D21.3

Application architecture for secure computation

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 / D21.3 / 1.0
Activity and WP:	Activity 2 / WP21
Due Date:	October 2016 - M36
Actual Submission Date:	2nd November, 2016
Responsible Organisation:	TUDA
Editor:	Ágnes Kiss
Dissemination Level:	Public
Revision:	1.0
Abstract:	This report is the outcome of Task 2.1.3 of the PRACTICE project and describes a general architecture for applications using secure multiparty computation. It extends deliverable D21.2 with general guidelines and already existing example applications, for including both task-specific protocols and generic secure computation in application development.
Keywords:	Architecture, Application, Secure Computation, Task-Specific Protocols

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

Editor

Ágnes Kiss (TUDA)

Contributors (ordered according to beneficiary numbers)

Florian Hahn (SAP)
Ahmad-Reza Sadeghi (TUDA)
Thomas Schneider (TUDA)
Peter Sebastian Nordholt (ALX)
Roman Jagomägis (CYBER)
Sander Siim (CYBER)
Matthias Schunter (INTEL)

Executive Summary

Work package WP21 unifies approaches for using secure computation in applications and programming tools. The main goal of the work package is to provide helpful guidance for designers of information systems that potentially could make use of secure computation. This includes providing developers with insight on how to combine the deployment and trust models of different techniques with the programmable secure computation technology to achieve better privacy and security guarantees.
Deliverable D21.2 [22] presents a general architecture for the Secure Platform for Enterprise Applications and Services (SPEAR) that allows for easy development and deployment of secure cloud applications. SPEAR relies on the Distributed Aggregation and Security Services (DAGGER) sub-platform in order to provide Cryptography-as-a-Service for privacy-sensitive cloud services and applications. Deliverable D21.2 also shows how SPEAR & DAGGER can be constructed in a number of alternative ways using different sets of secure computation technologies, presenting a general architecture as a result of the PRACTICE project.
In this deliverable we extend deliverable D21.2 with guidelines and example applications in order to help designers to rely on secure computation techniques when designing their applications. These guidelines are drawn from the experience gained throughout developing real-life secure applications. This deliverable covers not only generic secure computation protocols but also techniques to integrate task-specific protocols, such as private set intersection, into applications. A discussion on enhancing security using secure hardware is also included in the end of the deliverable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This deliverable presents general guidelines for integrating secure computation tools into applications, both task-specific and generic protocols are considered. We show how the existing *Secure Platform for Enterprise Applications and Services* (SPEAR) enables easy development and deployment of secure cloud applications where the *Distributed Aggregation and Security Services* (DAGGER) platform can be selected from the PRACTICE architecture. Helpful guidance is provided for application developers who would use secure computation for achieving better security and privacy guarantees. This deliverable discusses the additional factors that need to be taken into consideration when using secure computation – either generic or task-specific – as opposed to applications using conventional cryptographic algorithms. The guidelines given here are provided based on experience gained by developing and deploying applications throughout the PRACTICE project, e.g., in work packages WP23 and WP24. The example applications that we review here are or potentially can be deployed in the cloud. They all integrate secure computation tools from the PRACTICE general architecture and use them to achieve a particular funcionality for a use case application. In the end of the deliverable, we describe how hardware security can further enhance the security and privacy guarantees when deploying applications based on secure computation.

1.1 Scope

This report is the third deliverable of work package WP21 (*Architecture and Integration*) in the PRACTICE project. The main task of WP21 is to unify the approaches for using secure computation in applications and programming tools. A general architecture with general guidelines are to be provided for both secure computation services and applications. The work package analyses the secure deployment models of state-of-the-art secure computation protocol which is then combined with a generic architecture for building secure computation services and applications.
Previously, in deliverable D21.1 [95] we devised the secure deployment and trust models of secure computation technology as described in Task 2.1.1. Then, based on that work, in deliverable D21.2 [22] we designed the general architecture for building and deploying programmable secure computation systems on the cloud. The approaches of secure computation technologies were unified, and an architecture describing the integration of protocols with cloud applications and programming tools was provided, covering the Task 2.1.2.
In this deliverable, we build on top of the previous results from this and other work packages to provide general guidelines for designing applications using both generic and task-specific protocols. Figure 1.1 depicts the most important PRACTICE deliverables and work packages

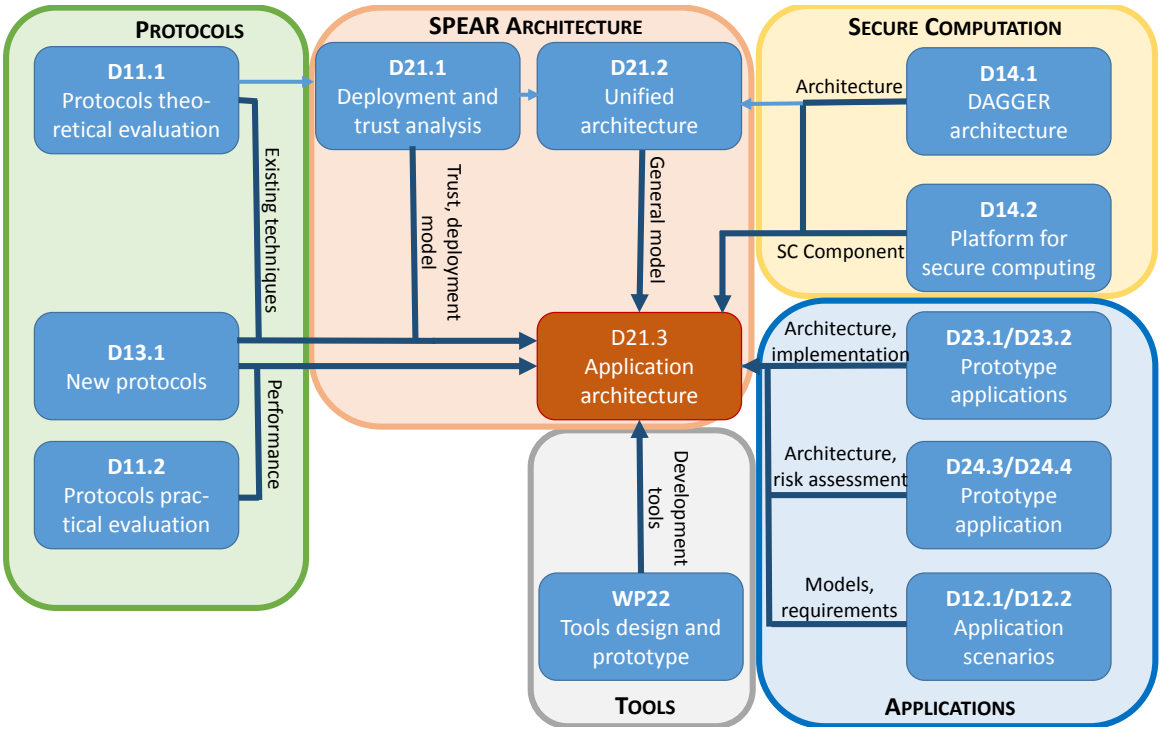


Figure 1.1: PRACTICE deliverables and work packages that influenced this work in D21.3, grouped into categories. The inputs that have been taken from the mentioned deliverables or work packages are indicated by the arrows.

that we build on throughout this work. On the figure we indicate what content had the main influence on this deliverable with the respective arrows. These can be grouped into larger categories such as

Protocols Existing secure computation protocols and their theoretical and practical evaluation can be found in D11.1, D11.2 and D13.1.

Secure Computation WP14 covers the secure computation engine with algorithm evaluation, and protocol integration into the engine and their usage. D14.1 describes the architecture and D14.2 the concrete implementation of the FRESCO engine.

SPEAR Architecture D21.1 provides the trust and deployment models, D21.2 provides a unified architecture for programmable secure computation, i.e., describes a VM based design. In the current deliverable, D21.3, a general architecture for applications that use secure computation (either VM-based or specific protocols) is given.

Tools The deliverables in WP22, i.e., D22.1, D22.2 and D22.3 describe the development tools.

Applications Application scenarios are described in D12.1 and D12.2, while prototype applications that we rely on throughout this deliverable are described in D23.1, D23.2, D24.3 and D24.4.

We summarize the scope of this deliverable as described in Task 2.1.3: „*Design of a general approach for using secure computation in applications*". General guidelines are to be described for including task-specific, as well as generic secure computation protocols in application development. We present a set of guidelines and examples on how to integrate as many kinds

of secure computation tools as possible. The general guidelines, among others, answer the following set of questions:

- What is the data that much be protected?
- How does this data need to be processed?
- What are the roles of the parties participating in the computation?
- What is the adversary model in the application?
- How should the DAGGER sub-platform be selected from the PRACTICE architecture?
- What algorithms does the application use and how do they process data?
- Where is the point of encryption?
- How is the data protected, what cryptographic primitives are used?
- Which performance optimizations can be included?
- How to deploy an application?

1.2 Organization

Chapter 2 of this deliverable provides the general guidelines on how to integrate secure computation tools into applications. It answers the questions described above and gives insights to application developers on how to develop and deploy their application that uses secure computation. In Chapter 3, several PRACTICE partners describe their applications that were developed (and possibly deployed) within the PRACTICE project. The application presented in Section 3.2 is based on task-specific protocols and the ones presented in Sections 3.1, 3.3, 3.4 and 3.5 are based on generic secure computation. All the presented applications are based on tools taken from the PRACTICE architecture presented in deliverable D21.2 [22]. In Chapter 4, we describe the potential integration of secure hardware into applications. Finally, we conclude in Chapter 5.

Chapter 2

Integrating SPEAR into Applications

Previously in deliverables D21.2 [22] and D14.1 [30] we have presented the architectures for the *Secure Platform for Enterprise Applications and Services* (SPEAR) and its *Distributed Aggregation and Security Services* (DAGGER) sub-platform that together allow building and using programmable secure computation systems on the cloud. We have also shown multiple examples of different technology stacks that could potentially be used for constructing such systems. In this chapter we will take a step further and describe a general approach for integrating SPEAR into applications. The main goal is to provide helpful guidance for designers of information systems that could use secure computation to achieve better privacy and security guarantees.
According to the SPEAR architecture the applications that use secure computation for processing data receive an additional dimension of data security when compared to conventional applications. This affects the way such applications are developed and deployed, as a number of important security related nuances must be taken into consideration throughout the application's life cycle. In the following we discuss these key aspects and propose the general approach of developing SPEAR applications based on the experience gained while developing several prototypes and real-life applications within and outside of PRACTICE documented in deliverable D21.1 [95] as well as work packages WP23 and WP24. We mainly focus on the specifics related to secure applications, as the rest of the software development process mostly follows a traditional approach.

2.1 Architectural Drivers

2.1.1 Problem statement

The development of a SPEAR application begins with determining the problem that can be solved using secure computation technology. The general motivation behind such problem would be the need to process private data in order to gain some kind of benefit. Processing private data allows learning new knowledge and can provide economic, strategic, social, medical or other added value to one or more stakeholders. However, the privacy of the data must not be compromised in the process, as otherwise the data owner's interests might be put at risk for personal, business or legal reasons.
Finding a Trusted Third Party (TTP) to process the data can be difficult and expensive, while the privacy issue would still persist. Alternatively, cryptographic secure computation techniques offer the trusted execution environment that acts as a TTP but significantly leverages the privacy issue. The SPEAR application will, where appropriate, utilize secure computation techniques to protect the sensitive data during processing.

To identify potential usage scenarios for secure computation one would need to analyze the current situation in one's business case and try to highlight the bottlenecks that exist as a result of data privacy issues. Some of the important topics to cover would be: a) Stakeholders and trust issues among them (see Section 2.1.2); b) Privacy issues and risks involved with data processing (see Section 2.1.3); c) Inefficiencies and inabilities resulting from the trust issues (e.g. ineffective/insufficient processes and decision making, inability to learn new knowledge/lack of data, inability to cooperate). This information will help to describe the bottlenecks and the reasons that prevent them from being solved, and allow to establish the links between the problems and the possible solutions. Then, by providing the privacy-preserving solutions to those bottlenecks new benefits can be unlocked.
There are multiple ways the private data can be processed. In deliverable D11.1 [107] three categories of usage scenarios for secure computation have been identified. Most applications should fit one of these categories, although there can be modifications.

Outsourced computation on one's own data This category will fit use cases where a data owner has data that needs to be processed and wants to outsource such processing to a service. This clearly represents the most common cloud computing scenarios where an individual or an organization outsources computation to a service provider. Secure computation is needed here to ensure that the service does not learn the confidential information or leak it to third parties.

Outsourced computation on collected data The second category handles the situations where the party who needs to process the data does not have the data and needs to collect it. This scenario occurs in surveys, government statistics, social studies, voting, medical research, auctions, and a wide range of corporate activities. Secure computation is needed to ensure that nobody except for the data owner has access to the data but, at the same time, data utility is retained.

Computation on shared data The third category covers use cases where similar parties combine their information to jointly learn new things. These scenarios occur in industrial consortiums, research collaborations and joint activities between coalitions of nations. Secure computations allow partners to share data while ensuring control of this data during its processing. The latter is achieved by having all parties participate in the actual computation (i.e., all parties being computing parties as defined in Section 2.1.2).

2.1.2 Actor roles and trust relations

In this section, we will describe how actors who participate in the use case scenarios of the application rely on and relate to each other: what roles they fulfill and what trust assumptions they have towards each other.

Roles in secure computation

An important step towards the SPEAR application design is to determine the *actors* that will participate in the use case scenarios enabled by the application. This is done by identifying the potential stakeholders of a business case and mapping their roles and goals in the application. As it was described in deliverables D21.1 [95] and D21.2 [22], the participants in a secure computation application can have four fundamental kinds of actor roles: *input parties*, *result parties*, *computing parties* and *verifier parties*. Each role has its own distinct generic behaviors and goals based on which it can be assigned to a suitable stakeholder to form an actor. Specific

Table 2.1: Actor roles in secure computation

Party	Definition
input party	possesses private input data and provides it to SPEAR; has complete control over private data; single or multiple
result party	wishes to gain benefits from computation outcomes; gets useful results; is interested in receiving the correct output
computing party	hosts and operates the SPEAR application and related infrastructure; protects private data; should not collude with other parties nor deviate from the application logic
verifier party	special kind of result party; verifies correctness of the result

to secure computation are the privacy related goals and trust assumptions. Hence, in order to properly assign the roles to stakeholders it is crucial to understand the trust relations among the stakeholders. In the following and in Table 2.1 we describe the named roles and provide assignment rationale for them.

Input party Input parties have the required input data and provide it to the SPEAR application for processing and storage. The private data is either owned by the input parties or entrusted to them under the agreement or legal obligation of non-disclosure. In any case, input parties are interested in retaining complete control of their private data, protecting it from any third party. An application can have one or more input parties, as the data may come from a single or multiple sources.

The potential stakeholders suitable for this role must be willing to participate in the application by providing their data. Among possible motivators for them are: a) having an additional role of the *result party* who benefits from the application outcomes; b) having obligations to participate.

Result party Result parties are the ones who want to analyze the private input data to answer their questions. They make queries to the SPEAR application and in return get useful computation results that can be interpreted to gain added value, but do not leak more information about the original inputs than allowed by design. The result parties are interested that the result values intended only for them are received from the application by nobody else but them. They are also interested in the correctness of the results. This role can be assigned to any stakeholders who wish to gain benefits from the computation outcomes.

Computing party Computing parties are responsible for hosting and operating the SPEAR application and related infrastructure. During the operation of the application the computing parties collaboratively conduct data processing and storage in accordance with the application logic and the DAGGER secure computing sub-platform used in its SPEAR architecture. The choice of DAGGER determines the number of computing parties in a deployed application and the trust model between them. In general, the computing parties are expected to protect the private data they process throughout the application's life cycle. Therefore, they should not collude (i.e. share data encryptions) with each other or any other parties, nor deviate from the agreed application logic and cryptographic protocols of DAGGER.

Stakeholders suitable for this role should have strong interest in preserving the privacy of the data providers, be as independent from each other as possible and share the incentive

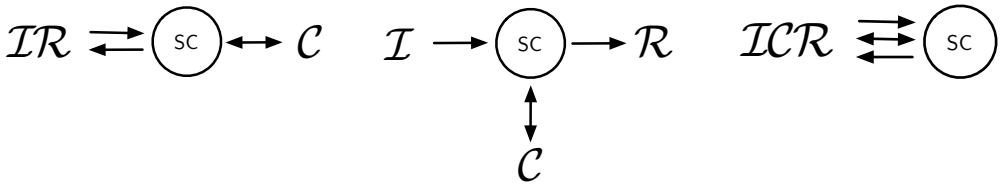
to host the application. Ideally, they would either be competing organizations, represent (or be one of) the *input parties* or have legal obligation to protect data. They should also be capable of independently arranging the necessary infrastructure and setting up the application runtimes ensuring best security of the setup.

Verifier party There is also a special kind of result party whose goal is to verify the correctness of the result after it has been computed. While the verifier does not necessarily participate in the computation, it still wishes to see the proof (e.g. a transcript of computation) that the computation has been performed correctly regardless of the trust assumptions of the DAGGER protocol implementation used in the application hosted by computing parties.

Any set of stakeholders interested in the additional verification of the result correctness can be assigned this role. Note, that this role can only be present in the system, if the chosen DAGGER protocols support verifiability by external party.

Roles in abstract usage models

The usage scenarios for secure computation from Section 2.1.1 can be described as abstract usage models of secure computing in terms of the roles we described above (as provided in D11.1 [107]). We denote input parties as $\mathcal{I}$, result parties as $\mathcal{R}$ and computing parties as $\mathcal{C}$.



(a) Process own data (b) Process collected data (c) Process shared data

Figure 2.1: Abstract usage models of secure computing

Outsourced computation on one's own data The model is shown on Figure 2.1a. The input party encrypts its input data and sends it to the computing party (e.g., a cloud service provider) who processes the data without decrypting it during the process. Once the computation has been completed, the encrypted result is returned to the data owner who can decrypt it.

Outsourced computation on collected data The model in Figure 2.1b differs from the previous model in the way that the input parties $\mathcal{I}$ and the result parties $\mathcal{R}$ are separated from each other. This is to signify the fact that the $\mathcal{I}$ parties do not trust the $\mathcal{R}$ parties.

Computation on shared data This model is shown on Figure 2.1c. Each party fulfills all roles by providing an input, contributing to the computation and benefiting from the results.

In all these settings, the computing parties (the $\mathcal{C}$ nodes) can be deployed on the cloud, because secure computation guarantees that the computing parties do not learn the private inputs.

2.1.3 Data and privacy risk assessment

When designing a secure computation application an additional analysis is required in order to get a good understanding of the data that needs to be processed by the application and the privacy issues involved with the data. This will allow to highlight the security threats and propose new security requirements.
The first step in this analysis would be to identify all the potentially useful data that each of the stakeholders has. Having a good overview of the data space in the form of clusters of data attributes centered around their owners would provide useful insight already in the problem statement and initial solution proposal phases. Based on that knowledge it would be possible to begin pinpointing connections between the business case bottlenecks discussed in Section 2.1.1 and the potential solutions by finding use for data that can become available via collaboration assuming that all the privacy concerns will be addressed to an acceptable degree.
Next, the privacy issues associated with the data should be learned, as it is crucial for the application designer to know which data in the data space must be protected and to what extent. The privacy issues may apply to a particular data attribute or their combination, i.e. it allows learning something sensitive that would not be possible to learn without different input sources combining their data. The stakeholders may be reluctant to share some of the data for personal, business or legal reasons. Also, there could be restrictions on who is or isn't allowed to know a particular data in the application. In any case, the privacy issues usually arise from the risks that could be imposed on the interests of individuals or organizations as a result of gaining access to sensitive or confidential data by third parties. By quantifying the privacy risks related with the data it is possible to determine the confidentiality level of the data and choose the appropriate level of protection.
The choice of methodology for privacy risk assessment depends on the specifics of a particular application problem statement and the goals of its stakeholders. In general it would involve performing a series of surveys or interviews with all the stakeholders to receive their view of the risks related to their data, and then applying a risk scoring mechanism that takes into account the parameters related with the application. An example risk measurement model has been developed for the collaborative cloud-based supply chain planning system in deliverable D24.3 [87]. From it we can derive a generalized approach as one possible way to assess privacy risks.
For each data attribute in data space the following questions are addressed:

1. What potential disadvantage (i.e. negative impacts resulting from (mis)using the data by stakeholders) may a data owner potentially incur when sharing private data?

2. What is the probability that a participating stakeholder (mis)uses the shared data to the disadvantage of the data owner?

3. To what extent is the data prior knowledge? It is reasonable to assume that the risks related with a certain data attribute are lower if it is already accessible to some of the stakeholders.

Then, based on questions 1 and 2 the criticality of a data attribute can be assessed by first calculating the criticality score of each individual negative impact (e.g. by multiplying the corresponding *impact* and *probability* scores), and then summing together the criticality scores of the individual negative impacts. The third question is used as a weight for determining the overall protection level. The result can then be interpreted wrt. the maximum possible score.

2.2 Design

2.2.1 Selecting DAGGER Engine

The *Distributed Aggregation and Security Services* (DAGGER) is a Platform-as-a-Service (PaaS) sub-system of the SPEAR architecture stack that implements the cryptographic secure computation techniques enabling the computing on encrypted data and provides the means for integrating these capabilities into the SPEAR applications. As such, it is the key enabling technology powering the subset of the application algorithms that are exclusively responsible for processing private data and allowing the application to achieve its objectives while leveraging privacy concerns.
DAGGER implementations differ in their functionality, security and efficiency aspects. When designing a SPEAR application it is therefore important to make a careful and informed choice of the DAGGER platform and its configuration by finding a suitable trade-off that satisfies the requirements of a particular problem. To do this, the application designer should be aware of different aspects of DAGGER implementations and understand how they affect the application. In the following we provide a guideline for selecting the DAGGER platform. Based on the previous architecture work in D21.2 [22] and D14.1 [30] we distinguish between two main components of the sub-system: a) DAGGER Engine (covered in this section) and b) DAGGER Protocol Suite (covered in Section 2.2.2).
A *Secure Computation Engine* (SCE) is the core engine of the DAGGER platform that organizes and facilitates the execution of secure applications utilizing Secure Computation Techniques. Once invoked by the application via the *Secure Service Interface* (SSI), the SCE evaluates the private data processing algorithms specified in the requested *Secure Computation Specification* (SCS) by applying secure functionalities (*Protocols*) of various Secure Computation Technique implementations (*Protocol Suites*) to enforce the security of data according to the specification. We can identify the following key aspects that can affect the choice of the DAGGER engine.

Available Protocol Suites

A Protocol Suite contains the implementation of a particular secure computation technique that an SCE can use to securely process data. As various techniques differ in their properties, the set of available Protocol Suites supported by the SCE defines the range of secure computing capabilities that can potentially be utilized by a SPEAR application. Not every technique is suitable for every application. For that reason, the availability of Protocol Suites suitable for the concrete problem strongly affects the choice of the SCE. In some cases it might even be desirable to combine the advantages of different secure computation techniques by using multiple Protocol Suites in a single application. Thus, a larger collection of supported Protocol Suites is a sign of a flexible and mature SCE implementation. The aspects that affect the choice of Protocol Suites will be covered separately in the next section.

Integration support

The DAGGER engine is integrated with the rest of the SPEAR application via the *Secure Service Interface* (SSI) that is used by the application to invoke the engine to perform secure computation. Hence, the SSI shall be implemented in a way that simplifies application development and satisfies its requirements. Aspects to consider here are as follows and are depicted in Table 2.2.

Table 2.2: Aspects impacting the integration of DAGGER with SPEAR

	Approaches	Provides
Coupling model (SSI and SCE)	*lower coupling*	+ more deployment flexibility - more difficult to implement inter-process communication
	higher coupling	+ easier to implement - lacks the deployment flexibility
Platform compatibility of SSI	*wide coverage*	+ allows for flexibility in choosing platform for application, increases SCE portability
	poor coverage	- new custom SSI wrappers need to be implemented, less flexibility
Invocation method of SCE	*command API*	+ better tailored for a particular SCE implementation
	query language	+ better conformance with standards, involves an interpreter - slightly less efficient
Expressive power of an SSI	*larger power*	+ dynamic and customized queries
	smaller power	- execution of only agreed-on SCSs with arguments

Coupling model This affects how much the SSI decouples the SCE from its consumer. An SSI mechanism with *lower coupling* provides more deployment flexibility by allowing: a) the SCE to be hosted on a different physical or virtual machine than the application backend; b) use SSI on the client side of the application to directly communicate with SCE. However, this option would assume some kind of inter-process communication protocol between the SSI and the SCE instance (e.g. Remote Procedure Call (RPC) or a custom message passing over Transmission Control Protocol (TCP)) to make secure queries to the SCE, and is more difficult to implement and configure. Examples of SCEs with lower coupling are Sharemind and SEEED.
Alternatively, an SSI mechanism with *higher coupling*, such as an API, could be used. In this case the SSI and SCE would be tightly coupled with the application backend in a single package. The advantages of this option are that it is much easier to implement and because of the non-existent communication layer the queries can be delivered to the SCE instantly. However, the latter advantage is also a disadvantage, as the highly coupled mechanism lacks the deployment flexibility discussed earlier. Examples of SCEs with higher coupling would be FRESCO and ABY.

Platform compatibility The SSI interface of an SCE is usually implemented by its authors using a certain programming language targeting certain platform(s). The reasons can vary from personal preferences and ease of implementation to author's specific needs. However, this sets limitation on the potential range of platforms and technologies that can be used for developing new SPEAR applications using that SCE due to compatibility issues. For example, if a suitable SCE for the application has its SSI implemented in C++, it cannot be immediately integrated into a Java application. This may become a problem, if some development platforms provide more benefits to the developer than others, but are not supported by the best suitable SCE. Hence, the compatibility of the SSI becomes an important aspect to consider.

Typically the compatibility with new platforms can be achieved by implementing additional SSI wrappers in the target languages, e.g. Java JNI can be used to wrap the C++ code and make it usable in a Java application. Alternatively, the SCE may provide SSI implementations originally written in different target languages, but this is less likely due to higher development and maintenance costs. In any case, SCEs with a wider coverage of platform compatibility will allow for more flexibility in choosing a platform for application development. Also, the ease of creating new custom SSI wrappers for the particular SCE is another important aspect to consider, as it increases SCE portability even if the required platforms are not supported out of the box. Chapter 4 of D21.2 [22] provides a nice overview of supported SSI languages for various existing SCE implementations.

Invocation method There are multiple ways the SCE can be invoked to perform a certain computation. The queries can be formed using a command API or a query language inspired by e.g. SQL or MDX. The command API is more likely to be used in a higher coupled SSI calling mechanism. Both the command API and the query language may be used in a lower coupled SSI to make queries to an SCE. The advantage of a command API is that it can be better tailored for a particular SCE implementation, while the advantage of a query language is its likely better conformance with standards. On the other hand, the query language would involve an interpreter and could be slightly less efficient than a command API.

Expressive power When forming a query one should be able to request the required computation and its parameters. The expressive power of an SSI defines how complex queries can be formed. In simpler cases it may only allow requesting the execution of particular agreed-on Secure Computation Specifications with given arguments. In other cases the user may have more freedom to craft and customize the query to dynamically trigger rather complex aggregations and data mining algorithms, that alternatively would have to be specified as dedicated procedures and invoked separately. Thus, the SSI with a larger expressive power allows to shift some logic complexity from the SCS and allow more dynamic and customized queries to be used in the application.

Programming paradigm

Programmability of the DAGGER engine indicates how and to what extent it allows the application developer to specify and customize the integrated procedures for computation on private data. It also affects how easy it would be to extend the application with new features. There are different approaches to program the Secure Computation Specifications, resulting in different SCS formats that the engine can understand. We discuss the identified ones below and in Table 2.3.

Interpreted programs Secure computation can be expressed in a Domain Specific Language (DSL) that is interpreted by the Secure Computation Engine in an on-demand fashion. For better performance a low-level DSL, such as a byte-code, is preferred. In these cases the computation itself is specified using a high-level DSL better suited for the task and then compiled into an interpreted low-level DSL that a particular SCE can understand. For example, the high-level SecreC language is compiled into the low-level Sharemind byte-code.
The interpreted language provides a clear abstraction between the concrete programming task at hand and the low-level cryptographic primitives it relies on. The secure computing task

Table 2.3: Approaches to program the Secure Computation Specifications

Approach	Advantages	Disadvantages	Example(s)
Interpreted programs: high-level DSL compiled to low-level DSL	simplifies code and application development; increases reuse of code; reduces program size; easier to read and verify; improves deployment, maintenance and portability	more difficult development and maintenance of VM that interprets the program; VM limits capabilities	high-level SecreC to SHAREMIND bytecode
Embedded DSL programs: builds on top of an existing programming language	reuse of language and development platform; capabilities and performance of host language; simplifies development of SCE	may lack features; syntax may be less clear; more difficult to write; native machine code may introduce vulnerabilities; compatibility issues	FRESCO and Java libraries
Compiled circuits: representation of the computing task as Boolean or arithmetic circuits	can be highly optimized; better performance; enables function hiding	lower level of abstraction; usually needs dynamical combination with SCSs based on interpreted or embedded DSLs	circuits in ABY
Task-specific protocols: implementation tailored to perform a specific task	efficient in a concrete task and compact; can be integrated into SPEAR; triggered from an SCS	development requires deep crypto knowledge; more difficult to combine with other operations	private set intersection

is, therefore, constructed from relatively high-level secure computation primitives, e.g. the operations represented by the DAGGER protocols.
This approach has multiple advantages. First, it significantly *simplifies the code and application development* as the developer only operates with easy-to-understand building blocks and does not have to deal with complex cryptography. Second, it *increases the reuse of code* and *reduces the program size*, as the low-level cryptographic primitives do not have to be copied throughout the computation. Third, the resulting simpler code is also *easier to read and verify* for undesired behavior. Fourth, it *improves the deployment and maintenance* of the overall SPEAR application, as one can update the program or the cryptographic protocols without having to update the other. Fifth, a tandem of a domain specific language compiler and well designed VM can make the overall application rather efficient. Last but not the least, because of the clear abstraction level the applications written in an interpreted language may be easier portable to interpreted languages of different DAGGER engines.
The potential disadvantage of interpreted languages is that developing and maintaining the virtual machine that interprets the program is a non-trivial task. If the virtual machine is not sufficiently optimized and does not apply clever run-time optimizations to the program, the resulting application can experience performance issues. Moreover, even if optimized, a

VM can sometimes hardly compete in terms of performance with hand-crafted task-specific applications. The expressive power of interpreted languages is also limited by the capabilities of the virtual machine.

Embedded DSL programs An embedded DSL is an alternative to custom built DSLs, that builds on top of an existing programming language and adds the secure computation related features. In the work done in WP14 it has been shown that, if architected smartly, an SCE programmable with an embedded DSL (e.g. the FRESCO and Java libraries) can have most of the advantages of the interpreted languages by providing a level of abstraction comparable with interpreted languages. However, this strongly depends on the capabilities of the host language, as it may not allow to fully implement a good abstraction on the SCE side.
The main advantage of an Embedded DSL is its reuse of an existing programming language and development platform. First, an embedded DSL largely inherits the capabilities and performance of its host language and platform. The more powerful the host platform, the more capabilities the SPEAR applications can immediately make use of. Second, this significantly simplifies the development of an SCE, as new languages and interpreters do not have to be built.
This approach does have drawbacks as well. Despite the power of its host language, an embedded DSL may not provide some features that an interpreted DSL/VM specially built for secure computation can have, and may lack in those regards. For example, the syntax may be less clear, harder to write and lack in the type system department. This may lead to implementation errors and make static analysis of private information flow harder. Some languages are compiled to native machine code directly and while this may contribute to execution performance, it may also introduce dependence on particular machine architecture and introduce new compatibility issues and attack vectors by allowing parts of the application tamper with the memory of other components running in the same memory space.

Compiled circuits The computing task can be represented in a form of Boolean or arithmetic circuits inspired from electronics. Indeed, one can imagine a Boolean circuit as an electrical circuit in a processor. Boolean circuits operate on a bit level with logical operations and are, therefore, very compact representations of a computing task. Arithmetic circuits work on elements of a group, ring or field and use the arithmetic operations available.
The common property of circuits is that they represent the complete secure computing task on a low level. As such, they can be highly optimized and offer higher performance compared to programs that combine independent higher level secure operations to perform the same task. Also, the compiled circuits have been shown to enable hiding of the particular function to be computed. An example application doing just that is described in Section 3.1.
Compiled circuits can be integrated into SPEAR applications via the Protocol Suite mechanism or as part of Task-specific protocols (see description below). In the protocol suite case it is potentially possible to dynamically combine the complex compiled circuits with secure computation specifications based on interpreted or embedded DSLs to optimize the overall performance of the secure application.

Task-specific protocols Some secure computing tasks (e.g. set intersection) can be implemented as protocols that are tailored to perform the necessary task very well. Developing such protocols requires deep cryptographic knowledge and may also be harder to combine with other secure computation operations due to the incompatibility of data representations or even deployment models. However, their efficiency and compactness can still easily justify their use.

Just like compiled circuits, the Task-specific protocols can sometimes be integrated with the SPEAR applications via the Protocol Suite mechanism that plugs them into the programmable interpreted or embedded DSL specifications. This way their execution can be easily triggered from an SCS even if the result cannot be combined with other protocols. In other cases the Task-specific protocol may have to be implemented as a completely separate SCE that can only execute that particular task and is integrated with the SPEAR application via its own respective SSI.
An advantage of Task-specific protocols is their performance and excellence at a concrete task. Also, it may be easier to audit a protocol that does just one thing than a fully generic secure computation runtime.

Performance level

While the performance of a SPEAR application largely depends on the chosen Protocol Suites, it can also be noticeably affected by the DAGGER engine itself. When running the same secure computation task and using the same cryptographic techniques, different DAGGER engines can provide different overall performance. There are several aspects that can affect the performance:

- The implementation language and platform of the engine.
- Supported program evaluation strategies (e.g. linear, parallel, streaming, etc. See D14.1 [30] and D14.2 [31] for more details.)
- Efficiency of the engine's virtual machine (i.e. how fast the program instructions are executed?).
- Efficiency of the engine's network layer (i.e. how fast the data is transferred over the network?).
- Overall quality of the engine implementation.

Sometimes the difference may seem like a small constant (e.g. 2, 5 or 10 times). However, if large-scale computation is performed then even a small constant may become a considerable factor. For example, running the computation for 2, 5 or even 10 days instead of one day can make a strong financial and time difference. It is therefore advised, at least in case of large-scale tasks, to choose the most efficient SCE that supports the required secure computation techniques.

Secure storage

Secure computation involves processing private data, and that data needs to be securely collected, stored and accessed in various amounts and shapes. One option is to use a general purpose database engine in the application backend. The data would be handled by the application backend at all times and only be passed to the DAGGER engine as arguments when a secure computation is invoked via the SSI. However this approach has certain issues. First, the stored data must be ordered correctly across all the SPEAR nodes making the application having to deal with keeping the order of input data. Second, the encrypted data must be accessed and modified efficiently (wrt. both processing time and memory footprint) regardless of its amount or shape, which may be harder to achieve if the algorithms can only operate with data passed as arguments and cannot access the database itself.

An alternative option is that the DAGGER engine can natively access the database layer. In this case the engine can have built-in database integrity protection and also have direct access to data at storage to reduce the need to have multiple copies of it in memory and be able to access and modify it at source. By utilizing built-in capabilities of the engine the application developer can save time on implementing these from scratch. Examples of DAGGER engines with database layer support are SEEED and Sharemind.

