving the scalability of the process to an unlimited number of gates. Additionally, the framework supports parametrization of input sizes, in order to generate more compact circuits. Other optimizations are also employed in the system, making use of techniques such as "free XOR" [35], garbled-row reduction [46] and oblivious transfer extension [29] to improve efficiency.

This approach can be directly applied in scenarios in which the trust level of all computing parties is assumed to be semi-honest or higher. In particular, Tax Fraud Detection, Privacy Preserving

[4] `http://www.mightbeevil.org/`

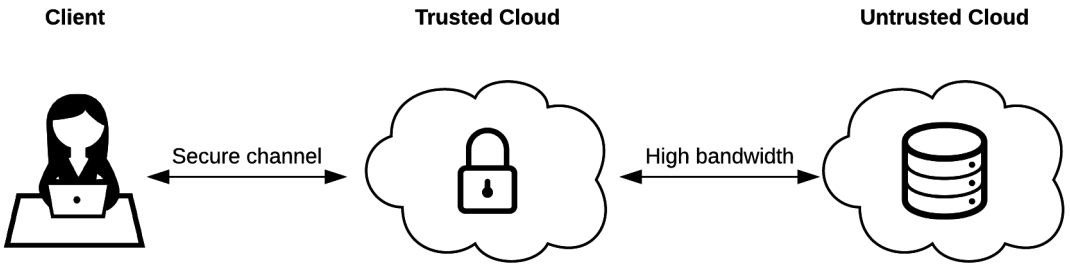

Figure 3.1: Twin clouds model: trusted cloud behaves as a proxy between the client and the commodity cloud.

Genome-Wide Association Studies Between Biobanks and Key Management [17] match this criteria, and could benefit from the framework.

Twin clouds [13] is an architecture for secure outsourcing of data and arbitrary computations to an untrusted cloud. In this approach, the user communicates with a trusted cloud that is responsible for encrypting and verifying stored data and operations performed in the untrusted commodity cloud. The computations are split such that the trusted cloud is mostly used for security-critical operations in the less time-critical setup phase, whereas queries to the outsourced data are processed in parallel by the faster cloud on encrypted data. In this model, a client wants to outsource data and computations into some cloud service in a secure way. This sensitive data must be confidentiality- and integrity-protected (assuming that the provider may be malicious), and the correctness of outsourced computations must be verifiable by the client. In order to satisfy these security requirements, the model (as depicted in Figure 3.1) uses a trusted cloud that provides a resource-restricted execution environment and infrastructure that is fully trusted by the client, as well as an interface for secure storage and computations to be performed in the commodity untrusted cloud. The client communicates with the trusted cloud via a low-bandwidth secure channel, whereas the two-clouds are connected via an insecure high-bandwidth channel.

Essentially, the trusted cloud serves as a transparent interface that enables security properties to the untrusted storage, while the architecture enables the computation to be performed in the more resourceful untrusted cloud. The scenario presented depicts only a single client accessing the outsourced data, but that setting can be extended to multiple programs and clients. At the high-level, the protocol works as follows: the client will begin by providing data D and programs P to the trusted cloud, which will be encrypted by the trusted cloud, and securely stored in the commodity cloud. Additionally, the trusted cloud will also re-encrypt D into a garbled equivalent $\tilde{D}$, and generate garbled circuits $\tilde{C}$ for P, that will also be stored in the untrusted cloud. From thereon, the client can issue queries q, that will be encrypted and sent to the untrusted cloud, that is responsible for computing the garbled result $\tilde{r} = \tilde{C}(\tilde{q}, \tilde{D})$ under encrypted values. The $\tilde{r}$ received from the commodity cloud is then verified by the trusted cloud to validate garbled circuit correctness and, if verification succeeds, the query result is decrypted and returned to the client. From the client's perspective, one has simply made a query on data that was previously sent to the cloud, and has received an appropriate result.

Observe that the role of the trusted cloud may be performed by some trusted authority, since the associated operations do not entail processor-intensive work, but rather the handling of sensitive information. From the selected application scenarios of PRACTICE, Platform for Auctions, Tax Fraud Detection, Joint Statistical Analysis Between State Entities, Privacy Preserving Genome-Wide Association Studied Between Biobanks, Privacy Preserving Personal Genome Analyses and Studies and Privacy Preserving Satellite Collision Detection [17] allow for some regulator entity

with these properties. Alternatively, when such level of trust cannot be placed in a participant, the trusted cloud can take the form of a cluster of virtualized cryptographic co-processors (such as the IBM Cryptographic Coprocessor 4764 [52, 3] or other Hardware Security Modules such as TPM or Software Guard Extensions [28]).

3.4 The verified engine as a sub-protocol

For a period of time, two different approaches to secure two-party computation co-existed, and were seen as alternatives to each other. One of them was based on garbled circuits [55], which was the building block of the protocols described in Section 3.2. Alternatively, secure function evaluation could be done using (somewhat) homomorphic encryption (HE), where one party sends its encrypted inputs to the other party, who then computes the intended function under encryption using the homomorphic properties of the underlying cryptosystem, and sends back the encrypted result. Popular examples are the additively homomorphic cryptosystems of Pailier [43] and Damgard-Jurik [15], as well as other fully homomorphic schemes [19, 54, 51]. Both approaches have their respective advantages and disadvantages. For example, garbled circuits require the transmission of a verbose circuit description, but allows pre-computation of almost all expensive operations; whereas most homomorphic encryption-based protocols require relatively expensive public-key operations in the online phase but can result in a smaller overall computation complexity. In this section, we consider the possibility of employing our verified secure computation framework as a sub-protocol for a broader deployment, and detail two solutions that leverage the combination of different approaches to secure computation to maximize performance.

TASTY (Tool for Automating Secure Two-partY computations) is a tool suite addressing secure two-party computation in the semi-honest adversary model [24]. TASTY incorporates a compiler and a high-level domain-specific description language named TASTYL. TASTY's goal is to prioritize the online phase for evaluating a function f, assuming a prior setup phase for the execution. To achieve the desired speedup, the tool suite allows the compilation and evaluation of functions using garbled circuits *and* (additively) homomorphic encryption schemes, e.g., Paillier, at the same time. Furthermore, TASTY can be extended to employ other primitives, such as fully-homomorphic encryption [20]. This hybrid approach combines garbled circuits and homomorphic encryption, as described in [34].[5] Specifically, for each function f to be evaluated, the programmer can define which parts of f should be computed by garbled circuits, and which parts should be computed by homomorphic encryption. One fundamental difference between such primitives relies on the concrete form of circuit representation that they enable. On one hand, arithmetic circuits (left side of Figure 3.2) are a poor solution to encode non-linear functions like comparisons, which includes Yao's millionaire problem and XORs [35]. On the other hand, boolean circuits (right side of Figure 3.2) are a poor solution to express arithmetic functions like multiplication. The combination of both approaches allows for a computation speedup [24, 32].

Prior to TASTY, compilers for secure function evaluation were restricted to compiling protocols using either garbled circuits or homomorphic encryption schemes. More recently, integrated protocols have been developed that permit obtaining combined benefits by using the most efficient of the above approaches for each sub-task in a larger protocol. Indeed, the use of secure and efficient composition of sub-protocols in several privacy-preserving applications was shown to yield important performance improvements [11, 12, 5, 49].