Development tools

A good DAGGER engine comes with a set of tools that simplify the development of SPEAR applications based on it. Here we list some of the most helpful tools to look for. Many of these are also provided with the PRACTICE SDK in work package WP22.

1. **Emulation-based secure computation runtime.** This runtime is capable of emulating various secure computation protocols in a delay-compatible manner, providing realistic application experience with a low resource footprint. The application can be developed against emulated DAGGER interfaces locally without having to deploy a full-blown system on the cloud. This significantly increases developer's productivity.

2. **Secure Language Compiler.** A high-level language for specifying DAGGER secure data analysis algorithms in SPEAR applications. A compiler for the DAGGER specification language.

3. **Developer's guide.** The support documentation that directs the developer through creating a complete application.

4. **Secure programming reference.** The reference includes API descriptions for the secure computation protocol suites, available secure operations, their performances, security assumptions etc.

5. **Standard library.** A set of (possibly advanced) algorithms that can be used in secure computation algorithm implementations.

6. **Integrated development environment.** An IDE for developing secure computation algorithms for the DAGGER platform. Supports syntax highlighting, compiler integration, emulator integration, documentation integration, debugging support, etc.

7. **Verification tools.** These tools can validate certain formal properties of a system. For example, a) analyze the secure computation algorithms for security leaks; b) verify the correctness of computation results; c) verify correctness of implementations, i.e., whether a desired protocol (the researchers intent) has been correctly translated into a specified protocol, and then correctly implemented in a given programming language; d) determine bottlenecks; and so on.

8. **Deployment tools.** Easy to use cloud provisioning and application deployment tools.

9. **Existing applications** A well-established DAGGER engine would have been used in a variety of existing application. The more the technology is used, the more experience its developers gain and the the larger community it manages to build. This which also means more community support, documentation, examples and new applications. Both the prior work and the community of an engine would also contribute to its quality and the overall experience.

2.2.2 Selecting DAGGER Protocol Suite

Based on the unified architecture defined in deliverable D21.2 [22] the DAGGER engine can be configured with a set of Secure Computation Protocol Suites. Each Protocol Suite implements a certain secure computation technique for computing on encrypted data and exposes its functionality to the engine via a special generic interface as described in more detail in deliverable D14.1 [30]. The engine can then apply these technique implementations according to pre-defined algorithm specifications in order to provide the actual security for the SPEAR applications during the processing of private data.
The Protocol Suites define many properties of the SPEAR applications, such as the security guarantees, functionality, performance and deployment model, as all of these can vary between different secure computation techniques. When choosing the appropriate Protocol Suites for a SPEAR application, an adequate trade-off between these properties should be found based on the requirements of the application. In Section 2.1 we discussed some of the main steps that lead to identifying the application requirements. In this section we describe the properties of Protocol Suites and discuss their potential effect on the applications.

Secure Computation Technique

Secure computation is a cryptographic method for computing a function on confidential inputs while keeping them secure even in the presence of the adversary who is trying to deliberately learn the inputs or affect the computation outcomes. Deliverables D11.1 [107] and D21.1 [95] provide a good overview of the existing secure computation techniques and their properties. Here we briefly reiterate on the types of secure computation and their main differences below and in Table 2.4.

Multi-Party Computation (MPC) The techniques in the MPC category involve multiple parties who collaborate to jointly compute a function over their privately held inputs. Each party may also receive a distinct private output of the computed function. The most common type of MPC techniques is based on *secret sharing*. Secret sharing allows the secret data to be split into *shares* that can then be distributed among the parties. The individual shares do not reveal any information about the secret value. The original value can only be reconstructed if sufficient amount of shares (e.g. all or at least some threshold) is held by a single party. The number of corrupt parties an MPC based technique can withstand is an important parameter

Table 2.4: Secure computation techniques

Technique	Protocol	Parties	Function
Multi-party computation	secret sharing based	multiple parties	agreed on in advance
Garbled circuits	Yao's secure function evaluation protocol, Boolean circuit-based	two parties (garbler and evaluator)	agreed on in advance
Homomorphic encryption	computing function of encrypted inputs results in the encrypted output	two parties	not defined in advance
Trusted hardware	used as TEE	multiple parties	not defined in advance

to consider when choosing such a technique to meet the security requirements derived from the application stakeholder trust and data analysis.

Garbled Circuits (GC) GC is a set of two-party MPC techniques based on Yao's secure function evaluation protocol. The general idea is that the previously agreed upon function to be evaluated is represented as a Boolean circuit. One of the parties, the garbler, encodes his inputs by creating a garbled circuit from the original one. He then sends that garbled circuit to another party, the evaluator, who evaluates the received circuit with her own inputs and the respective keys, and sends the result back to the garbler. In the process, the evaluator is unable to uncover the inputs of the garbler from the circuit.

Homomorphic Encryption (HE) HE is a group of techniques that allow one party to encrypt their input values and let another party to compute a function on the encrypted values. By computing function on encrypted inputs a new encryption with the respective result is created. For example, in an additively homomorphic scheme adding two encryptions together would result in a new encryption of the sum of the values of in the original encryptions. We can distinguish between Partially Homomorphic Encryption (PHE), Semi- and Somewhat-Homomorphic Encryption (SHE) as well as Fully Homomorphic Encryption (FHE). A PHE scheme supports only one type of operation with no restriction on the amount of operations that can be performed. The Semi-HE schemes are essentially additively homomorphic with some limit on the amount of homomorphic additions the scheme can tolerate before the encryption starts to degenerate and can no longer be decrypted. The SHE schemes support arbitrary functions of limited size. Finally, the FHE schemes support the evaluation of arbitrary functions on the ciphertexts. One property that differentiates this type of technique from the others is that the function to be computed does not have to be agreed in advance, as the encrypted values can be reused in different functions, potentially not determined at the time of encryption.

Trusted Hardware As an alternative to software-based techniques, trusted hardware can be used as Trusted Execution Environment to securely compute on private data. This topic is covered in more depth in Chapter 4.

Deployment model

One of the properties that distinguishes the types of secure computation techniques discussed above is their deployment model. We can clearly say that they can have a minimum and a maximum number of supported computing parties. The homomorphic encryption and trusted hardware mostly support one computing party, the garbled circuits support two computing parties, and the multi-party computation can have a minimum of two and maximum of n computing parties.
As such, each technique can fit one of the canonical deployment models of secure computation applications as identified in deliverable D21.1 [95]. These models are: *centralized secure computation*, *distributed secure computation* and *casual secure computation*. A centralized secure computation model is appropriate in situations where all input providers $\mathcal{I}$ can agree on some trusted computing party $\mathcal{C}$ to perform computation. The input providers send their encrypted input to the $\mathcal{C}$ who then performs the computation in a secure domain and finally sends the encrypted results to the result parties $\mathcal{R}$. In a distributed secure computation model a number of computing parties $\mathcal{C}$ cooperate to perform the computation securely. A casual secure

computation is a modification of the distributed model, where a coordinating server is placed as a buffer for network communication, allowing for asynchronous exchange of messages and execution of secure computation. Please refer to D21.1 for more details on the proper choice of secure computation techniques based on the trust relations/assumptions of these canonical deployment model.

Usage model

In Section 2.1 we also mentioned three usage models for secure computation. As not all secure computation techniques are well suited for each type of application model, we provide here a recommendation with that regard.

- *Outsourced computation on one's own data.* This usage model is basically covered by the centralized secure computation deployment model above. Relevant secure computation techniques for this category include techniques that support a single computing party, such as homomorphic encryption, property-preserving encryption and trusted hardware.
- *Outsourced computation on collected data.* This usage model can be implemented with all three deployment models described above. One requirement is that the technique allows combining data from different sources. Examples of suitable secure techniques for outsourced computation on collected data include property-preserving encryption and secure multi-party computation based on secret sharing or garbled circuits.
- *Computation on shared data.* This usage model is better suited for the distributed and casual deployment models, as each data provider is also a computing party and a result party. Hence, the secure multi-party computation based on secret sharing or garbled circuits ideally fits this usage model.

Level of security

The protocols of secure computation techniques are designed to provide security against an adversary that has a certain amount of power to attack the computation. An adversary may control a subset of the parties participating in the protocol and might operate in different ways. Thus, an appropriate level of security should be chosen for the developed application. Below we discuss the main adversary properties to consider (cf. Table 2.5).

Adversary behavior The adversary model defines what kind of behavior an adversary may dictate to the parties that it controls. Aumann and Lindell [14] distinguish three adversary models:

- *Malicious adversaries* (also known as active adversaries) are adversaries that may behave arbitrarily and are not bound in any way to follow the instructions of the specified protocol. Protocols that are secure in the malicious model provide a very strong security guarantee for the user.
- *Covert adversaries* have the property that they may deviate arbitrarily from the protocol specification in an attempt to cheat, but do not wish to be "caught" doing so. Protocols secure in the covert model guarantee that an adversary is caught cheating with at least a defined probability ϵ.

Table 2.5: Adversary properties to be considered

Aspect	Category	Description
Adversary behavior	malicious	may behave arbitrarily, strongest
	covert	may behave arbitrarily but does not wish to be caught, protocols provide probability for adversary being caught
	semi-honest	follows protocol but attempts to learn information, weaker security notion
Computational complexity	polynomial time	adversary can run polynomial-time algorithms wrt. a security parameter
	computationally unbounded	adversary has no computational limits, e.g., can run exponential algorithms as well
Corruption strategy	static corruption	adversary controls pre-defined fixed set of parties, achieves better performance
	adaptive corruption	adversary dynamically corrupts parties

- *Semi-honest adversaries* (also known as passive, or honest-but-curious, adversaries) correctly follow the specified protocol, yet they may attempt to learn additional information by analyzing the transcript of messages received during the execution. Security in the presence of semi-honest adversaries provides a weaker security guarantee, yet might already be sufficient if the adversary is given limited access to the computation, e.g. through defined interface to framework executed in isolation (like trusted hardware).

The adversary model can be determined for each actor and each specific data attribute of the application based the trust and risk analysis described in Section 2.1.

Computational complexity An additional parameter of adversarial power identifies how much computing power an adversary is assumed to have. This defines the difficulty of the problems that it can solve.

- *Polynomial-time* complexity is the common bound used in the theory of computer science for defining efficient computation. It is assumed that the adversary is allowed to run in (probabilistic) polynomial-time wrt. a security parameter, such as the length of cryptographic keys. This means that the adversary cannot perform computations that take super-polynomial time (e.g. exponential time).

- *Computationally unbounded* complexity assumes that the adversary has no computational limits. It can run arbitrarily long brute force attacks and break any encryption scheme that does not have information theoretic security.

Corruption strategy We can distinguish between two main ways in which the adversary might corrupt the parties participating in the protocol.

- In the *static corruption* model the adversary decides before the start of the computation a fixed set of parties whom it controls. The techniques assuming this corruption strategy are usually easier to design and have better performance.

- In the *adaptive corruption* model the adversary is able to dynamically corrupt parties during the computation. This model is more complex to achieve and can have negative impact on performance.

Out of the three properties, the adversary behavior model identifies an adequate level of security, because a higher security level usually has negative impacts on the performance. If more advanced security is required, then the computational complexity and corruption strategy can also be considered in a more complex risk analysis algorithm.

Verifiability and Integrity

Another security related aspect of secure computation techniques to consider when choosing Protocol Suites is their ability to indicate that the computation is actually working as expected and produces the correct results with desired security guarantees.
The secure computation techniques can support different approaches to protocol integrity. One approach is that the protocol is able to detect outside attacks and as a result either attempt to recover or else identify a failure of the protocol.
Another approach is the support for the verifiability of computation correctness. In *Verifiable Computation* the protocols are capable of producing proof evidence (e.g. transcripts) that is sufficient for the external verifiers to validate the correctness of computation results. There are two kinds of verifiability: *public (universal) verifiability* and *designated verifiability*. In the universally verifiable scheme any verifier can perform the verification, while in the designated scheme only a chosen verifier with specific characteristics (e.g. the capability of authenticated interaction) is able to perform the verification.
Not all techniques support these approaches, as they introduce additional complexity to the protocols and negatively affects the performance. For more details on verifiability please refer to deliverable D13.2 [15].

Functionality

Each secure computation technique provides a number of *operations* it is capable of performing on values on certain *data types*. When choosing a suitable Protocol Suite to perform the secure computation subset of the SPEAR application it is important that the Protocol Suite supports all the operations and data types required to express the computation on private data.
In the programmable setting, these operations typically represent fundamental functions (e.g. addition, multiplication, etc.) that are composable into more complex programs and can be thought as building blocks for SPEAR applications. In many cases more complex composable functions can also be available, such as sorting or finding a minimum. In a task-specific setting, the operations may be not composable with other operations, but represent a larger function that performs a rather complex well defined task in a very optimized way. Examples of such operations include private set intersection or private function evaluation. The supported data types may include, but not limited to: signed and unsigned integers of different lengths, floating and fixed point numbers of different lengths and strings.

Performance

Different properties of the protocol suites have different effect on the performance the protocol suites provide to he application. Below and in Table 2.6 we give an overview of the main contributors to the performance of protocol suites.

Table 2.6: Performance contributors of protocol suites

Performance contributor	Impact	Example
Construction of secure computation techniques	performance is affected by the underlying fundamental crypto primitives as well as their combination	secret-sharing based MPC, garbled circuits or trusted hardware; crypto primitives (e.g. MACs, rings/fields) and their properties (e.g. homomorphism)
Level of security	stronger security results in lower performance	adversarial power, adversary's corruption strategy
Deployment model	communication with other parties degrades performance, minimal number of computing parties is preferred	centralized or distributed secure computation
Verifiability	additional work during runtime, negative impact on performance	provides correctness proof and ensures integrity
Operations	operations have performance curve, depends on input size, might be parallelized, specialized protocols might have better performance	implement a function by combining multiple operations or provide specialized protocols (e.g. finding a minimum)
Data types	supported data types can have different performance, smartly converting between them might lead to better performance	data types of shorter length perform better than data types of longer length

Construction of Secure Computation Techniques Secure computation techniques build on top of cryptographic primitives that directly affect how efficient the techniques are in computing a function on encrypted data. First, these primitives have their own cost of computation. Second, the cryptographic primitives have properties (e.g. homomorphic properties) that simplify and make inexpensive some kinds of fundamental operations while making difficult and expensive the other operations. The secure computation techniques make use of this fact by combining a set of primitives with best possible properties in certain ways to achieve the desired outcome. Hence, the way these primitives are combined also largely affects the performance of the final construction.

In practice the *secret-sharing* based MPC is generally quite fast. The operations supported by the homomorphic properties of the secret sharing schemes can be performed locally. Other operations require network communication between the computing parties to be computed. Such operations (protocols) have round complexity proportional to the depth of the circuit being evaluated, which introduces dependence on latency and can be problematic in high latency setups.

The techniques based on garbled circuits are also considered to have good performance. The computation performed by the parties is linear in the size of the circuit and relies on efficient symmetric cryptographic primitives. The computation is also highly parallelizable as the garbling of gates can usually be done independently. Furthermore, the round complexity of GC is constant as it does not depend on the circuit size.

Trusted hardware can be used either for increasing the performance of other types of secure computation techniques or to provide efficient trusted execution environments as alternative

secure computation techniques. In either case, the performance advantage comes from the highly optimized hardware implementation.
The performance of the state-of-the-art protocols as well as the improved protocols of PRACTICE is analyzed in deliverables D11.2 [106] and D13.1 [70] respectively.

Level of security Performance of secure computation techniques crucially depends on the level of protection they provide (i.e. the supported adversarial power). Typically the stronger protection results in significantly lower computational performance.

Deployment model In previous sections we have identified two main categories of deployment models that secure computation can have: *centralized secure computation* and *distributed secure computation*. While these can not be compared directly due to the additional dependence on the concrete technique implementation, their main difference comes from the need to communicate with other parties participating in the computation.
The centralized secure computation enjoys the fact that it typically does not require communication with other parties to compute a function, while in case of distributed secure computation such need exists and forms the basis of related techniques. With larger number of computing parties the amount of communication would typically grow resulting in quick performance degradation. When choosing and configuring techniques based on distributed secure computation, it is advised to use the minimal number of computing parties as possible to achieve the desired application functionality.

Verifiability Verifiable computation typically involves additional work to produce the correctness proofs during runtime as well as to ensure integrity of the computation. Thus, it inevitably has a negative impact on the performance of secure computation techniques that support it.

Operations Each operation supported by a protocol suite has its own performance curve. The level of performance can vary depending on the input size, as different parallelization optimization can be applied and have more pronounced effect with larger inputs.
When specifying the secure computation algorithms one should also pay attention on which operations are available and how these can be utilized to achieve the desired functionality with the best performance. Often there will not be another way, but to implement a function (e.g. finding a minimum) by combining multiple basic operations (e.g. addition, multiplication and comparison) on a higher level. However, specialized protocols for specific functions may be preferable to generic protocols in terms of performance. A protocol suite may provide more complex but better optimized task-specific operations implemented as whole low-level protocols that compute the desired outcomes with much better performance.
Finally, the operations required to implement a specific function may have significantly better performance in a different secure computation technique than the one used for the rest of algorithms. If the conversion cost is small, it may be a good idea to combine multiple protocol suites in a single application to achieve better overall performance and functionality of the application.

Data types The data types supported by a protocol suite can also have different performance. Typically the data types of shorter lengths perform faster than the data types of longer lengths. Hence, incorrect use of data type lengths (e.g. using large integers to hold small values) may lead to performance decreases of multiple times. When minimizing the data types to be used, one

should keep in mind that during the computation the values may grow. Thus, when choosing an appropriate data type the maximum possible values should be considered to avoid overflows leading to incorrect results. By smartly converting between the data types supported by the protocol suite (or even different protocol suites) it is possible to gain better performance of the application.

Resource requirements

Finally, the choice and configuration of a Protocol Suite can to some degree be affected by the memory, CPU and network requirements of the secure computation technique it implements, as the amount of resource consumption directly translates into financial cost. The MPC and GC based techniques heavily rely on the network latency and bandwidth. If well parallelized, they can also use more memory and CPU. The FHE techniques are known to be highly intensive on memory and the CPU, and can easily consume powerful hardware.
It must be noted, that the required amount of resources also depends on the amount of data to be processed. Hence, in the design phase of the application it is a good idea to have an understanding of the potential amounts of data the application will have to process in a limited time frame and test the application with generated test data and various protocol suite configurations to detect the resource requirements of the real deployment.

2.2.3 Constructing SPEAR Application

Once there is a general understanding of the problem statement and application requirements, the next step is to construct the SPEAR application. In D21.2 [22] we have presented the general SPEAR architecture designed specifically for services and applications that need to securely process private data. The deliverable describes the components of a SPEAR application, the processes related with them, their distribution into packages as well as their deployment. As part of the work we also covered multiple examples of how the architecture could be implemented based on specific existing platforms and technology stacks. These examples can be used as the starting point in developing a new application, or new stacks of suitable technologies compatible with each other may be considered. The architecture is generic in nature, but still flexible enough to allow building rather complex applications based on it and even can tolerate a degree of deviations, e.g. by merging or reallocating some of the functionality. The design space for constructing SPEAR applications is wide and while it is mostly driven by the requirements of a specific application, we can still generalize a number of important architectural aspects to consider in the application design.
The SPEAR architecture follows a client-server model where the server hosts the backend application logic (both the general application service and the secure computation) which is consumed by the client frontend according to the client's role. The layers and components of the architecture in D21.2 [22] are presented in Figure 2.2 and Figure 2.3 respectively for the reader's reference.
In order to construct an application based on the SPEAR architecture the following general steps need to be undertaken:

1. Select the DAGGER Engine and Protocol Suites that will power the secure computation on encrypted data. This was covered in Sections 2.2.1 and 2.2.2.

2. Design the general Application Backend that integrates with the DAGGER engine.

3. Design the general Application Frontend allowing the end users to access the backend.

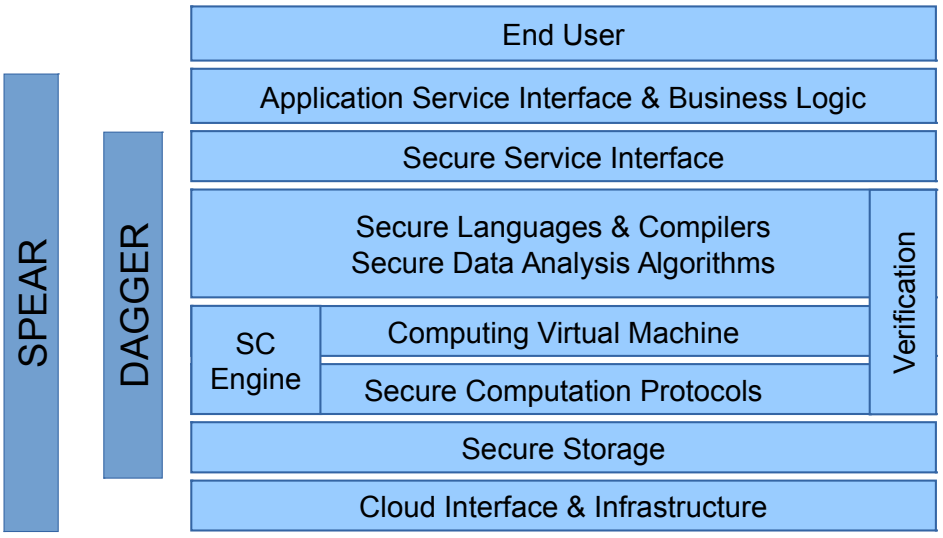


Figure 2.2: The layers of the overall SPEAR architecture.

4. Design the secure computation algorithms for analyzing encrypted data. The algorithm specifications will be executed by the selected DAGGER Engine.

In the following sections we discuss the latter three steps and the design choices related with them in more detail.

Application Backend

On the server side an application consists of a general application service backend and the DAGGER engine that it integrates with. Depending on the choice of the DAGGER platform and the application requirements these components can be either two separate layers or one unified layer including both functionalities. For generality purposes the architecture represents them as separate layers.
Aspects to consider when designing the Application Backend and the DAGGER engine are depicted in Table 2.7 and described in detail below in this section.

Implementation platform The *Application Backend* powers the service logic that the end users access via their client software. We can see three options to implement this component.

- *Web platform.* In the cloud setting it is natural to implement the application backend as a web service accessible via a web browser. Web technologies are widely used because their delivery method is simple and flexible. There is a wide choice of web platforms for implementing the backend, e.g. Java, PHP, Ruby, Node.js or others. All of these provide a full stack of standard technologies and tools supported by large communities. This significantly simplifies and accelerates the development process.

- *Custom software.* Building a custom service software instead of a standard web service is another option, should more customization be desired. However, this option would require to spend additional resources on implementing the client-server communication and other required functionality that may already be included in the existing web platforms.

- *DAGGER engine.* As a third alternative, the whole application backend could potentially be implemented as part of the chosen DAGGER engine, if the latter is powerful enough

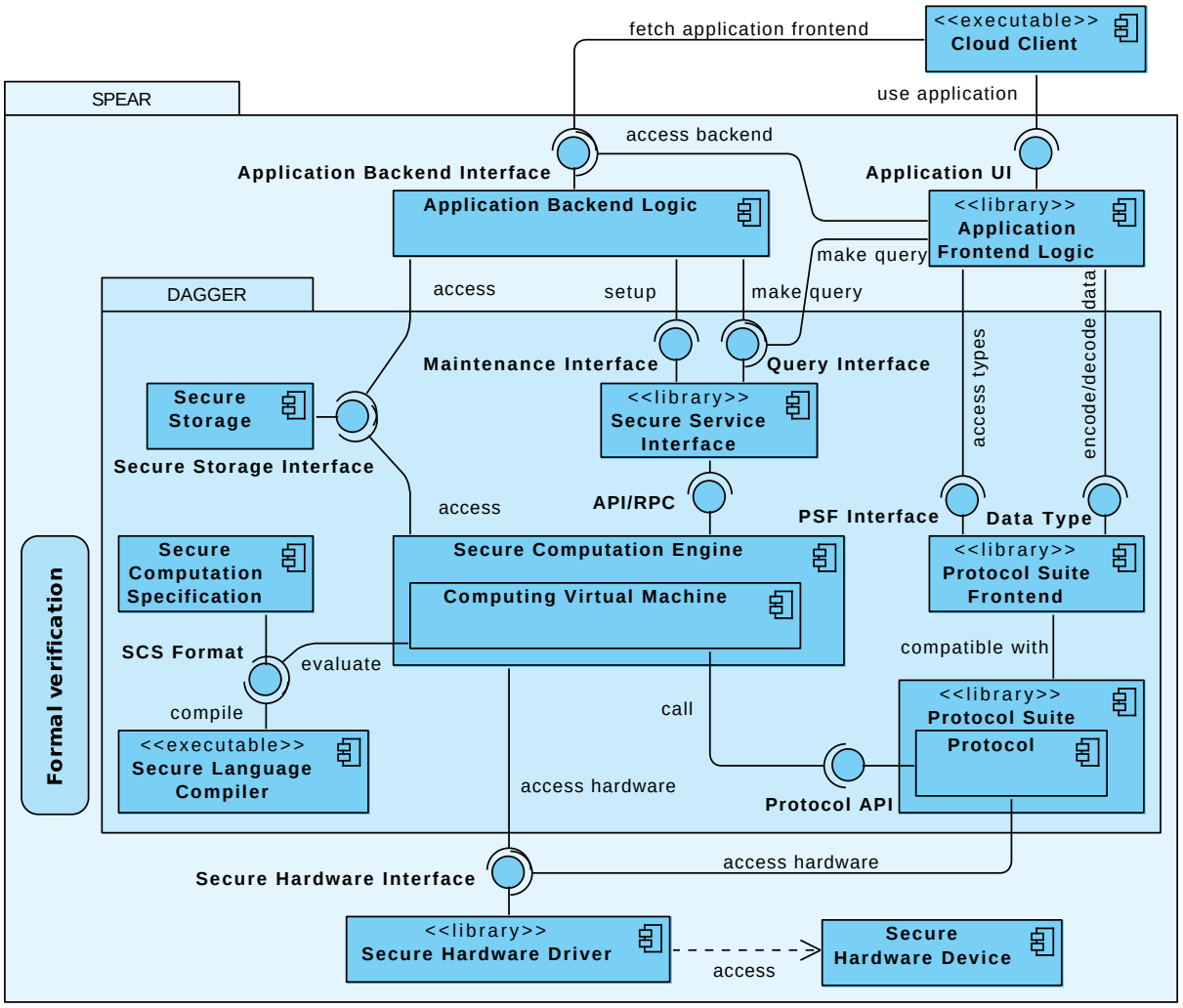


Figure 2.3: The component view of the SPEAR architecture.

to support serving general purpose logic to multiple end user clients (i.e. support session management).

Integration with DAGGER The application backend relies on the DAGGER engine whenever it is required to process private data. The integration with the DAGGER engine is done via the *Secure Service Interface* (SSI) that the application can use in order to invoke the engine. There are three integration options to consider:

- *Single specific DAGGER.* The application may be built using a single specific best suitable DAGGER platform and optimize for it.
- *Multiple specific DAGGERs.* The application may be built using multiple different specific DAGGER platforms and combine their best capabilities in a single application.
- *Abstract DAGGER.* The application may be built for an abstract DAGGER platform by implementing an integration layer for multiple DAGGER platforms. This would minimize the integration surface with the application backend and allow switching between various platforms. However, it may require the secure computation algorithms to be optimized for each platform separately. While this option doesn't always allow to achieve the best

Table 2.7: Aspect to consider when designing the SPEAR Application Backend.

Aspect	Category	Description
Implementation platform of Application Backend	web platform	a web service accessible via a web browser, simple delivery, wide choice, simplified and accelerated development process
	custom software	allows for better customization, requires implementing the client-server communication and other required functionalities
	DAGGER engine	implemented as part of the DAGGER engine, if powerful enough to support multiple end user clients
Integration with DAGGER via SSI	single specific DAGGER	optimize for a single best suitable DAGGER platform
	multiple specific DAGGER	multiple different DAGGER platforms and combine their best capabilities in an application
	abstract DAGGER	an abstract DAGGER platform via an integration layer for multiple DAGGER platforms, minimizes integration surface, allows switching between platforms, does not always achieve best performance but helps leveraging the vendor lock-in problem
Storage engine	handled by DAGGER	better if direct manipulation of encrypted data is required at source
	external general purpose storage engine	preferred if data does not need to be handled privately at source
Participants of secure computation	independent computing parties	computing parties are independent from the input and result parties, this scenario is directly supported by the SPEAR architecture
	end user involved in computation	interaction with end user client is required to compute a function: either take the role of an additional computing party or can be allowed to perform additional interactive operations
Authenticated vs anonymous queries	authenticated	authentication of queries is needed in order to access control the input and result phases
	anonymous	keep the user unidentified, where tokens are not associated with the user but only identify a unique data entry

performance without platform-specific optimizations, it does help to leverage the vendor lock-in problem.

The implementation of the integration layer can be approached from two different directions: a) one could first choose the desired DAGGER platforms and then write an integration layer over their SSI interfaces; b) one could first implement a good abstract SSI interface and then (himself or let the DAGGER platform developers do it) implement

the translation modules for each particular DAGGER platform. The option (a) is easier to achieve than option (b).

Depending on the chosen DAGGER and its configuration, the whole SPEAR stack (the application service, the DAGGER component and the underlying infrastructure) may have to be replicated across multiple SPEAR nodes, each running the same software with minor configuration differences (e.g. keys, ip addresses, the role in the chosen secure computation technique and the amount of computational resources required for that role) and hosted by an independent computing party. This also means that the end user client has to communicate with all the nodes of the SPEAR application, e.g. when providing input or querying for results. In the replicated setups the public data of the application is typically identical across the nodes while the encryptions of data handled by the DAGGER platform are different.

Storage Engine The selection of the *secure storage* depends on the application requirements and the DAGGER capabilities. The storage can be handled by the DAGGER platform natively or an external general purpose storage engine can be used via the application backend. The native support may be preferred if efficient direct manipulation of encrypted data at source is required, as it allows the engine to apply its optimizations based on the data representation in the database. In both cases various ODBC/JDBC, NoSQL and other types of storage options should be considered, since all of them have their advantages for certain types of tasks.

Participants of Secure Computation The general case handled by the SPEAR architecture is that the secure computation part of the application is hosted and performed only by the computing parties and independently from the input and result parties. However, one can imagine an application, where the secure computation application hosted on the cloud requires interaction with the end user client in order to compute a function. Such requirement can arise if a) the chosen secure computation technique like FHE does not have enough key material to perform some operation and needs to interact with the user who has the key; or b) the task-specific protocol is utilized that by design or by purpose assumes interaction with the end user. Section 3.2 describes an example application use case involving the participation of both the server and the client in a secure computation. The SPEAR architecture does not specifically disallow such a scenario. In fact, it can be implemented in multiple ways.

- *Reduce to server side multi-party model.* One way to achieve this functionality is to make the end user to take an additional role of a computing party and host one of the SPEAR nodes. However, this does require that the Protocol Suite with the secure computation technique in question contains the versions of the protocols suitable to be deployed in a multi-party setup with an additional computing party in mind. Otherwise, additional work may have to be undertaken in order to implement/port the protocols in the Protocol Suite to work in a multi-party setup. This option also allows the task-specific protocols to be integrated with the programmable secure computation algorithms.
- *Add interactive client-server operations.* Another option to implement this scenario is by supplementing the Protocol Suite Frontend component with the additional interactive operations besides encryption and decryption, that are expected of the client to be performed during computation. The interactive operations would need to exchange messages with the corresponding DAGGER protocol suite whenever a query is made by the client. The messages can travel either via the whole application stack (frontend, backend and DAGGER engine) until they reach the destination, or take a shorter path by traveling

directly to the DAGGER engine. In the former case each of the three components will have to provide the required software infrastructure for transferring the messages back and forth. In the latter case only the DAGGER component would have to pass through the messages, but this would also require a more advanced DAGGER platform that can handle direct client requests in the first place. In a sense, this seems like the matter of shifting the implementation complexity between the application developer and the DAGGER developer.

Authenticated vs anonymous queries The input and result parties submit their queries to the application over the encrypted channels. In some scenarios the parties need to be authenticated and in others they do not.

- *Authenticated queries.* Authentication is needed in order to access control the input and result phases of the application. For example, the authentication of queries can be performed by the means of a central authentication server. After logging in the user may be granted some kind of authenticated access tokens for performing specific kinds of operations. When the Application Backend receives a query and the corresponding token from the Application Frontend, it can verify the token by bouncing it through the authentication server. Later the token can be invalidated after the required operations have been completed.

  Alternatively, the application can distribute a number of pre-generated access tokens among the list of users, who can then perform the operations related with the token.

- *Anonymous queries.* When it is desired to keep the user unidentified (e.g. gather input data anonymously), the anonymous access tokens or session IDs could be generated by the application for each input query. In this case the token/ID would not be associated with any particular user, but only identify a unique data entry. The tokens can later be used to verify that the entries were submitted to all the SPEAR nodes successfully.

Query communication model

In the data collection phase the private data must be securely collected from the input parties for storage and processing. In the analysis phase, the result parties would like to receive results to their queries. We can think of the following models for organizing these processes.

- *Distributed queries.* An input party can provide the encryptions of its input data directly to the computing parties according to the DAGGER configuration of the SPEAR application. In this case, the input party would connect with the SPEAR node of each computing party separately over secure channels, authenticate with each computing party and securely send the inputs directly to them without any proxies. The results are received in a similar way. This model provides a good level of control over the query submission process. The client receives instant feedback about the status of its query. Furthermore, the private data in the query can be verified and processed at the time of submission, since all the computing parties are online. However, it may require additional measures to be taken in order to synchronize the queries across all SPEAR nodes.

- *Centralized queries.* The queries may also be made in a centralized model, where a single coordinating party first collects the inputs and later feeds them to the SPEAR application for processing. When the results are available, they are passed to the result parties

in a reverse direction. Actually, the whole public backend of the SPEAR application could potentially be represented by that single coordinating node. This resembles the *Casual Secure Computation* deployment model described in D21.1 [95]. The advantage of centralized queries is that the computing parties do not have to be online during the data collection phase or after data analysis phase. The main disadvantage is that it introduces a single point of trust.

The raw data encryptions cannot always be handed to that single node due to the security assumptions of many secure computation techniques. If a single node holds enough of data key material, it may be able to reconstruct the original values which destroys the security. Hence, before sending the input values v encrypted with the chosen secure computation technique T (lets denote these encryptions as $E_T(v)$) to the coordination party, the input party would additionally encrypt them with the public keys of the intended computing parties $E_{\mathcal{C}_{pk}}(E_T(v))$. This way the coordinator will not be able to see the input encryptions, while the computing parties will be able to decrypt the inputs intended for them using their secret keys $D_{\mathcal{C}_{sk}}(E_{\mathcal{C}_{pk}}(E_T(v))) = E_T(v)$ and use the uncovered input encryptions in secure computation. This model requires the clients to authenticate the computing parties.

Having computed a function on private data, the computing parties would need a way to securely send results to the result parties via the coordinator. Although this can be similarly done using an asymmetric keypair, the result party may not always have an asymmetric keypair (e.g. in case of anonymous queries). Hence, a symmetric key can be agreed to between the result party and computing party. When making the query, the result party can generate a symmetric key for each computing party and encrypt it with the public key of the corresponding computing party. This way only the result party and the computing party will know a common key. The computing party can then encrypt the results with the symmetric key it received and send these to the coordinating node.

If both the client and the computing party have an asymmetric keypair, they can both sign the sent data to achieve even more security by allowing the other party to verify who the data came from.