[5] A preliminary version appeared as [33]

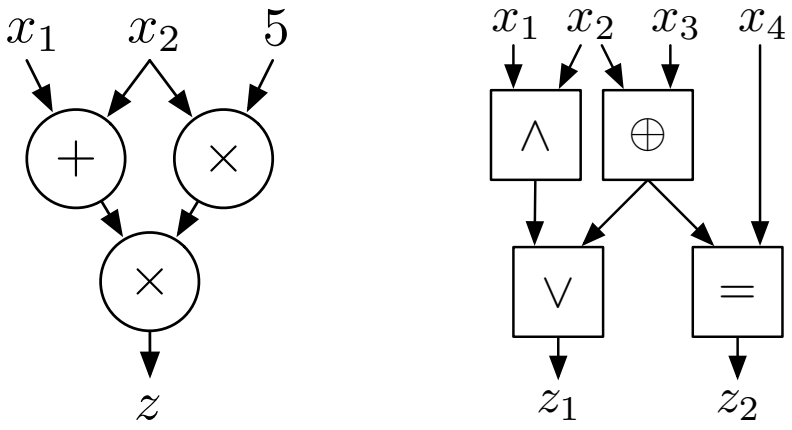

Figure 3.2: Arithmetic (left) and boolean (right) circuits.

TASTY executes protocols as follows. Each participant has its own input x. Firstly, the server and the client agree on a protocol specification written in TASTYL. After an initial validation for syntactic correctness, the server converts x into an encrypted value using an homomorphic encryption scheme under its own public key, while the client converts x into a garbled value. TASTY implements the algorithms required to convert into different representations on-the-fly (garbled circuits to homomorphic encryption and vice-versa) [34]. After the setup phase, and with these conversion algorithms, f may be evaluated by the protocol using both mechanisms simultaneously.

The ABY Framework [16] is also a similar mixed-protocol approach that aims to efficiently combine secure computation schemes based on arithmetic sharing, boolean sharing and Yao's garbled circuits, enabling interchangeability among the three. ABY has records of being able to outperform other mixed-protocol frameworks [24, 31, 34] by employing more efficient methods for multiplication, faster conversion techniques between secure computation protocols and by batching pre-computing cryptographic protocols. ABY originally considers protocols for secure two-party computations that can be used for a large variety of privacy-preserving applications, but can be expanded into multiple input and output parties.

The major strength of this framework is the possibility of combining three secure computation schemes, and employing them in a way that maximizes efficiency. This is accomplished via parallel batching during the setup phase (encompassing most of the cryptographic operations) and improved constructions for conversion algorithms with regards to computation and communication costs.

Combining secure function evaluation techniques is an advantageous approach when performance is a core requirement. Furthermore, for the PRACTICE project, Aeroengine Fleet Management, Platform for Benchmarking, Consortium Gathering Information From its Members, Tax Fraud Detection, Joint Statistical Analysis Between State Entities, Privacy Preserving Personal Genome Analyses and Studies and Platform for Surveys on Sensitive Data [17] explicitly require either high bandwidth or large computation capabilities, and therefore may greatly benefit from such a performance-oriented approach. Such tool suites could employ our verified implementation of Yao as a sub-protocol for the computations that employ garbled circuits.

3.5 Towards more challenging security models

We will now recall techniques that permit enhancing solutions we have described so far so that they can deal with stronger adversaries. The provided description will be very brief, since this encompasses

secure computation solutions that have already been evaluated in detail for WP11 [45] and adversarial definitions presented in WP12 [10].

The previously depicted approaches consider the semi-honest adversarial model, in which security is guaranteed only against adversaries that follow the protocol specification. However, many real-world scenarios involve entities that are willing to deviate from established behavior for goals such as acquiring sensitive data or forcing unfair protocol results. As such, stronger adversaries must be considered in order to provide adequate multi-party computation protocols.

If the security model assumes covert adversaries [4], we should consider the possibility of adversaries acting arbitrarily, as long as their probability of being caught is not higher than some deterrence factor. This scenario is explicit in the requirements for application scenarios of the PRACTICE project, such as the Consortium Gathering Information From its Members [17], but can also be considered as an alternative to more restrictive settings, in which the required performance levels cannot be met by protocols that provide stronger security guarantees.

According to [41], the cut-and-choose technique is often employed to extend garbled circuit evaluation to this adversarial model, and it works as follows. The first party constructs n copies of some garbled circuit and sends them to the second party. Afterwards, half of the circuits are opened, in order to verify the correctness of their construction. If this subset is correct, then it is guaranteed that a majority of the unopened half is also correct, with an exceptional probability that is negligible in n. Finally, the participants jointly evaluate the remaining $n/2$ circuits, and the second party considers the output value that appears in most of the evaluated circuits. Recent research in [27, 38] has systematically improved the performance of protocols using this cut-and-choose paradigm, reducing the overhead in computation and communication.

Following [14], malicious (or active) adversaries are a more powerful variation of covert adversaries, i.e. also capable taking full control of corrupt parties and behaving arbitrarily, that are not concerned about being caught. This is the strongest adversary for modeling multi-party computation protocols and also a very common threat when considering real-world applications of multi-party computation. In particular, 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 and Mobile Data Sharing [17] are PRACTICE application scenarios with requirements that either consider all computation parties to be malicious (albeit not all colluding), or a mixed setting of semi-honest and malicious adversaries.

Passively secure protocols can be transformed into actively secure ones by employing zero-knowledge proofs-of-knowledge (ZK PoK) and verifying secret sharing schemes, following a construction known as the GMW protocol compiler [21, 37]. Despite ZK PoK techniques being usually considered inefficient for practical use, developments such as [38] present improvements to previous cut-and-choose strategies against malicious adversaries [39, 41, 42, 50] that greatly decrease the number of garbled circuits necessary for the computation of large circuits (which is usually taken to be the case in real-world implementations). These approaches allow us to expand the scope of our verified secure computation engine to more demanding scenarios, including use cases that consider covert or active adversaries.

Chapter 4

Formal verification requirements

In the previous chapters we have discussed in detail the motivation and scope for the formal verification work that will be carried out in PRACTICE. We have also given a high-level view of the functionalities provided by the different components in the PRACTICE high-assurance secure computation framework, and clarified their integration in the PRACTICE architecture. In this chapter we will refine the specification of the three components in this framework, providing a detailed description of the required functional and security properties, and outlining our approach to formally verifying these properties. The material in this chapter will guide the development and formal verification work that is being undertaken in the PRACTICE project. The following subsections separately address the three components in the formally verified framework: secure computation engine, computation description generation and high-level specification analysis.

4.1 Secure computation engine

In this section, we state the requirements that arise in the verification of the correctness and security of an implementation of Yao's protocol, as described in the previous chapter. We begin by introducing EasyCrypt, a framework for computer-aided cryptographic proofs that we will be using to achieve our goals, and then present a formalization of the properties that will need to be verified for our concrete use case. Our description follows [2], where we presented the results of a preliminary feasibility analysis of undertaking such a task in EasyCrypt, carried out in collaboration with the EasyCrypt developers.

Computer-aided cryptographic proofs

EasyCrypt [6] is a tool-assisted framework for verifying the security of cryptographic constructions in the computational model. Following suggestions by Bellare and Rogaway [9] and Halevi [22], EasyCrypt adopts a code-based approach, in which cryptographic constructions, security notions and computational assumptions are modeled as probabilistic programs in a core (but extensible) probabilistic programming language with imperative constructs and procedure calls. Procedures can be concrete, in which case they are provided with a body consisting of a command and a return expression, or abstract, in which case only a type signature is provided. The primary purpose of abstract procedures is to model adversaries, but they are also an essential ingredient for compositional reasoning.

Program logics. Reasoning in EasyCrypt is supported by two program logics:

1. a probabilistic relational Hoare logic (pRHL), that allows reasoning about judgments of the form:
$$[c_1 \sim c_2 : \Psi \Longrightarrow \Phi]$$
where c_1 and c_2 are probabilistic programs, and Ψ and Φ are relations on memories;

2. and a probabilistic Hoare logic (pHL), that allows reasoning about judgments of the form:
$$[c : \Psi \Longrightarrow \Phi] \diamond \delta$$
where c is a probabilistic program, Ψ and Φ are predicates on memories, δ is a real-valued expression, and $\diamond$ is a comparison operator (i.e. $\leq$, $=$, or $\geq$).

The logics are respectively used to establish a formal connection between two games in game-based proofs, and to resolve the probability of an event in a game. As an illustration, consider two programs c_1 and c_2 that are equivalent up to a failure event F in the execution of c_2; the equivalence is formalized in pRHL by the judgment:
$$[c_1 \sim c_2 : \mathsf{true} \Longrightarrow \neg F\langle 2\rangle \Rightarrow \equiv]$$
where $\equiv$ denotes equality of memories, and $\langle 2\rangle$ indicates that the interpretation of F is taken in the output memory of c_2. Moreover, assume that the probability of F in c_2 is upper-bounded by some constant δ; this bound can be formalized in pHL by the judgment:
$$[c_2 : \mathsf{true} \Longrightarrow F] \leq \delta$$

EasyCrypt also provides rules to convert pRHL and pHL judgments into probability claims. Using these rules, one obtains that, for every event E:
$$\Pr[c_1 : E] - \Pr[c_2 : E] \leq \delta$$

Note that one can recover the more traditional formulation of the Fundamental Lemma (where the left-hand side of the above inequality is replaced by its absolute value) by strengthening the post-condition of the pRHL judgment with the assertion $F\langle 1\rangle \Leftrightarrow F\langle 2\rangle$.

EasyCrypt also provides an ambient logic to reason about operators. It can be used, for instance, to state that a decoding function is the inverse of an encoding function. A novel feature of EasyCrypt 1.0 is to allow pRHL and pHL judgments as first-class formulae in the ambient logic. In other words, one can use the ambient logic to reason about formulae that freely use pRHL and pHL judgments; for instance, one can perform a case analysis on the probability of an event to build an adversary such that a reduction is valid; or, one can use the available induction principles in the ambient logic to formalize hybrid arguments.

Modules. EasyCrypt features a module system that provides a structuring mechanism for describing cryptographic constructions. A module consists of global variable declarations and procedure definitions. (By construction, all the procedures of a given module share memory.) Modules are mainly used for representing cryptographic games - either concrete or abstract. For example, the ElGamal encryption scheme is represented as the following concrete module (where the code for enc and dec has been omitted):

```
module ElGamal = {
  fun kg() : skey * pkey = {
    var x : int = $[0..q-1];
    return (x, g^x);
  }
  fun enc(pk:pkey, m:plaintext) : ciphertext = { ... }
  fun dec(sk:skey, c:ciphertext) : plaintext = { ... }
}.
```

The constituents of a module and their types are reflected in their module type: a module M has module type I if all procedures declared in I are also defined in M, with the same type and parameters. For instance, the previously defined ElGamal module can be equipped with the following module type:

```
module type Scheme = {
  fun kg () : skey * pkey
  fun enc(pk:pkey, m:plaintext) : ciphertext
  fun dec(sk:skey, c:ciphertext) : plaintext
}.
```

Not only can modules use previously defined modules, but they can also be parameterized. For example, the following parameterized definition captures chosen-plaintext security, where the public-key encryption scheme S and the adversary A are module parameters:

```
module type ADV = {
  fun choose (pk:pkey) : msg * msg
  fun guess (c:cipher) : bool
}.

module CPA (S:Scheme, A:ADV) = {
  fun main () : bool = {
    var pk,sk,m0,m1,b,b',challenge;
    (pk,sk) = S.kg();
    (m0,m1) = A.choose(pk);
    b <$- {0,1};
    challenge = S.enc(pk, b?m1:m0);
    b' = A.guess(challenge);
    return b' = b;
  }
}.
```

The key to compositional reasoning in EasyCrypt is its ability to quantify (either universally or existentially) over modules in its ambient logic. An essential feature of the module system is to allow restricting memory access between quantified modules.

THEORIES. EasyCrypt features a theory mechanism for organizing and reusing the axiomatizations of the different algebraic and data structures used in cryptographic constructions. In its simplest form, a theory consists of a collection of type and operator declarations, and a set of axioms; cyclic groups, finite fields, matrices, finite maps, lists or arrays are instances of such forms of theories in EasyCrypt's core libraries. Theories might also contain modules, allowing the definition of libraries of standard games depending on abstract algebraic and data structures.

Theories enjoy a cloning mechanism that is useful when formalizing examples that involve multiple objects of the same nature, e.g. cyclic groups in bilinear pairings. Moreover, operators of a theory can be realized, i.e. instantiated by expressions, during cloning. We also use cloning as a substitute for polymorphic modules.

In the following example, the ElGamal scheme is defined in the scope of an ElGamalT theory that first clones a fresh copy of the theory of cyclic groups. The ElGamalT theory is then cloned as InstantiatedElGamalT. The cyclic group of InstantiatedElGamalT is no more abstract but is realized using a concrete cyclic group.

```
theory CyclicGroup.
  type t.
  op g : t. (* generator *)
  ...
end CyclicGroup.

theory ElGamalT.
```

```
  clone import CyclicGroup as CG.
  module ElGamal = { ... (* g is CG.g in this context *) }.
end ElGamalT.I

clone ElGamalT as InstantiatedElGamalT
  with CG.t ← Z*_5, CG.g ← 2, CG.( * ) x y = x * y mod 5, proving * by smt.
```

Code generation. EasyCrypt offers an extraction mechanism that generates OCaml code from functional programs written in EasyCrypt, allowing the production of correct-by-construction implementations from an EasyCrypt proof. With the nature of the source and target languages being close, the extraction mechanism is simple enough that one can have a high confidence in the generated code.

The presence of abstract types/operators does not prevent the use of the extraction mechanism. Indeed, when encountering an abstract library, stubs for its operators, that have to be filled manually, are generated. For instance, EasyCrypt provides OCaml implementations for most of the algebraic and data structures that have been abstractly formalized in the core libraries, such as fixed-length bitstrings or functional arrays.

Security Claims. Security claims in EasyCrypt are expressed in the form of reductions relating the advantages of two algorithms as in $\mathbf{Adv}_{\mathcal{A}}(\lambda) \leq \mathbf{Adv}_{\mathcal{B}(\mathcal{A})}(\lambda)$. Such statements have the advantage of making the adversary $\mathcal{B}$ explicit, and of supporting concrete as well as asymptotic security. For instance, EasyCrypt claims could be converted into asymptotic security claims by making the notion of security parameter implicit, and by requiring in all computational assumptions and security definitions that the adversaries and simulators execute in polynomial-time. In a similar way, EasyCrypt claims could be converted to concrete security claims by reasoning about the execution time of algorithms. Currently, EasyCrypt offers no support for reasoning about program complexity, so the only way to check that our reductions and simulations are indeed performed in polynomial time is by direct inspection of the code. However, adding support for reasoning about complexity is work in progress; once available, it will be possible to take full advantage of existential quantification over modules.

Formalizing Yao's Protocol in EasyCrypt

Two-Party Protocols. We first start by generically defining two-party protocols, that generalize both Secure Function Evaluation and Oblivious Transfer, and their security. In EasyCrypt, declarations pertaining to abstract concepts meant to be later refined can be grouped into named theories such as the one shown in Figure 4.1. Any lemma proved in such a theory becomes also a lemma of any implementation (or instantiation) where the theory axioms hold.

```
theory Protocol.
 type input1, output1.
 type input2, output2.
 op validInputs: input1 → input2 → bool.
 op f: input1 → input2 → output1 * output2.

 type rand1, rand2, conv.
 op prot: input1→ rand1→ input2→ rand2→ conv * output1 * output2.
  ...
end Protocol.
```

Figure 4.1: Abstract Two-Party Protocol.