Preserving order of inputs

In a distributed setting the SPEAR application consists of multiple SPEAR nodes, each hosted by a single computing party. When the users make queries (e.g. provide inputs) to the SPEAR nodes, they do so using secure network channels. The network is a complex topology, where the packets are routed between the endpoints in a dynamic way based the distance, ping and other real-time information about the network nodes along the route which can change very quickly. As the SPEAR nodes are deployed in different locations, it can take different amount of time to transfer the packets of similar size from a single client to each particular SPEAR node. In fact, even if a single client repeatedly made a number of similar queries to the same SPEAR nodes, the data might reach the nodes in different time-frames for each query. This would actually hold even if the client was connected with the SPEAR nodes via direct wires, since the CPU load of each physical computer at any given time can differ and thus cause the packets to be processed at different times.
This introduces a problem when several clients attempt to make queries to the SPEAR application, as the queries of different clients may be processed in different order at different SPEAR nodes. In the data collection phase this might result in data entries from a single input query

being stored in different order at each SPEAR node. If in such situation the secure computation were executed, wrong values would have been handed for processing to the underlying protocols, effectively resulting in completely wrong computation outcomes. Hence, additional mechanisms are required to ensure the correct order of entries in the databases of SPEAR nodes. We describe a couple of example solutions below.

- *Use session identifiers and transactions.* When the client connects to the SPEAR nodes, they could engage in a communication round to agree on a session ID for the client before it can make a query. The session ID could also be generated and served to the client by a central authority where the users authenticated himself. Then, all client's queries can be associated with that particular session ID and the SPEAR nodes can use it to make synchronized transactions to the database. In the case with the central authority, the SPEAR nodes may need to verify the session ID with the central authority to check if the ID is valid (i.e. exists and not expired). This logic can be implemented in either the application backend or in DAGGER engine. The option allows to detect any errors related with data entry at the very early stage, but requires more synchronization rounds than the option below.

- *Use data entry identifiers.* Alternatively, each data entry could be assigned a certain unique identifier, either generated by the user client, provided to the user by the application, or somehow derived from the user query. The data entries are stored by the SPEAR nodes in the order they are received, but the entry identifiers are included with the data. Later, in the computation phase, each data entry will have the same unique identifier across all SPEAR nodes. The individual entries could be picked by the identifier, or all the entries could be sorted by the identifier to get the same order across the deployment. Before the computation, it may be required for the SPEAR nodes to communicate to agree on the common intersection (a subset) of entries that exists on all the nodes. If a SPEAR node does not have an entry with an identifier suggested by another node, then it is discarded. This option requires only one synchronization round before the computation, but requires more storage space because of the need to store the identifiers. It also doesn't guarantee that all the data entries will be successfully stored on all the nodes.

Fault tolerance A well designed application is capable of recovering from various unintended behaviors. While there are many such behaviors, some are especially important to consider in case of secure computation applications.
A SPEAR application is built as a distributed service with the chosen DAGGER sub-platform potentially performing joint computation over the network. The crashing of a SPEAR node and network errors can introduce significant problems to the application if the software contains bugs or is not well designed. For example the nodes hosted by other computing parties may become out of sync, locked in a certain state, or even crash altogether. Therefore, the application should be able to restart the crashed or interrupted operations and either a) continue where previously stopped; or b) rollback and restart from a known checkpoint. Transactional processing of data may help deal with such kinds of problems.
The application backend shall also verify the data sent from clients to avoid buffer overflows, SQL injections, cross site request forgery and other attacks. An application processing private data shall follow the best practices for application development.

Application Frontend

On the client side the application can represent different actor roles. A data entry application would implement the behavior of the input parties, while a data analysis application would implement the behavior of the result parties. Hence, the application frontend should have all the required functionality to support both use cases.
A general client application consists of the *Application Frontend Logic* that handles the user interface and communicates with the *Application Backend Logic*, and also the *Protocol Suite Frontend* that handles data encryption and decryption in a way that is compatible with the selected DAGGER engine on the server side of the application. In cases, where the Application Backend Logic is non-existent or is fully implemented as part of DAGGER engine, the client may also contain a *Secure Service Interface* to communicate directly with the DAGGER engine. The application designer has the following design choices related with client applications (Table 2.8):

Implementation platform There are two ways to implement the client application.

- **Web application.** In this case the application is intended to be used with the standard web browsers on PC and mobile devices, and is therefore implemented in languages (i.e. HTML, CSS and JavaScript) supported by modern browsers and platforms. In theory it can also be implemented as a Java applet (or similar alternative), but that would require the user to install an additional browser plugin, which can become problematic due to security policies, user permissions, user skill, platform compatibility and other reasons. The advantage of the web browser-based option is that the application can be dynamically downloaded over the network from the backend service. This significantly improves the ease of deployment as well as the maintainability of client side parts of the application.

- **Standalone client software.** Alternatively, a standalone client can be designed as a self-contained type of software that includes an integrated UI (e.g. Graphical User Interface or a Command Line Interface) and the means for communicating with the backend. A standalone client can be implemented using a range of compiled or interpreted programming languages and therefore provides more customization flexibility (e.g. a custom protocol for communicating with the SPEAR nodes). This option involves installing custom software to the user's device, and as such it allows more mechanisms to ensure that software deployment is performed securely and the software itself is legitimate. However, the ease of deployment and maintainability of the software are greatly reduced compared to the web client option.

Point of encryption The idea of protecting data comes from the need to protect the interests of the data owners. The data should be accessible only to its owner and nobody else. For that reason, all the the data must be encrypted at the owner's control boundary, before it is handed to the application for processing. Depending where the boundary lies we can identify two possible points of encryption in the SPEAR architecture.

- **At the client.** The most common scenario is that the point of encryption lies right at the input party's client. First, the Application Frontend Logic component deployed to the client uses the proper Protocol Suite Frontend component to encrypt the data. Next, the encrypted data is submitted to the application backend or directly to the DAGGER engine for further storage and processing. This model protects the data of each individual

Table 2.8: Aspects to consider when designing the SPEAR Application Frontend and client application.

Aspect	Category	Description
Implementation platform of Application Frontend	web application	used with web browsers, can be dynamically downloaded, improves the ease of deployment and maintainability
	standalone client software	self-contained software with integrated UI, provides more customization flexibility and more secure deployment
Point of encryption	at the client	right at the input party's client, this protects the data of each individual input party
	at the proxy	well monitored proxy encrypts the data before passing it to the application
Bootstrapping trust	trust based on user preference	the user may load the application by accessing the SPEAR node hosted by the computing party of the user's preference
	auditing	auditing the application he receives from any party, potentially multiple nodes and compare to detect lying party
	central trusted authority	central trusted authority with legal or contractual obligations to host initial access point of application
	signed applets	for web applications: run signed code via plugins and verify authenticity
	manual installation	install standalone version, signed by all the computing parties, least flexible option in terms of delivery
Source of randomness	Web Cryptography API	well-established cryptographic PRNGs
	Combine randomness from multiple sources	various techniques are available to seed the software PRNG which then provides large amounts of randomness
Result verification	at Application Frontend	should verify that all the nodes send identical plaintext data
UI design	of SPEAR application	can greatly improve the user's sense of security if designed correctly

input party and fits well with both the centralized and distributed secure computation techniques.

- **At the proxy.** Alternatively, we can imagine a situation, where the control boundary includes more than one client, e.g. an organization with a number of employees. The goal of the organization could be to allow its workers to securely process the corporate data. Hence, it could set up a well monitored proxy that encrypts the data of its employees

before passing it to the application. In a sense the organization with all its employees acts as a single input party. If desired, this option can be further modified to include the Application Backend inside the organization control boundary and let it play the proxy role in communicating with the DAGGER engine hosted on the cloud. An example application based on this model is described in Section 3.4.

Bootstrapping trust The idea of cloud-based applications is that besides their scalability and ease of deployment they can also be easily accessed by the users over the network. The most common type of cloud applications is a web application that is accessed by the user via a web browser. However, there is an inherent fundamental problem with deploying secure computation applications in the web. The initial application code is loaded from a single server with a certain domain, while secure computation may assume a distributed deployment. As web applications cannot be signed, the user must either trust the web server providing the application web page or audit the entire client side code before using it. In a sense, it is a variant of the "trusting trust" problem [127]. Below we discuss some of the options to approach the problem when designing a SPEAR application.

- *Trust based on user preference.* In the distributed setting a SPEAR web application would be replicated across a number of SPEAR nodes, each running identical software. The user may load the application by accessing the SPEAR node hosted by the computing party of users preference. For example, if a computing party is a representative of a group of input parties, then that group would trust such computing party not to tamper with the client software and prefer accessing the application via that party. A worse option would be to try reduce the odds of accessing a SPEAR node under the adversary control by randomly accessing one of the nodes.

  From the technical side each SPEAR node could have either a completely independent domain name, or use a numbered or named subdomain of a commonly agreed domain name. In case of independent domains the known Same Origin Policy [1] of the browsers will need to be bypassed. Fortunately, this can be done in a controlled manner by using, e.g., HTML5 Web Messaging API [2], CORS [3] headers or WebSockets [4] [125].

  Since the initial application is loaded from a single server, then it could potentially lie about the domains or IP addresses of other SPEAR nodes. To leverage this to some extent, trusted certificates are used by each SPEAR node to prove their identity. This would require that the users know in advance (possibly from out-of-band channels) by whom the application is hosted, so they can compare the identity with their knowledge.

- *Auditing.* In a setting similar to the previous one a user may have no preferred computing party to load the application from. In this case the user could try auditing the application he receives from any party. Furthermore, he could try accessing multiple nodes and comparing the application code received from them for any differences (either by hand or using some automated tool). This can help detect a lying party. If no differences are detected, then either any one of them can be used, or none at all, as that would mean that everyone lies. In the latter case the security assumptions of the application would be broken, which defeats the whole point of using secure computation technology and, thus, should never happen.

[1] Same Origin Policy – `http://www.w3.org/Security/wiki/Same_Origin_Policy/`
[2] HTML5 Web Messaging – `http://www.w3.org/TR/webmessaging/`
[3] Cross-Origin Resource Sharing – `http://www.w3.org/TR/cors/`
[4] WebSocket API – `http://www.w3.org/TR/websockets/`

- *Central trusted authority.* An application could rely on a central commonly trusted authority to host the initial access point of the application. For example, it could be the same entity that authenticates the user. It could be independent or audited by all the computing parties of the application. Such an entity can also have legal or contractual obligation to ensure security of the application by participating and auditing efforts.

- *Signed applets.* A possible approach to the trust problem of web applications could be to utilize the ability of browsers to run signed code via plugins. For example, the browsers allow installing a Java plugin and then running signed Java applets loaded from the web. If the client software of a SPEAR application is developed as a Java applet and signed by a trusted party, then the user could verify its authenticity. Also, such an application is packaged in a self-contained jar file, which is easier to verify using alternative mechanisms, such as hash verification.

- *Manual installation.* Probably the most secure way for the user is to install a standalone version of the application. It can be signed by all the computing parties and the application developer after everyone has audited and agreed to its code and even the hardcoded addresses for communicating with the computing nodes. However, this option is the least flexible in terms of delivery.

Source of randomness On the client side the application frontend is required to encrypt the user inputs and this requires cryptographically secure randomness for the security assumptions to hold. One of the problems with the browsers is their ability to generate cryptographically secure randomness. For a long time there was little to no support for such functionality. There exist software implementations of pseudorandom number generators in JavaScript, but the randomness they produce is rather weak for secure computation as they are not seeded with good entropy. The possible options in that regard are:

- Newer browsers support the *Web Cryptography API* [5] that can generate cryptographically random values using well-established cryptographic pseudo-random number generators seeded with high-quality entropy.

- On the browsers that do not support the Web Cryptography API, it is necessary to gather and combine randomness from multiple sources to get good enough entropy that could be used for seeding the software PRNG in JavaScript. While there are various techniques, including based on mouse movement, a good idea would be to ask a small amount of cryptographically secure randomness from each SPEAR node and then, e.g., bitwise xor it all together. The resulting randomness will also be cryptographically secure and good to seed the PRNG, which can then be used for getting larger amounts of randomness necessary for encrypting data inputs.

Result verification The client uses the SPEAR application by making requests and receiving responses. In a distributed setting the requests are made to all the SPEAR nodes of the application. Based on security assumptions of the underlying DAGGER configuration, a number of SPEAR nodes may be expected to behave maliciously. They can also do so unwillingly by error due to developer's fault.
To reduce any errors, the application frontend should verify that all the nodes send identical plain-text data. The encrypted values are difficult to verify, as the encryptions sent by the nodes

[5] Web Cryptography API – `http://www.w3.org/TR/WebCryptoAPI/`

are expected to differ by design. Some assessment can be made by looking at the decrypted value. If a computing party lied, the decryption may result in an unreasonable value.

UI design The User Interface of a SPEAR application can greatly improve the user's sense of security if designed correctly, or destroy the trust otherwise. It has to communicate the security to the user in the best way possible. A good UI would allow the user to distinguish secure data collection from standard web forms, so that the user can identify the moment when his extra attention is necessary. For example, the UI could visually identify the computing parties and communicate the state of input data encryption and processing.

Secure Computation Algorithms

Secure Computation Algorithms represent the business logic that exclusively processes private data by operating with primitives exposed by the Protocol Suites. The algorithms are implemented as Secure Computation Specifications by following a special SCS format that can be understood by the DAGGER secure computation engine. These specifications are then executed by the DAGGER engine. The development of secure computation algorithms is carried out in the following four steps, that should be witnessed by the computing and result parties to build understanding of the joint data analysis task.

1. **Describe the data analysis task.** First, the general problem statement given by the stakeholders should be analyzed, followed by a description of how secure computation systems can be used to solve the problem. The analysis task defines the requirements for the system, e.g. which parties provide what data, how the data is combined and what the results of the analysis are. The choice of a suitable secure computation system can be guided by prototyping the task in plain-text pseudocode, as it helps gaining better understanding of the kinds of algorithms and operations potentially required to solve the problem. The topics related with this step are covered in Sections 2.1.1 and 2.1.2.

2. **Specify the data model.** The data model should be specified, using the descriptions of the input and output data, which can depend on the structure of the existing databases of the input parties. The input data has to be converted into the primitive data types of the chosen DAGGER engine before it can be processed. The data model should also show which data can and cannot be revealed to the secure computation system. More details on this are provided in Section 2.1.3

3. **Develop the business logic specification.** Next, the application logic should be written in the DAGGER language to perform the data analysis task. Since the secure operations are much slower than their public counterparts, it is important to select the right algorithms for each specific analysis task. For that reason the secure implementations may end up somewhat different from the initial plain-text prototypes.

4. **Integrate into the SPEAR application.** Finally, the developed secure computation algorithms have to be integrated into the SPEAR application. The algorithms compiled to the SCS format are first deployed to the DAGGER engine. Then, for each kind of query that the SPEAR application backend is designed to handle, the DAGGER engine must be invoked to execute the corresponding SCS implementation.

2.2.4 Protection of data

In order to securely process data, a SPEAR application relies on the DAGGER platform, i.e. a secure computation engine and its protocol suites, that provide Security-as-a-Service to the application. While their choice is crucial for the security of the final application, the way these technologies are utilized by the data analysis algorithms is equally important. DAGGER encapsulates and abstracts away most of the cryptographic details of the underlying secure computation techniques. However, the application developer is still required to have certain understanding of the base principles and techniques for implementing secure data analysis algorithms in such a way that the processed data is protected at all times. In the following we provide the guidelines for implementing secure computation algorithms based on the DAGGER platform.

Programming model of Secure Computation

The DAGGER engine can be programmed to operate with a multitude of secure computation techniques implemented as Protocol Suites, each representing a unique combination of data representations, algorithms and protocols for storing and computing on encrypted data. A data analysis algorithm can employ these techniques to process data securely. Depending on the application requirements, it may use one specific technique, multiple different techniques or multiple similar techniques with different configurations.
An instantiation of a secure computation technique represents an independent environment for secure computation (also referred to as *protection domain* [21], *secure environment* or *security type*) containing a set of data that is protected with the same resources based on that technique. Thus, a secure computation algorithm would operate with a number of such secure environments, each powered by a specific secure computation technique configuration and securely processing certain private data according to the algorithm. In this programming model we also have an extra environment for storing and processing *public* data, that does not apply any protection and is powered by the DAGGER virtual machine and the underlying infrastructure. Next, we discuss the security aspects of this model with regard to the information and control flow.

Information flow In secure computation algorithms we can clearly distinguish between a number of environments in which the data is processed. Namely, we have an environment for public data and one or potentially more environments for private data. When the data providers encrypt the data, it is transferred from the public environment to one of the secure environments. Then, the data can move from between the environments to achieve certain application goals.
The information flow between the different environments must be strictly controlled because the environments provide different levels of protection and moving data from a stronger to a weaker level of protection is a security risk. Especially undesired is the unintended or too early declassification (release) of information from a protected environment to the public environment. The algorithms should not allow to determine inputs based on outputs. For that reason, all parties must be informed of any points in the algorithm where the data crosses the boundaries of secure environments, so that they can understand what data is changing level of protection (or is being released completely) and assess the risks associated with that. This kind of assessment can be automated by static security analysis tools that can track the information propagation through these boundary crossing points and perform formal verification to ensure the algorithm does not leak more data than desired.

Control flow In traditional computer execution models the control flow of the program is public, i.e. all branching decisions are made based on public unencrypted data. In secure computation there is an additional notion of private data that is handled inside the secure environments. This allows us to discuss two different kinds of control flow in secure computation algorithms.

- *Public control flow.* In this case the algorithm specification (i.e. the list of instructions) is public and is executed by a public VM or a CPU, just like any traditional software. The branching is performed using public values and is therefore fast. The difference from the traditional applications is that the private data is stored in encrypted form inside a secure environment and is processed as such by the corresponding secure computation technique. The secure computation algorithm publicly specifies which secure non-branching operations need to be performed on any particular piece of private data, and the respective secure implementations of the specified operations are invoked by the DAGGER engine when executing the algorithm specification.

  This is a common way to implement MPC applications. It should be noted though, that in this execution model the branching cannot be done based on private data. If it is necessary to branch based on private value, that value must first be made public. However, this may leak some information about the inputs and must be done very carefully: a) on a minimum possible scale; b) only on values with low privacy risk; c) as late as possible, preferably for the final results.

- *Private control flow.* There exist unconventional techniques that allow hiding the decision made based on private data without disclosing any information. On the algorithm level this can be achieved by computing both branches obliviously. A simple mathematical example would be replacing the conditional statement `if (b) x = y; else x = z;` with the formula `x = b * y + (1 - b) * z;`. Depending on the Boolean value of `b`, the variable `x` will receive the value of either `y` or `z`; However, since both branches must be computed and combined using secure computation, branching on private data can quickly become a performance bottleneck and must therefore be used with caution.

  Alternatively, there exist cryptographic techniques that allow evaluating arbitrary private functions. An example of such technique based on universal circuits is presented in Section 3.1. Private Function Evaluation can be useful whenever there is a need to hide the function being computed (e.g. banks not willing to disclose the credit worthiness checking procedure, or Tax and Customs Board not willing to disclose fraud detection algorithms). The cryptographic techniques for hiding functions are usually built as low-level task-specific protocols and can be integrated via the Protocol Suite mechanism.

Maximizing the entropy

One of the security goals is to maximize the entropy (measure of uncertainty) of the intermediate values and the final outputs that are to be made public by the algorithm.

- *Aggregation of data.* One way to increase entropy is to apply data aggregation techniques. Aggregation combines the individual data values into various trends and statistics. This breaks the dependence of outputs on the inputs and makes it much harder to recover the original values from the outputs. The more complex aggregation is, the better entropy it can produce.

- *Large datasets.* The amount of data in the aggregation also contributes to the entropy of results. The larger the dataset, the harder it is to derive original values from the aggregated result. As a nice bonus, more data also ensures statistically more interesting results.

Hiding links to data sources

In many cases it will be necessary to reveal a vector of values indicating some information about the original input entries. For example, one could learn whether or not some entry matches certain conditions or belongs to a set. The problem is that the computing party can monitor who submitted which encrypted entries during the data collection phase, e.g. by looking at IP addresses or authentication information, and may be able to link that new information to the data owner, which is not always desirable.
The links between the data and its provider can be removed during the computation phase by obliviously shuffling (randomly permuting) the encrypted database rows. This hides the initial order of rows and therefore removes the link between the order of input. Sometimes new information about more than one column of a database may be revealed. In these cases shuffling each column separately can also help.

Handling malicious queries

A SPEAR application must also consider protection against malicious queries by result parties, who may deliberately attempt to exploit the query mechanism to learn new information or influence the outcomes. The application developer should consider the following ideas to secure the query mechanism.

- *Access control* of input and result parties can prevent unauthorized access to the application or to certain types of queries. However, it does not protect against authorized parties.

- *Exceptional input detection.* Corrupt input parties may attempt to provide exceptional (invalid) values as inputs. This can affect the computation outcomes or even leak other inputs. One should therefore reduce the dependence of outputs on exceptional values. Since the private values are encrypted and cannot be seen, then the oblivious filtering techniques (e.g. outlier detection) must be used. For example, the comparison operations can be used for finding filters, while the addition and multiplication can be used to apply filters.

- *Minimum amount of entries.* If there are too few input entries in the database, this can be used to exploit simple algorithms, such as sum or mean. This can compromise the privacy of input parties who provided their data in good faith. The queries a result party can make must leak minimum amount of information about the inputs in the database. Hence, limitations should be set on the minimum amount of entries in the database before a data analysis algorithm can be executed by the result party. This relies on the idea we discussed previously, that a larger amount of data provides better entropy of the aggregation result.

- *Differential privacy techniques*, such as mixing controlled random noise into the computation, can be used to reduce chances of deducing inputs from outputs. However, this also reduces the accuracy of outputs.

2.2.5 Performance optimizations

Secure computation techniques have a significant impact on the performance of SPEAR applications compared to conventional applications that compute only on public data. Some techniques are faster than the other, but in general they are all designed to protect data by performing heavy-duty highly resource intensive cryptography. The distributed secure computation is typically mainly bottlenecked by the network communication, while the centralized secure computation relies more on the computational resources. Depending on the techniques employed by the data analysis algorithms the developer must apply appropriate optimization techniques to get the most performance out of the application given its fixed configuration. In the sections below we describe a set of optimization techniques applicable to secure computation applications.

Data parallelism

Data parallelism is a form of parallelization that focuses on distributing the *data* across different processor nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel. Data parallelism is closely related to Flynn's SIMD (Single Instruction, Multiple Data) classification [43].

Communication overheads The MPC based secure computation techniques extensively rely on the network communication to compute secure operations. Efficient use of the network can improve the overall performance of secure computation algorithms. The data is transmitted over the network in packets, each containing a header and a payload. However, the transmission process involves certain overheads:

1. Each packet has a header that has to be transmitted causing bandwidth overhead.
2. Each packet has to be routed to the destination requiring a certain amount of routing decisions resulting in time delays.
3. Each packet involves network protocol processing and device interrupts resulting in time delays.

All these overheads can be reduced by sending few large packets instead of many small packets. This will allow to save up on bandwidth as well as reduce communication rounds. In practice this can mean a significant throughput improvement and consequently a strong performance boost.

Operation vectorization The observation discussed above can be utilized to significantly improve performance of secure computation algorithms by increasing the amounts of data each secure operation processes in parallel. A vector processing technique like SIMD can be applied here. The DAGGER engine can be thought of as a programmable computer with SIMD instructions (assuming it supports vectorized operations). Then, instead of executing many distinct secure operations on single values, the operations should be executed on vectors of values. This kind of vectorization should be practiced thought the algorithm to make all its operations more efficient in the underlying protocols.
In reality the performance boost can be significant even if there is very little to process. Hence, the programmer should choose to implement well parallelizable algorithms. At the same time, vectorization should be practiced smartly, since too extensive use of this method may unwillingly

result in a very large memory footprint. For example, the algorithms that process data in a breadth-first manner will result in larger vectors, while the algorithms with smaller depth will require less iterations to compute and therefore have less communication rounds. One should also consider tailoring the algorithm for better balance between memory requirement and performance [65].

Task parallelism

Task parallelism is a form of parallelization of computer code that focuses on distributing *tasks* – concretely performed by processes or threads – across different processors in parallel computing environments.

- *Local task parallelism.* In a multiprocessor system, task parallelism is achieved when each processor executes a different thread or process on the specified data. A secure computation algorithm could utilize multithreading or multicomputing to compute independent branches in parallel. This can help speed up the local computation.
- *Scalable distributed computing.* The processors in a distributed system run concurrently in parallel. The task can be parallelized by splitting it into smaller independent subtasks and distributing these among a cluster of processors for completion, similar to the MapReduce model. In case of a SPEAR application, the DAGGER part of SPEAR can be replicated onto additional hardware resources and marshalled by the SPEAR application to compute a number of similar or different subtasks in parallel. This technique allows to securely process large data sets in a massively scalable way. An example application utilizing this scalable distributed computing approach is described in Section 3.3.

Data Type and Operation Optimization

Various secure computation techniques can have very different performance levels. In Section 2.2.2 we have already covered how different properties as well as the secure operations and data types of various techniques affect application performance. We therefore refer the reader to revisit that section for more details on the topic. For the sake of completeness and to signify the importance of data type and operation optimizations we here provide the list of key optimizations covered earlier.

Operation optimizations

1. Target efficiency of operations that have most weight in the algorithm.
2. Combine fastest operations from multiple protocol suites.
3. Utilize optimized task-specific operations.
4. Experiment with input sizes to learn the performance curve of the operation.

Data type optimizations

1. Prefer smallest and simplest data types in the data model as these are generally faster. For example, there is no point to hold small numbers in a 64-bit integer type.

2. Combine fastest data types from multiple protocol suites. For example, operations on integers may be faster in protocol suite A, while the operations on floating point numbers are faster in protocol suite B.

3. Reduce complex data values to simpler types, when possible. For example, currency can be converted from floating-point numbers to integers by multiplying by a large enough constant to include enough decimal places after comma. After computation the values can be divided by the same constant.

Efficiency-privacy trade-off

Algorithms can often be optimized at the cost of leaking some information. The general idea involves reducing the amount of private data that has to be computed securely by the data analysis algorithm. Since secure computation is very expensive, a reduced amount of data would have a positive effect on the performance of the algorithm. The data reduction can be achieved by making small controlled leaks of information that is known to have low enough privacy risk, and then discarding part of the private data based on the revealed information. The optimization boils down to finding an acceptable efficiency-privacy trade-off for the algorithm. The applicability of such optimizations largely depends on the particular algorithm.
An example of this optimization would be computing a predicate on a table and declassifying the availability vector to discard the entries that do not satisfy the condition. Otherwise the secure computation algorithm would have to account for the whole availability vector and compute on the whole table, resulting in much higher amounts of data that need to be processed.

Precomputation

Precomputation is the act of performing an initial computation before the run time to prepare or generate some data in advance with the purpose of increasing algorithmic efficiency. We can identify the following points where precomputation can be applied.

- *Preparing unencrypted private data.* The last chance to compute on private data in unencrypted form is at the client side of an input party before the data collection phase. There is no need to compute a function with secure computation if it can be computed much faster in clear text. Thus, the algorithms should be designed such that the computation in encrypted form is minimized and the expected inputs are submitted to the SPEAR application in the most prepared form eliminating the need or possibility to perform any further preparation prior to secure computation.

- *Preparing submitted private data.* Another precomputation idea is that the input data is "prepared" after the data collection phase but before the data analysis phase. This may allow to significantly boost the performance of the analysis phase, especially if many queries are to be made against the prepared private input data. This optimization has been applied in the Tax Fraud Detection example in Section 3.3 where between the upload and risk analysis phases there is also an aggregation phase. Similarly, in the Generic Data Collection Application example described in Section 3.5, there is a separate data preparation phase.

- *Precomputation in protocols.* Precomputation allows a secure computation technique to perform the majority of the expensive computations in an offline phase. Typically, the

offline phase does not depend on the computational task at hand. However, the requirement for precomputing may limit the usability of the protocols in certain scenarios where quick reaction times are needed immediately after starting the secure computation application. Task-specific protocols may similarly benefit from precomputation. For example, the task-specific protocol for malware checking via Private Set Intersection described in Section 3.2 has an offline phase.

Caching

The idea of caching is to store intermediate computation results in memory or in the database in order to allow reusing these results in the future and avoiding their recomputation to increase performance. Since secure computation is expensive, caching should be utilized by both the DAGGER secure computation algorithms and the overall SPEAR application backend to store any secure computation results that may become useful in the future. In a sense, caching has similarities with precomputation, as both are storing some computation results for the future use. However, caching can by applied at any point in time and code where any new computation results emerge.

Optimizing for the underlying protocol

When designing applications based on secure computation, a developer also has to consider which optimizations are or can be included in the underlying secure computation protocol. This naturally depends on the application as well, but depending on the underlying protocol and its optimizations, one can also benefit from optimizing the developed application for the given protocol. This might allow the developer to achieve better performance than without considering the properties of the underlying protocol and optimize the application.
As an example use case, when designing private function evaluation (PFE) using universal circuits (UCs) as described in [72] and in Section 3.1, it can be discovered that a universal 2-input gate, i.e., a gate that can be programmed to compute any gate with 2 inputs and 1 output by one party without revealing the gate functionality to the other party, can be achieved with Yao's garbled circuit protocol with the third of the communication than with the GMW protocol. The reason for this is that in Yao's protocol we can assume that the party who programs the universal gate to compute a specific gate is the garbler and therefore after garbling the gate, he does not have to put additional effort into hiding the gate functionality. However, for the case of the GMW protocol, the functionality of the gates are known to both parties and therefore one needs to design a universal block that can be programmed to evaluate any gate with two inputs and one output. The most efficient such blocks consists of 3 AND and 6 XOR gates and therefore results in at least 3 times more communication between the parties during evaluation for any such universal block or gate.

2.3 Deployment

Having designed and developed a SPEAR application one would have to deploy it to the real world. In deliverables D21.2 [22] and D14.1 [30] the development and deployment model for SPEAR applications and its DAGGER platform has already been covered. In this section we provide some additional practical guidelines with regard to deployment of secure computation applications.

Hosting and delivery A typical SPEAR application would consist of the DAGGER platform (engine, protocol suites, storage) and the SPEAR application (application backend, frontend and the secure computation algorithms written in the DAGGER language. The DAGGER engine, application backend and secure computation algorithms represent the server side node of the SPEAR application and as such are hosted either on a number of cloud providers (one per computing party) or in the premises of the computing parties, where they can fully control the deployment. In the cloud deployments the cloud service providers act on behalf of and in agreement with computing parties. A mix of cloud and private hosting can also be considered. The application frontend must be delivered securely and in a trustworthy manner to the input and result parties. For desktop client applications, this means ensuring the correctness and authenticity of the code by techniques such as code signing. For the web applications it is also important to inform the users of the correct URL.

Security From the security point of view the SPEAR nodes of the application must be guaranteed to be physically well protected from the third parties so that the data encryptions from the node are not stolen. Also, since non-collusion is difficult to achieve technically, personal motivation and contractual/legal obligations shall be seeked as additional countermeasures to reduce the risks. This can be a serious requirement in case of e.g. governmental institutions.
The SPEAR nodes must be interconnected with reliable communication channels (LAN or WAN), that are encrypted and authenticated (e.g. TLS). This is one of the assumptions for most MPC secure computation techniques as well. It must be noted, that the typical network encryption is secure only against computationally bounded adversaries.

Exchanging credentials The computing parties will require a set of credentials in order to be able to authenticate each other over the network. Each computing party must generate itself an asymmetric keypair and securely exchange the public key credentials with the other computing parties. Key exchange can be done via a separate authenticated out-of-band communication channel. For example, the key could be handed over on a personal meeting after having visually identified each other, or in encrypted form over the network. In the latter case the password still has to be exchanged or, e.g., a national PKI infrastructure could be utilized instead.

Agreement During the deployment phase the computing parties have to agree with each other, that they:

1. use the same version of SPEAR application.
2. use the same DAGGER platform and configuration, e.g. the engine and protocol suite versions and parameters.
3. agree to the secure computation specified in the algorithms and in the backend of a SPEAR application.

The first two goals can be achieved by exchanging the digests of the respective software packages and verify the digital signatures of the code. The application can potentially also be programmed to perform such checks.
To achieve the third goal, each party must at least witness the development process. Ideally, each party should also be able to independently audit the application source code and build the final binaries. Auditing can be automated by the means of formal verification tools.

Compatibility A computing party may have certain internal corporate policies which can impose hardware or operating system requirements on the deployment environment. Large companies and institutions can therefore prefer older or less common platforms, which can become problematic due to compatibility reasons. Hence, the deployment requirements should be taken into account early to avoid confusions in the deployment phase.

Testing A good idea is to test the production deployment before using it with real data. The aspects to pay attention to are:

- *Bugs.* Secure computation or optimizations could have introduced bugs during development or deployment. One way to check for bugs is to have the same computation algorithm implemented on plain text and compare if it gives the same results with same data as the deployed secure computation version.
- *Accuracy.* Secure data types may have cut corners to gain performance at the cost of accuracy. It is therefore important to check if the secure application has any accuracy loss compared to the plain text version
- *Performance.* One should also test performance with good generated test data similar to the real one. This is to ensure that no surprises occur after deployment. For example, on a development machine everything could have worked, but after moving to production machine the differences in deployment and environments could negatively impact performance.

Retirement After the application has completed its mission and is no longer needed, it is important to correctly retire it. The data should be securely erased from the machines so that it cannot be leaked or reused later. In the distributed setups it may often be enough erasing the data one one single SPEAR node, since the values of all nodes are required to reconstruct the secrets.

Chapter 3

Example Applications

In this chapter, we provide example applications that were developed on top of the PRACTICE architecture presented in deliverable D21.2 [22] by partners in the PRACTICE project. The design decisions described in Chapter 2 has been made when constructing these applications, the respective partners provide description of the most important aspects considered. The applications presented below are also highly optimized in most scenarios to achieve good performance, which represents the capabilities of the secure computation technologies. In the following sections, we present applications that use secure computation technologies, letting the application requirements speak for themselves.
The applications are first presented in an overview section, after which their architecture is given and possibly details on the implementation is provided. In the end of every section, we draw conclusions about the application. In Section 3.1, privacy-preserving credit checking based on private function evaluation via universal circuits is described. This prototype application uses the ABY and Fairplay secure computation frameworks along with a tool developed for compiling circuits into universal circuits to be able to evaluate functions in a private manner. Section 3.2 presents an application based on a task-specific protocol and its implementation using the ABY framework: a malware checking mobile application between a large database holding known malwares and a computationally weak mobile device with a couple of applications installed. Private set intersection can be used to determine the malwares on the user's mobile device. In Section 3.3, privacy-preserving tax fraud detection using parallel computation is described and is based on the Sharemind secure computation framework. This application has already been deployed in Estonia and uses generic secure computation techniques to analyze value-added tax declarations in a privacy-preserving manner. SEEED-proxy is presented in Section 3.4 that is based on encrypted databases (SEEED) and enforces privacy of the data for cloud based web applications. A generic data collection application, based on the FRESCO secure computation framework, is detailed in Section 3.5.