Two parties want to compute a *functionality* f on their joint inputs, each obtaining their share of the output. This may be done interactively via a *protocol* prot that may make use of additional randomness (passed in explicitly for each of the parties) and produces, in addition to the result, a *conversation trace* of type conv that describes the messages publicly exchanged by the parties during the protocol execution. In addition, the input space may be restricted by a validity predicate validInputs. This predicate expresses restrictions on the adversary-provided values, typically used to exclude trivial attacks not encompassed by the security definition.

Following the standard approach for secure multi-party computation protocols, security is defined using simulation-based definitions. In this case we capture honest-but-curious (or semi-honest, or passive) adversaries. We consider each party's *view* of the protocol (typically containing its randomness and the list of messages exchanged during a run), and a notion of *leakage* for each party, modelling how much of that party's input may be leaked by the protocol execution (for example, its length). Informally, we say that such a protocol is secure if each party's view can be efficiently simulated using only its inputs, its outputs and the other party's leakage. Formally, we express this security notion using two games (one for each party). We display one of them in Figure 4.2, in the form of an EasyCrypt *module*. Note that modules are used to model games and experiments, but also schemes, oracles and adversaries.

Module type $\mathcal{Adv}_i^{\mathsf{Prot}}$ ($i \in \{1,2\}$) tells us that an adversary impersonating Party i is defined by two procedures: i. choose that takes no argument and chooses a full input pair for the functionality, and ii. distinguish, that uses Party i's view of the protocol execution to produce a boolean guess as to whether it was produced by the real system or the simulator. Since the module type is not parameterized, the adversary is not given access to any oracles (modelling a non-adaptive adversary). We later show how oracle access can be given to both abstract and concrete modules. We omit module types for the randomness generators R_1 and R_2, as they only provide a single procedure gen taking some leakage and producing some randomness. We also omit the dual security game for Party 2.

The security game, modeled as module Sec_1, is explicitly parameterized by two randomness-producing modules R_1 and R_2, a simulator S_1 and an adversary $\mathcal{A}_1$. This enables the code of procedures defined in Sec_1 to make queries to any procedure that appears in the module types of its parameters. The game implements, in a single experiment, both the real and ideal worlds. In the real world, the protocol prot is used with adversary-provided inputs to construct the adversary's view of the protocol execution. In the ideal world, the functionality is used to compute Party 1's output, which is then passed along with Party 1's input and Party 2's leakage to the simulator, producing the adversary's view of the system. We prevent the adversary from trivially winning by denying him any advantage when he chooses invalid inputs.

A two-party protocol prot (parameterized by its randomness-producing modules) is said to be secure with leakage $\Phi = (\phi_1, \phi_2)$ whenever, for any adversary $\mathcal{A}_i$ implementing $\mathcal{Adv}_i^{\mathsf{Prot}}$ ($i \in \{1,2\}$), there exists a simulator S_i implementing Sim_i such that

$$\mathsf{Adv}^{\mathsf{Prot}^i_\Phi}_{\mathsf{prot},\mathsf{S}_i,\mathsf{R}_1,\mathsf{R}_2}(\mathcal{A}_i) = |2 \cdot \mathsf{Pr}[\mathsf{Sec}_i(\mathsf{R}_1,\mathsf{R}_2,\mathsf{S}_i,\mathcal{A}_i) : \mathsf{res}] - 1|$$

is small, where res denotes the boolean output of procedure main.

Intuitively, the existence of such a simulator S_i implies that the protocol conversation and output cannot reveal any more information than the information revealed by the simulator's input.

Oblivious Transfer Protocols. We can now define oblivious transfer, restricting our attention to a specific notion useful for constructing general SFE functionalities. To do so, we *clone* the Protocol theory, which makes a literal copy of it and allows us to instantiate its abstract declarations with

```
type leak1, leak2.   op φ1 : input1 → leak1.   op φ2 : input2 → leak2.
type view1 = rand1 * conv.   type view2 = rand2 * conv.

module type Sim = {
  proc sim1(i1: input1, o1: output1, l2: leak2) : view1
  proc sim2(i2: input2, o2: output2, l1: leak1) : view2
}.

module type Sim_i = {
  proc sim_i(i_i: input_i, o_i: output_i, l_{3-i}: leak_{3-i}) : view_i
}.

module type Adv_i^Prot = {
  proc choose(): input1 * input2
  proc distinguish(v: view_i) : bool
}.

module Sec1(R1: Rand1, R2: Rand2, S: Sim1, A1: Adv_1^Prot) = {
  proc main() : bool = {
    var real, adv, view1, o1, r1, r2, i1, i2;
    (i1,i2) = A1.choose();
    real <$- {0,1};
    if (!validInputs i1 i2)
      adv <$- {0,1};
    else {
      if (real) {
        r1 = R1.gen(φ1 i1);
        r2 = R2.gen(φ2 i2);
        (conv,_) = prot i1 r1 i2 r2;
        view1 = (r1, conv);
      } else {
        (o1,_) = f i1 i2;
        view1 = S.sim1(i1, o1, φ2 i2);
      }
      adv = A1.distinguish(view1);
    }
    return (adv = real);
  }
}.
```

Figure 4.2: Security of a two-party protocol protocol.

concrete definitions. The partial instantiation is shown in Figure 4.3. We restrict the input, output and leakage types for the parties, as well as the leakage functions and the functionality f. The chooser (Party 1) takes as input a list of boolean values (i.e., a bit-string) she needs to encode, and the sender (Party 2), takes as input a list of pairs of messages (which can also be seen as alternative encodings for the boolean values in Party 1's inputs). Together, they compute the array encoding the chooser's input, revealing only the lengths of each other's inputs. We declare an abstract constant n that bounds the size of the chooser's input. This introduces an implicit quantification on the bound n in all results we prove. Defining OT security is then simply a matter of instantiating the general notion of security for two-party protocols via cloning. Looking ahead, we use $\mathsf{Adv}^{\mathsf{OT}^i}$ to denote the resulting instance of $\mathsf{Adv}^{\mathsf{Prot}^i_{(\mathsf{length},\mathsf{length})}}$, and similarly, we write $\mathcal{Adv}_i^{\mathsf{OT}}$ for the types for adversaries against the OT instantiation.

To define a concrete two-party OT protocol, one now only has to define the types of randomness and conversations and the protocol itself, with its individual computation and message exchange steps.

Garbling schemes. Garbling schemes [8] (Figure 4.4) are operators on *functionalities* of type func. Such functionalities can be evaluated on some input using an eval operator. In addition, a functionality can be *garbled* using three operators (all of which may consume randomness). The operator funG produces the garbled functionality, inputK produces an input-encoding key, and outputK produces an output-encoding key. The garbled evaluation evalG takes a garbled functionality and some

```
clone Protocol as OT with
  type input1 = bool array,
  type output1 = msg array,
  type leak1 = int,
  type input2 = (msg * msg) array,
  type output2 = unit,
  type leak2 = int,
  op φ1 (ī1: bool array) = length ī1,
  op φ2 (ī2: (msg * msg) array) = length ī2,
  op f (ī1: bool array) (ī2: (msg * msg) array) = ī1ī2.
  op validInputs(ī1: bool array) (ī2: (msg * msg) array) =
      0 < length ī1 ≤ nmax ∧ length ī1 = length ī2,
...
```

Figure 4.3: Instantiating Two-Party Protocols into Abstract OT.

encoded input and produces the corresponding encoded output. The input-encoding and output-decoding functions are self-explanatory. In practice, we are interested in garbling functionalities

```
type func, input, output.
op eval : func → input → output.
op valid: func → input → bool.

type rand, funcG, inputK, outputK.
op funcG : func → rand → funcG.
op inputK : func → rand → inputK.
op outputK: func → rand → outputK.

type inputG, outputG.
op evalG : funcG → inputG → outputG.
op encode: inputK → input → inputG.
op decode: outputK → outputG → output.
```

Figure 4.4: Abstract Garbling Scheme.

encoded as boolean circuits and therefore fix the func and input types and the eval function. Circuits themselves are represented by their topology and their gates. A topology is a tuple $(n, m, q, \mathbb{A}, \mathbb{B})$, where n is the number of input wires, m is the number of output wires, q is the number of gates, and $\mathbb{A}$ and $\mathbb{B}$ map to each gate its first and second input wire respectively. A circuit's gates are modeled as a map $\mathbb{G}$ associating output values to a triple containing a gate number and the values of the input wires. Gates are modeled polymorphically, allowing us to use the same notion of circuit for boolean circuits and their garbled counterparts. We only consider *projective schemes* [8], where boolean values on each wire are encoded using a fixed-length random *token*. This fixes the type funcG of garbling schemes, and the outputK and decode operators.

Following the Garble1 construction of Bellare et al. [8], we will construct our garbling scheme using a variant of Yao's garbled circuits based on a pseudo-random permutation, via an intermediate Dual-Key Cipher (DKC) construction. We denote the DKC encryption with E, and DKC decryption with D. Both take four tokens as arguments: a tweak that we generate with an injective function and use as unique IV, two keys, and a plaintext (or ciphertext).

The privacy property of garbling schemes required by Yao's SFE protocol is more conveniently captured using a simulation-based definition. Like the security notions for protocols, the privacy definition for garbling schemes is parameterized by a leakage function upper-bounding the information about the functionality that may be leaked to the adversary. (We consider only schemes that leak at most the topology of the circuit.) Consider efficient non-adaptive adversaries that provide two procedures: i. choose takes no input and outputs a pair (f,x) composed of a functionality and some input to that functionality; ii. on input a garbled circuit and garbled input pair (F,X), distinguish outputs a bit b representing the adversary's guess as to whether he is interacting with the real or

ideal functionality. Formally, we define the $\mathsf{SIM\text{-}CPA}_\Phi$ advantage of an adversary $\mathcal{A}$ of type $\mathcal{A}dv^{\mathsf{Gb}}$ against garbling scheme $\mathsf{Gb} = (\mathsf{funcG},\mathsf{inputK},\mathsf{outputK})$ and simulator S as

$$\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Phi}_{\mathsf{Gb},\mathsf{R},\mathsf{S}}(\mathcal{A}) = |2 \cdot \Pr[\mathsf{SIM}(\mathsf{R},\mathsf{S},\mathcal{A}) : \mathsf{res}] - 1| \,.$$

A garbling scheme Gb using randomness generator R is $\mathsf{SIM\text{-}CPA}_\Phi$-secure if, for all adversary $\mathcal{A}$ of type $\mathcal{A}dv^{\mathsf{Gb}}$, there exists an efficient simulator S of type Sim such that $\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Phi}_{\mathsf{Gb},\mathsf{R},\mathsf{S}}(\mathcal{A})$ is small.

```
type leak.
op Φ: func → leak.

module type Sim = {
  fun sim(x: output, l: leak): funcG * inputG
}.

module type Adv^Gb = {
  fun choose(): func * input
  fun distinguish(F: funcG, X: inputG) : bool
}.

module SIM(R: Rand, S: Sim, A: Adv^Gb) = {
  fun main() : bool = {
    var real, adv, f, x, F, X;
    (f,x) = A.gen_query();
    real <$- {0,1};
    if (!valid f x)
      adv <$- {0,1};
    else {
      if (real) {
        r = R.gen(Φ f);
        F = funcG f r;
        X = encode (inputK f r) x;
      } else {
        (F,X) = S.sim(f(x),Φ f);
      }
      adv = A.dist(F,X);
    }
    return (adv = real);
  }
}.
```

Figure 4.5: Security of garbling schemes.

Yao's SFE Construction. We now explain how garbled circuits and oblivious transfer can be combined to provide a general secure function evaluation protocol, and formalize a generic security proof goal, parameterized by a secure garbling scheme and a secure oblivious transfer protocol. The goal is subsequently to instantiate this abstract result with a concrete oblivious transfer protocol and a concrete garbling scheme.

As discussed briefly above, we model SFE as a two-party protocol. We consider the functionality to be evaluated, encoded as a circuit, as part of Party 2's input and set up the leakage function to let it become public. We denote by $\mathsf{Adv}^{\mathsf{SFE}^i}$ and $\mathcal{A}dv_i^{\mathsf{SFE}}$ the instantiations of $\mathsf{Adv}^{\mathsf{Prot}^i}$ and $\mathcal{A}dv_i^{\mathsf{Prot}}$ to the SFE construction. We preface the definition of our generic SFE construction with two named clones of the abstract garbling scheme and oblivious transfer theories. This allows us to essentially parameterize the SFE protocol with a garbling scheme and an oblivious transfer protocol. Instantiating these parameters is simply done by instantiating the Gb and OT theories with concrete definitions and proofs. In Figure 4.6, we formalize a standard SFE construction explained informally in the previous chapter.

```
clone Garble as Gb.
clone OT as OT.

clone Protocol as SFE with
  type rand1 = OT.rand1,
  type input1 = bool array,
  type output1 = Gb.output,
  type leak1 = int,
  type rand2 = OT.rand2 * Gb.rand,
  type input2 = Gb.func * bool array,
  type output2 = unit,
  type leak2 = Gb.func * int,
  op f i1 i2 = let (c,i2) = i2 in Gb.eval c (i1 || i2),(),
  type conv = (Gb.funcG * token array * Gb.outputK) * OT.conv,
  op validInputs (i1:input1) (i2:input2) =
    0 < length i1 ∧ Gb.validInputs (fst i2) (i1 || snd i2),
  op prot (i1:input1) (r1:rand1) (i2:input2) (r2:rand2) =
    let (c,i2) = i2 in
    let fG = Gb.funG c (snd r2) in
    let oK = Gb.outputK c (snd r2) in
    let iK = Gb.inputK c (snd r2) in
    let iK1 = (take (length i1) iK) in
    let (ot_conv, (t1,_)) = OT.prot i1 r1 iK1 (fst r2) in
    let GI2 = Gb.encode (drop (length i1) iK) i2 in
    (((fG,GI2,oK),ot_conv), (Gb.decode oK (Gb.evalG fG (t1 || GI2)),())).
```

Figure 4.6: Abstract SFE Construction.

Formal verification goals: protocol correctness and security

Our formal verification process will be constructed using the following results. First, we will show that Yao's construction is secure when instantiated with a secure oblivious transfer protocol and a secure garbling scheme.

Theorem 1 (Abstract SFE security) *For any oblivious transfer protocol OT and any garbling scheme Gb, let SFE_a be the SFE protocol built using Yao's construction from OT and Gb. For all SFE adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ implementing type $\mathcal{Adv}^{\mathsf{SFE}} = (\mathcal{Adv}_1^{\mathsf{SFE}}, \mathcal{Adv}_2^{\mathsf{SFE}})$, we can construct an efficient simulator that estabishes its security under the assumption that OT and Gb are themselves secure.*

We will then formalize in detail and prove the security of a concrete oblivious transfer protocol (SomeOT) and a concrete garbling scheme construction (SomeGarble), leading to the following theorems.

Theorem 2 (OT-security of SomeOT) *For all $i \in \{1, 2\}$ and OT^i adversary $\mathcal{A}_i$ of type $\mathcal{Adv}_i^{\mathsf{OT}}$ against the SomeOT protocol, there exists an efficient simulator that establishes its security under standard computational assumption.*

Theorem 3 (SomeGarble is SIM-CPA-secure) *For every SIM-CPA adversary $\mathcal{A}$ against SomeGarble, there exists an efficient simulator that establishes its security under the assumption that the underlying dual key cipher scheme is secure.*

Finally, we will instantiate our first abstract result with the previous concrete theorems, to derive the following final main theorem stating the security of the resulting Concrete SFE protocol.