3.1 Privacy-Preserving Credit Worthiness Checking Using Private Function Evaluation

Private function evaluation (PFE) enables two parties to jointly compute a secret function f, specified by one of the parties on the other party's input x. The party providing the function can also provide an input to the computation by including his input into the function description as a constant. Since the function will be kept secret, his private input is also protected in this way. An often cited example application of private function evaluation is credit worthiness checking

in a privacy-preserving manner. On the one side, a client is applying for a loan at a bank, and on the other side, the bank would like to check the credit worthiness of this customer. This checking procedure, however, is desired to kept private by the bank. This means that the bank's checking criteria becomes the private function that is to be computed on the client's private inputs.
There exist specific protocols designed for PFE with linear complexity in the circuit size based on homomorphic encryption and protocols with logarithmic overhead $\mathcal{O}(n \log n)$ based on oblivious transfer (OT). However, the concrete efficiency of these solutions is not clear since no practical implementation is available. Universal circuits (UCs) and secure function evaluation (SFE) or secure two-party computation provide another efficient solution for PFE. A UC can be programmed to evaluate a circuit up to a given size, where the programming p becomes an input to the universal circuit, i.e., $f(x) = UC(x, p)$. Evaluating a UC with secure function evaluation provides a generic solution for PFE, while UCs can be applied in various further scenarios as well.

3.1.1 Application Overview

Any computable function $f(x)$ can be represented as a Boolean circuit with $x = (x_1, \ldots, x_u)$ input bits. Universal circuits (UCs) are programmable circuits, which means that beyond the true u inputs, they receive $p = (p_1, \ldots, p_m)$ program bits as further inputs. By means of these program bits, the universal circuit is programmed to evaluate the function, such that $UC(x, p) = f(x)$. The advantage of UCs in general is that one can apply the same UC for computing different functions of the same size. An analogy between universal circuits and a universal Turing machine allows to turn any function into data in the form of a program description.
The most prominent application of universal circuits is the evaluation of private functions based on *secure function evaluation* (SFE) or *secure two-party computation*. SFE enables two parties P_1 and P_2 to evaluate a publicly known function $f(x, y)$ on their private inputs x and y, ensuring that none of the participants learns anything about the other participant's input. SFE ensures that both P_1 and P_2 learn the correct result of the evaluation. Many secure computation protocols use Boolean circuits for representing the desired functionality, such as Yao's garbled circuit protocol [132, 80] and the GMW protocol [59]. In some applications the function itself should be kept secret. This setting is called *private function evaluation* (PFE), where we assume that only one of the parties P_1 knows the function $f(x)$, whereas the other party P_2 provides the input to the private function. P_2 learns no information about f besides the size of the circuit defining the function and the number of inputs and outputs.
PFE can be reduced to SFE [3, 113, 105, 76] by securely evaluating a UC that is programmed by P_1 to evaluate the function f on P_2's input x. Thus, P_1 provides the program bits for the UC and P_2 provides his private input x into an SFE protocol that computes a UC. The complexity of PFE in this case is determined mainly by the complexity of the UC construction. The security follows from that of the SFE protocol that is used to evaluate the UC. If the SFE protocol is secure against semi-honest, covert or malicious adversaries, then the PFE protocol is secure in the same adversarial setting.
Efficient constructions considering both the size and the depth of the UC were proposed. The first approach was the optimization of the size by Valiant [129], resulting in a construction with asymptotically optimal size $\mathcal{O}(k \log k)$ and depth $\mathcal{O}(k)$, where k denotes the size of the simulated circuits. In this deliverable, due to the applications of private function evaluation, e.g., diagnostic programs, blinded policies and database queries, we concentrate on the existing

size-optimized UCs and note, that the asymptotically optimal size is $\Omega(k \log k)$ [129, 131].

Applications of Private Function Evaluation

As mentioned before, UCs can be used to securely evaluate a private function using a generic secure computation protocol. [27] shows an application for secure computation, where evaluating UCs or other PFE protocols would ensure privacy: when *autonomous mobile agents* migrate between several distrusting hosts, the privacy of the inputs of the hosts is achieved using SFE, while privacy of the mobile agent's code can be guaranteed with PFE. Privacy-preserving *credit checking* using garbled circuits is described in [48]. Their original scheme cannot represent any policy, though by evaluating a UC, their scheme can be extended to more complicated credit checking policies. [98] show a method to *filter remote streaming data* obliviously, using secret keywords and their combinations. Their scheme can additionally preserve data privacy by using PFE to search the matching data with a private search function. Privacy-preserving evaluation of *diagnostic programs* was considered in [25], where the owner of the program does not want to reveal the diagnostic method and the user does not want to reveal his data. Example applications for such programs include medical systems [16] and remote software fault diagnosis, where in both cases the function and the user's input are desired to be handled privately. In the protocol presented in [25], the diagnostic programs are represented as binary decision trees or branching programs which can easily be converted into a Boolean circuit representation and evaluated using PFE based on universal circuits. Besides, PFE can be applied to create *blinded policy evaluation protocols* [47, 49]. [47] utilizes UCs for so-called oblivious circuit policies and [35] for hiding the circuit topology in order to create one-time programs. Further applications of PFE given in [88] are *evaluation of branching programs on encrypted data* [63] and *privacy-preserving intrusion detection* [96]. Since PFE using UCs utilizes general secure computation protocols, it is possible to outsource the function and the data to two or multiple servers (using XOR secret sharing) and then run private queries on these. This is not directly possible with other PFE protocols, e.g., with the protocol presented in [67] or the homomorphic encryption-based protocols from [88, 89].

Applications of Universal Circuits Beyond Private Function Evaluation

Besides being used for PFE, UCs can be applied in various other scenarios. Efficient *verifiabile computation* on encrypted data was studied in [41]. A verifiable computation scheme was proposed for arbitrary computations and a UC is required to hide the function. [54] make use of universal circuits for reducing the verifier's preprocessing step. In [55], a *multi-hop homomorphic encryption* scheme is proposed that also uses a universal circuit evaluator to achieve the privacy of the function. When the common reference string is dependent on a function that the verifier is interested in outsourcing, then the function description can be provided as input to a UC of appropriate size. In [99, 42], universal circuits are used for hiding *queries in database management systems* (DBMSs). The Blind Seer DBMS was improved in [99] by making use of a simpler UC for evaluating queries, which does not hide the circuit topology. The authors mention that in case the topology of the SQL formula and the circuit have to be kept private, a UC can be utilized. As described in [13], the *Attribute-Based Encryption* (ABE) schemes for some polynomial-size circuits can be turned into ciphertext-policy ABE by using universal circuits. The ABE scheme of [53] also uses UCs.

3.1.2 Application Architecture

In this section, we show how to construct a universal circuit from a standard circuit description and how to program it accordingly. We validate our results with an implementation, creating a novel toolchain for private function evaluation, using two existing frameworks as frontend and backend from the PRACTICE architecture [22, Fig. 4.1] along with our tool. We emphasize that our tool for constructing and programming UC is generic and can easily be adapted to other secure computation frameworks or other applications of UCs.
The architecture of our toolchain for PFE using UCs is shown in Figure 3.1. In this section, we describe its different artifacts and its use of the Fairplay [82] and ABY [34] frameworks. We designed our UC compiler to work with these frameworks as frontend and backend, but it can easily be adapted to other circuit formats and SFE tools as well.

Step 1. Compiling Input Circuits from High-Level Functionality using Fairplay: Due to its easy adoptability, we use the Fairplay compiler [82, 18] with the FairplayPF extension [76] to translate the functionality described in the high-level Secure Function Description Language (SFDL) format to the Fairplay circuit description called Secure Hardware Definition Language (SHDL). The FairplayPF extension already converts circuits with gates with an arbitrary number of input wires into gates with at most two input wires, which is required for Valiant's construction as well. However, in case of Valiant's UC construction, there is another restriction on the input circuit. It has to have fanout 2, i.e., the output wires of all the gates and input wires can only be used as the input of at most two later gates.
In case the input circuit does not follow this restriction, an algorithm places a binary tree in place of each gate with fanout larger than 2, following Valiant's proposition: *,,Any gate with fanout $x+2$ can be replaced by a binary fanout tree with $x+1$ gates"* [129, Corollary 3.1]. This is done using so-called *copy gates*, i.e., identity gates, each of them eliminating one from the extra fanout of the original gate. An upper bound can be given on the number of copy gates. The class of Boolean functions with u inputs and v outputs that can be realized by acyclic circuits with k gates and arbitrary fanout, can also be realized with an acyclic fanout-2 circuit with $k \leq k^* \leq 2k+v$ gates [129, Corollary 3.1]. In [73], we give concrete examples on how this conversion changes the input circuit size for practical circuits and show that in most cases, the resulting number of gates remains significantly below the upper bound $2k+v$.

Step 2. Obtaining the Graph of the Circuit: From the SHDL description of a C circuit with at most two input and at most two output wires for each gate, the graph G^C of the circuit C can be directly generated: with the number of inputs u, the number of outputs v and the number of gates k^* in circuit C, G^C has $u+v+k^*$ nodes and the wires are represented as edges in the graph. The first u nodes in the topological order correspond to the inputs, the last v nodes to the outputs and the nodes in between them to the k^* ordered gates. The resulting G^C graph has at most two incoming and at most two outgoing edges for each node, which is denoted by $G^C \in \Gamma_2(n)$ class of graphs.

Step 3. Generating Edge-Universal Graph U_n: Knowing the number of input bits u, the number of gates k^* and the number of output bits v one can construct the corresponding edge-universal graph U_n, where $n = u+v+k^*$, with our input-output optimization from [72]. We note that no knowledge is necessary about the topology or the gate tables in circuit C for this step. In Valiant's universal circuit construction, two edge-universal graphs for graphs with

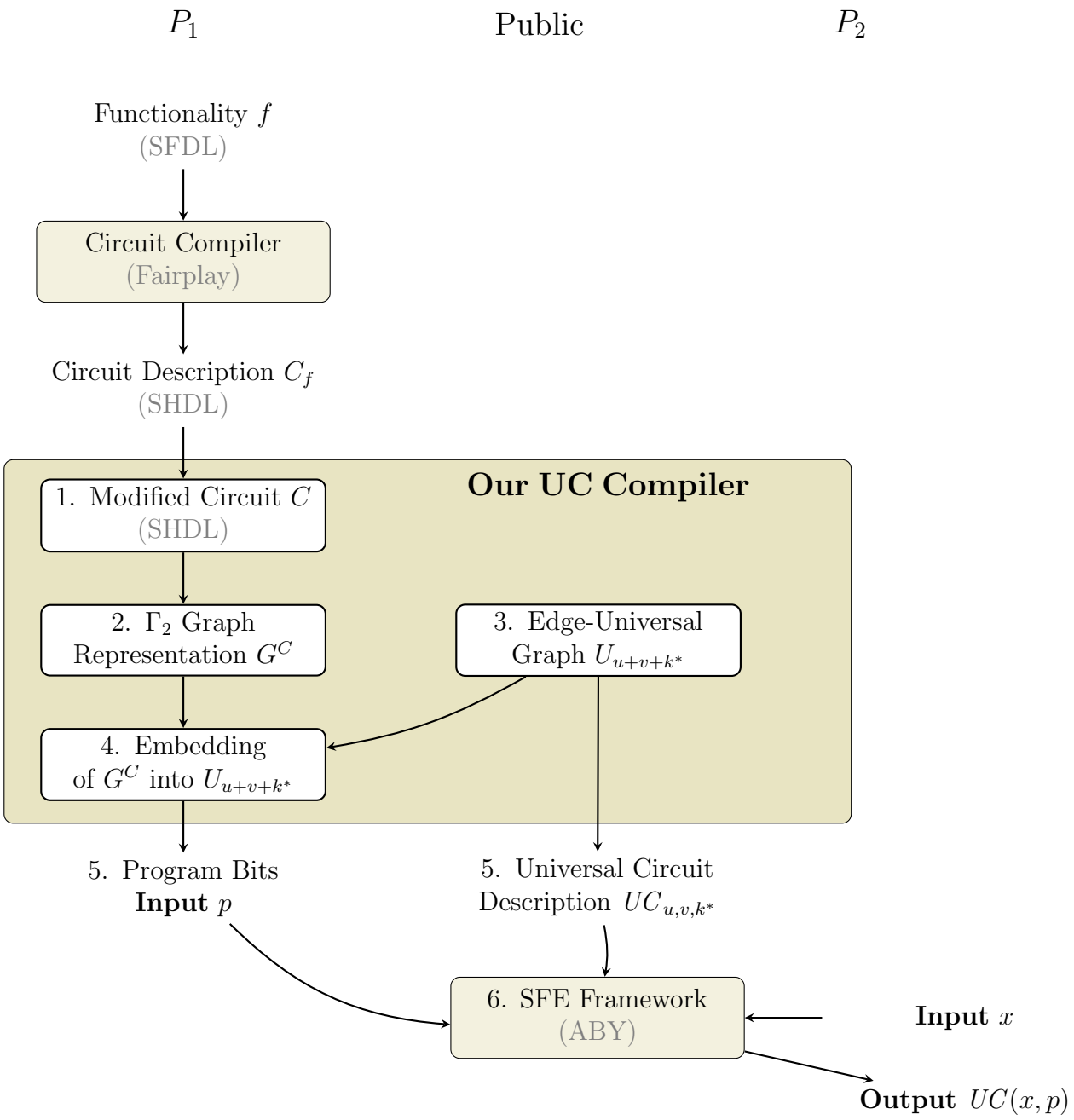


Figure 3.1: Universal circuit compiler and toolchain for private function evaluation.

at most one incoming and at most one outgoing edges for each node, i.e. $U_{n,1} \in \Gamma_1(n)$ and $U_{n,2} \in \Gamma_1(n)$, are merged in order to obtain an edge-universal graph for graphs with at most two incoming and at most two outgoing edges for each node, such that the poles are merged and the edges coming into and going out from them become as follows: the edges in $U_{n,1}$ will be the first input and output for each pole, the edges in $U_{n,2}$ will be the second input and output. For efficiency reasons, we directly generate the merged edge-universal graph, i.e., an edge-universal graph for $\Gamma_2(n)$, with the poles as common nodes.
We include our optimization for the input and output nodes from [72] and Valiant's optimizations for $n \in \{2, 3\}$, but do not consider Valiant's optimizations for $n \in \{4, 5, 6\}$. We include an additional optimization, by means of which we can neglect those extra nodes that have only one incoming edge when the graph is translated into a universal circuit description, since the edges passing through them can be represented as wires.
We note that the edge-universal graph (with undefined function tables and control bits for the universal switches) can be publicly generated. However, the party programming it has to either generate or receive a copy of it for programming the control bits according to the topology of the simulated circuit (i.e., to edge-embed G^C into U_n).

Step 4. Programming U_n According to an Arbitrary G^C Graph: The Γ_2 graph of the circuit G^C with n nodes is partitioned into two $\Gamma_1(n)$ graphs G_1^C and G_2^C which are embedded into the two edge-universal graphs that build up U_n. Valiant proved in [129] that for any topologically ordered $\Gamma_1(n)$ graph, for any $(i, j) \in E$ and $(k, l) \in E$ edges there exist edge-disjoint paths in U_n between the i^{th} and the j^{th} poles and between the k^{th} and the l^{th} poles. Valiant's method is described and the algorithm that uniquely defines these paths in U_n is detailed in [72]. We note that a graph with less than n nodes can also be embedded into U_n, in which case the result of the additional dummy gates are not outputted.
Once the embedding is performed, for defining the control bits, each node x has at most two nodes that have ingoing edges to x, one is represented as the left parent and one as the right parent of x in the edge-universal graph. The two consecutive nodes are also saved as left and right children of x. Now, when x is a switching node and we take edges (v, x) and (x, w) in the path, we save for x if parent v and child w are on the same or on the opposite side in the edge-universal graph. This defines the control bit of each universal switch in the translated universal circuit, where left and right parent and child translate to first and second input and output, respectively. We note that in order to program U_n correctly, we require that if x is the left (right) parent of v in the edge-universal graph, then v is the left (right) child of x as well.

Step 5. Generating the Output Circuit Description and the Programming of the Universal Circuit: After embedding the graph of the simulated circuit into the edge-universal graph U_n, we write the resulting circuit in a file using our own circuit description. In the edge-universal graph, each node stores the program bit resulting from the edge-embedding (control bit c of the corresponding universal switch) and each pole stores four bits corresponding to the simulated circuit (the four control bits of the function table, c_0, c_1, c_2, c_3). Thus, after topologically ordering U_n, one can directly write out the incoming and outgoing wires for each gate into a circuit file and the program bits to a programming file.
Our circuit description format starts with enumerating the inputs and ends with enumerating the outputs. We have universal gates denoted by U, universal switches denoted by X or Y depending on the number of outputs (X with two outputs and Y with one). We replace any gates that have only one input by wires in the UC. The wires are represented in the following

manner:

$$\begin{array}{llll} U & \text{in}_1 & \text{in}_2 & \text{out}_1 \\ X & \text{in}_1 & \text{in}_2 & \text{out}_1 \quad \text{out}_2 \\ Y & \text{in}_1 & \text{in}_2 & \text{out}_1 \end{array} \tag{3.1}$$

denotes that wire out_1 (and possibly out_2) is coming from a gate with input wires in_1 and in_2. The program bits are not represented in the circuit format, but in a separate file, for each universal gate we save a four-bit number representing the control bits and for each universal switch we store a control bit. The output nodes are outputs of Y universal switches and are marked in the end of the file as $O \quad \text{o}_1 \quad \text{o}_2 \quad \ldots \quad \text{o}_v$. The circuit and its programming are given in plain text files.

Step 6. Evaluating Universal Circuits for PFE in ABY: As an example application of UCs, we implement PFE using SFE of a universal circuit. We adapted the ABY secure two-party computation framework [34] for this purpose. Firstly, since ABY uses the free-XOR optimization from [75], we construct universal gates and switches with low ANDsize and ANDdepth. With the cost metric we consider, X and Y gates have the same AND complexity, optimized in [75] and are obtained as

$$\begin{aligned} \text{out}_1 = Y(\text{in}_1, \text{in}_2; c) &= (\text{in}_1 \oplus \text{in}_2)c \oplus \text{in}_1 \\ (\text{out}_1, \text{out}_2) = X(\text{in}_1, \text{in}_2; c) &= (e \oplus \text{in}_1, e \oplus \text{in}_2) \text{ with } e = (\text{in}_1 \oplus \text{in}_2)c \end{aligned} \tag{3.2}$$

with ANDsize and ANDdepth of 1 for both universal switches. X gates are realized with one additional XOR gate compared to Y gates.
Our efficient implementation of generic universal gates uses Y gates yielding

$$\text{out}_1 = U(\text{in}_1, \text{in}_2; c_0, c_1, c_2, c_3) = Y[Y(c_0, c_1; \text{in}_2), Y(c_2, c_3; \text{in}_2); \text{in}_1] \tag{3.3}$$

with $\text{ANDsize}(U) = 3$ and $\text{ANDdepth}(U) = 2$. This universal gate implementation is generic and works in all secure computation protocols. However, for Yao's garbled circuits protocol, one can further optimize it to $\text{ANDsize}(U) = \text{ANDdepth}(U) = 1$, as in some garbling schemes such as the garbled 3-row-reduction [93] the gate being evaluated remains oblivious to the evaluator. After constructing the efficient building blocks, the output circuit file of our UC compiler is parsed, a circuit is generated accordingly and programmed with the input program bits. We conclude that our toolchain is the first implementation of Valiant's size-optimized universal circuit that supports efficient private function evaluation.

3.1.3 Implementation Optimized for Efficient Private Function Evaluation

In this section we describe the challenges we faced while integrating three different tools to perform the secure function evaluation of universal circuits via private function evalution.

Fairplay: Frontend for Translating Functionality

We use the Fairplay framework [82] as a circuit compiler, more specifically, its FairplayPF extension [76], which translates an SFDL function description to an SHDL circuit description, allowing only gates with at most two inputs. The SHDL circuit description starts with enumerating the inputs, which is followed by the topologically ordered gates with their function

tables and their input wires. If the gate is an output gate, it is specified in the same line (cf. Listing 3.1).

Listing 3.1: Small example SHDL circuit

```
0 input
1 input
2 input
3 input
4 gate arity 2 table [ 1 0 0 0 ] inputs [ 1 2 ]
5 gate arity 2 table [ 0 1 0 0 ] inputs [ 0 4 ]
6 gate arity 2 table [ 0 0 0 1 ] inputs [ 4 3 ]
7 output gate arity 2 table [ 0 1 1 1 ] inputs [ 1 2 ]
```

Our UC Compiler

Firstly, we transform the circuit outputted by FairplayPF with fanin-2 and arbitrary fanout into a circuit with fanin-fanout-2, denoted by C. This is an automatic transformation that takes place via adding identity gates into the original circuit. This step is described in Section 3.1.2 as Step 1. Thereafter, the graph of this modified circuit G^C is generated as described in Step 2. The construction of Valiant's universal circuit and the embedding of G^C takes place next, which is the core of our UC compiler.
The output of the UC compiler consists of two files: one for the topology of the universal circuit (cf. Listing 3.2 corresponding to Listing 3.1) and one for the programming corresponding to the input circuit (cf. Listing 3.3). For every X and Y gates, one programming bit is specified while for a universal gate U, an integer between 0 and 15 is given, specifying the function table of the gate using four bits.

Listing 3.2: Universal Circuit

```
C 0 1 2 3
X 1 0 4 5
X 0 1 6 7
X 3 2 8 9
X 2 3 10 11
X 10 4 12 13
X 7 9 14 15
X 5 11 16 17
X 8 6 18 19
X 16 5 20 21
X 12 17 22 23
X 18 15 24 25
X 6 19 26 27
U 24 23 28
X 28 25 29 30
X 22 28 31 32
U 29 32 33
X 33 30 34 35
X 31 33 36 37
X 34 27 38 39
Y 14 35 40
Y 36 13 41
X 20 37 42 43
Y 26 39 44
X 38 40 45 46
Y 42 21 47
X 41 43 48 49
U 45 49 50
X 50 46 51 52
X 48 50 53 54
U 51 54 55
Y 55 52 56
Y 53 55 57
Y 56 44 58
Y 47 57 59
Y 58 59 60
O 60
```

Listing 3.3: Programming

```
0
1
0
0
0
1
1
1
0
1
0
0
1
1
0
2
1
0
0
1
0
1
0
0
1
0
8
1
1
14
1
0
1
0
0
```

ABY: Backend for Function Evaluation

In order to be able to evaluate the UC outputted by our UC compiler, we needed to add a new gate type to ABY, that is, a universal gate. The three gate types of the universal circuit are recapitulated in Listing 3.4, and their size-efficient implementation is detailed in Section 3.1.2.

Listing 3.4: ABY gates for PFE

```
/* Universal gate */
uint32_t output_wire = PutUniversalGate(uint32_t input_wire1,
        uint32_t input_wire2, uint32_t function_num);
/* X gate */
vector<vector<uint32_t> > output_wires = PutCondSwapGate(uint32_t input_wire1,
        uint32_t input_wire2, uint32_t control_bit);
/* Y gate */
wires[tokens[2]] = PutVecANDMUXGate(uint32_t input_wire1,
        uint32_t input_wire2, uint32_t control_bit);
```

3.1.4 Conclusion

In this section, we have presented private function evaluation along with universal circuits as an application of secure computation. Using the universal circuit compiler [72] for translating any functionality into programming bits, i.e., input to the computation, any secure computation framework can be enabled to compute this function, preserving privacy of both the user's input and of the functionality. Therefore, this application is usable in the programmable SPEAR setting. In order to use this application, the function provider only needs to run our UC compiler to retrieve his input corresponding to the desired computation and run the generic secure computation framework for performing secure function evaluation.

3.2 Malware Checking via Private Set Intersection

Private set intersection (PSI) enables two parties to determine the intersection of their private input sets without revealing anything but the elements in the intersection. This means that given two parties with two sets X and Y they are able to compute $X \cap Y$ without revealing any information about the elements outside the intersection. PSI has been widely studied in the literature and different approaches appeared for various application scenarios.
On the one end, the naive approach is that the two parties send cryptographic hashes of their inputs to each other which are then compared. This solution is the most efficient and widely used in real-world applications but insecure due to the low entropy inputs. On the other end, using generic secure computation would result in a secure but inefficient solution. Therefore, more efficient, specific protocols were proposed for PSI.

3.2.1 Application Overview

The application we are looking at is **malware checking** between a large database D and a mobile device M who has a few applications and restricted resources. Assume a company having such a malware database wants to provide a malware checking application to its customers, keeping both its database and the customers' sensitive inputs private. In this scenario, the customer, once having installed a number of applications on his mobile device M, can check if any of these installed applications are in the malware database D or not. An overview of this application is represented in Figure 3.2: here, we only require that the user learns which of his applications are in the malware database.
Using private set intersection (PSI), the intersection of the malware database and the user's applications can be determined without revealing anything other than the intersection of the two sets, i.e., the malwares installed on the user's device. Our aim is to minimize the computation on the user's side and the communication between the server and the mobile device in the online phase.

Proposed Solution The communication complexity of the state-of-the-art solution for PSI [108] is $\mathcal{O}(N \log N)$ where N is the size of the database. This is not desirable in our application scenario due to the restricted power of the client device. We design a solution where the server sends $\mathcal{O}(N)$ data in the preprocessing phase and for each query we have $\mathcal{O}(1)$ communication in the online phase. Our solution for communication-efficient PSI utilizes several efficient protocols and data structures for minimizing the communication in the online phase: 1-out-of-2 oblivious transfers (OTs), Yao's garbled circuit protocol (GC) and counting Bloom filters (CBFs).

1-out-of-2 oblivious transfer (OT) enables two parties to exchange a message obliviously, i.e., the sender sends two messages s_0 and s_1, the receiver sends a choice bit b and receives the message that corresponds to his choice bit, i.e., s_b. The sender does not get to know which message was received by the receiver, and the receiver only gets to know the message corresponding to his choice bit. OT extension protocols [11, 62] allow for the precomputation of a small number of base OTs using public-key operations from which any polynomial number of OTs can be computed using only more efficient symmetric-key operations.

Yao's garbled circuit protocol is a generic secure two-party computation protocol that allows for the secure evaluation of any Boolean circuit. Linear (XOR) gates are evaluated without additional communication or cryptographic operations and therefore the complexity depends on the number of non-linear (AND) gates. Yao's garbled circuit (GC) protocol can be used as follows: the server garbles the circuit by assigning symmetric keys to input wires and encrypting the output wires of a gate using the keys on the input wires. The server sends the garbled circuit to the receiver, who needs to receive the keys corresponding to his own input bits via oblivious transfers (OTs). Then, he can evaluate the garbled circuit by decrypting the wires corresponding to his inputs. In our protocol, we result in an efficient online phase by precomputing GCs offline.

A Bloom filter is a probabilistic data structure that can be used for efficiently testing if an element is in a set or not. An n-bit Bloom filter corresponding to a set with m elements is initialized s.t. all the bits are set to 0. The filter uses k hash functions with range that map each of the m elements randomly into $\{1, \ldots, n\}$. These bits in the Bloom filter are then set to 1. For checking if an element is in the set, we check whether all hashes of the element are set to 1. We note that false positives can occur but false negatives are not possible. A **counting Bloom filter** (CBF) extends the notion of Bloom filters by storing an array of counters instead of a bit for each element. It allows for deletion of elements as well by decrementing the counters.

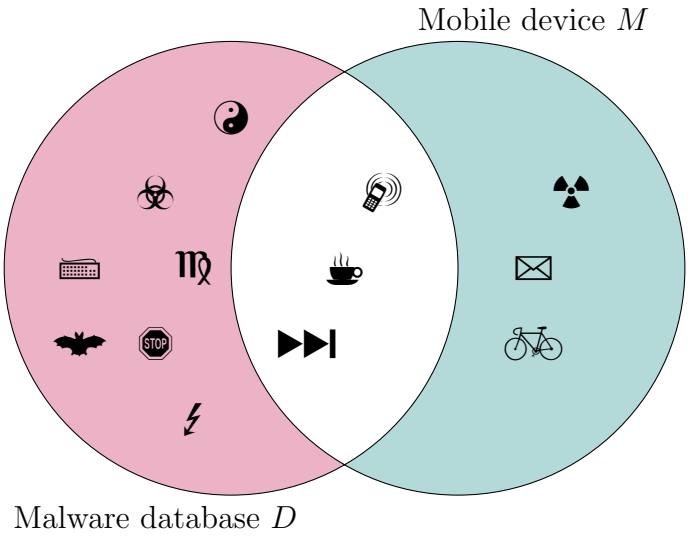


Figure 3.2: Malware checking application

We now detail the high level overview of the offline and online phases of our protocol. Assume that there are N elements in the malware database owned by the server, and the client has a small number of n applications installed on his mobile device.

Offline Phase

With shifting most of the computation in the precomputation phase, the offline phase becomes as follows:

1. The server initializes a counting Bloom filter (CBF).
2. The server generates a 128-bit key k.
3. The server encrypts his N inputs using AES-128 under key k and inserts the encryption of each element into the CBF.
4. The server garbles (at least) n AES circuits.
5. The server sends the CBF and the garbled circuits to the client.
6. The server and the client perform the initial OT phase of OT extension and precompute (at least) $128n$ OTs.

Online Phase

After having the precomputation done in the offline phase, the **online phase** is as follows:

1. The client, for all his elements x_i, performs a 1-out-of-2 OT for each bit to evaluate the garbled AES circuit with the server's input k and his input x_i and therefore retrieves the AES encryption of x_i using the key k.
2. The client, for all encrypted elements $\widetilde{x}_i$, checks if the encryption is in the counting Bloom filter CBF or not. If yes, the application is a malware stored in the database with high probabilty.

3.2.2 Application Architecture

In this section, we show how we implement our PSI protocol using existing secure computation frameworks. The architecture of our toolchain for PSI for malware checking is depicted in Figure 3.3. We describe its different artifacts and its use of the ABY [34] and ObliVM [81] frameworks. For implementing our Android demonstrator, we use the ObliVM secure computation framework which is implemented in Java.

Step 1. Constructing counting Bloom filter CBF and key k As a first step, we generate a 128-bit secret key k (later used for AES-128) and a counting Bloom filter CBF. The size of the empty counting Bloom filter is $1.44\epsilon N \log\log N$, where ϵ is a pre-defined security parameter. The server chooses a 128-bit key k and encrypts each of his elements using AES-128 with this key. Thereafter, he inserts all of these encrypted inputs into the CBF and sends over the CBF to the user.

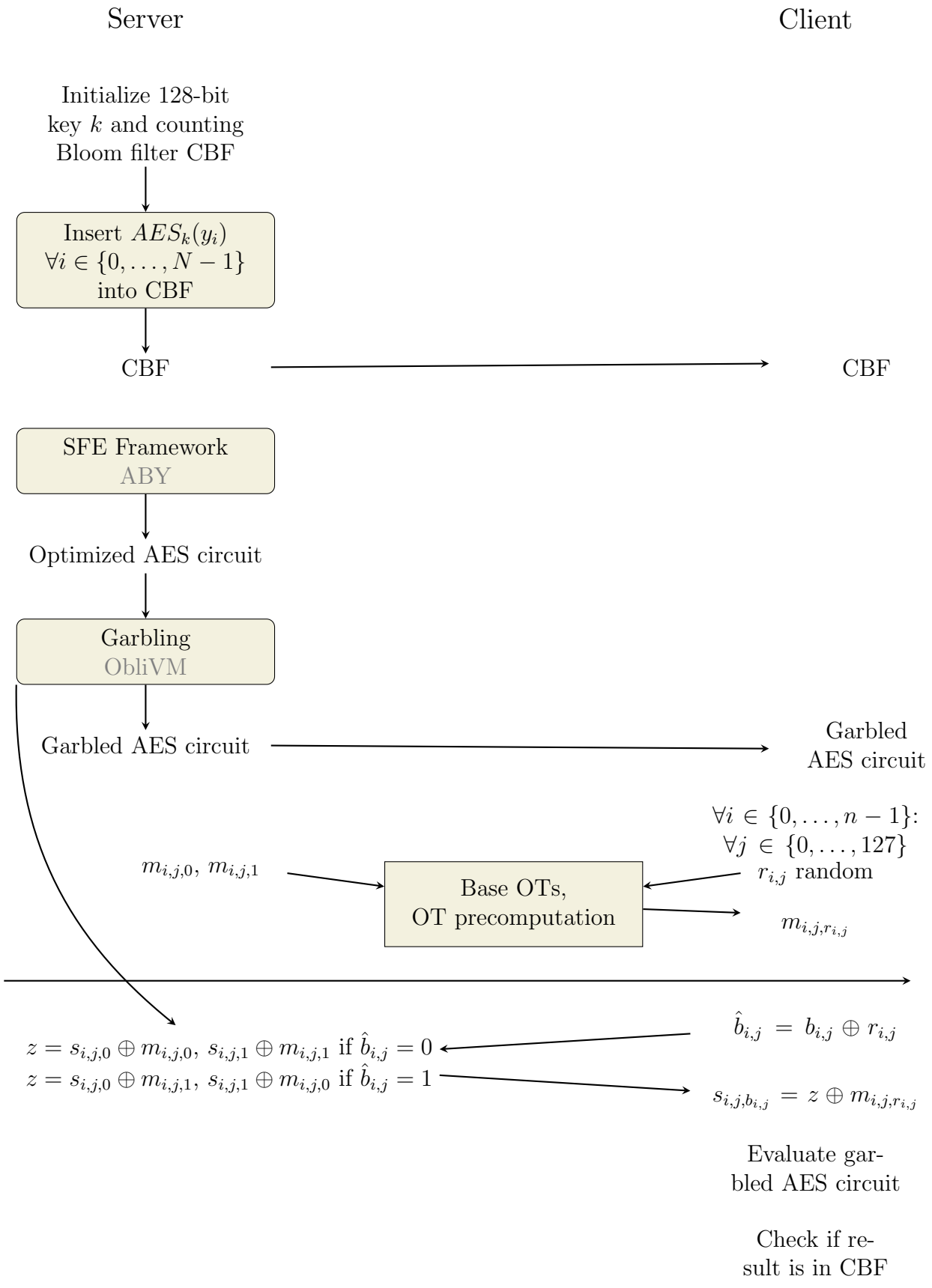


Figure 3.3: Communication-efficient PSI framework.

Step 2. Prepare garbled AES circuit We retrieve the plain AES circuit description from the **ABY secure computation framework** [34]. This AES circuit is highly optimized with 5 440 AND gates. In order to be able to retrieve the AES circuit from ABY, we need to write every generated input, gate and output to a circuit file. We choose the SHDL circuit description format for this and enable ABY to write out any circuit while building the circuit in the framework. After retrieving the plain AES circuit in SHDL format, we read the file in using the **ObliVM secure computation framework** and the server creates the garbled circuit in the offline phase. The garbled labels are then written into a file and are sent to the evaluator (client). For using it in the online phase, the server stores two garbled labels (corresponding to 0 and 1) for each input bit of the client ($s_{j,0}, s_{j,1}$ for all $j \in 0, \ldots, 127$).

Step 3. Precomputing oblivious transfers Still in the offline phase, we compute the base OTs of the OT extension of [62] using public-key operations and then using OT extension we perform the OT precomputation from [17]. For each AES circuit, we precompute 128 oblivious transfers since 128 OTs are needed to be performed in the online phase for each element. The client chooses a random bit r while the server chooses two random masks m_0 and m_1. The parties run an OT protocol using OT extension on these randomly chosen values and the client thereafter retrieves the random mask corresponding to her random bit m_r. We implemented this in the **ObliVM framework** based on the already existing OT extension implementation of [62].

Step 4. Evaluate AES circuit After precomputing 128 OTs, in the online phase the client masks his input bit b with r and sends this masked bit to the server. The server, having s_0 and s_1 messages, sends the receiver $m_0 \oplus s_0$ with $m_1 \oplus s_1$ of the received masked bit was 0 and $m_0 \oplus s_1$ with $m_1 \oplus s_0$ if the masked bit was 1. The client can then retrieve the correct s_0 or s_1 depending on the value of m_r. Using OT precomputation, we eventually perform all the OTs in the offline phase and only XOR operations in the online phase. After retrieving all the keys for her input wires, the client can iteratively evaluate the AES circuit and retrieve the result. Then it is checked if the result is in the counting Bloom filter or not.

3.2.3 Conclusion

In this section, we have presented another application of secure computation. This application however is task-specific, i.e., is usable only in case the parties want to compute the intersection between their two input sets. A special, more-efficient protocol is designed to solve this problem instead of using generic secure computation. This enables two parties, a server and a computationally weaker client to compute their set intersection in an interactive, private manner.
In the final application, the behavior of the parties is the same as for generic secure computation: they input the elements of their private sets to the framework and receive as output the elements that are present in both their sets, i.e., in the intersection. This enables them to compute the intersection privately, they do not get to know any information about the elements outside of this intersection. Our motivating malware checking application scenario is an example use case, our framework can be useful in other real-life applications as well. A possible example is an airline comparing the list of passangers for a given day with the so-called ,,no fly list" of people who are prohibited to travel in or out of the given country.

3.3 Privacy-Preserving Tax Fraud Detection using Parallel Computation

In this section, we demonstrate how the performance of large-scale MPC applications can significantly be improved in some cases by using parallel computation methods inspired by well-known parallel programming paradigms such as MPI [44] and MapReduce [32]. We observe that these methods are also advantageous in the MPC domain for optimizing the communication cost of oblivious algorithms. The cloud environment is ideally suited for performing such computations, since they require a large number of processor cores to run many independent computations in parallel. The elastic cloud allows to seamlessly scale the amount of used hardware to support larger volumes of input data.
We describe the general principles for such parallel computation methods and exemplify their use with a tax fraud detection prototype application [20] built on the Sharemind MPC platform [19]. Our benchmark experiments of the prototype performed in the AWS EC2 public cloud demonstrate unprecedented cost-efficiency in processing data volumes of this size using MPC[1]. Note that the general method described here is agnostic of the underlying MPC primitives and not limited to the Sharemind platform, as we consider optimal scheduling of abstract oblivious algorithms.

3.3.1 Application Overview

Tax fraud detection is described as one of the application scenarios for secure computation in PRACTICE deliverable D12.2. We summarize the scenario also here and discuss a proposed deployment and prototype in the context of Estonia [20].
A tax fraud detection system collects transaction data from companies, which is analyzed by the government's tax authority to detect fraud. This scenario is an ideal use-case for MPC, since honest companies do not want to disclose their private transaction data and the government is only interested in identifying the fraudulent enterprises.
As part of the PRACTICE project, a prototype application that analyzes value-added tax (VAT) declarations in a privacy-preserving manner was built on the Sharemind platform [20]. Sharemind uses a passively secure secret sharing based protocol suite to enable extracting meaningful information from private data while maintaining confidentiality.
A Sharemind deployment consists of many computing servers, each hosted by a different entity. Private data is first secret-shared and then loaded into Sharemind with each server-hosting party receiving a random share of the data. The parties can then jointly perform secure computations on the data, without actually seeing it. Afterwards, the result of the computation can be declassified only if all parties give their consent. Currently, the most efficient protocols on Sharemind require three non-colluding computing parties [19]. If the parties are chosen with clearly non-collusive relations, the direct perception of security for data owners is greatly improved. For the tax fraud detection scenario, a possible deployment model in Estonia is depicted on Figure 3.4.
Risk analysis computations are performed jointly by the different organizations. Instead of sending the VAT declarations directly to the tax board, the VAT declarations are secret-shared between the computing parties. The performed computations are agreed upon beforehand when each party is satisfied that the algorithms do not disclose private information. Auditing and verification methods can be used to ensure that the servers do not deviate from these agreed-upon algorithms [104, 78].

[1] We are thankful to Amazon Web Services for their generous support in performing these experiments.

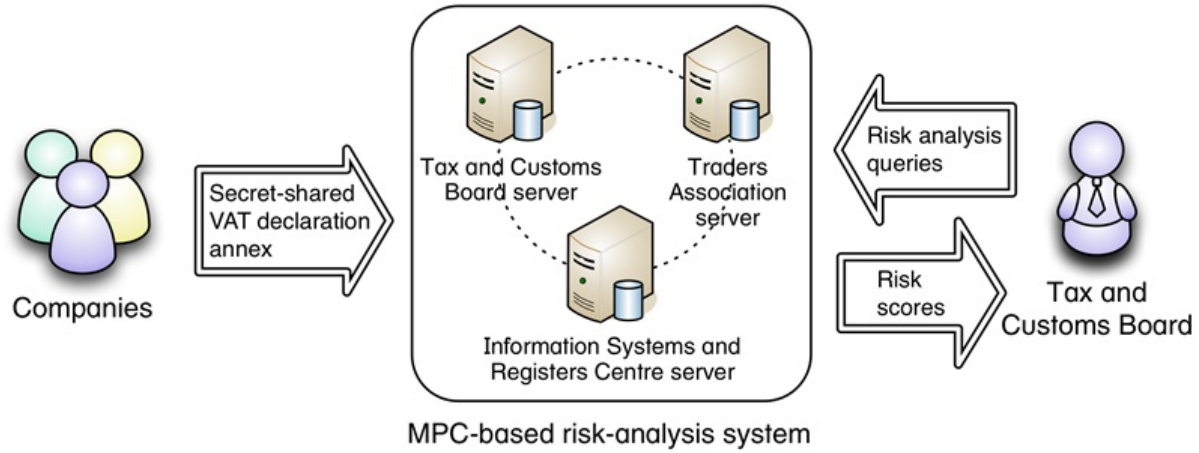


Figure 3.4: Deployment model of a tax fraud detection system using secure multi-party computation

As output of the risk analysis, only the tax board receives the risk scores for companies with suspicion of fraud, and can then investigate further. The transactions of honest companies need not be revealed. The companies maintain a degree of control over their data, since the Traders Association as a representative of the private sector is one of the server hosts. The third server host in this model acts as a neutral party. Alternatively, the application could be set up as a two-party computation between the tax authority and a representative organization of the private sector, using for example efficient two-party secure computation methods described in PRACTICE deliverable [70].

3.3.2 Application Architecture

The general idea common to all parallel computation techniques is to divide the input data into a number of smaller blocks that can be processed independently in parallel. The intermediary results of these subtasks can then be used in further computation or combined to a final result. This simple idea can be used to greatly enhance the running time and reduce the total communication cost for various oblivious algorithms.
We used this idea to significantly speed up aggregation of transaction data in the tax fraud detection application. In our prototype, the whole computation process is divided into three distinct phases:

1. **Upload** – the secret-shared tax declarations are uploaded into Sharemind and initial data validation is performed.

2. **Aggregation** – the data from each declaration is aggregated to enable risk analysis to be performed very efficiently. This is the most computation-intensive phase, however, data from each declaration can be processed independently, which allows for a high degree of parallelization.

3. **Risk analysis** – The results of the parallel aggregation are merged into a single large analysis table on which the risk analysis algorithms are performed. The output of this phase is the list of companies' registry codes with suspicion of fraud.

The computation process is also depicted graphically on Figure 3.5.

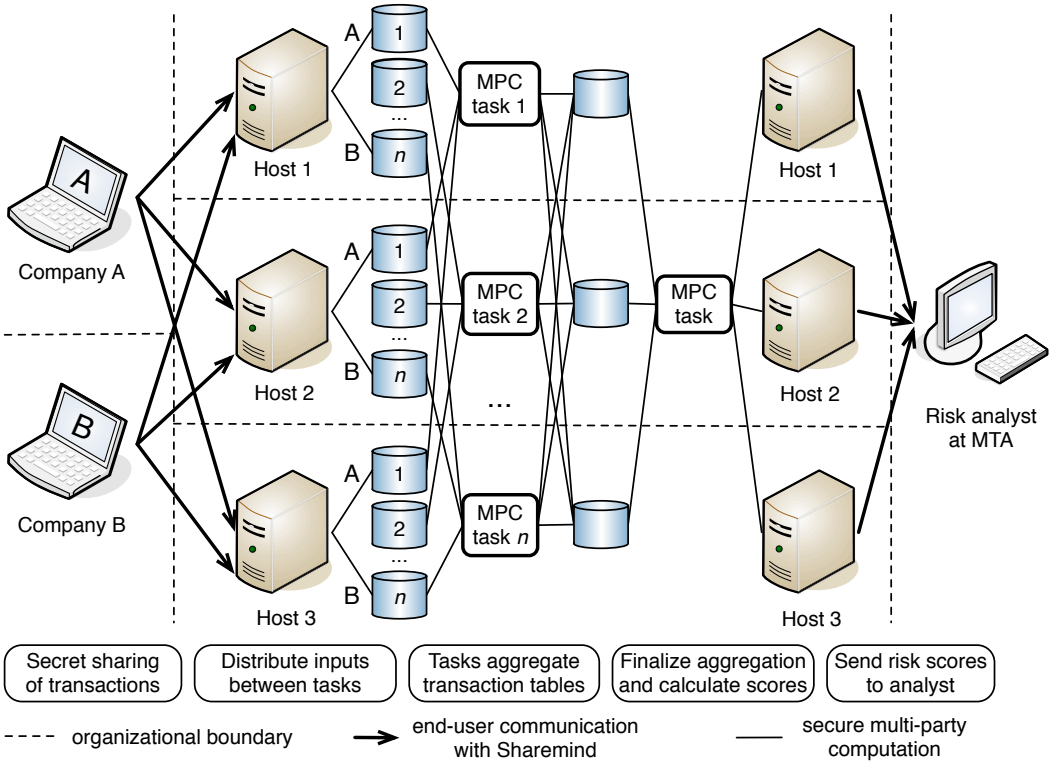


Figure 3.5: The performed computations in the prototype MPC tax fraud analysis system

The most costly computation that is performed in the aggregation phase is the oblivious aggregation of transaction sums for all unique pairs of transaction partners, essentially a GROUP BY query in SQL terms. In general, we can describe the computation on a database of two column vectors containing pairs (k_i, x_i), where $k_i \in K$ are the key column values and $x_i \in S$ are the data values that need to be aggregated. That is, we want to aggregate the x_i values that have a common key value k_i. The result is an aggregated table with pairs (k_j, y_j), where each $k_j \in K$ is unique, and the corresponding y_j value is an aggregate of x_i values, whose key value is k_j. Different aggregation functions can be considered besides summing the values, such as computing the average or finding the minimum/maximum element.

This kind of grouped aggregation is very natural to divide into independent computational tasks that aggregate a subset of the data. The intermediary tables can be later concatenated and aggregated to the final result. Moreover, if the data can be divided into disjoint subsets according to key values, then the final aggregation can be omitted as the intermediary results can simply be combined to get the final aggregated table.

Using this approach, we divided the transaction data of different companies into separate database tables already in the upload phase. The aggregation phase could then be run in parallel on these separate data sets. For the final risk analysis phase, the aggregated tables were concatenated into a single large table, on which the risk analysis algorithms were performed, as described in [20].

There are two reasons why this parallel approach is gainful in terms of performance in the MPC setting. First, if the round or communication complexity for the algorithm is superlinear, we can reduce the total complexity by running the computation on smaller batches of the input. Also, the independent parallel subtasks can be divided between different physical nodes, which allows to use more network links for the whole computation, increasing the available bandwidth

capacity. This is especially relevant for MPC as communication tends to be the bottleneck for most state-of-the-art protocols. Naturally, scaling the number of computation nodes is much easier to do in a cloud computing environment.
In our cloud experiments running the tax detection application, we also noticed that independent Sharemind computation processes running on separate threads but on the same physical nodes were still able to leverage the available network capacity more efficiently than a single process using SIMD parallel computations. This is due to the fact that parallel processes can dynamically interleave their local computations and network communication, such that the network link has a constant high load. This property is more relevant for secret sharing based protocols, as the ones used in Sharemind, since they have a higher round-complexity but overall lower communication cost compared to constant-round protocols, such as Yao's garbled circuits protocol [80].

3.3.3 Cloud Deployment

We now describe how we deployed and benchmarked our prototype in the AWS EC2 environment. We estimated that the first version of the prototype system could perform risk analysis on a month of Estonian economy in 10 days using about €20 000 worth of hardware in a local deployment. Following the interest from the PRACTICE advisory board on assessing the viability of performing these computations in the cloud, we performed some algorithmic optimizations to the prototype and prepared it for a large-scale cloud deployment.
In the cloud setting, the computation servers would still be managed by the same three non-colluding organizations, however the actual physical servers would be hosted by one or many cloud service providers (see Figure 3.6).

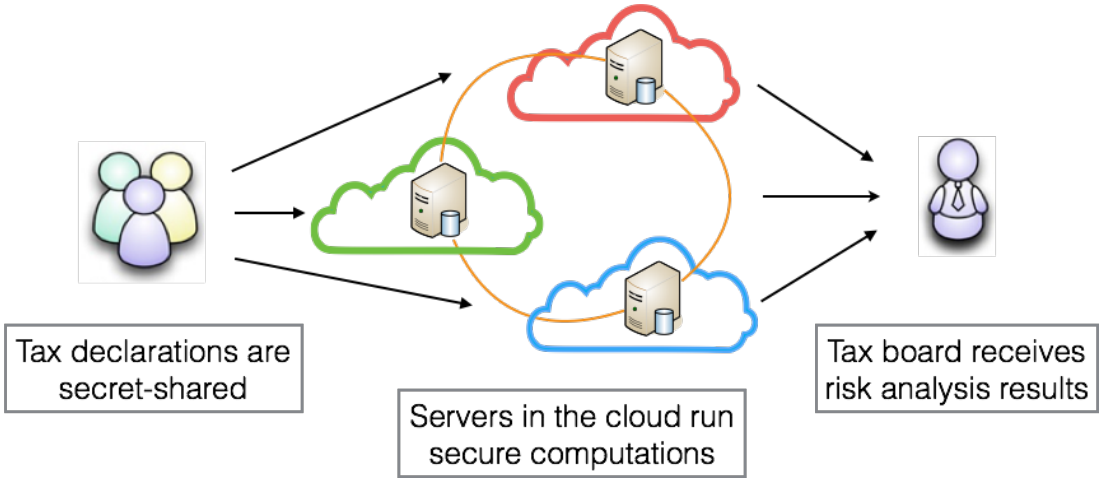


Figure 3.6: Tax fraud detection system deployed in the cloud. Different party's servers are hosted by one or many cloud providers.

The confidentiality of the data is similarly protected against the organizations managing the servers. However, additional trust assumptions about the cloud providers need to be made. Using different cloud service providers for hosting each party's servers provides the best security guarantees, but is not ideal for performance. Different possible deployment models are described below in Table 3.1.
The different models offer a trade-off between security assumptions and performance. In the future, secure hardware solutions such as Intel SGX could help reduce the trust assumptions that need to be made about the cloud provider as described in Section 4. In our benchmarks

Table 3.1: Descriptions of different possible cloud deployment models

Cloud deployment model	Possible attacks	Security assumptions	Performance
Single cloud provider – all Sharemind servers are hosted by the same cloud provider	If the cloud provider has access to all computing servers, it can read all the private data	The cloud provider must be trusted not to access the data on the servers	The servers can be connected in a LAN, offering the highest performance
Two cloud providers – two out of three parties host their servers using one cloud provider, the third party uses a different cloud	The cloud provider hosting two servers can deduce the private data over time by reading the contents of encrypted network communication of both servers	The cloud provider hosting two party's servers must be trusted not to access the private keys of the servers' communication channels	Performance degrades due to latency as the physical distance of the two clouds increases
Three cloud providers – all parties use different cloud providers	If two cloud providers collude and monitor communication they can deduce the private data over time	The cloud providers must be non-colluding	Performance degrades further due to latency between all pairs of servers

on AWS EC2, we simulated all three of these models by running the computation servers in different geographical EC2 regions to introduce latency.
To allow for a high number of parallel processes in the aggregation phase, we deployed each of the three party's servers as a group of four EC2 instances, totaling in 12 computing instances. Each set of 3 instances were running 20 Sharemind processes in the aggregation phases, whereas the risk analysis phase uses only a single process. An additional instance acted as the client that uploaded data into the computing nodes. Figure 3.7 illustrates this instance deployment using two EC2 Europe-based regions.
For Sharemind and many other methods of secure computation, a fast network connection is critical for performance. Thus, we chose to use EC2 c3.8xlarge instances in all our benchmarks, since it was the cheapest instance type having a 10 Gigabit network connection at the time of our experiments, and also supported Amazon's Enhanced Networking technology, improving overall network performance[2]. The number of instances and parallel processes to use was then estimated by profiling the application with the largest used data set in a local deployment.
To be able to to compare results, we used the same instance type setup in all the benchmarks. In Table 3.2, we summarize the different regional settings that we benchmarked. These correspond to the cloud deployment models described in Table 3.1.

3.3.4 Benchmark Results

We used three input data sets with different size in our benchmarks (see Table 3.3). The largest data set corresponds to the estimates of Estonia's Tax and Customs Board on the number of

[2]The c3.8xlarge instance was one of the largest EC2 instance types at the time for computation-intensive applications with each instance having 32 CPU cores and 60GiB of memory. It supports also Amazon's Enhanced Networking, which is claimed to increase packets per second performance and reduce network jitter and latencies for the instance, which is advantageous for secure computation performance.

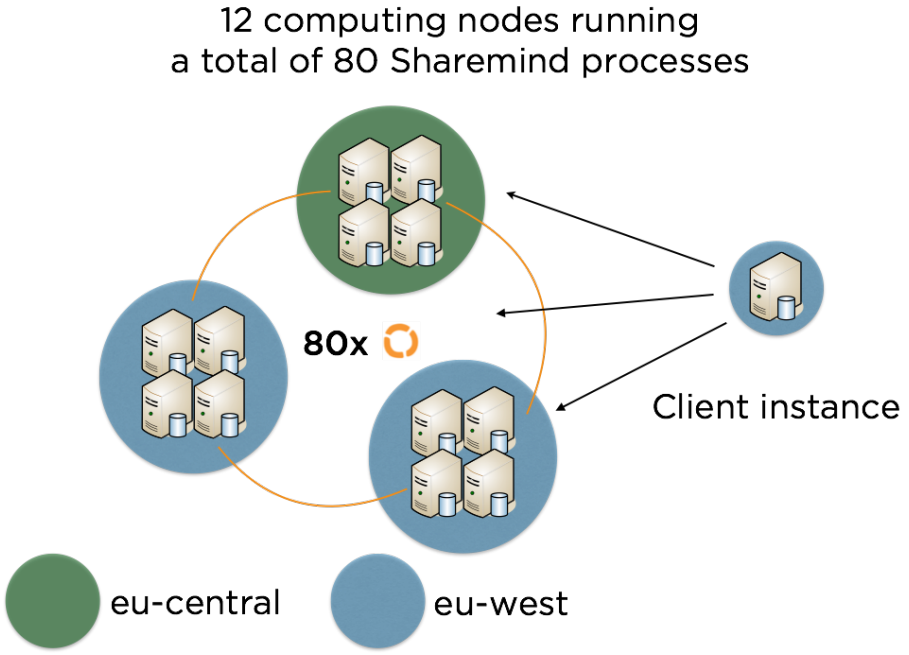


Figure 3.7: Amazon EC2 deployment within 2 Europe-based regions using a total of 80 Sharemind processes to aggregate data in parallel

Table 3.2: The three regional instance deployments used, modelling one or many cloud providers

Regions	Client	Computation servers	Latency (round-trip)
1	us-east - c3.8xlarge	us-east - 12x c3.8xlarge	< 0.1ms between all nodes
2	eu-west - c3.8xlarge	eu-west - 8x c3.8xlarge eu-central - 4x c3.8xlarge	< 0.1ms between eu-west nodes 19ms – eu-west, eu-central
3	us-east - c3.8xlarge	us-east - 4x c3.8xlarge us-west - 4x c3.8xlarge eu-west 4x c3.8xlarge	77ms – us-east, us-west 133ms – us-west, eu-west 76ms – us-east, eu-west

taxable persons and performed business transactions in one month in Estonia. Each company's tax declaration is an XML-file consisting of a summary report for the current taxation period and a detailed list of all sales and purchase transactions performed with different business partners.

Table 3.3: Descriptions of the three data sets used in the benchmarks

No. of companies	No. of transaction partner pairs	Total no. of transactions
20 000	200 000	25 000 000
40 000	400 000	50 000 000
80 000	800 000	100 000 000

In the upload phase, declarations were uploaded to the 80 Sharemind processes, each process receiving a single declaration at a time. After aggregating the data, the results were moved together into a single process running on three instances, and the remaining instances were closed. Note that each party only moves data shares between instances that it controls. The single process then merged the data and performed the risk analysis computations.

The running times of all computations are presented on Figure 3.8. The performance of the prototype has significantly improved compared to the earlier version and is well within practical limits as the analysis only needs to be performed once in a single tax period (each month). We were unfortunately unable to perform the benchmark with the largest data set in the multi-continent deployment due to some technical synchronization issues that occurred in the high-latency environment.

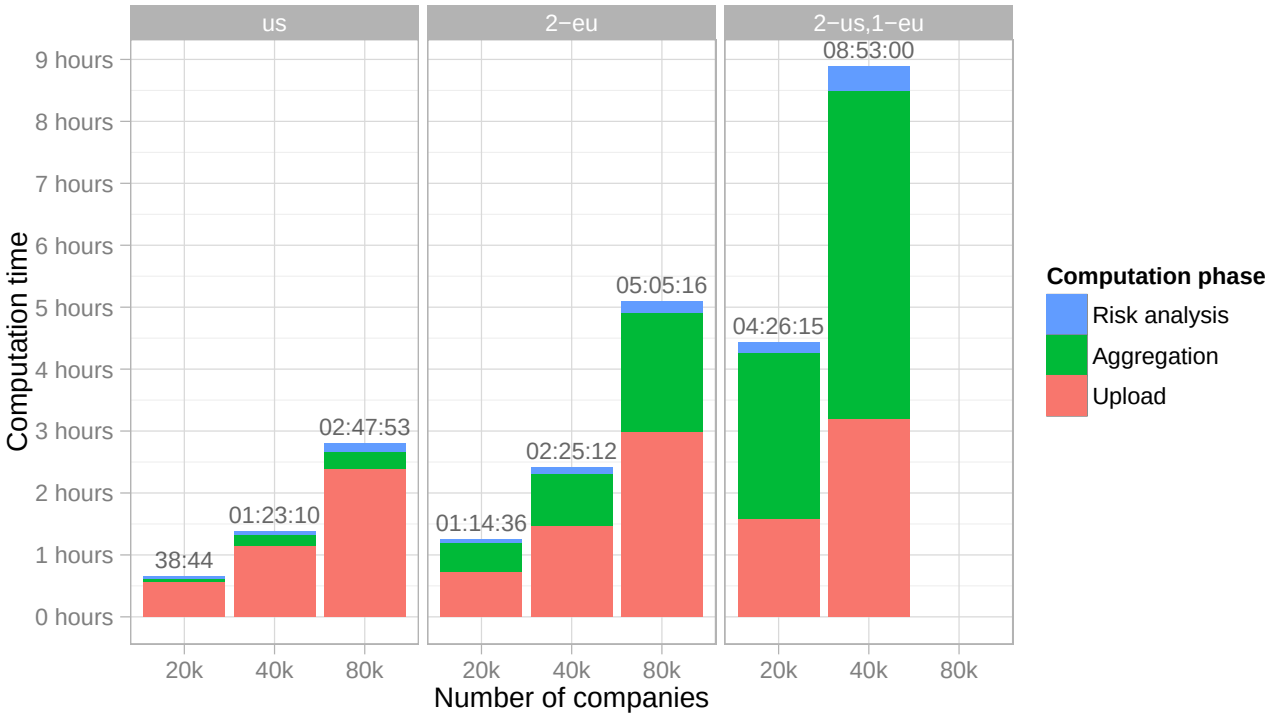


Figure 3.8: Running times of the computations in different deployments and varying amount of data

As can be expected, in multi-region deployments the computations are slower due to the increased latency. The aggregation phase is affected most, as the bulk of the computations are done there. Upload times are also affected since some secret-shared data validation is required. The risk analysis itself is very fast. However, the optimized algorithm introduces some admissible leakage. In particular, for every company that is flagged with potential fraud, we leak the number of transaction partners that the company reported in the current period. This information might be used to identify a specific company, if it has a very unique number of business partners. The computing parties would then learn that this company has been flagged with suspicion of fraud. Ideally, only the tax authority should get this information.
The leakage occurs since the risk analysis procedure requires to also flag all the business partners of a suspicious company. We also implemented a more brute-force approach, that obliviously iterates over the entire analysis table of companies when flagging the business partners to hide their amount.
The total cost of a single run of the entire computation is very low for a privacy-preserving computation of this scale (see Table 3.4 and Table 3.5). Total communication refers to the total sum of one-way communication between all computing instances during the whole computation, measured by incoming network messages. Instance costs are calculated by charging to the full hour separately for the parallel phases (upload, aggregation) and the risk analysis step.

The depicted costs include the price for running the instances and also data transfer between different EC2 regions (communication within a single region is free). Data transfer costs become increasingly important in multi-region deployments, forming up to 12% of the total cost. In a real-world deployment, these costs would also be higher if the servers are hosted by different cloud providers, as communication between AWS regions is cheaper than communication to the public Internet. The depicted costs do not reflect expenses for data storage, which would be added for a persistently deployed system that stores all data from previous periods.

Table 3.4: Running times, total exchanged communication and running times of benchmarks using the fast risk analysis algorithm

Deployment	**Input data size**	**Total communication (GB)**	**Data transfer cost**	**Total time (hour:min:s)**	**Instance cost**	**Total cost**
us	20k	290.5	-	38:44	$26.88	**$26.88**
us	40k	587.8	-	01:23:10	$48.72	**$48.72**
us	80k	1202.2	-	02:47:53	$70.56	**$70.56**
2-eu	20k	307.1	$3.99	01:14:36	$56.82	**$60.81**
2-eu	40k	619.4	$8.05	02:25:12	$82.28	**$90.33**
2-eu	80k	1264.0	$16.43	05:05:16	$133.21	**$149.63**
2-us,1-eu	20k	308.1	$6.13	04:26:15	$119.11	**$125.25**
2-us,1-eu	40k	625.5	$12.46	08:53:00	$210.18	**$222.64**

Table 3.5: Running times, total exchanged communication and running times of benchmarks using the risk analysis algorithm with total privacy

Deployment	**Input data size**	**Total communication (GB)**	**Data transfer cost**	**Total time (hour:min:s)**	**Instance cost**	**Total cost**
us	20k	1324.8	-	02:55:40	$36.96	**$36.96**
us	40k	4744.0	-	09:29:57	$89.04	**$89.04**
us	80k	17859.1	-	33:34:07	$221.76	**$221.76**
2-eu	20k	1383.2	$17.44	22:38:25	$180.46	**$197.90**
2-eu	40k	4958.3	$62.28	48:41:02	$353.13	**$415.41**
2-eu	80k	21643.3	$271.34	111:16:25	$757.34	**$1028.67**

In a real-life scenario, the data would be uploaded over a longer period of time and aggregation would also be continuous, processing new data as it is uploaded. An elastic cloud-computing environment would allow scaling the amount of hardware used dynamically, without requiring all the instances to run during the whole period. As such, the hardware costs would not differ much overall.

The c3.8xlarge instance type provides 32 CPU cores and 60GB of RAM. However, during aggregation, peak usage did not exceed 78% of total available CPU and 15% of RAM in any experiment. Average loads were 40% and 10% respectively for CPU and RAM. Although maximum bandwidth used was measured up to 4 Gbit/s for a single instance, which suggests more data could have been processed in a single process during aggregation to saturate the network connection without slowing down the computation, thus increasing cost-efficiency. With the largest dataset, a total of 1.2 terabytes of one-way communication was performed for the fast

risk analysis implementation, which also stresses the importance of a fast network connection to achieve good performance.

3.3.5 Conclusion

The results of our cloud benchmarks fully demonstrate that deploying and running a large-scale application performing secure computations has become very cost-efficient, particularly in an elastic cloud computing environment. We observed that parallelized distributed computation methods can significantly improve the performance of MPC applications processing large volumes of data.
By performing these experiments, we gained much experience and insight into deploying large-scale MPC applications in the cloud. In the future, automatic service provisioning tools to deploy MPC in the cloud would make such deployments easier and bring MPC further toward the cloud.
Our prototype application could still be further improved, especially the uploading process. Currently, we manually divided data between Sharemind processes, but an automatic load balancer would be a more general and convenient solution for future applications and help make the current uploading phase faster. Also, tighter integration with cloud provider specific technologies and best practices could increase performance and practical security.

3.4 SEEED-proxy

The SEEED-proxy application presented in this document is based on our technique for encrypted databases (SEEED) described in previous deliverables D22.2 [90] and D22.3 [23]. Particularly, we refer to deliverable D22.2 Section 6.2 for adjustable encryption, how to transform SQL queries to their encrypted version and how SEEED supports joins. Furthermore, advanced techniques like SQL query splits that enable complex SQL queries on encrypted data are described in D22.3 Section 5.4.2. Finally, details of the prototypical implementation of our encrypted database are discussed in D22.3 and its placement within the SPEAR & DAGGER framework is elaborated in deliverable D21.2 [22]. The trust assumptions and attacker model are the same as in the previous deliverables and can be summarized as follows:

1. Data confidentiality preserved amongst honest-but-curious SaaS and DBaaS providers
2. The application computing on sensitive data is trusted
3. Hardware, operating systems and browsers of end users are trusted
4. SEEED(-proxy)'s software components at cloud customer are trusted

SEEED-proxy enforces data confidentiality for *cloud based web applications* that are backed by a database in this model. It enables company employees to be able to access privacy-preserving business web applications in a simple and familiar way by using their browsers. Despite the fact that the applications only process encrypted data, the applications must allow the employees to process and manipulate the data according to their daily work. This includes data insertions, updates and deletions and especially data queries. Also, very complex analytic queries that are not uncommon in modern business applications must be supported. All processing steps have to be possible without revealing unencrypted data to the cloud providers. Furthermore, it should be easy for the companies to set up and manage all software components that are required to protect their business application. It is infeasible to deploy and control the software

components on every device of the end users, e.g. the employees should be able to access the web applications with their browsers without installing additional software. Therefore, a proxy solution should be used that is deployed at a centralized point in the company's network. Every message between the end users' browsers is routed through this proxy, has to be inspected and sensitive data must be protected.
Thereby, the proxy provides the company with full control over its sensitive corporate data. Furthermore, the proxy should be application agnostic, i.e., it must not be required to update the proxy if the application changes or new applications are installed.

3.4.1 Application Overview

In summary, SEEED-proxy presented in this document aims to fulfill the following requirements:

1. **Application agnostic:** Applications are modified or explicitly written for SEEED-proxy, but no software components deployed at the cloud customer must be updated or modified by anybody to support new/updated applications.
2. **Application at SaaS and database at DBaaS:** A company wants to use the full potential of cloud computing – the application and the database are outsourced to the cloud.
3. **Data confidentiality at SaaS and DBaaS provider:** Confidentiality for the company's data is enforced at SaaS and DBaaS providers abiding the defined trust model (see Section 3.4).
4. **Structured data:** The company's data is stored as structured data in a database.
5. **HTTP based web application:** The end users utilize their browsers to access the web application that is hosted by the SaaS provider. The communication between the browser and the application is based on the HTTP(S) protocol.
6. **Control over encryption keys at cloud customer:** All encryption keys are stored in the cloud customer's realm. The customer is able to configure which keys are allowed to be published.
7. **Collaboration amongst end users possible:** The end users (employees) must be able to collaborate while the company's secrets are protected.
8. **Support for complex queries:** The application allows the employees to fulfill their business tasks. Complex analytic queries that contain range queries, sorts and sum operations must be directly possible on encrypted data at the DBaaS provider.

The main idea is to offer end users the possibility to access and process data in the cloud while data is protected to a certain degree. Therefore, a protocol is required that allows data requests and manipulations. The standard protocol we chose for this task is OData [2]. OData is a resource-based web protocol standard originally defined by Microsoft and standardized at OASIS in Version 4.0. The main idea is to establish uniform semantics for a client-server communication by defining the creation and consumption of REST APIs. All accessible resources are identified with uniform resource indicators (URI). Clients request and edit resources using simple HTTP(S) [40, 109] requests.
The OData protocol defines the communication between a client and OData services. An application that utilizes OData offers one or multiple OData service endpoints that provide

a data interface for the client-side application code. The protocol allows any client to access data that is exposed by a source. The source might be a database, spreadsheets or custom applications. The HTTP methods GET, POST, PUT and DELETE are used to query, insert, update and delete data, respectively. Responses from an OData service are either XML-based AtomPub [60] or JSON based messages [1].

3.4.2 Applicability assessment of different approaches

In the following we analyze a typical web application setup and discuss different approaches for installing SEEED-proxy. The application itself resides on an application server in the company's network or in the cloud at an SaaS provider. The same applies for the database, which resides at an on-premise database server or at a DBaaS provider. A high-level overview is given in Figure 3.9.

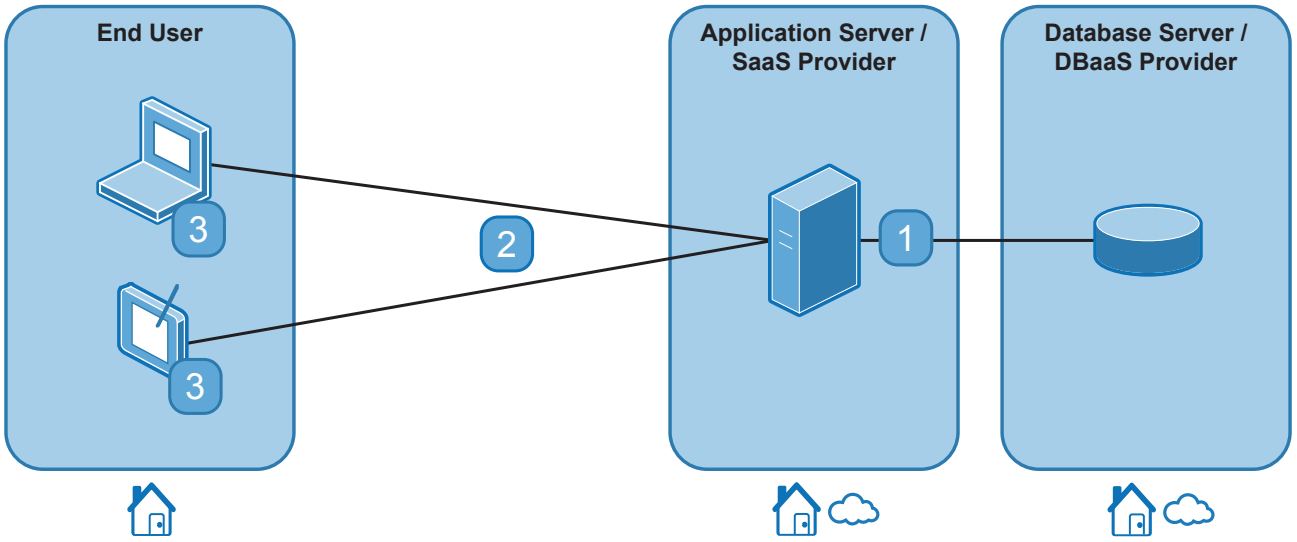


Figure 3.9: Typical web application scenario with different encryption points

In the remainder of this text, it will be stated explicitly if the applications or databases reside on-premise. There are varying approaches that utilize different points to enforce data confidentiality (blue boxes in Figure 3.9). We will reference these points as *encryption points* throughout this discussion. The choice of an encryption point has distinct advantages and disadvantages. Components on the right hand side of an encryption point handle only encrypted data and the components on the left hand side handle plaintext. As the region left of the encryption point must be trusted to guarantee security of the data, it is also called *trusted computing base* (TCB).