Theorem 4 *For all SFE adversaries $\mathcal{A}$ against the Concrete SFE protocol, we construct an efficient simulator S that establishes its security, under the assumption that the underlying dual key cipher is secure and other stantard computational assumptions.*

Our formalization will also include correctness results for each of the proof steps highlighted here.

4.2 Computation description generation

The verified computation description generation component will be implemented as an extension to the CompCert verified compiler. We begin by introducing CompCert and the relevant features, and then we present the functionality and formal verification goals for our extension, which will convert C programs into computation descriptions represented as Boolean circuits.

Background on CompCert

CompCert is a formally verified optimizing C compiler [36]. It produces target code with strong correctness guarantees and reasonable efficiency when compared to general purpose compilers. CompCert supports the C language (with almost complete coverage of the ISO C 90 / ANSI C standard) and produces assembly code for the PowerPC, ARM, and IA32 (x86 32-bits) architectures. CompCert is mostly implemented in Coq, and its development is subdivided into 19 compiler phases, each of which builds a semantic preservation proof between semantically defined intermediate languages.

Formally, CompCert's correctness theorem establishes a strong notion of semantic preservation, referred in CompCert terminology as a *simulation*. This guarantees that, if a source program P^{C} is successfully compiled into P^{asm}, then the observable behaviour of this last program is an admissible behaviour of the original program. The proof of this result is based on a formalization of the semantics of both the compiler's source and target languages (C and assembly), as well as of all the compiler passes. Behaviours are captured by a possibly infinite sequence of events that model interactions of the program with the outside world, such as accesses to volatile variables, calls to system libraries, or user defined events (so called annotations).

More in detail, CompCert's semantic preservation result establishes that, conditioning on similar interactions with the environment, the observable behaviour for the compiled program in any of the target and intermediate languages, matches the observable behaviour of the source C program. The notion of observable behaviour of a program in any of the languages manipulated by CompCert considers only programs with a well-defined entry point (the main function); it permits adding annotations to source code that explicitly create visible events in the execution trace; and it captures the possibility that programs go wrong or do not terminate.

We now detail the various aspects in which we enchanced CompCert and the formal correctness guarantees that it provides, in order to obtain verified Boolean circuit generation.

Certified generation of circuits

What does it means to generate a circuit from a C program? The overall idea is to focus on the impact of the C program on two pre-specified memory regions: an *n-bit input buffer*, and an *m-bit output buffer*. The generated circuit should then implement a boolean function $f^{\mathsf{circ}} : 2^n \to 2^m$ that captures precisely the effect of running the program on a memory state where the input buffer is filled with some n-bit string, and observing the m-bit string obtained on the output buffer when the program finishes. In order to make sense of the previous discussion, two major restrictions should be considered on C programs:

1. Programs should avoid any form of interaction with the environment (such as systems calls, use of volatile variables, etc.). In particular, the behaviour of programs should be fully deterministic (depending only on the contents of the input buffer);

2. Programs should always terminate for every possible input data.

Regarding the first point, we note that interaction with the environment is precisely what is normally registered on the behavioural trace produced by programs. In fact, we are interested in programs that are essentially "silent" with respect to observable events. But since the CompCert's semantic preservation result is expressed resorting to those traces, we will build on a CompCert extension developed in [1], that adds two special builtin operations to the CompCert framework: read and write. The former emulates the non-deterministic generation of an arbitrary global input, registering it in the observable behaviour of the program. The latter will be used to expose a memory region that stores output of the program (circuit) in its observable behaviour. Using these constructs, we will anchor our tool on a program entry-point with the following shape:

Function main( )
read($\vec{i}$)
circuit( )
write($\vec{o}$)

Here circuit represents the procedure that acts as entry-point for the program that implements the Boolean circuit in C, $\vec{i}$ represents global variables registered as circuit inputs, and $\vec{o}$ represents global variables registered as circuit outputs. The two special builtins read and write are responsible for the whole interaction with the environment: read fills the input variables with values obtained from the environment, write externalise the corresponding output variables. Since read and write are the only constructs in our programs that induce observable events, we note that traces became essentially a way of registering the intended input/output behaviour of the circuit program. Hence, the correctness result of CompCert ensures preservation of this behaviour along the compilation process.

Regarding the second point, termination will be ensured by restricting the allowed control structure of C programs. Indeed, and following what is done in other related works [44, 18], we will consider a subclass of C programs for which:

- All function calls can be statically inlined.
- Loops can be fully statically unrolled.
- No dynamic memory allocation is used. In particular, all arrays are statically allocated.

Intuitively, the input program must be prone to static transformations that give rise to an equivalent C implementation in a single loop-free procedure, taking no inputs, that reads information from a well-defined set of global input variables and writes to a well-defined set of global output variables (which may overlap).

Tool architecture

As mentioned above, the certified circuit generation tool will be built on top of the CompCert certified compiler. Specifically, we will rely on the the CompCert's C frontend and on the first compiler passes up to the RTL (Register Transfer Language) intermediate language. Once at the RTL level, an abstract circuit description will be produced, where conditional branches are translated to equivalent data-flow selection gates. A concrete Boolean circuit is then produced by instantiating elementary operations (e.g. 32bit addition) with externally provided optimized gates. Figure 4.7 details the flow between the source C program and the final circuit.

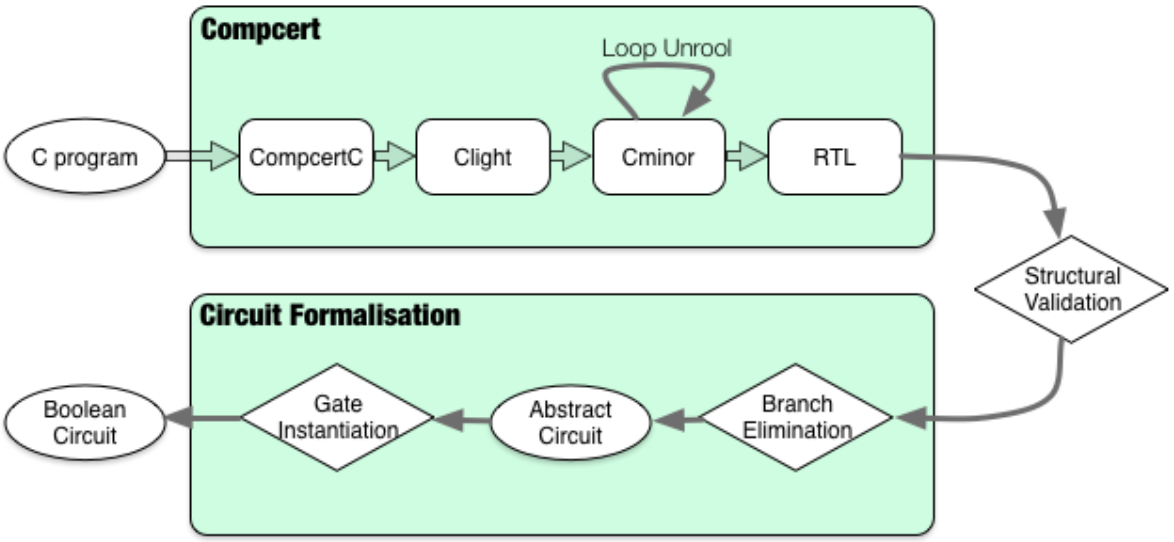

Figure 4.7: Architecture of the Certified Circuit Generation Tool

Formal verification goals: the C side

We will benefit from CompCert's C language formalisation, as well as the first few compiler passes. This would allow us to leverage CompCert's framework as a stepping stone to define/formalise a semantic preserving translation to Boolean circuits.