Encryption point 1: Encryption point 1 leads to the largest TCB: everything but the DBaaS must be trusted to achieve an acceptable level of data security. Either a trusted SaaS provider must be assumed or the application must be deployed at an application sever in the realm of the company. This shows inherent drawbacks of this design: if the cloud customers want to use cloud based applications, the solutions do not provide any protection against attackers targeting the SaaS provider.

Encryption point 2: The encryption keys are stored in a key store next to the proxy, because the proxy takes care of the encryption and decryption of data. As a result the company

can focus on protecting this one key store and the proxy itself from attackers and curious administrators. Additionally, the access can be restricted and monitored.

Encryption point 3: The general idea of solutions deployed in the end users browser is to minimize the TCB. Only the hardware, operating system and browser must be trusted. No central server is required, but the solution must be deployed and updated on every device and the end users have to use a supported browser. Managing application specific settings on every device and synchronizing keys between all devices already becomes a complicated tasks with a few end users. Since every device needs a key store, the attack surface is bigger than with other encryption points and it is infeasible for the company to restrict and monitor attacks on every device.

We chose encryption point 2 since it is an efficient trade-off between minimizing trust assumptions (drawback of encryption endpoint 1) and maximizing maintainability (drawback of encryption endpoint 3): It supports a rich set of functionalities required for complex business applications to process data in the cloud. Additionally, this approach is application agnostic, i.e., the proxy does not have to be updated if the application changes or a new application is used.

3.4.3 Application Architecture

In the following we elaborate on the use-case in a more detailed fashion, namely in the perspective of the chosen encryption point. Subsequently we present the message protocol used internally by SEEED-proxy to fulfill the requirements mentioned in Section 3.4.1.

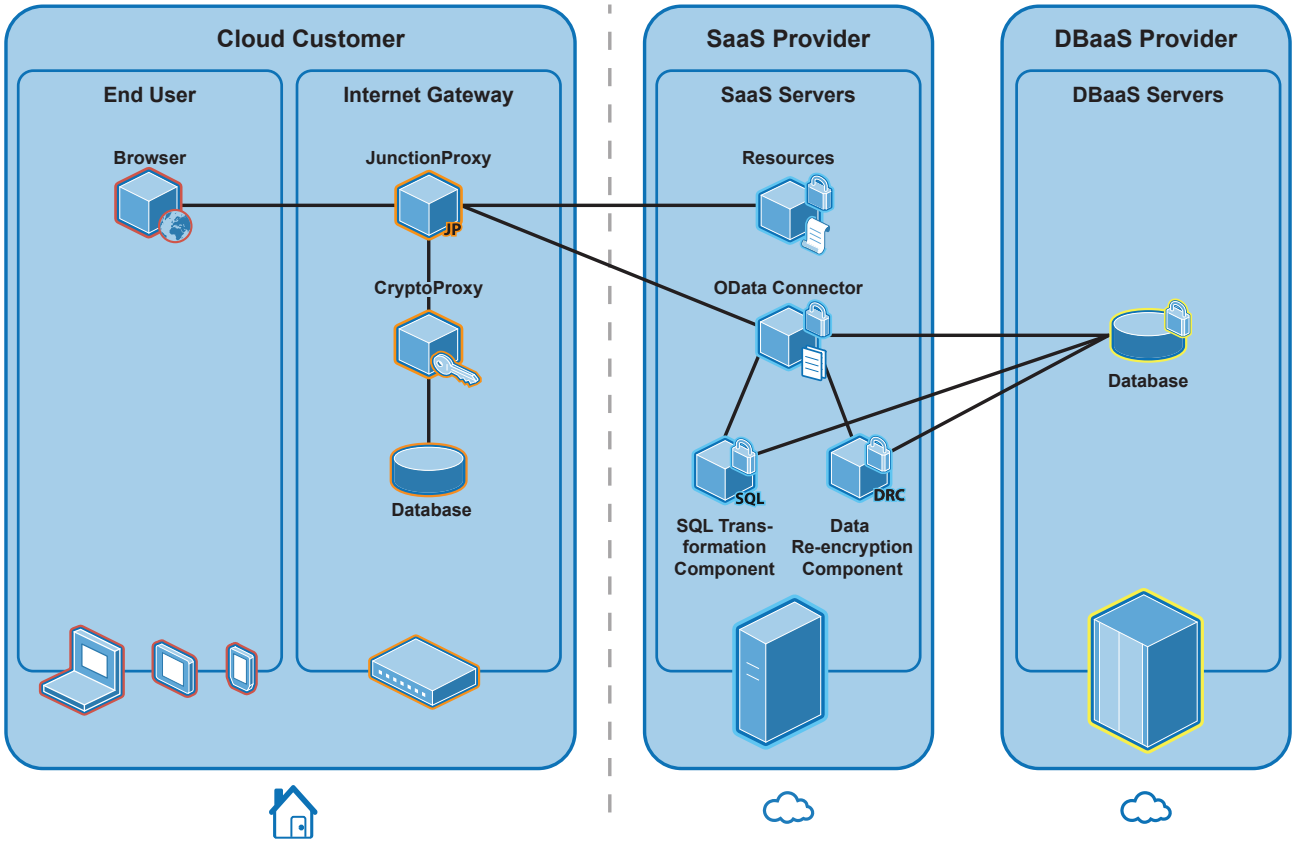


Figure 3.10: SEEED-proxy architecture

DBaaS provider

The company's structured data is stored in an SQL database at a DBaaS provider (cf. requirement 4). All data items in this database should be protected, but applications still have to be able to perform as many calculations as possible directly on the protected data without changing the database management system (DBMS). Thereby, the strength of databases – fast data processing – can be utilized.
SEEED-proxy uses adjustable encryption (see Deliverable D22.2, Section 6.2) to reach this goal, because this encryption approach offers a rich set of operations on encrypted data and the only necessary change at the DBMS is an additional operator to sum up encrypted values. With adjustable encryption even complex business queries can be executed at the DBaaS provider (cf. requirement 8).

SaaS provider

One important requirement for SEEED-proxy is that it is application agnostic, i.e., it must be able to protect web applications without any updates of SEEED-proxy's software components. Therefore, SEEED-proxy must be able to extract all sensitive data values from all transferred messages without knowing specifics about the applications that are deployed at the SaaS provider. We use the following notations to differentiate messages that are transferred between the SaaS provider and the end users' browsers:

***Content data* message:** messages that contain sensitive data which has to be protected by SEEED-proxy.

***Metadata* message:** messages that contain meta information about the content data.

***Application resource* message:** messages that contain client-side application resources such as HTML, CSS and JavaScript.

Two concrete tasks must be solved by SEEED-proxy: all content data and metadata messages have to be identified and relevant data values must be extracted. As a first step, SEEED-proxy assumes that content data and metadata messages are strictly separated from application resource messages. Figure 3.10 shows this separation: an OData connector contains the application logic and a separate interface for resources is provided. For the information extraction, SEEED-proxy assumes that a standard data transport protocol is used for the transfer of content data and metadata messages. This allows a fast extraction of information from a predefined structure. Without loss of generality, OData is used as a standard data transport protocol throughout this section to keep the text comprehensible.
For the design of SEEED-proxy, we focus on applications in which the application logic is encapsulated in database queries. The OData connector publishes OData services that are callable (via HTTP(S)) by client-side code. The connector is a (nontrivial) bridge between OData requests and the database. The main task is to transform the requests that are received at an OData service to SQL queries, executing them at the database and transforming the result sets back to OData responses.
Several libraries exist that are able to perform these tasks for OData messages for several database management systems, but they only support plaintext databases, i.e., all incoming OData requests are transformed to SQL queries for plaintext databases. For that reason, we extend the original connector with external components to support adjustable encrypted databases. A *SQL transformation component* converts the created queries to onion SQL queries that support adjustable encrypted databases. For other processing steps, the extended OData

connector uses a *data re-encryption component*. This data re-encryption component is able to decrypt and re-encrypt values, but only if the corresponding keys were published before. Upon re-encryption, it performs a proxy re-encryption which means that one ciphertext is re-encrypted to another ciphertext without revealing the plaintext. The data re-encryption component is not able to create encryption keys by itself, and the publishing decision is solely in the hands of the cloud customer.

Internet gateway

Before messages can be processed, there has to be a possibility to intercept all messages. The Internet gateway of the cloud customer is used to deploy proxies that fulfill this task, because it is a single point between the local network and the Internet where all messages pass through. The proxy components – *JunctionProxy* and *CryptoProxy* – form the heart of SEEED-proxy. The JunctionProxy intercepts and examines every request and response passing through the Internet gateway. Its main task is to decide for every intercepted message whether it should be redirected to the CryptoProxy or directly forwarded to the application server. The CryptoProxy performs the customer-side processing of OData messages. It uses a database that is deployed on the same server to handle complex data queries. Note that the CryptoProxy is the only entity in the setup with access to the keys.

End user

The end users (employees) use a web browser on their preferred device to connect to SEEED-proxy protected web applications. The end users' devices must be connected to the company's network to call SEEED-proxy protected web applications, because otherwise the connections are not routed through the Internet gateway.

3.4.4 Message flow overview

Figure 3.11 gives an overview of SEEED-proxy's message flow when the end user calls a web application in his browser. Many details are left out because this message flow overview should just give an insight in the complexity of the problem that is solved by SEEED-proxy. For that reason, this message flow does not show the separation of the proxy in two components. Instead, one combined proxy is assumed here.
The message flow starts with an end user that enters the application's URL in his browser's address bar or clicks on a link to the application. Firstly, the browser requests the application's resources. The corresponding messages are passed through the proxy. Yet, they are not processed, because they do not contain any sensitive information. The resource requests are left out in all subsequent message flows.
Afterwards, the browser sends the OData request (message 1). The proxy at the Internet gateway intercepts and examines the request and processes it if necessary. For this basic example, we assume that no sensitive information is contained in the request and therefore no processing of such is required (in the following we explain the processing of sensitive data values in a request). The request is then forwarded to the OData service provided by the OData connector (message 2).
The OData connector converts the OData request to an SQL query for plaintext databases and passes this query to the SQL transformation component (message 3). This component transforms the incoming query to one or multiple onion SQL queries. The transformation depends on the current onion and layer configuration. Therefore, the SQL transformation

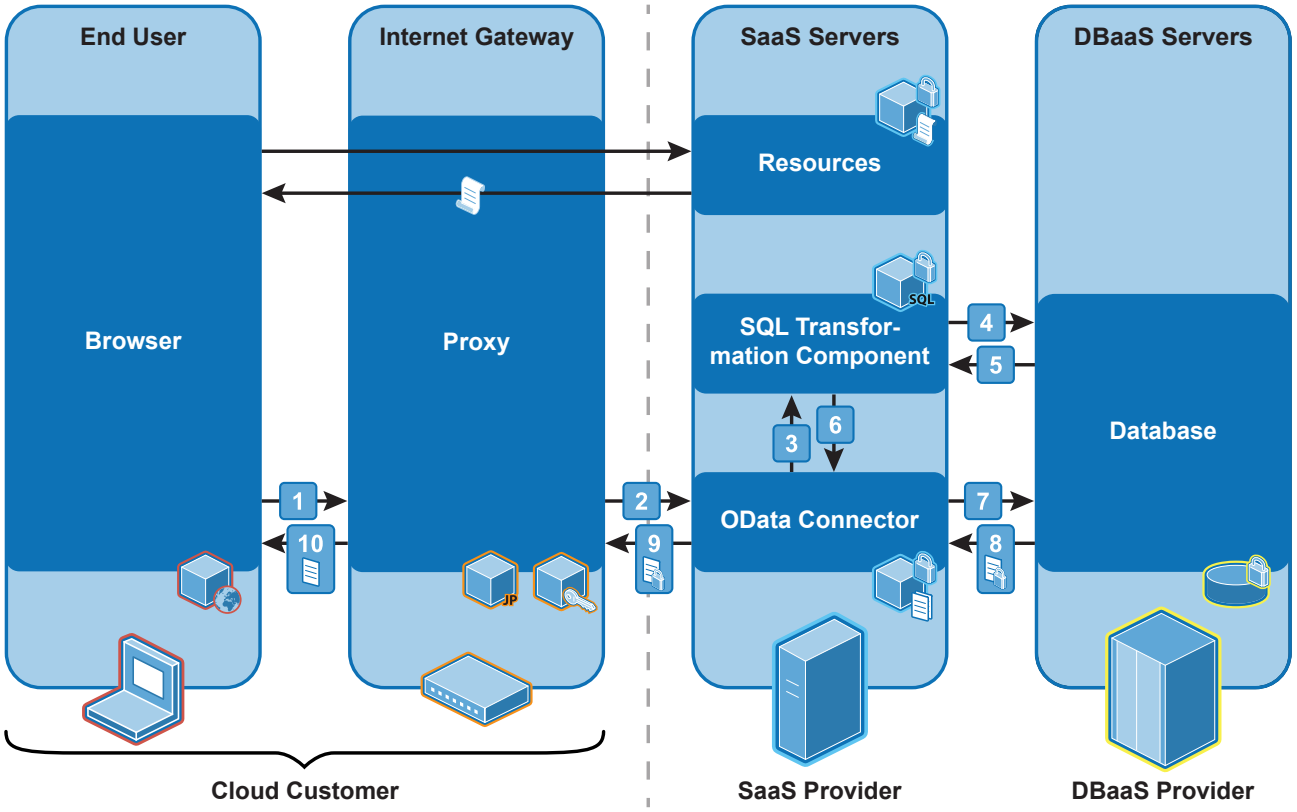


Figure 3.11: Message flow in SEEED-proxy overview

component requests the current onion and layer configuration from the database at the DBaaS provider (messages 4, 5). Then, it performs the transformation of the incoming SQL query to an onion SQL query as explained in previous deliverable D22.2 [90]. The difference between the incoming and transformed query is that the attribute, table and operation names (only for sum operations) are replaced to fit the current settings in the database that utilizes adjustable encryption. No sensitive information is contained in the onion SQL query, because we have assumed that the OData request does not contain any sensitive information.
A transformation might not be possible if no layer (encryption scheme) with a required functionality is exposed in the database. The necessary steps to expose these layers are explained in Section 6.3 of deliverable D22.2. Furthermore, a split of the onion SQL query and post processing might be necessary for complex queries (see deliverable D22.2). For the message flow in this section, we assume that no layer removal is required and the onion SQL query is fully executable at the DBaaS provider.
After the transformation, the SQL transformation component returns the result back to the OData connector (message 6) and the connector executes the onion SQL query on the database (message 7). The database only contains encrypted data, therefore a result set with encrypted values is passed back to the OData connector at the SaaS provider (message 8). The OData connector converts the result set to an OData data response (the data values are still encrypted) and returns it back to the proxy (message 9). The proxy decrypts the values in the content data response and forwards the processed message to the end user's browser (message 10). The end user receives the response from the web application he requested. All content data is presented in plaintext and the end user does not even notice that the content data was stored in an encrypted database in the back end.
Business web applications normally offer a set of *predefined content data requests*. The appli-

cation vendors design these for a large group of customers, a specific business area or a specific customer. These predefined content data requests are loaded when the end users open a web application in their browser. The results are displayed in, e.g., tables and figures. However, the application vendor cannot anticipate all needs of end users and therefore strives to provide a flexible application: the end users can influence the content data request by interacting with a UI. For instance, the end users may want to sort, load further or filter for specific entries. However, the interaction only influences predefined content data requests. Usually, it is not possible for the end users to execute arbitrary data requests. OData offers a server side data model, i.e., all calculations are done at the server and only the result is transferred. In many cases, only a subset of the requested data is loaded if the result has too many entries. The request URL determines which data resource is loaded from the application server and how many entries should be transferred. Mechanisms such as keys, associations and order by statements are used to limit and order the data request and thereby the data response.

Example: Simple data request

First, we examine the most simple content data request to explain the message flow: retrieving an entity set. This content data request is simple, because it does not require data values inside the URL that are used to request the OData resource. The following is an example URL for the retrieval of an entity set:

```
http://host/service/orders
```

Firstly, the JunctionProxy intercepts the HTTPS GET request, finds a `X-SEEED-proxy` header and redirects the request to the CryptoProxy. No processing of the original request is done in this simple example, because the URL does not contain any sensitive data values. The content data request is passed back to the JunctionProxy and forwarded to the OData service at the OData connector.
The OData connector transforms the OData request to an SQL query that could be executed if a plaintext database was used. The query is then sent to the SQL transformation component, the current onion and layer setting is requested, the onion SQL query is loaded from a cache of already transformed queries or created dynamically and the onion SQL query is passed back. The OData connector executes the onion SQL query at the DBaaS provider and requests the onion and the layer configuration of the encrypted values. Then, it creates the OData data response, together with the layer and onion information. Only the layer is dynamic metadata, but the CryptoProxy does also need the onion for decryption. An example line in an OData data response has the following form:

```
<d:totalSales m:type="Edm.Decimal" s:onion="1" s:layer="2">
  Epw738ceS+Ao7yP1JA5gCAoa6uJhiWqHIWT4r...
</d:totalSales>
```

The OData connector adds the `X-SEEED-proxy` header with the value `dataResponse` to the response and passes it back to the JunctionProxy where the custom header is recognized and the message is redirected to the CryptoProxy. The CryptoProxy extracts the onion, layer and property name of every line with an encrypted value, retrieves the corresponding static metadata from its cache and decrypts every value in the OData response. All ciphertexts are replaced by the corresponding plaintexts and all previously inserted metadata is removed to recreate the original expected OData response. This exemplary line in the OData response corresponds to the encrypted line above:

```
<d:totalSales m:type="Edm.Decimal">72701.8</d:totalSales>
```

Finally, the plaintext OData response is passed back to the JunctionProxy and is forwarded to the end user's browser.

Example: Complex content data request

So far, we only examined the request of an entity set. The processing becomes more complex if sensitive data values are involved in the request. For instance, OData uses keys in the request URL to distinctly identify a single entry in an entity set. The most important query options are: restricting the content data request with a filter parameter (`$filter`), ordering results according to one or multiple defined properties with `$orderby`, paging entries with `$skip` and `$top` or any combination of these query options.
`$orderby`, `$skip` and `$top` do not require to add sensitive data values to the request and thus do not need further processing. The message flow equals the flow explained so far. As for the other content data requests, OData requires to add data values to the request. Listing 3.5 shows examples for OData request URLs with data values.

```
http://host/service/orders(53)
http://host/service/weekday(day=30,month=04,year=2015)
http://host/service/categories(1)/products
http://host/service/categories(1)/products(20)/price
http://host/service/orders?$filter=totalSales gt 10
http://host/service/orders?$filter=totalSales gt 10 and customerName eq 'Bob'
```

Listing 3.5: Exemplary OData request URLs with data values

Every key string in the URL consists of the column name and a key value (if more than one key is used). For instance, the second URL in Listing 3.5 contains the key values `30`, `04` and `2015` for the column names `day`, `month` and `year`, respectively. The filter strings contain a column name and a constant filter value, e.g., constant value `10` and the column name `totalSales` in the fifth URL.
All constant data values and column names contained in URLs have to be used in the SQL query created by the OData connector, because they restrict the expected result set. However, the data values in the URLs may leak sensitive information about the request and subsequently about the response if they are not encrypted before they reach the SaaS provider. For instance, if every employee with a salary greater than $500,000$\$ is requested and only one row is returned, an attacker might be able to infer this person. Only the CryptoProxy has access to the keys, therefore it has to perform the encryption.
The main change is that CryptoProxy encrypts the data values before it is passed back to the JunctionProxy. Handling the filter entries is complex, because the required encryption depends on the concrete operation: e.g., equality checks (`eq`, `neq`) need deterministic encryption and comparisons such as greater than `gt` or less than `lt` require order preserving encryption. The CryptoProxy must therefore inspect the URL in detail and decide which type of encryption is required. It then reads the static metadata from its cache and utilizes the stored usage attribute to choose the correct onion. However, it might be necessary to remove a layer before the operation can be performed, because the used layer does not match the current layer in the database. For that reason, the CryptoProxy must transfer the used layer to the OData connector.
The CryptoProxy encodes the used layer in the URL. The OData connector transfers the URL to the data re-encryption component. There, the layer information is extracted and layers at the encrypted data values are removed if necessary. Listing 3.6 shows processed URLs

that correspond to the URLs in Listing 3.5. Every ciphertext is Base64 encoded and `X'...'` surrounds every binary value (default OData behavior). Note, that the corresponding SQL

```
http://host/service/orders(X'gGd2H6aAiz9JX1pnG13EF...':2)
http://host/service/weekday(day=X'Za4G1y...':1, month=X'jgCLDx...':2,
   year=X'MR0WMLX...':2)
http://host/service/categories(X'7b4N7l...':2)/products
http://host/service/categories(X'7b4N7l...':2)/products(X'gGd2H...'':3)/price
http://host/service/orders?$filter=totalSales gt X'AReTa/...':1
http://host/service/orders?$filter=totalSales gt X'AReTa/...':1
   and customerName eq X'Ce3ERa...':3
```

Listing 3.6: Exemplary OData request URLs with data values after processing

query for plaintext tables that is created by the OData connector already contains encrypted data values. For that reason, there is no need for the SQL transformation component to handle the data values during transformation but only the correct encryption layers must be chosen. The following example shows an SQL query that is passed from the OData connector to the SQL transformation component:

```
SELECT weekday FROM weekdays
WHERE day = 'Za4G1y...' AND month = 'jgCLDx...'
AND year = 'MR0WMLX...'
```

An exemplary corresponding final onion SQL query is:

```
SELECT weekday_RND FROM ENC_weekdays
WHERE day_DET = 'Za4G1y...' AND month_DET = 'jgCLDx...'
AND year_DET = 'MR0WMLX...'
```

3.4.5 Conclusion

In this section we highlighted how SEEED can be extended to enforce data confidentiality for cloud based web applications amended by database systems. The main idea is to utilize intelligent message flows between the HTTP based web application and the CryptoProxy. Thereby, SEEED-proxy adds the security benefits of adjustable encryption to a complex setup where the application is deployed at an SaaS provider and a protected database is deployed at a DBaaS provider.
Although this complex application environment setup is assumed, data confidentiality is enforced by only two proxy components and no plaintext values leave the cloud customer's infrastructure without being encrypted if not explicitly specified by the cloud customer. Even layer removals – which weaken security characteristics – are only possible if the keys have been published by the CryptoProxy. This decision is solely in the hands of the cloud customer and the publishing is restrictable by fine-grained security policies. Furthermore, every key publishing request is logged and can be reviewed by the cloud customer at any time.
Additionally, SEEED-proxy is application agnostic. Neither the JunctionProxy, nor the CryptoProxy have to be updated to support new applications. The applications only have to abide the defined messages flows and use the components provided by SEEED-proxy.
Finally, the end users can collaborate with each other, because the proxy components at the Internet gateway intercept every message and all encryption and decryption operations are done there. No key synchronization or special hardware is necessary for the collaboration of

end users. Instead, they are allowed to use any device, as long as every connection goes through the proxies. While providing a rich set of functionalities, the same security guarantees as for SEEED apply to SEEED-proxy.

3.5 A Generic Data Collection Application

A number of potential applications using secure computation proceed in the following two phases: First, data is collected from some set of parties we will call *data providers*. The data collected is private to each data provider, and is therefore protected in a secure computation system such that it will not be revealed to any other party. Second, once data has been collected from the data providers a (typically distinct) set of parties, we will call *data users*, can use the secure computation system to perform various analyses on the collected dataset. For the data providers the secure computation system will ensure that the the data users only learn results of analyses approved by the data providers (such as a limited set of statistics), and no other information about the collected data set. For the data users the system can ensure that the results of analysis they request are only learned by the requesting party. This means that no other party in the system will learn which analysis the data users are interested in.
Such applications are useful whenever, for legal or business reasons, the data providers can or will not provide the data users or any other party with the collected data set in the clear. In such cases the secure computation system can allow for data sets to be utilized in ways that simply can not be done in any other way.
One example of such an application is the benchmarking prototype presented in D23.2 of PRACTICE. In this prototype, financial data on a large amount of customers in a particular business segment is collected from a consultancy house. Banks then use the secure computation system to benchmark potential customers against the best practices in the particular segment. The application thus allows the consultancy house to keep their data set secret (which it is obligated to do by its customers) while allowing the banks to benchmark their own costumers. In the scenario described in D23.2 the consultancy house is the only data provider and the banks using the system are the data users. However, an obvious extension, which was also suggested in D23.2, would be to add more data providers to add additional data to the data set and make it even more valuable as a data foundation for the banks.
Another potential example could be in medical research. Here medical researchers could be interested in doing statistic analysis on confidential health records held across multiple hospitals and general practitioners. However, the hospitals and doctors are not allowed to share information on their patients. In this setting we could first securely collect the joint data set of health records from all hospitals and general practitioner and then let medical researchers run statistical analyses on the dataset using secure computation.
In this section we will describe an application supporting the first phase of the above mentioned application pattern, i.e., the data collection phase. Additionally, the application supports the what we call a *data preparation phase*, where generic operations are done to the collected data sets order to produce a single consistent data set. The data collection application can then be used to bootstrap any father application following the described application pattern.
The rest of this section is structured as follows. In Section 3.5.1 we give a high level overview of the data collection application. In Section 3.5.2 we describe the architecture of the application focusing on its logical components, their physical deployment and their interaction.

3.5.1 Application Overview

The application described in this section is being developed in the project *Big Data by Security* (BDbS) (`http://www.bigdatabysecurity.dk/`) founded by the Danish Industry Foundation. The project focuses on demonstrating how secure computation can be used to enable Big Data analytics by allowing analysis on data sets that could otherwise not be collected or analyzed for privacy and security reasons. In this context the generic data collection application will be used to support prototypes of applications in the financial and energy sector all following the above application pattern. There is no formal connection between the BDbS project and PRACTICE apart from technologies developed in PRACTICE being used in BDbS project.
The data collection application is focused on providing a simple and user friendly interface for the data providers to upload bulk data to the secure data collection system, which can then later be used for analysis. It also provides an interface for a special *organizer* to design and manage the data collection process. Additionally, the data collection application provides the ability to transform the individually collected data sets from each data provider into a single coherent data set, appropriate for the later data analysis. These can be simple operations such as removing duplicate entries or linking records between data sets provided by distinct data providers. It can also be a set of more complicated procedures such as checking for data corruption. We call this *data preparation*.
In the following we first describe the types of users involved in the application and then describe in more detail the data collection process of the application.

Users

There two main types of users involved in the application namely the *organizer* and *data providers*. The third user type the *secure computation hosts* as mentioned above are mainly there to ensure the security requirements of the application.

- A single organizer defines and manages the data collection process. I.e., the organizer decides what data is needed from each of the data providers in order to create the desired data set. He also decides how data provided will be prepared to form the desired data set. Additionally the organizer drives the data collection process by deciding when to go from one phase of the data collection process to another, as described below. We stress that the organizer does not have full control of data collection system. In particular, he does not have access to the data provided by the data providers, he simply dictates what types of data is required from each provider.

- Data providers hold the raw data needed in for the data collection and provides it as instructed by the organizer. I.e., if the instructions of the organizer are acceptable to the data provider she uploads her data to the data collection system.

  Note that different types of data may be collected from different data providers. E.g., in the medical research example above general practitioners may be able supply data on patients not held by the hospitals and vice versa.

  The data provider should be involved as little as possible in the application and should ideally only have to upload a single file to the system.

- The secure computation hosts are the parties that host the underlying secure computation servers implementing the data collection application. They do not play an active role in the data collection. However, if they collude they will be able to break the confidentiality

of the collected data. Thus they must be chosen such that the data providers can trust that at least one of the hosts will act honestly. This can typically be achieved by letting at least one of the hosts be a trusted representative of the data providers.

Data Collection Process

In the data collection application the final data set to be collected is assumed to be structured as a single table with each row containing a, possibly hidden, ID identifying uniquely the data record in the row. Each row is assumed to have a label identifying the type of information held in the given row (such as a customer ID or similar).
Here we go through the complete process of such a data collection using the application. We take the point of view of the end user treating the data collection application abstractly as a single service implemented in the cloud. However, as we shall see the in following section, the application is concretely implemented as a distributed system using secure computation. At the end-user level, however, the process proceeds in the following three phases:

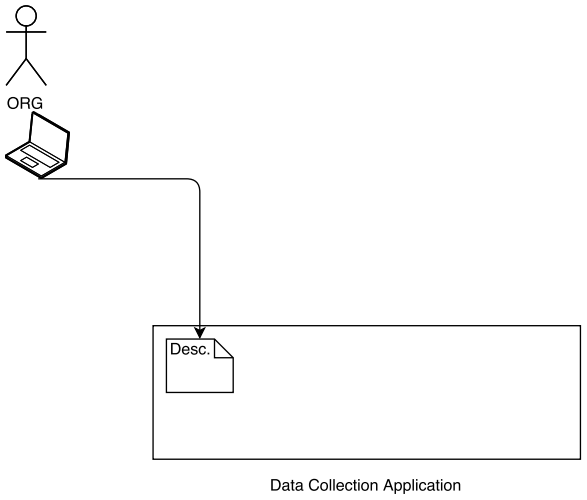


Figure 3.12: Definition phase: The organizer (ORG) uploads a job description.

Definition (Figure 3.12) Before initiating a data collection the organizer must decide what data must be collected from each of the data providers and how these data sets must be prepared to form the final data set. Once these decisions are made the organizer start the data collection by specifying them using a web-interface to the application.

He first designs a template for each of the data tables to be collected from the data providers. The template describes what data the data providers must provide and in which format.

The organizer then commits to which method to use for any data preparation procedure. As described above this can include, e.g., removal of duplicate records from multiple data providers. We note that this decision must be fixed before the data collection is started and should not be allowed to change at a later stage.

Finally the organizer gives a short textual description for the data collection and its purpose. We call the collection of all this information the *job description* of the data collection. Once the organizer has finalized it, it is uploaded to the data collection application and the actual data collection process can begin.

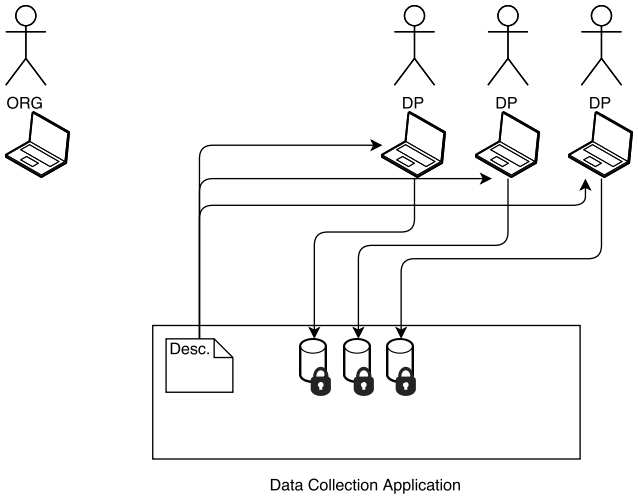


Figure 3.13: Collection phase: Data providers (DP) reads job description and uploads data.

Collection (Figure 3.13) The data providers can now via a web-interface log in to the data collection application in order to read the description of the data collection job. If the data provider finds the job description acceptable (including how data will be manipulated in the data preparation phase), he will upload his data to the data collection application. He does so by downloading the template table he has been assigned by the organizer in a convenient format such as a csv or excel file. The data provider then fills out the template with actual data and uploads the file to the application.

All uploaded data is securely stored in the data collection application using the secure internal representation of the underlying secure computation system (typically secret sharing). This way, once it leaves the data providers client, no data is available to the application in clear text.

During this phase the organizer can monitor the status of the data collection process. Once sufficient data has been collected or a set time limit has be reached the organizer can stop the data collection phase.

Preparation (Figure 3.14) Once the data collection phase is over and if sufficient data has been collected the organizer starts the data preparation phase. Using secure computation this phase transforms the data tables collected individually from each data provider into a single coherent data table. This is done using whatever method was committed to by the organizer in the definition phase. If the data preparation process involves checking if the provided data tables are valid or corrupted the organizer will be informed the result and can choose to cancel the data collection effort. Otherwise, the data collection process ends successfully and the collected dataset will be ready for analysis.

3.5.2 Application Architecture

The application architecture is an example of the so called *outsourced secure computation* model. Namely, end users of the application simply act as clients who securely provide input to and read output from the application. To achieve the security requirements of the core of the application is implemented using a secure multi-party computation system. However, only a few selected parties host the underlying secure computation servers. This is in contrast to the

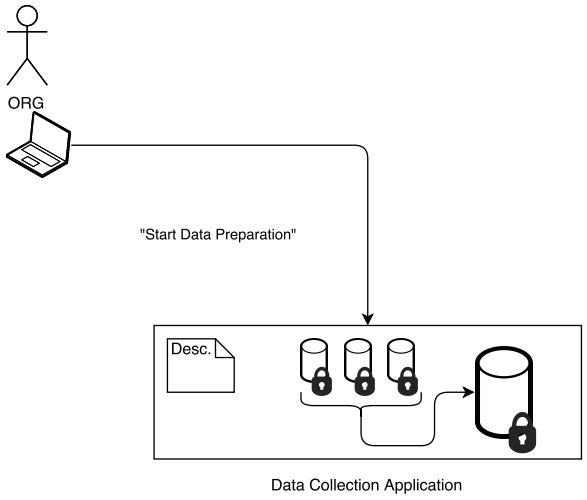


Figure 3.14: Preparation phase: Organizer starts data preparation. Data sets are transformed.

more traditional model of a secure computation system, as seen in the theoretical literature, where each user is control their own secure computation server.
The outsourced model has a number of benefits: It is easier to manage since the complex task of deploying and maintaining the secure computation servers can be left to a few parties with the required technical skills. Since the performance of many secure computation technologies decreases as the number of secure computation servers increases, the outsourced model also performs and scales better as the number of servers can be kept to a small constant regardless of the number end-users. The outsourced model is also more flexible as end-users can be added and removed dynamically without having to re-deploy the secure computation system. Finally, outsourcing the secure computation minimizes the required online time for the end-users. I.e., the end-users only have to connect to the application in order to actively give input and read output. This means that when the end-user is not actively using the application, he can forget about it and turn-off his machine. In contrast, in the traditional model each end-user would have to be keep their secure computation server live for the duration of the applications life time, even when they are not actively interacting with the system.
The main drawback of the outsourced model is that the end-users have to trust the parties hosting the secure computation servers to do the secure computation honestly. However, depending on the secure computation technology used, if only a single (or small group) of these parties behaves honestly security will not be broken. Thus, this trust issue can usually be solved by choosing this set of parties such that each end-user feel they have a trusted representative within the set.
We note that a similar architecture to the one described here was used for the *confidential benchmarking* prototype described in WP23. This is no coincidence, in fact the architecture of the application described in this chapter was heavily based on our experience from the benchmarking prototype.

Components

The diagram in Figure 3.15 illustrates the major software components of the data collection application and how they are connected. Below we describe each of these components in more detail.

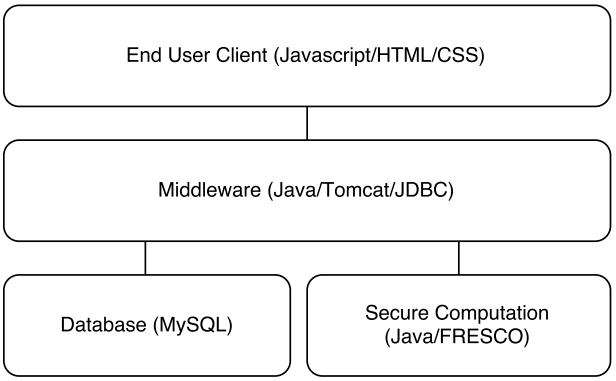


Figure 3.15: Software components of the data collection application.

End User Client The client component mainly consists of the UI for the users (i.e., the organizer and data providers). It takes the form of a simple web-application that the users interact with via their browsers and is implemented in a combination of HTML, JavaScript and other standard web-technologies. The clients for the organizer and data providers are very similar sharing a common look and feel. However, since the data providers use their client to upload confidential data to the application it must include special functionality to securely share the data across the secure computation servers.

Middleware The main responsibility middleware component is to connect the various other components in the system. It is implemented in Java using a number of subcomponents. E.g, It contains a web-server in order to serve and communicate with the end user clients, a JDBC component for interaction with the database and small custom component for interaction with the FRESCO based secure computation component.

Database All data is stored in a standard database. Concretely we use MySQL, but essentially any off-the-shelf database could be used. The confidential data provided by the data providers is secured by collecting and storing it in the internal representation of the secure computation system, such as secret sharing. This way the data is protected from the secure computation hosts holding the database.

Secure Computation The secure computation component handles any secure computation done by the application. I.e., this is the component used for the data preparation phase. The component is implemented in Java using the FRESCO framework as developed in deliverable D14.2. It in turn consists of two main subcomponents: one component implementing the concrete secure computation to be done for the various data preparation procedures (what is known as the SCS component in the deliverable D21.2), and another component implementing the secure computation technology used to actually perform the computation (known as the SCE in D21.2).

The flexibility of the FRESCO framework means that the underlying secure computation technology does not need to be fixed. Instead it can be adapted to the concrete scenario without changing the application significantly. This means that for a concrete usage scenario we can pick a secure computation technique best suited for the job. This is key, for a very generic application such as this one, because it is intended to be use to support many different scenarios.

Physical Deployment

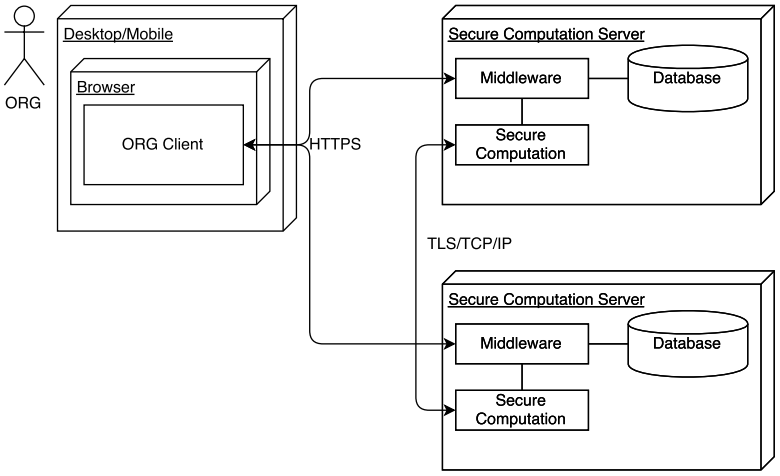


Figure 3.16: Physical deployment of the data collection application.

As described above the application is an example of the outsourced secure computation model. I.e., the secure computation is outsourced to a few secure computation servers, while the end users simply act as clients to the system by securely providing inputs and reading outputs from the secure computation servers. The diagram in Figure 3.16 illustrates how the deployment of application could look like in the case of two secure computation servers. In the diagram we only illustrate the organizer as end user since the picture would more or less be the same for the data providers.
As also mentioned above the end users client is a simple web-application in the browser of his mobile phone or desktop computer. The client communicates over HTTPS with the middleware components at each of secure computation servers. Note, that the client is connected to both servers. This is in order to securely share his input across the servers (in case of a data provider). Each of the secure computation servers are essentially identical and host both the middleware, data base and secure computation components. The middleware component is connected to the data base using JDBC and interacts with the secure computation component using regular a regular Java interface. Once, secure computation is started the secure computation components communicate directly using TCP/IP connections and using TLS for any communication that needs to be secured.
In this setup the secure computation servers should preferably be high-performance machines as the secure computation process can be rather demanding. However, secure computation is only actually used in the data preparation phase. In the definition and collection phase the secure computation servers may not have as much power. Therefore, as a cost saving mechanism in the cloud computing setting, we could consider a setup where the servers are moved to an expensive high-performance machine only during the data preparation phase.

3.5.3 Conclusion

In this section we have described a generic application for confidential data collection based on secure computation. The application is meant as a building block for data analytics applications involving analysis against a data foundation collected in a early phase of the application lifetime. Having such a generic building block is valuable, in order to considerably shorten the development time of applications following a certain application pattern that we often see in secure computation based data analytics scenarios.

We have additionally presented the architecture of the application which is heavily inspired by our work on the prototypes in WP23 of PRACTICE. In particular the Secure Credit Assessment and Survey prototypes. The architecture follows the same outsourced secure computation model where used in the PRACTICE prototypes which has a number of benefits in terms of usability, performance and scalability.
A prototype of the described application will be implemented and used in the project Big Data by Security founded by the Danish Industry Foundation, to build prototypes solving confidentiality problems in the banking and energy sector. The implementation will be based on the secure computation framework FRESCO developed in WP14 and described in D14.2. Thus, this application nicely demonstrates how the work of PRACTICE both in terms of architecture of secure computation applications and secure computation development frameworks has been useful even outside of the PRACTICE project.

Chapter 4

Hardware-Enhanced Security for Secure Multi-Party Computing

As outlined in Chapter 5 of D23.1 [101], algorithms for secure multi-party computation (SMPC) usually provide security based on distributed cryptographic algorithms.
Minimizing trust assumptions and using only a minimal trusted computing base is a common goal of IT security. Traditional security on the application layer is usually dependent on the complete software stack (operating system, firmware, and the application itself). Using hardware to protect security critical components can (a) reduce the trusted computing base, (b) increase efficiency by providing a verifiable TCB, and may allow replacing complex multi-party protocols with a simple computation that is protected by trusted hardware.
For SMPC, the traditional cryptography-only approach of SMPC has several shortcomings that can be mitigated by adding hardware-based security protections:

High Cost Depending on the function to be computed, the corresponding circuit can be complex and thus the computation and memory cost of the function evaluation is very high. This may render SMPC for certain functions too expensive for many practical applications.

Limited Scalability In practice, computations may involve many players (e.g. voting) or may require large data sets (databases or even big data). For these scenarios SMPC may not scale well enough to be viable in practice.

State Similarly, if computations require a state to be kept between different function evaluations (e.g. a database), SMPC may not be the optimal approach in practice.

Software Vulnerabilities A final word of caution is that SMPC assumes correct implementation and undisturbed execution of the specified algorithms at the parties that are deemed to be trusted. Similarly, keys and state need to be kept confidential and integrity protected. Hardware security may help implementing these assumptions.

SMPC assumes that certain "trusted" participants follow the protocol as specified. This includes the centralized compute node in the traditional model as well as, e.g., certification authorities in the cryptographic protocols. One way to further reduce this high level of trust is to augment the compute architecture by introducing hardware-enhanced security.
The following two sections (published as [12]) survey capabilities and relevant products for hardware-based security.

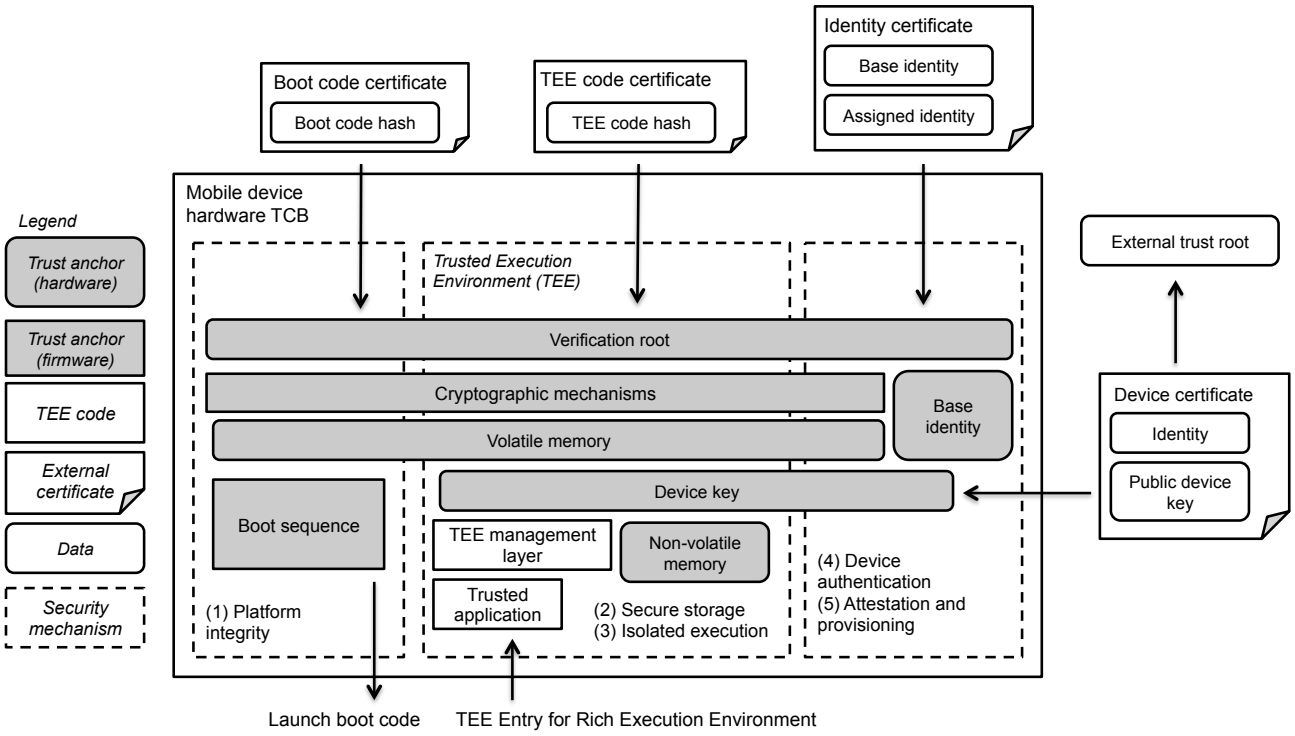


Figure 4.1: Common hardware security concepts devices (adapted from [38]).

4.1 Basic Concepts of Hardware-enhanced Security

The *trusted computing base* (TCB) of a device consists of hardware and firmware components that need to be trusted unconditionally. The remainder of the platform is usually denoted as "Rich Execution Environment (REE)". In this chapter we denote such hardware and firmware components as *trust anchors* of the computing system.
Figure 4.1, adapted from [38], illustrates trust anchors present in a typical device. Individual trust anchors are shown in gray. The numbered dotted boxes (1-5) represent common security mechanisms and illustrate the trust anchors needed to implement each mechanism. In the following subsection (Sections 4.1.1) we describe the security mechanisms. We use **bold font** whenever we introduce a concept shown in the figure for the first time.

4.1.1 Basic security mechanisms

Platform integrity

The integrity of platform code (e.g., the device OS) can be verified either during system boot or at device run-time. This allows device manufacturers and platform providers to prevent or detect usage of platform versions that have been modified without authorization. Two variations of boot time integrity verification are possible.
In *secure boot*, the device start-up process is stopped if any modification of the launched platform components is detected. A common approach to implement secure boot is to use code signing combined with making the beginning of the **boot sequence** immutable by storing it within the TCB (e.g., in ROM of the device processor chip) during manufacturing [7]. The processor must unconditionally start executing from this memory location. **Boot code certificates** that contain hashes of booted code, signed with respect to a **verification root**, such as the device manufacturer public key stored on the device, can be used to verify the integrity of the booted

components. The device must be enhanced with **cryptographic mechanisms** to validate the signature of the system component launched first (e.g., the boot loader) that can in turn verify the next component launched (e.g., the OS kernel) and so on. If any of these validation steps fail, the boot process is aborted. Integrity of the cryptographic mechanisms can be ensured by storing the needed algorithms in ROM. The immutable boot sequence and a verification root together with an integrity-protected cryptographic mechanism provide the needed trust anchors for secure booting.
In *authenticated boot*, the started platform components are measured but not verified with respect to any reference values. Instead these measurements are logged in integrity-protected **volatile memory**. The boot loader measures the first component launched which in turn measures the next one and so on. The recorded measurements represent the state of the platform components after boot, and can be used for local access control enforcement or remote attestation (cf. later this section). Two trust anchors are used to implement authenticated boot: integrity-protected volatile memory and a cryptographic mechanism.
Boot time integrity alone is not sufficient if an attacker can modify the system after it has been booted. In *runtime platform integrity* verification, a trusted software (or firmware) component monitors the integrity of the platform code continuously [102] and repairs modified components automatically if possible [71]. The integrity of the monitor itself can be verified using the above described boot integrity verification techniques.

Secure storage

A mechanism to store data on the device to disallow unauthorized access by Rich Execution Environment (REE) components is called secure storage. Sensitive data kept in secure storage should not leak to an attacker even if the REE is compromised. A common way to implement secure storage is to augment the device hardware configuration with a confidential and integrity-protected device-specific key that can be accessed only by authorized code. Such a **device key** may be initialized during manufacturing and stored in a protected memory area on the processor chip. To protect against key extraction by physical attacks, manufacturing techniques like protective coatings may be used. In addition to the device key, implementation of secure storage requires trusted implementations of necessary cryptographic mechanisms, such as an authenticated encryption algorithm. Data rollback protection requires the inclusion of writable **non-volatile memory** (e.g., a monotonic counter) that persists its state across device boots.
To summarize, two trust anchors are needed for secure storage: a device key and cryptographic mechanisms. Note that securely storing cryptographic keys is useful only if cryptographic algorithms using these keys are protected as well.

Isolated execution

The term “isolated execution” refers to the ability to run security-critical code outside the control of the untrusted Rich Execution Environment (REE). Isolated execution combined with secure storage constitutes a trusted execution environment (TEE), which allows implementation of various security applications that resist REE compromise. We explain possible TEE architectures in Section 4.1.2. Here we introduce the trust anchors needed to implement a TEE, which are a subset of the device hardware TCB. Conceptually the TEE can be seen as a component of the TCB.
A TEE can expose the functionality of predefined cryptographic mechanisms to the REE with the guarantee that the cryptographic keys never leave the TEE. While predefined common cryptographic operations are sufficient for many services, certain applications require isolated

execution of application-specific algorithms. Proprietary one-time password algorithms for on-line banking constitute one such example. To support isolated execution of *arbitrary* code, the device hardware configuration must provide an interface (**TEE entry**) through which the executable code (**trusted applications**) can be loaded for execution using the protected volatile memory.
A **TEE code certificate** can authorize code execution within the TEE and authorize trusted applications to access the device key and other device resources such as confidential data (e.g., DRM keys) and hardware interfaces (e.g., the cellular modem or NFC). Furthermore, the access that any trusted application has to the device key and other device resources may be controlled based on the platform state that was measured and saved during an authenticated boot process.
A software or firmware component called **TEE management layer** provides a runtime environment for trusted applications and enforces access control to protected resources like the device key (more details in Section 4.1.2). The integrity of the management layer must be verified either as part of the boot time platform integrity verification (and runtime monitoring) or on demand when trusted applications are loaded for execution [85]. Realization of isolated execution can make use of the following trust anchors: isolated memory (volatile or non-volatile), cryptographic mechanisms and verification root.

Device authentication

An external service provider can use device authentication to verify the identity of the device (and its TEE). The identity may include device manufacturer information that can imply compliance to external service provider requirements.
The device hardware configuration typically has a unique immutable **base identity** which may be a serial number from a managed name space or a statistically unique identifier initialized randomly at manufacture. A combination of a verification root and the base identity allows flexible *device identification*. An **identity certificate** that is signed with respect to the aforementioned verification root binds an assigned identity to the base identity. International Equipment Identifier (IMEI) and link-layer identities such as Bluetooth and WiFi addresses are examples of device identities.
A **device certificate** signed by the device manufacturer can bind any assigned identity to the public part of the device key. Signatures over device identities using the device key provide device authentication towards external verifiers.

Attestation and provisioning

An externally verifiable statement about the software configuration running on a device is called *remote attestation*. Remote attestation allows an external service provider to verify that a device is running a compliant platform version. A common way to implement remote attestation is to provide statements signed with the certified device key over authenticated measurements (e.g., cryptographic hash digests) of the firmware and software components loaded at boot time.
The process of securely sending secrets and code to the TEE of the target device is called *provisioning*. Many security services require a security association between an external service provider and the TEE of the correct user device. For example, a bank might want to provision a key to the TEE of a customer device for on-line banking authentication. In some cases, service providers also need to provision TEE code that operates on the provisioned secrets, such as proprietary one-time password algorithms. Device authentication provides the basis for TEE provisioning. Data can be provisioned encrypted under a certified device key. Device

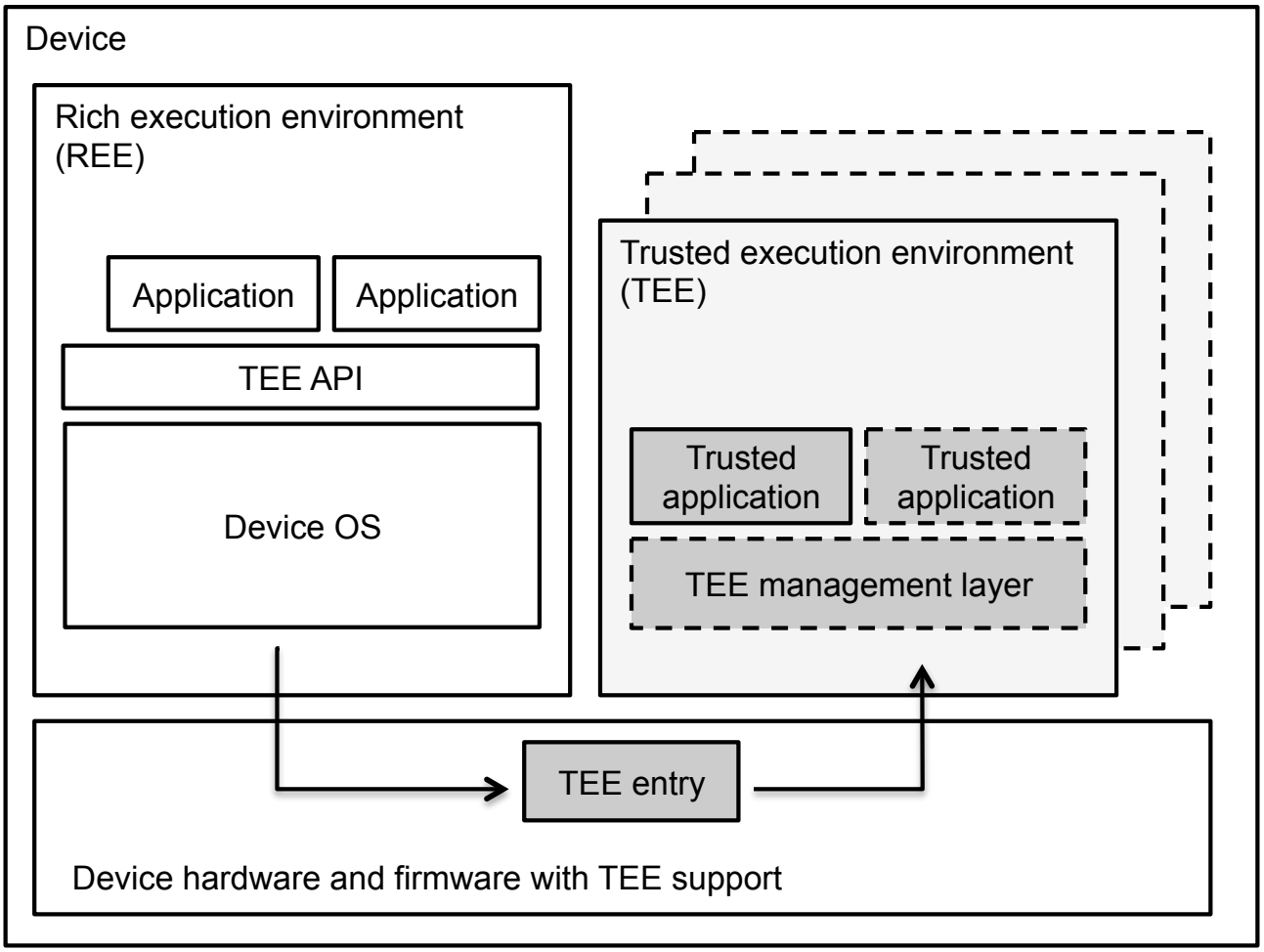


Figure 4.2: Generic TEE architecture model (adapted from [57]). Trusted applications are executed within a TEE instance that is isolated from the REE device OS. One or more TEE instances may be supported. Dashed boxes illustrate entities that are not present in all TEE architectures, grey boxes are not controlled by the REE device OS.

certificates do not include user identities and thus provisioning user authentication must be implemented by other means.
Note that all cryptographic keys needed for secure storage, isolated execution, device authentication, attestation and provisioning can be derived from the same device key.

4.1.2 TEE architecture

The isolation needed for a TEE can be realized in various ways, ranging from separate security elements to secure processor modes and virtualization. Depending on the used isolation technique, different TEE architectures are possible. Figure 4.2, adapted from [57], depicts a generic, high-level TEE architecture model that applies to different TEE architecture realizations.
We call a processing environment that is isolated from the REE device OS as *TEE instance*. A TEE architecture realization may support one or more TEE instances. In TEE architectures that are based on dedicated security chips [128] and processor modes [8, 121], typically a single TEE instance is available. Virtualization [84] and emerging processor architectures [97, 86] are TEE examples in which each REE application may create its own TEE instance. TEE instances are created (or activated) and accessed using the TEE entry interface. Applications running in

the REE device OS access TEE services through a *TEE API* that allows REE applications to execute trusted applications and to read and write data to and from them.
If only a single TEE instance is available, the same TEE instance typically allows execution of multiple trusted applications. The TEE management layer can be implemented in software as a full-fledged embedded OS, a set of libraries, a small interpreter that runs within the TEE or in device hardware and firmware. It provides the interface through which trusted applications communicate with REE applications and invoke cryptographic operations within the TEE. In terms of size and complexity, the management layer is likely to be significantly smaller than the REE device OS and thus, its attack surface is smaller. In TEE architectures, where each REE application creates its own TEE instance, a management layer may not be used.

4.2 Research Solutions

In the following we discuss various concepts and research efforts that continue to extend and improve trusted computing research. Observe that we mainly focus on the recent research directions in embedded systems. A detailed survey on research in traditional trusted computing is available in [100].

4.2.1 Alternative trusted computing designs

One of the earliest works that describe the use of secure co-processors to assure isolated execution (cf. Section 4.1.1) is the report on the 4758 Secure Coprocessor [37]. It describes the design of a physically isolated, tamper-resilient execution environment which implements a TEE that communicates with the CPU to execute certain tasks securely on a separate processor and memory [133]. Following this work, it was investigated how remote parties can gain assurance that a particular application has been executed in the TEE of some particular device (cf. Section 4.1.1). A trust chain was devised by which the TEE itself can vouch for the execution of a particular code, which in turn may load and execute other code [120]. The device key of the TEE is embedded by its manufacturer, who vouches for the correct operation of that TEE.
Drawbacks of secure co-processors are the high additional costs and the generally low computation performance. Copilot [102] alleviates this problem by using the co-processor only to monitor and assure the integrity of the actual computation performed by the main CPU. Overshadow [29] uses hardware-assisted virtualization to enforce different views on memory for user applications and OS kernels, thus ensuring the integrity and confidentiality of applications despite OS compromise.
Some works have also investigated the extension of the CPU itself to enable the measurement of executing code and to establish a TEE. For instance, the AEGIS system architecture [124] extends the CPU interface with facilities for loading, measuring and authenticating software modules, and uses these facilities to provide authenticated execution of tasks in real (non-virtual) memory. Similarly, it was proposed that a CPU vendor could provide *trusted software modules* (TSMs) [36]. The code segments of TSMs are extended with authentication codes which are automatically verified when they are loaded into the cache banks of a CPU during execution.
Leveraging such a trusted loader or regular secure/authenticated boot (cf. Section 4.1.1), a minimal security kernel can be launched which then in turn ensures a measured and isolated execution of software tasks. In particular, the PERSEUS system architecture [103] proposes to leverage a secure microkernel for strong isolation between a multitude of software security services. The NGSCB [39] proposes an ultimately trusted security kernel to support secure

applications in a secure world mode, while Terra [52] argues that a Chain of Trust must be established from platform initialization to the hypervisor and the individual executed applications. Trusted hypervisors such as sHype [111] and TrustVisor [84] follow this design and use a minimal security kernel that provides strong isolation between higher layer applications.

4.2.2 Remote attestation

Remote attestation (cf. Section 4.1.1) begins with the initial measuring of the bootloader and OS [68, 116]. IMA [112, 64] extends the Linux kernel with facilities to measure loaded code and data according to predefined policies. During attestation, the software measurements maintained by the kernel can then be signed by the device key (cf. Section 4.1.1) and the kernel in turn can be verified based on the measurements performed by the boot loader and platform firmware. As an alternative, secure OS kernels such as PERSEUS or TrustVisor only measure certain isolated security services, which are then used by regular applications to perform secure transactions on their behalf [103, 84]. The security services are designed to provide maximum flexibility while maintaining low internal complexity and external dependencies, thus simplifying the process of measuring, validating and establishing trust in a particular software [5, 115].
When extending a secure channel protocol with remote attestation, care must be taken that the reported measurements actually originate from the particular platform that is to be attested [58]. Multiple works have proposed protocol extensions for secure channels such as SSL and IPsec [9, 115] and extend the resulting networks into security domains of assured distributed access control enforcement (e.g., [83, 26]).
A general problem in remote attestation is the disclosure of the often privacy-sensitive software state to the verifying entity (verifier). To address the problems of privacy but also scalability when dealing with large amounts of software integrity measurements, property-based attestation [110, 28] proposes to attest concrete properties of software. For this purpose, the loaded software is equipped with property-certificates which ensure that the software has certain properties. During attestation, the platform then proves the existence of the required properties of the loaded software to the verifier. However, the identification and extraction of the desired software security properties from software remains an open problem [92].

4.2.3 Low-cost trusted execution environments

With the rise of resource-constrained embedded systems as part of complex monitoring and control infrastructures, a recent line of research investigates the possibility to perform attestation (cf. Section 4.1.1) and isolated execution (cf. Section 4.1.1) even on such low-end devices. These works typically assume that common approaches like secure co-processors or complex CPU modes are too expensive in terms of production cost or energy consumption. Instead, they aim to provide a limited trusted computing functionality for the purpose of automated verification and trust establishment in larger IT infrastructures.

Software-based attestation

If the device does not support a hardware-protected key needed for remote attestation (as described in Section 4.1.1), attestation can be implemented in software. A typical *software-based attestation* scheme exploits the computational constraints of a device to make statements about its internal software state [118, 117]. The prover must compute a response to a given attestation challenge within a certain time. When receiving the correct response in the expected time, the verifier has assurance that only a specific attestation algorithm could have been

executed within that time frame. The attestation algorithm is typically implemented as a specific checksum function that iteratively merges information gathered from the device. A formal analysis of software-based attestation [10] has shown the challenges of formalizing the underlying assumptions.
Several variations and extensions to software-based attestation have been proposed, ranging from implementations for different platforms to more fundamental changes to the software-based attestation concept, such as repeated challenge-response procedures [66, 79] or using memory constraints [51, 130], and self-modifying or obfuscated algorithms to prevent manipulation of the attestation algorithm [119, 56, 117]. Multiple works consider the combination of software-based attestation with hardware trust anchors such as TPMs [114, 77] and SIM-cards [66] to authenticate the prover device.

Minimal Attestation Hardware

The Secure Minimal Architecture for Root of Trust (SMART) [33] is designed to enable remote attestation and isolated execution at the lowest possible hardware cost (see also [46]). SMART realizes this using a custom access control enforcement on the memory bus, allowing access to a particular secret key in memory only if the current CPU instruction pointer (IP) points to a known trusted code region in ROM (secure storage). This way, the secret key is only accessible when the CPU is executing that trusted code and can thus be used to authenticate the execution of that ROM code to other parties. In particular, by letting the trusted ROM code measure and execute arbitrary code, the design can be extended to a freely programmable trusted execution mechanism or simply be used to attest the local platform.
While SMART is more efficient and easier to validate than software-based attestation, it suffers from certain practical drawbacks. In particular, SMART offers no exception or interrupt handling, requiring a platform reset and memory clearing in case of unexpected errors. To prevent interruption of the trusted code, the hardware access control in SMART assures that the corresponding code region can only be entered at the first address and exited at its last address. However, memory protection based on the CPU instruction pointer may still be exploited with code re-use attacks, where the semantics of code is changed based on stack or other data manipulation [45].

CPU-based task protection

Another approach to isolated execution and possibly low-cost trusted execution are Software Protected Modules (SPM) [123]. They extend the CPU instructions to provide trusted execution based on execution-dependent memory protection, allowing tasks to request protected memory regions and query the protection status of other tasks in the system directly from the CPU. This way, protected tasks can inspect and attest each other in local memory. For communication and multi-tasking, protected tasks can declare code entry points which may be called by other tasks with the desired arguments, while other portions of code are protected by the platform. However, when communicating with other tasks on the local platform, one needs to assure that the other task's entry points and protection status have not been changed since the last local attestation.
Sancus [97] extends an openMSP430 CPU to implement SPM in hardware. However, the problem of handling interrupts and unexpected software faults remains unsolved, and additional modifications are required to sanitize the platform memory upon device reset. To assure to local tasks that a particular other task has not been modified (e.g., by malware), the CPU provides a number of cryptographic tokens and secure hashes of individual loaded tasks. As a result,

Sancus imposes relatively high hardware costs for the targeted low-end systems, imposing a 100% area increase for providing eight secure modules [97]. By managing tasks through CPU instructions, Sancus imposes certain restrictions on the memory layout of a task, .e.g., limiting capabilities for shared memory or peripherals I/O. Another implementation of SPMs is provided in the Fides hypervisor [122]. Fides can provide secure interruption and communication between processes, which, however, seems to be achievable also with typical task isolation by trusted hypervisors or security kernels.

Execution-aware memory protection

TrustLite [74] extends the concepts of SMART [33] and SPM [123] to provide a programmable, execution-aware memory protection subsystem for low-cost embedded devices. TrustLite's EA-MPU allows running a number of protected tasks (Trustlets) in parallel without requiring additional CPU instructions. Moreover, the EA-MPU can be programmed to provide individual Trustlets with shared memory and exclusive peripherals access, enabling the construction of secure device drivers and other platform services. TrustLite also proposes a modified CPU exception engine to prevent information leakage to OS interrupt handlers. This allows the OS to perform preemptive multitasking of Trustlets similar to regular tasks, thus facilitating integration of Trustlets with existing software stacks.
To address the assumption of SMART and Sancus that all system memory is cleared on platform reset, TrustLite deploys a Secure Loader that initializes the EA-MPU at boot-time, thus allowing an efficient re-allocation and protection of sensitive memory regions prior to REE invocation. Additionally, instead of having the hardware managing identification tokens for secure inter-process communication (IPC) as in Sancus, TrustLite assumes that low-cost embedded systems do not require the re-allocation or upgrade of TEE tasks at runtime but that TEEs can remain in memory until platform reset.

4.3 Hardware-enhanced Security in Commercially Available Products

Over the past years, several trusted computing research concepts have been realized in industry products, and in many cases such products have fostered new opportunities to build and research trusted systems. In the following sections we review some of the main technologies as well as standardization efforts.

4.3.1 Virtualization and dynamic root of trust

Many mobile and ultra-mobile laptop platforms feature hardware-assisted virtualization technology, such as Intel®Virtualization Technology (Intel VT). A central design goal of Intel VT was to simplify the implementation of robust hypervisors. Intel VT adds two new operation modes: VMX root mode for hypervisors and VMX non-root mode for virtual machines. VMX root mode is very similar to the normal Intel Architecture without Intel VT while VMX non-root mode provides an Intel Architecture environment controlled by a hypervisor. A Virtual-Machine Control Structure (VMCS) was introduced to facilitate transitions between VMX root mode and VMX non-root mode and can be programmed by the hypervisor to establish boundaries on a VM, including access to memory, devices and control registers. While operating in VMX non-root mode, the execution of certain instructions and events cause a transition to VMX root mode called a VMexit. The hypervisor can retrieve details as to the cause of the VMexit by

reading the VMCS and process the event accordingly [94]. Intel VT introduced a generalized IO-MMU architecture which enables system software to define constraints on DMA devices, restricting their access to specific subsets of physical memory allowing for a smaller TCB [4].
Another major capability of modern systems is the DRTM. Available as Intel®Trusted Execution Technology (Intel TXT) or AMD Secure Virtual Machine, this technique enables a CPU to perform a runtime re-initialization and establish a new software TCB (TEE payload), irrespective of the trustworthiness of previously loaded software. For this purpose, the TCG TPM was extended with a set of DRTM PCRs which can be reset at runtime by the CPU by sending a TPM command from the appropriate operation mode (TPM locality). The Intel GETSECS[SENTER] instruction initiates the DRTM. The CPU resets the DRTM PCRs and loads an Authenticated Code Module (ACM) into an isolated execution environment. The ACM performs a series of platform configuration checks, configures DMA protection for the TEE payload and extends the TEE payload hashes into the TPM PCRs.
DRTM technology has been used to securely execute critical software payloads such as SSH logins, X.509 eMail signatures, or to protect banking secrets [85, 50, 24]. Intel TXT has also been used in combination with Intel VT to initiate a trusted hypervisor, which in turn provides multiple TEEs to the individual running VMs [84]. The generalized IO-MMU allows hypervisors to be "disengaged", i.e., to only perform an initial configuration of VM boundaries, thus providing only a minimal external interface and complexity [69]. Alternatively, a "disaggregated" hypervisor may reduce its TCB by delegating drivers for peripherals control to other VMs [91], or to construct a trusted path, providing secure user I/O for TEEs [134].