Nevertheless, we shall adapt and extend some parts of the CompCert development to meet the specific characteristics of the envisaged tool. Most notably, we will add additional compilation passes that will address some missing processing stages for circuit generation, namely:

Loop unroll — this compiler pass will unroll each loop by a externally provided bound. The bound is expected to be sufficiently large so that the compiler can statically ensure the safety of the loop removal. The choice of performing this pass at the *Cminor* level is based on the fact that at lower levels the loop structure is disguised in general `if/goto` patterns, which would complicate its treatment.

Structural validation — this pass is a validation check that will ensure that all the restrictions placed on the allowed C programs are met. Specifically, it will check that entry-point exibihts the required shape (mentioned above); that all function calls were fully inlined; and that all loops were removed (after being unrolled). Any violation of the above requirements would halt the compilation.

After successfully validated, the resulting RTL program is a single function whose CFG (Control-Flow Graph) is an acyclic graph.

Formal verification goals: the circuit side

The restricted RTL obtained from the modified CompCert shall be further transformed to remove all the branches of the CFG. Conceptually, this transformation amounts to the translation of the RTL program into SSA (Static Single Assignment) form, replacing then the ϕ-nodes into appropriate selection-gates. This is, however, a translation where some clever program manipulations might be

called upon to achieve reasonably compact circuits; and this is why we consider this compilation pass as a good candidate for a translation-validation pass[1].

Having the program as a pure *straight-line program* in SSA form allow us to perform the intended semantic shift on its interpretation – instead of variables stored in memory we shall consider wire-buses connecting Boolean gates. To this end, we will formalise the evaluation of Boolean circuits in Coq, relating it to the correspondent C-side semantics. At this level, primitive operations are viewed as high-level gates (with a well-defined interface and semantics).

Finally, to produce a standalone circuit, high-level gates should be replaced by concrete circuit instantiations. The aim is to rely as much as possible on existing optimized implementations developed in CBMC-GC, possibly adding some validation checks using automatic provers such as SMTs and SAT solvers. In other cases, verified (non-optimized) alternatives might be adopted by bootstrapping them in the tool.

4.3 High-level specification analysis

SecreC is the language in which multi-party computations are specified in the Sharemind framework and it was designed for statistical analysis and data mining. Syntactically, the language is quite similar to C though the similarities are very superficial and, for instance, lacks many of features that make C an easy to misuse language (precisely to avoid the potential security issues), such as raw pointers. One of the major features of SecreC is a type system designed to differentiate publicly available data from private data. A programmer can designate some data as public and some other as private, and the type system guarantees that private data does not become public unless the programmer explicitly declassifies it, i.e., accepts to disclose such private data publicly. As a simple example, consider the following code:

Listing 4.1: A simple SecreC example

```
1 import shared3p;
2 domain private shared3p;
3 void main () {
4     private float[[1]] x = argument ("x");
5     private float sqrLen = dot (x, x);
6     bool res = declassify (sqrLen < 1.0);
7     ...
8 }
```

This snippet imports a library shared3p which provides various functions specific to the additive 3-party secret sharing scheme (and possibly defines more efficient operations than those included in the standard library). Next, a private protection domain of kind shared3p is declared – the protection domain must be configured with concrete physical addresses on the Sharemind installation in which this code runs. This is required because Sharemind allows for a single installation to be running multiple protection domains of the same kind, and it is also possible for a program to be using more than one domain at the same time. The main procedure of the program reads a private argument x, performs some private computations on it, and finally declassifies (reveals) the data to all parties participating in the computation. Note that the code would be rejected by the type checker without

[1] In certified compilers, a translation-validation pass is a compiler pass where the translation itself is not independently certified. Instead, it is delegated to a non-certified component, and only its output is submitted to a validation pass the checks if it meets the semantic requisites.

the explicit declassification call, since this would be interpreted as a non-intentional release of private information. For a more complete overview of the SecreC language, the interested reader is referred to PRACTICE deliverable D22.1.

SecreC already does assist the developer via the type system and language design choices targeted at the SMC domain, but much more could be possible via static code analysis. Here, we discuss possibilities for improving the developer experience and increasing the security guarantees that SecreC provides when manipulating secure computation specifications. These will be addressed using formal verification techniques, and the resulting analyses will be integrated into the Sharemind framework.

Formal verification goals: a program analysis framework

SecreC has a very basic code analysis framework that first translates SecreC programs into a simpler intermediate representation language, and then runs the corresponding analyses on top of the generated intermediate representations [48]. Among a few others analyses, the SecreC analysis framework implements a so-called *reachable-declassify analysis*, to check if a private input directly reaches a declassification call (without being processed by a non-trivial intermediate computation). For some simple cases, this analysis can be used to detect unwanted data flows originating from sensitive private information that is not advisable for publication. However, this analysis has not proved useful in practice because (the framework itself) is not context- nor path-sensitive. To understand its limitations, consider a simple example:

Listing 4.2: Limitations in information flow control

```
1 private int aggregate (private int[[1]] array) {
2     return array[0];
3 }
4 void main () {
5     private int[[1]] array = argument ("array");
6     private int result = aggregate (array);
7     print (declassify (result));
8 }
```

The main procedure gets an array of integers as an input, performs some aggregation on it and declassifies the result. However, the function to aggregate the data is defined in such a way that it returns the first element of the array. This means that a private input reaches a declassify call directly through a function call, what should raise a warning. The current analysis, however, does not catch this data flow due to lack of context sensitivity. If the aggregation function would instead compute the average of the elements of the input array, then the raw private inputs would not be directly declassified and the analysis should not raise a warning.

We will study how a type system for language-based information flow analysis can assist in the reasoning about the flow of private information in a SecreC program. In this setting, we can use type inference to overapproximate which declassify calls can be reached (directly or not) from a set of private inputs. Still for the above example, it may be difficult to reason about whether leaking the total aggregate is acceptable or not. A quantitative analysis of the amount of leaked data per declassification call will be a possible further refinement to help in that direction.

Formal verification goals: validating optimizations

Any typical secure multi-party algorithm involves a declassification of private information before publishing its results. Assuming that releasing such information is acceptable, it is often possible to improve the performance of the algorithm without changing its security guarantees, by refactoring it into a version that makes sure to declassify such released information as early as possible, thus minimizing the number of (inefficient) secure computations. In order to perform this optimization automatically, a secure multi-party computation compiler must be able to identify which extra declassified data can be inferred from the publicly available data (either known from the start or published after running the computation).

This problem of *knowledge inference*, has been researched in [47, 53], and we will integrate the existing state-of-the-art into the SecreC analysis framework. Implementing these approaches directly over SecreC remains, however, challenging since typical SecreC programs may span dozens of functions and thousands of lines of code. Moreover, (the implementation of) a knowledge inference technique is often tailored to a specific secure multi-party computation dialect and may place non-obvious restrictions on the subclass of programs that can be analyzed, with different performance/expressiveness tradeoffs. A more robust solution for SecreC will likely require the combination of different existing approaches.

Formal verification goals: controlling leakage

Consider a piece of code that counts how many times some element occurs in an array and declassifies the result.

Listing 4.3: A more elaborate declassification example

```
1  uint count (private uint[[1]] xs, private uint z) {
2      private bool[[1]] bs = (xs == z);
3      private uint[[1]] cs = (uint) bs;
4      private uint count = 0;
5      for (uint i = 0; i < size (cs); ++ i) {
6          count = count + cs[i];
7      }
8
9      return declassify (count);
10 }
```

First, the input array is compared point-wise to the input element, yielding the boolean array bs. Next, the boolean array is converted to unsigned integers, which are summed using a for loop, and the declassified sum is finally returned.