4.3.2 Userspace trusted execution

Intel®Software Guard Extensions (Intel SGX) are a set of new instructions and memory access changes to the Intel Architecture to support TEEs. The extensions provide the ability to instantiate one or more TEEs (*enclaves*) that reside within an application inside a REE. Accesses to the enclave memory area against software (not resident in the enclave) are prevented by hardware. This restriction is enforced even from privileged software, such as operating systems, virtual machine monitors and BIOS.
The enclave lifecycle begins when a protected portion of an application is loaded into an enclave by system software. The loading process measures the code and data of the enclave and establishes a protected linear address range for the enclave. Once the enclave has been loaded, it can be accessed by the application as a service or directly as part of the application. On first invocation, the enclave can prove its identity to a remote party and be securely provisioned with keys and credentials. To protect its data persistently, the enclave can request a platform specific key unique to the enclave to encrypt data and then use untrusted services of the REE.
To implement Intel SGX memory protections, new hardware and structures are required. The Enclave Page Cache (EPC) is a new region of protected memory where enclave pages and structures are stored. Inside the EPC, code and data from many different enclaves can reside. The processor maintains security and access control information for every page in the EPC in a hardware structure called the Enclave Page Cache Map (EPCM). This structure is consulted by the processor's Page Miss Handler (PMH) hardware module, as shown in Figure 4.3. The PMH mediates access to memory by consulting page tables maintained by system software, range registers and the EPCM. A Memory Encryption Engine (MEE) protects the EPC when using main memory for storage [86].
Enclave binaries are loaded into the EPC using new instructions. ECREATE starts the loading process and initializes the Intel SGX Enclave Control Structure (SECS) which contains global

information about the enclave. EADD loads a page of content into a free EPC page and records the commitment into the SECS. Once the EPC page has been loaded, the contents of the page are measured using EEXTEND. After all the contents of the enclave have been loaded into the EPC, EINIT completes the creation process by finalizing the enclave measurement and establishes the enclave identity. Until an EINIT is executed, enclave entry is not permitted.
Once an enclave has been loaded, it can be invoked by application software. To enter and exit an enclave programmatically (e.g., as part of a call/return sequence), new instructions, EENTER and EEXIT, are provided. While operating in enclave mode, an interrupt, fault or exception may occur. In this case, the processor invokes a special internal routine called Asynchronous Exit (AEX) which saves and clears the enclave register state and translation lookaside buffer (TLB) entries for the enclave. The ERESUME instruction restores the processor state to allow the enclave to resume execution.
To enable attestation and sealing, the hardware provides two additional instructions EREPORT and EGETKEY. The EREPORT instruction provides an evidence structure that is cryptographically protected using symmetric keys. EGETKEY provides enclave software with access to keys used in the attestation and sealing process. A special quoting enclave is devoted to remote attestation. The quoting enclave verifies REPORTs from other enclaves on the platform and creates a signature using a device specific (private) asymmetric key [6].
Intel SGX minimizes the TCB of trusted applications to the critical portion of the application and hardware. This simplifies the creation and validation of remote attestation reports, as remote verifiers no longer have to understand multiple TEE management layers and their dependencies. While requiring CPU extensions, Intel SGX does not require any dependencies on the TPM, a hypervisor or a separate trusted operating system. Further, it is protected against hardware and software attacks on RAM. Finally, Intel SGX enables application developers to directly deploy trusted applications inside REE applications [61].

4.4 Combining Hardware Security and SMPC

We will now discuss viable approaches for combining security through cryptographic protocols with novel hardware-security mechanisms. All three approaches are complementary and can be combined to increase the efficiency, practicality, and security of SMPC.

Protecting Cryptographic Algorithms and Storage The first and most straightforward approach is to use platform security features to improve the security of a deployed SMPC scheme. The goal is to deploy a given SMPC algorithm on a given compute platform while maximizing use of the security features of a platform. This can be done by following the subsequent set of best practices that leverage capabilities of a trusted execution environment:
The first approach to enhance SMPC protocols is to use hardware acceleration to speed up crypographic primitives. On modern Intel Platforms, optimized hardware support is usually offered for AES and SHA. Furthermore, optimized libraries for RSA, ECC, and long-integer arithmetic are available.
The second approach is to isolate the execution of SMPC algorithm from interferences and attacks. A simple form is task or virtual machine isolation. Moving SMPC tasks into a separate VM can render attacks from other VMs more difficult.
The strongest form of isolation are so-called Trusted Execution Environments (TEEs) (such as Intel SGX discussed above). A TEE allows to execute a SMPC algorithm in isolation. Unlike Virtual Machine and Task isolation, TEEs also protect the workload from attacks by a

compromised virtual machine monitor or operating system. Furthermore, a TEE allows other parties to validate the software integrity by means of attestation.

Resolving Performance Bottlenecks The second approach is to analyze a SMPC algorithm for performance bottlenecks and re-implement the functionality of these performance bottlenecks within a trusted execution environment.
One example is to use Intel SGX to implement encrypted search or private membership tests and then use these higher-level primitives in a larger SMPC system.
This approach can be implemented gradually. In the extreme case, a complete algorithm can be distributed into multiple TEEs, executed in confidentiality and isolation, and can then output the desired results after cross-validation with its peers. This would replace a SMPC algorithm by TEE-protected computation of the corresponding circuit/function.
The drawback of using TEEs is a change into a fundamentally different trust model where the players need to trust the hardware security capabilities offered by the hardware vendor. In the traditional security model of MPC, the goal is that each participant only needs to trust itself.

Implementing Trusted Third Parties A third approach to leverage trusted execution environments is to use these services to implement and execute trusted third parties (TTP) required by a protocol. The idea is that one party executes the trusted third party services in a TEE. Other parties can use the attestation capability to verify that the TTP is correctly implemented and executed.

4.5 Using Hardware to Enhance Protection of Applications

In the previous sections, we surveyed hardware capabilities such as Trusted Execution Environments (TEE) that can be used to augment the capabilities of the PRACTICE architecture. In Section 4.4 we then discussed generic approaches to use such hardware security capabilities to enhance security and efficiency of secure multi-party computations.
We will now have a closer look at some of the proposed applications that have been presented in this document and discuss potential integration and benefits of secure hardware.

4.5.1 Malware Detection using Private Set Intersection

The goal of this application described in Section 3.2 is to determine whether a given application signature (e.g. an android app identifier) is listed in a large database of known malware. Technically, the goal is to compute an intersection between a small set (installed application identifiers) and a larger set (the malware database).
One approach to solve these application requirements is to use the primitive of "Private Set Intersection (PSI)" that is described in deliverables D11.1 [107] and D13.1 [70]. This approach already uses hardware cryptographic acceleration offered by today's platforms.
If one requires trust into TEEs, then one viable approach is to use the TEE to implement the complete functionality of private set intersection. This approach has been described in [126].
The goal of the paper is to provide malware look-up such that the server does not learn what malware exists on what device and the client does not learn the complete malware database. The key idea of the design is that the server provides a TEE that can be used as a look-up proxy by a client. The client can the load his malware identifier into this TEE (privacy of the

apps) while the TEE cycles through the database (loading / unloading portions). After cycling through the encrypted malware database, the TEE can declare the subset of applications that are malware. The cycling through the database hides the applications that are looked up from the server. The execution is performed on a server since the database is larger and the server is usually more powerful than a client (e.g. a phone).

4.5.2 Privacy-enhanced Tax Fraud Detection

Section 3.3.1 discusses the detection of fraudulent claims to reimburse value-added tax using MPC. The goal is to scan transactions while determining whether certain transactions are deemed fraudulent. The requirement from the government is to detect fraudulent transactions. The privacy requirement from the monitored enterprises is to not reveal transactions that are not fraudulent.
The first approach to increase efficiency is to scale the computing power available by parallel computation. Sharemind, part of the PRACTICE architecture, offers different trust models and different efficiency levels. As indicated in Section 3.3.1, the most efficient algorithms require three independent parties that are assumed to work correctly and that are assumed not to collude.
As a consequence, the three independent parties can best be protected using three independent Trusted Execution Environments (TEEs). This provides similar independence and isolation while allowing to re-use the existing software components. Unlike the normal Sharemind deployment using three independent (additional) parties, we can now provide three hardware TEEs that provide the same service using only one additional third party.
Instead of placing three TEEs at independent parties, we can now add one TEE to the services at the tax authority, one TEE can be implemented at the monitored enterprise, and a single independent party is needed to provide the third TEE. This substantially enhances the efficiency of the Sharemind system while reducing the requirement for independent parties. This scenario adds hardware protection to the cryptographic protections of Sharemind.
If all parties agree that trust in hardware alone is sufficient and no additional layer of cryptographic defense is needed, then a similar service could be implemented using two TEEs: One TEE would be deployed on a server at the tax authority that implements the fraud detection system. Another TEE is deployed at the monitored companies to collect transaction data. Since the company can validate (attest) the TEE at the tax authority, it can verify that its privacy is fulfilled. The tax authority can also attest a company TEE to ensure that the right data is monitored and submitted.

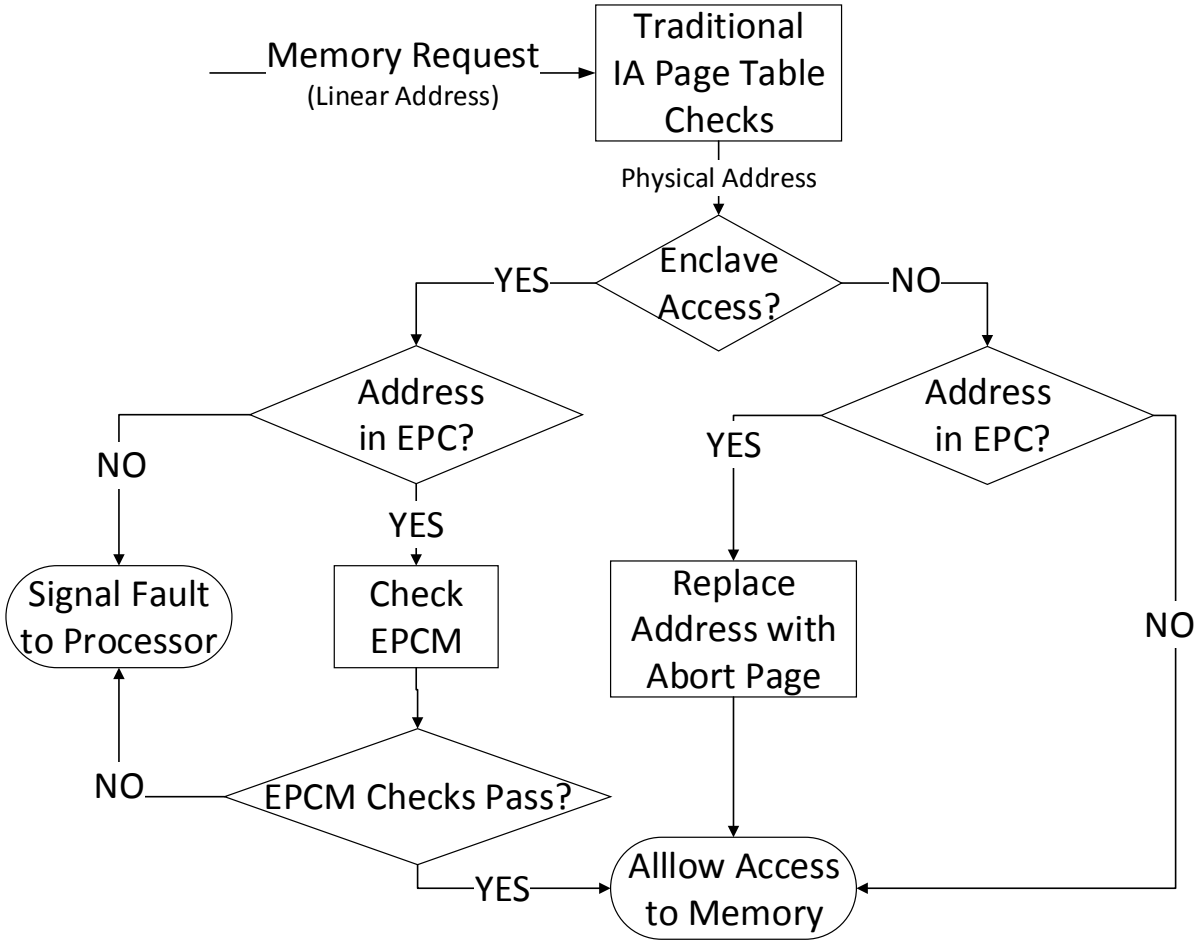


Figure 4.3: High-level architecture of Intel SGX Page Miss Handler (PMH). Processor memory requests are sent to the PMH for translation and access control checks. As part of Intel SGX, the PMH has been extended to check whether a memory access was initiated by an enclave. For non-enclave accesses, the PMH redirects any access to the EPC to non-existent memory (abort page). For an enclave access (an access by enclave to its protected linear address range), the PMH checks that the translated address is an EPC page. Furthermore,the PMH consults the EPCM to verify that the EPC page belongs to enclave requesting access, the correct linear address was used to access the page, and access permissions are consistent with the request.

Chapter 5

Conclusion

In this document we extended the description of a general architecture using secure computation started in deliverable D21.2 [22]. The goal of this work was to present general guidelines for developing and deploying actual applications that use secure computation technologies, e.g., from the PRACTICE architecture. Along with these, we have detailed several example applications that have been developed by project partners on top of the tools from the PRACTICE architecture, e.g., building on top of the ABY, Sharemind, SEEED and FRESCO frameworks. Moreover, we provide guidelines on how to include secure hardware into applications that use secure computation, i.e., how these can benefit from using additional security guarantees provided by the hardware used.
The main contributions of this work can be summarized as follows:

- Chapter 2 serves as the core of this work, where the general guidelines are drawn from the architecture and example applications. Firstly in Section 2.1 we describe **architectural drivers** such as a) identifying the problem for which we use secure computation, b) identifying the usage scenario of secure computation and the expected roles of the actors who participate in the secure computation, and c) analyzing data and assessing the risk corresponding to losing privacy. Then, we consider aspects when it comes to the **design**, i.e., to the **selection of the DAGGER engine** (Section 2.2.1), including its available protocol suite, integration support and programming paradigm as well as the performance level, secure storage, and development tools it provides. **Selecting the DAGGER protocol suite** (Section 2.2.2) depends on the underlying secure computation technique, the deployment and usage models it supports, the level of security it provides, if verifiability and integrity checks are provided, as well as the functionalities it can implement, its performance and the corresponding resource requirements. Furthermore, we discuss aspects in connection with the **construction of the SPEAR application** (Section 2.2.3). This includes a discussion on how to construct the Application Backend on the server side, what are the possible query communication models and solutions for preserving the order of inputs in the databases of SPEAR nodes, as well as the design choices related to the construction of the Application Frontend and the necessary steps for the development of secure computation algorithms. Thereafter, guidelines are given for implementing secure computation algorithms based on the DAGGER platform to achieve the desired **protection of data** (Section 2.2.4). This includes aspects related to the security of the programming model of secure computation, how to maximize the entropy, i.e., the measure of uncertainty of the intermediate values, how to hide links to data sources and handle malicious queries within the SPEAR application. Last but not least, an assessment of the possible **performance optimizations** is given (Section 2.2.5),

which includes techniques such as parallelization of data and tasks, optimization for data type and operations, for the underlying protocol, precomputation and caching as well as a discussion on how trading off some privacy might result in more efficiency.

- Chapter 3 describes **example applications** that show examples for different applications utilizing these design choices. Firstly, **privacy-preserving credit worthiness checking** (Section 3.1) is described based on private function evaluation, where one of the parties possesses a private function to be computed on the other party's input. This is achieved using the ABY secure computation framework and public, so-called universal circuits that are programmed to implement the desired private circuit. Then, a **malware checking mobile application** (Section 3.2) is presented, using private set intersection between a large database and a small amount of applications installed in a mobile phone with restricted resources. Thereafter, **privacy-preserving tax fraud detection** (Section 3.3.1) is described, based on SHAREMIND, making use of parallelization to achieve performance improvements that enable use of secure computation technologies for a real-world application scenario, i.e., detecting tax fraud in Estonia in a privacy-preserving manner. **SEEED-proxy** (Section 3.4), based on encrypted databases, achieves data privacy for cloud based web applications. As our last example application, a **generic data collection application** (Section 3.5) is described, based on Fresco. Here, the computing parties collect data from so-called data providers, after which they allow data users to perform analysis on the collected dataset, using secure computation technologies.

- Chapter 4 then describe aspects when it comes to using **secure hardware** alongside with secure computation technologies. After providing the **basic concepts for hardware-enhanced security** (Section 4.1), the latest **research solutions** for trusted computing are recapitulated (Section 4.2), including alternative trusted computing designs, a description of remote attestation techniques, and low-cost trusted execution environments. Thereafter, **commercially available products**, provided by Intel, that make use of **hardware-enhanced security** are described (Section 4.3). **Combining hardware security and secure computation** and its possible approaches are described afterwards (Section 4.4) and two examples for **using hardware to enhance protection of applicatins** from Chapter 3 are described (Section 4.5).

In this document, we present a large set of general guidelines with novel examples on how to integrate different secure computation tools into applications developed and how to deploy them in the cloud. Our guidelines describe advantages and disadvantages of available solutions from which developers can choose from, which can provide useful information for the designers of secure computation based applications. However, as can be seen in our application scenarios, every application has its unique requirements and possibilities, and therefore, a throrough analysis of available techniques is needed for the design of novel applications.
We believe that this deliverable, alongside with D21.2 [22] and D14.1 [30] can be sensefully used by developers of real-life applications as guideline for their design process and it helps providing insights to the different available techniques and research solutions.

Chapter 6

List of Abbreviations

ABE	Attribute-Based Encryption
ACM	Authenticated Code Module
AEX	Asynchronous Exit
AWS	Amazon Web Services
BF	Bloom Filter
CBF	Counting Bloom Filter
DAGGER	Distributed Aggregation and Security Services
DBaaS	Database as a Service
DBMS	Database Management System
DP	Data Provider
DSL	Domain Specific Language
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
FHE	Fully Homomorphic Encryption
GC	(Yao's) Garbled Circuit
HE	Homomorphic Encryption
IMEI	International Equipment Identifier
Intel TXT	Intel®Trusted Execution Technology
Intel SGX	Intel®Software Guard Extensions
Intel VT	Intel®Virtualization Technology
MEE	Memory Encryption Engine
MPC	Multi-Party Computation
MPI	Message Passing Interface
ORG	Organizer
OT	Oblivious Transfer

PaaS	Platform-as-a-Service
PFE	Private Function Evaluation
PHE	Partically Homomorphic Encryption
PMH	Page Miss Handler
PRACTICE	Privacy-Preserving Computation in the Cloud
PSI	Private Set Intersection
REE	Rich Execution Environment
RPC	Remote Procedure Call
SaaS	Software as a Service
SCE	Secure Computation Engine
SCS	Secure Computation Specification
SECS	Intel SGX Enclave Control Structure
SFE	Secure Function Evaluation
SFDL	Secure Function Description Language
SHDL	Secure Hardware Description Language
SHE	Somewhat Homomorphic Encryption
SMART	Secure Minimal Architecture for Root of Trust
SMPC	Secure Multiparty Computation
SPEAR	Secure Platform for Enterprise Applications and Services
SPM	Software Protected Module
SSI	Secure Service Interface
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
TSM	Trusted Software Module
TTP	Trusted Third Party
UC	Universal Circuit
URI	Uniform Resource Indicator
VAT	Value-Added Tax
VM	Virtual Machine
VMCS	Virtual-Machine Control Structure
WP	Work Package

Bibliography

[1] JavaScript Object Notation (JSON). `http://www.json.org/`.

[2] OData - the protocol for REST APIs. `http://www.odata.org/`.

[3] Martín Abadi and Joan Feigenbaum. Secure circuit evaluation. *J. Cryptology*, 2(1):1–12, 1990.

[4] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[5] Ammar Alkassar, Michael Scheibel, Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Security architecture for device encryption and VPN.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Hardware and Architectural Support for Security and Privacy (HASP)*, New York, NY, USA, 2013. ACM.

[7] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Security and Privacy (S&P)*, Washington, DC, USA, 1997. IEEE.

[8] ARM. ARM security technology — Building a secure system using TrustZone technology. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html`, April 2009.

[9] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. An efficient implementation of trusted channels based on OpenSSL. pages 41–50.

[10] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Conference on Computer and Communications Security (CCS)*. ACM, November 2013.

[11] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS'13*, pages 535–548. ACM, 2013.

[12] N. Asokan, Jan-Erik Ekberg, Kari Kostiainen, Anand Rajan, Carlos V. Rozas, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. Mobile trusted computing. *Proceedings of the IEEE*, 102(8):1189–1206, 2014.

[13] Nuttapong Attrapadung. Fully secure and succinct attribute based encryption for circuits from multi-linear maps. *IACR Cryptology ePrint Archive*, 2014:772, 2014.

[14] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.

[15] Manuel Barbosa, Bernardo Portela, Berry Schoenmakers, Niels de Vreede, Guilaume Scerri, and Bogdan Warinschi. PRACTICE Deliverable D13.2: efficient verifiability and precise specification of secure computation functionalities, 2016.

[16] 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 *European Symposium on Research in Computer Security – ESORICS'09*, volume 5789 of *LNCS*, pages 424–439. Springer, 2009.

[17] Donald Beaver. Precomputing oblivious transfer. In *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 1995.

[18] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM CCS'08*, pages 257–266. ACM, 2008.

[19] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[20] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *LNCS*, pages 227–234. Springer, 2015.

[21] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic programming of privacy-preserving applications. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14, pages 53–65. ACM, 2014.

[22] Jonas Böhler, Florian Hahn, Ágnes Kiss, Kasper Lyneborg Damgård, Peter Sebastian Nordholt, Reimo Rebane, Roman Jagomägis, Matthias Schunter, Bernardo Portela, Manuel Barbosa, and Niels de Vreede. PRACTICE Deliverable D21.2: unified architecture for programmable secure computations, 2015.

[23] Jonas Böhler, Florian Hahn, Raad Bahmani, Daniel Demmler, Ágnes Kiss, Thomas Schneider, Michael Stausholm, Reimo Rebane, José Bacelar Almeida, Manuel Barbosa, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. PRACTICE Deliverable D22.3: Software development kit and tools prototype (1^st version), 2015.

[24] Franz Ferdinand Brasser, Sven Bugiel, Atanas Filyanov, Ahmad-Reza Sadeghi, and Steffen Schulz. Softer smartcards — Usable cryptographic tokens with secure execution. In *Financial Cryptography*, pages 329–343.

[25] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *ACM CCS'07*, pages 498–507. ACM, 2007.

[26] Serdar Cabuk, Chris I. Dalton, Konrad Erikkson, Dirk Kuhlmann, HariGovind V. Ramasamy, Gianluca Ramuno, Ahmad-Reza Sadeghi, Matthias Schunter, and Christian Stüble. Towards automated security policy enforcement in multi-tenant virtual data centers. *Computer Security*, 18:89–121, 2010.

[27] Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller. One-round secure computation and secure autonomous mobile agents. In *International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *LNCS*, pages 512–523. Springer, 2000.

[28] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. Property-based attestation without a trusted third party. In Tzong-Chen, Wu Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security Conference (ISC)*, volume 5222 of *LNCS*, pages 31–46. Springer, September 2008.

[29] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 42(2):2–13, March 2008.

[30] Kasper Lyneborg Damgård, Peter Sebastian Nordholt, Roman Jagomägis, Hugo Pacheco, and Manuel Barbosa. PRACTICE Deliverable D14.1: Architecture, 2015.

[31] Kasper Lyneborg Damgård, Peter Sebastian Nordholt, and Thomas Jakobsen. PRACTICE Deliverable D14.2: Platform for Secure Computation, 2015.

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communincations of the ACM*, 51(1):107–113, January 2008.

[33] Karim E. Defrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2012.

[34] 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'15)*. The Internet Society, 2015. Code: `http://encrypto.de/code/ABY`.

[35] Konrad Durnoga, Stefan Dziembowski, Tomasz Kazana, and Michal Zajac. One-time programs with limited memory. In *Information Security and Cryptology (INSCRYPT'13)*, volume 8567 of *LNCS*, pages 377–394. Springer, 2013.

[36] Jeffrey Dwoskin and Ruby Lee. Hardware-rooted trust for secure key management and transient trust. In *Computer and Communication Security (CCS)*, pages 389–400, October 2007.

[37] Joan Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 Secure Coprocessor. *IEEEC*, 34(10):57–66, 2001.

[38] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. The untapped potential of trusted execution environments on mobile devices. *IEEE Security and Privacy Magazine*, 2014. `http://dx.doi.org/10.1109/MSP.2014.38`.

[39] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *IEEE Computer*, 36(7):55–63, 2003.

[40] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[41] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In *ACM CCS'15*, pages 844–855. ACM, 2014.

[42] Ben Fisch, Binh Vo, Fernando Krell, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steven M. Bellovin. Malicious-client security in Blind Seer: A scalable private DBMS. In *IEEE Symposium on Security and Privacy (S&P'15)*, pages 395–410. IEEE, 2015.

[43] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[44] Message Passing Interface Forum. MPI: A message passing interface standard, version 3.1, June 2015.

[45] Aurelien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. Systematic treatment of remote attestation. `http://eprint.iacr.org/2012/713.pdf`, 2012. Cryptology ePrint Archive.

[46] Aurélien Francillon, Quan Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe*, pages 1–6. IEEE, March 2014.

[47] Keith B. Frikken, Mikhail J. Atallah, and Jiangtao Li. Attribute-based access control with hidden policies and hidden credentials. *IEEE Transactions on Computers*, 55(10):1259–1270, 2006.

[48] Keith B. Frikken, Mikhail J. Atallah, and Chen Zhang. Privacy-preserving credit checking. In *ACM Electronic Commerce (EC'05)*, pages 147–154. ACM, 2005.

[49] Keith B. Frikken, Jiangtao Li, and Mikhail J. Atallah. Trust negotiation with hidden credentials, hidden policies, and policy cycles. In *Network and Distributed System Security (NDSS'06)*, pages 157–172. The Internet Society, 2006.

[50] Sebastian Gajek, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. TruWallet: Trustworthy and migratable wallet-based web authentication. pages 19–28.

[51] Ryan W. Gardner, Sujata Garera, and Aviel D. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *Trans. Info. For. Sec.*, 4(4):638–650, 2009.

[52] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. pages 193–206.

[53] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Fully secure attribute based encryption from multilinear maps. *IACR Cryptology ePrint Archive*, 2014:622, 2014.

[54] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Advances in Cryptology – EUROCRYPT'13*, volume 7881 of *LNCS*, pages 626–645. Springer, 2013.

[55] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable Yao circuits. In *Advances in Cryptology – CRYPTO'10*, volume 6223 of *LNCS*, pages 155–172. Springer, 2010.

[56] Jonathon T. Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *Annual Computer Security Applications Conference (ACSAC)*, pages 23–32, Washington, DC, USA, 2005. IEEE.

[57] Global Platform. TEE system architecture. `http://www.globalplatform.org/specificationsdevice.asp`, December 2011.

[58] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. pages 21–24.

[59] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *ACM Symposium on Theory of Computing (STOC'87)*, pages 218–229. ACM, 1987.

[60] J. Gregorio and B. de hOra. The Atom Publishing Protocol. RFC 5023 (Proposed Standard), October 2007.

[61] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Hardware and Architectural Support for Security and Privacy (HASP)*, New York, NY, USA, 2013. ACM.

[62] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[63] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In *Theory of Cryptography Conference (TCC'07)*, volume 4392 of *LNCS*, pages 575–594. Springer, 2007.

[64] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-reduced integrity measurement architecture. In *Symposium on Access control models and technologies (SACMAT)*, pages 19–28, New York, NY, USA, 2006. ACM.

[65] Roman Jagomägis. SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[66] Markus Jakobsson and Karl-Anders Johansson. Retroactive detection of malware with applications to mobile platforms. In *Workshop on Hot Topics in Security (HotSec)*, Washington, DC, August 2010. USENIX.

[67] Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In *Advances in Cryptology – ASIACRYPT'11*, volume 7073 of *LNCS*, pages 556–571. Springer, 2011.

[68] Bernhard Kauer. OSLO: improving the security of trusted computing.

[69] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized cloud infrastructure without the virtualization. In *International Symposium on Computer Architecture (ISCA)*, pages 350–361, New York, NY, USA, 2010. ACM.

[70] Florian Kerschbaum, Florian Hahn, Thomas Schneider, Michael Zohner, Pille Pullonen, and Claudio Orlandi. PRACTICE Deliverable D13.1: A Set of New Protocols, 2015.

[71] Michael S. Kirkpatrick, Gabriel Ghinita, and Elisa Bertino. Resilient authenticated execution of critical applications in untrusted environments. *IEEE Transactions Dependable Secure Computing*, 9(4), 2012.

[72] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In *Advances in Cryptology – EUROCRYPT'16*, LNCS. Springer, 2016.

[73] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. Cryptology ePrint Archive, Report 2016/093, 2016. `http://eprint.iacr.org/2016/093`.

[74] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *European Conference on Computer Systems (EuroSys)*. ACM, April 2014.

[75] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[76] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography and Data Security (FC'08)*, volume 5143 of *LNCS*, pages 83–97. Springer, 2008. Code: `http://encrypto.de/code/FairplayPF`.

[77] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *Security and Privacy (S&P)*. IEEE, May 2012.

[78] Peeter Laud and Alisa Pankova. Preprocessing-based verification of multiparty protocols with honest majority. Cryptology ePrint Archive, Report 2015/674, 2015. `http://eprint.iacr.org/`.

[79] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: Verifying the Integrity of PERipherals' firmware. In *ACM Conference on Computer and Communications Security*, pages 3–16. ACM, October 2011.

[80] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[81] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society, 2015.

[82] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*.

[83] Jonathan M. McCune, Stefan Berger, Ramon Caceres, Trent Jaeger, and Reiner Sailer. Bridging mandatory access control across machines. Research Report RC23778, IBM Research Division, November 2005.

[84] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. pages 143–158, Oakland, CA, May 2010. IEEE.

[85] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*, pages 315–328. ACM, 2008.

[86] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Hardware and Architectural Support for Security and Privacy (HASP)*, New York, NY, USA, 2013. ACM.

[87] Mario Mnzer, Florian Hahn, Cem Kazan, Elif Özdoğan, Buket Serper, Fabian Taigel, Julian Kurz, Jan Meller, Richard Pibernik, Antonio Zilli, Angelo Corallo, Giuseppe Grassi, Marianna Lezzi, Stelvio Cimato, Ernesto Damiani, and Kurt Nielsen. PRACTICE Deliverable D24.3: Industrial Settings, 2015.

[88] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *Advances in Cryptology – EUROCRYPT'13*, volume 7881 of *LNCS*, pages 557–574. Springer, 2013.

[89] Payman Mohassel, Seyed Saeed Sadeghian, and Nigel P. Smart. Actively secure private function evaluation. In *Advances in Cryptology – ASIACRYPT'14*, volume 8874 of *LNCS*, pages 486–505. Springer, 2014.

[90] Tobias Mueller, Niklas Büscher, Hiva Mahmoodi, Janus Dam Nielsen, Peter S. Nordholt, Dan Bogdanov, Manuel Barbosa, Johannes U Jensen, and Kurt Nielsen. PRACTICE Deliverable D22.2: tools design document, 2014.

[91] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen security through disaggregation. In *SIGPLAN/SIGOPS Virtual Execution Environments (VEE)*, pages 151–160, New York, NY, USA, 2008. ACM.

[92] Aarthi Nagarajan, Vijay Varadharajan, Michael Hitchens, and Eimear Gallery. Property based attestation and trusted computing: Analysis and challenges. In Yang Xiang, Javier Lopez, Haining Wang, and Wanlei Zhou, editors, *Network and System Security (NSS)*, pages 278–285. IEEE, 2009.

[93] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *ACM Electronic Commerce (EC'99)*, pages 129–139, 1999.

[94] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[95] Janus Dam Nielsen, Florian Hahn, Daniel Demmler, Hiva Mahmoodi, Thomas Schneider, Peter Sebastian Nordholt, Michael Stausholm, Roman Jagomägis, Matthias Schunter, Meilof Veeningen, Niels de Vreede, Antonio Zilli, Johannes Ulfkjaer Jensen, and Kurt Nielsen. PRACTICE Deliverable D21.1: Deployment Models and Trust Analysis for Secure Computation Services and Applications, 2014.

[96] Salman Niksefat, Babak Sadeghiyan, Payman Mohassel, and Seyed Saeed Sadeghian. ZIDS: A privacy-preserving intrusion detection system using secure two-party computation protocols. *Comput. J.*, 57(4):494–509, 2014.

[97] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*. USENIX, August 2013.

[98] Rafail Ostrovsky and William E. Skeith III. Private searching on streaming data. In *Advances in Cryptology – CRYPTO'05*, volume 3621 of *LNCS*, pages 223–240. Springer, 2005.

[99] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind Seer: A scalable private DBMS. In *IEEE Symposium on Security and Privacy (S&P'14)*, pages 359–374. IEEE, 2014.

[100] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. pages 414–429, Oakland, CA, May 2010. IEEE.

[101] Marko Jõemets Reimo Rebane Meril Vaht Johannes Ulfkjær Jensen Kurt Nielsen Peter S. Nordholt, Roman Jagomägis. PRACTICE Deliverable D23.1: secure survey prototype - a supplementary report, 2015. Available from `http://www.practice-project.eu`.

[102] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2004. USENIX.

[103] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research, April 2001.

[104] Tiit Pikma. Auditing of Secure Multiparty Computations. Master's thesis, Institute of Computer Science, University of Tartu, 2014.

[105] Benny Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explorations*, 4(2):12–19, 2002.

[106] Benny Pinkas, Florian Kerschbaum, Florian Hahn, Thomas Schneider, Michael Zohner, and Reimo Rebane. PRACTICE Deliverable D11.2: An Evaluation of Current Protocols based on Identified Model, 2015. Available from `http://www.practice-project.eu`.

[107] 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, 2015. Available from `http://www.practice-project.eu`.

[108] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 515–530, Washington, D.C., August 2015. USENIX Association.

[109] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000.

[110] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms.

[111] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor.

[112] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. pages 16–16.

[113] Tomas Sander, Adam L. Young, and Moti Yung. Non-interactive cryptocomputing for NC^1. In *Foundations of Computer Science (FOCS'99)*, pages 554–567. IEEE, 1999.

[114] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote attestation on legacy operating systems with Trusted Platform Modules. *Sci. Comput. Program.*, 74(1-2):13–22, 2008.

[115] Steffen Schulz and Ahmad-Reza Sadeghi. Secure VPNs for trusted computing environments. pages 197–216.

[116] Marcel Selhorst and Christian Stüble. TrustedGRUB project. `http://sf.net/projects/trustedgrub/`, 2010.

[117] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. pages 1–16.

[118] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. SWATT: SoftWare-based ATTestation for embedded devices.

[119] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In Refik Molva, Gene Tsudik, and Dirk Westhoff, editors, *Security and Privacy in Ad-hoc and Sensor Networks*, volume 3813 of *LNCS*, chapter 3, pages 27–41. Springer, Berlin/Heidelberg, 2005.

[120] Sean W. Smith. Outbound authentication for programmable secure coprocessors. pages 72–89.

[121] Jay Srage and Jerome Azema. M-Shield mobile security technology, 2005. TI White paper. `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`.

[122] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *Computer and Communications Security (CCS)*, pages 2–13. ACM, 2012.

[123] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In Sushil Jajodia and Jianying Zhou, editors, *Security and Privacy in Communication Networks (SecureComm)*, volume 50 of *LNCS*, chapter 20, pages 344–361. Springer, Berlin, Heidelberg, September 2010.

[124] Edward Suh, Charles O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random function. In *International Symposium on Computer Architecture (ISCA)*, pages 25–36, May 2005.

[125] Riivo Talviste. *Applying Secure Multi-party Computation in Practice*. PhD thesis, University of Tartu, 2016.

[126] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. *CoRR*, abs/1606.01655, 2016.

[127] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[128] Trusted Platform Module (TPM) Specifications. `https://www.trustedcomputinggroup.org/specs/TPM/`.

[129] Leslie G. Valiant. Universal circuits (preliminary report). In *ACM Symposium on Theory of Computing (STOC'*.

[130] Amit Vasudevan, Jonathan Mccune, James Newsome, Adrian Perrig, and Leendert Van Doorn. CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*. ACM, May 2012.

[131] Ingo Wegener. *The complexity of Boolean functions*. Wiley-Teubner, 1987.

[132] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *Foundations of Computer Science (FOCS'86)*, pages 162–167. IEEE, 1986.

[133] Bennet S. Yee. *Using secure coprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1994. CMU-CS-94-149.

[134] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. pages 616–630, Washington, DC, USA, 2012. IEEE.