In this example, however, one would need to go one step further and assist the programmer in reasoning about whether or not declassifying the boolean array holding the results of the comparisons (what could lead to a much more efficient protocol) is acceptable from the point of view of the concrete application in which this computation is integrated (e.g., how much more sensitive is the overall released information). Two options will be considered for this functionality. The first one is to allow the programmer to specify what *acceptable* leakage is, independently of the program itself. It should then be possible to verify if the optimized implenentation satisfies this requirement. Another direction in which assistance can be provided, is to construct a knowledge basis of program transformations that can lead to more efficient solutions, and then construct a tool that is able to recognize (or even automatically suggest) these re-writing patterns in order to validate the optimizations.

Chapter 5

Conclusion

In this document we have presented the requirements for the formal verification activities that are currently being undertaken in the PRACTICE project. In order to pinpoint these requirements, we have also refined the description of the proof-of-concept high-assurance secure computation framework that will be developed, and clarified how this fits into the overall architecture of the software stack that is being created in the project. We have also analysed which use-cases and application scenarios this framework can cater to, and validated our requirements accorgingly.

The activities leading to the completion of this framework are split between work packages WP14 and WP22, where the development work of the various tools is taking place. They will continue until the end of the project at M36. A first snapshot of this development and verification work will be provided in the prototype deliverables that will be produced at M24 in WP22 (i.e., in Deliverable 22.3). Also, a detailed description of how this framework will fit into the global PRACTICE architecture and into the lower level secure computation engine architecture that are being specified in WP21 and WP14 will also be included in the M24 deliverables that are being written in these work packages (i.e., D21.2 and D14.1, respectively).

Chapter 6

List of Abbreviations

CFG	Control-Flow Graph
DKC	Dual-Key Cipher
HE	Homomorphic Encryption
MPC	Multi-party Computation
OT	Oblivious Transfer
pHL	Probabilistic Hoare logic
pRHL	Probabilistic relational Hoare logic
RTL	Register Transfer Language
SFE	Secure Function Evaluation
SSA	Static Single Assignment
TASTY	Tool for Automating Secure Two-partY computations
ZK PoK	Zero-knowledge proofs-of-knowledge

Bibliography

[1] J.B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *Conference on Computer and Communications Security, CCS 2013*. ACM, 2013.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Guillaume Davy, François Dupressoir, Benjamin Grégoire, and Pierre-Yves Strub. Verified implementations for secure and verifiable computation. Cryptology ePrint Archive, Report 2014/456, 2014. `http://eprint.iacr.org/`.

[3] Todd W Arnold and Leendert P Van Doorn. The ibm pcixcc: A new cryptographic coprocessor for the ibm eserver. *IBM Journal of Research and Development*, 48(3.4):475–487, 2004.

[4] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography*, pages 137–156. Springer, 2007.

[5] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Secure evaluation of private linear branching programs with medical applications. In *Computer Security–ESORICS 2009*, pages 424–439. Springer, 2009.

[6] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Heidelberg, 2011. Springer.

[7] Amos Beimel. Secret-sharing schemes: a survey. In *Coding and cryptology*, pages 11–46. Springer, 2011.

[8] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 784–796. ACM, 2012.

[9] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, Heidelberg, 2006. Springer.

[10] 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`.

[11] Justin Brickell, Donald E Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 498–507. ACM, 2007.

[12] Justin Brickell and Vitaly Shmatikov. Privacy-preserving classifier learning. In *Financial Cryptography and Data Security*, pages 128–147. Springer, 2009.

[13] Sven Bugiel, Stefan Nurnberger, A Sadeghi, and Thomas Schneider. Twin clouds: An architecture for secure cloud computing. In *Workshop on Cryptography and Security in Clouds (WCSC 2011)*, 2011.

[14] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation, an introduction. *Contemporary cryptology*, pages 41–87, 2009.

[15] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[16] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby–a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security, NDSS*, 2015.

[17] Hiva Mahmoodi Ferdinand Brasser and Editors Ahmad-Reza Sadeghi. D12.2: Adversary, Trust, Communication and System Models. PRACTICE Deliverable, 2014.

[18] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: an ANSI C compiler for secure two-party computations. In Albert Cohen, editor, *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8409 of *Lecture Notes in Computer Science*, pages 244–249. Springer, 2014.

[19] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[20] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, March 2010.

[21] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2004.

[22] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[23] Isabelle Hang, Ferdinand Brasser, Niklas Buescher, Stefan Katzenbeisser, Ahmad Sadeghi, Kai Samelin, Thomas Schneider, Jakob Pagter, Peter Sebastian Nordholt Janus Dam Nielson, Kurt Nielsen, Johannes Ulfkjaer Jensen, Dan Bogdanov, Roman Jagomägis, Liina Kamm, Jaak Randmets, Jaak Ristioja, Reimo Rebane, Jaak Ristioja, Sander Siim, Riivo Talviste, Manuel Barbosa, Bernardo Portela, Rui Oliveira, Stelvio Cimato, and Ernesto Damiani. PRACTICE Deliverable D22.1: tools: State-of-the-art analysis, 2013. Available from `http://www.practice-project.eu`.

[24] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*, pages 451–462, 2010.

[25] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 772–783. ACM, 2012.

[26] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[27] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology–CRYPTO 2013*, pages 18–35. Springer, 2013.

[28] Intel. Software guard extensions programming reference, 2013.

[29] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[30] Roman Jagomägis. Secrec: a privacy-aware programming language with applications in data mining. *Master's thesis, University of Tartu*, 2010.

[31] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *Applied Cryptography and Network Security*, pages 566–584. Springer, 2014.

[32] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS*, volume 5888 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2009.

[33] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. From dust to dawn: Practically efficient two-party secure function evaluation protocols and their modular design. *IACR Cryptology ePrint Archive*, 2010:79, 2010.

[34] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 21(2):283–315, 2013.

[35] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[36] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages, POPL 2006*. ACM, 2006.

[37] Yehuda Lindell. *Composition of Secure Multi-Party Protocols: A Comprehensive Study*, volume 2815. Springer Science & Business Media, 2003.

[38] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology–CRYPTO 2013*, pages 1–17. Springer, 2013.

[39] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology-EUROCRYPT 2007*, pages 52–78. Springer, 2007.

[40] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptol.*, 22(2), April 2009.

[41] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25(4):680–722, 2012.

[42] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *Advances in Cryptology–CRYPTO 2013*, pages 36–53. Springer, 2013.

[43] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology - EUROCRYPT'99*, pages 223–238. Springer, 1999.

[44] 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.

[45] 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`.

[46] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[47] Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A. Hammer. Knowledge inference for optimizing secure multi-party computation. In Prasad Naldurg and Nikhil Swamy, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, pages 3–14. ACM, 2013.

[48] Jaak Ristioja. An analysis framework for an imperative privacy-preserving programming language. Master's thesis. University of Tartu., 2010.

[49] Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Efficient privacy-preserving face recognition. In *Information, Security and Cryptology–ICISC 2009*, pages 229–244. Springer, 2010.

[50] Chih-hao Shen et al. Two-output secure computation with malicious adversaries. In *Advances in Cryptology–EUROCRYPT 2011*, pages 386–405. Springer, 2011.

[51] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography–PKC 2010*, pages 420–443. Springer, 2010.

[52] Sean W Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[53] Raphael Urmoneit and Florian Kerschbaum. Efficient secure computation optimization. In Martin Franz, Andreas Holzer, Rupak Majumdar, Bryan Parno, and Helmut Veith, editors, *PETShop'13, Proceedings of the 2013 ACM Workshop on Language Support for Privacy-Enhancing Technologies, Co-located with CCS 2013, November 4, 2013, Berlin, Germany*, pages 11–18. ACM, 2013.

[54] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in cryptology–EUROCRYPT 2010*, pages 24–43. Springer, 2010.

[55] Andrew Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.