D14.1
Architecture

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 / D14.1 / 1.0
Activity and WP:	Activity 1 / WP14
Due Date:	October 2015 - M24
Actual Submission Date:	31st October, 2015

Responsible Organisation:	CYBER
Editor:	Roman Jagomägis
Dissemination Level:	Public
Revision:	1.0

Abstract:	This document describes the architecture for integrating the Secure Computation Technique implementations into DAGGER systems.
Keywords:	Architecture, Protocol, Protocol Suite, Secure Computation, Integration, Virtual Machine, Translation, Resources

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

Editor

Roman Jagomägis (CYBER)

Contributors (ordered according to beneficiary numbers)

Kasper Lyneborg Damgård(ALX)
Peter Sebastian Nordholt (ALX)
Roman Jagomägis (CYBER)
Hugo Pacheco (INESC PORTO)
Manuel Barbosa (INESC PORTO)

Executive Summary

The objective of work package WP14 is to construct implementations of proposed solutions. Beyond that, it also aims to provide a library for those in the academic community and industry who are interested in secure computation technology. The existence of such a library will bring secure computation another step closer to being a tool that is used in practice.
Deliverable D14.1 provides the architecture for the generic secure computing platform that allows to easily implement and integrate secure computation technologies. The purpose of the following design is to make technology implementations portable and flexible to use, while simplifying their internal complexity. This work builds upon the deliverable D21.2 [2] by presenting a more detailed perspective on a Secure Computation Technology related subset of the more general SPEAR architecture and covers its layers related to the integration of secure computation into DAGGER. The two deliverables D14.1 and D21.2 [2] complement each other in the sense that D21.2 [2] describes the general structure of the architecture for building secure computation services and applications, whereas the detailed information on the implementation and integration of Secure Computation Technologies into the DAGGER part of the larger architecture lies within D14.1. This document describes in depth the architecture of the interfaces and interactions between the actual cryptographic protocols, the Secure Computation Engine and the Secure Computation Specification as described in D21.2 [2] and therefore fits into the larger SPEAR & DAGGER picture. The architecture proposed in D14.1 will be implemented in deliverable D14.2 [1].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose

This deliverable presents the architecture for implementing *Secure Computation Technologies* and integrating them into existing *DAGGER (Distributed Aggregation and Security Services) Platforms*. These technologies are often developed using non-standard ad-hoc approaches that require more effort to implement and are more difficult to maintain and use in practice. The purpose of the following design is to make technology implementations portable and flexible to create and use while simplifying their internal complexity. This would improve the productivity of those in the academic community and the industry, and remove the compatibility issues making the final implementations easier to compare and integrate into existing secure computation systems and applications. We achieve these goals by breaking dependence of technology implementations on the rest of DAGGER, by defining mechanisms for integrating them into DAGGER, and by splitting the responsibilities between the technology implementations and the DAGGER.

1.2 Scope

This document builds upon the deliverable D21.2 [2] by presenting a more detailed perspective on a Secure Computation Technology related subset of the more general SPEAR (*Secure Platform for Enterprise Applications and Services*) architecture described in D21.2. The two deliverables D14.1 and D21.2 [2] complement each other in the sense that D21.2 [2] describes the general structure of the architecture for building secure computation services and applications, whereas the detailed information on the implementation and integration of Secure Computation Technologies into the DAGGER part of the larger architecture lies within D14.1. This document describes in depth the architecture of the interfaces and interactions between the actual cryptographic protocols, the Secure Computation Engine and the Secure Computation Specification as described in D21.2 [2] and therefore fits into the larger SPEAR & DAGGER picture. Both documents use similar terminology and structure.

We begin by identifying the goals of the stakeholders. We continue by analyzing the use cases involving the stakeholders and derive the functional requirements that will enable the designed system to achieve the goals. Next, we discuss the quality attributes which we believe would be the advantages of implementing the system according to the architecture. In chapter 3, we describe the architecture using different views which should give a good understanding of how to implement the architecture. Thoughout the document we supply the reader with explanatory diagrams that follow the standard UML 2.0 conventions.

1.3 Glossary

To help the reader with terminology and abbreviations used in this document, we include the following glossary of the most common terms.

Secure Multiparty Computation (SMC) The act of computing in a secure manner between multiple parties.

Secure Computation Protocol Suite (SCPS) Is the implementation of an entire secure computation technique such as Yao circuits or SPDZ. It consists of several protocols.

Secure Computation Protocol (SCP) Is a single-purpose protocol, which can be invoked via some interface by the Secure Computation Engine. I.e., think of a multiplication protocol or an addition protocol.

Secure Computation Specification (SCS) Is a component that describes the secure computation segment of a concrete application, in such a way that it can be understood by the SCE.

For example, in Sharemind the SCS is a byte-code program read by the SCE at runtime. The byte-code is produced by compiling high-level code written in the SecreC language.

A more general approach is that of FRESCO. Here the SCS is essentially a stream of instructions which are generated at runtime. Thus the SCS is produced by writing a succinct program that can generate this stream of instructions when needed.

Secure Computation Engine (SCE) Is the system that provides the software infrastructure necessary to organize and facilitate the successful execution of secure applications (SCS) utilizing Secure Computation Technologies implemented by SCPS. The engine reads an SCS and evaluates it by calling the appropriate SCPs provided by SCPS.

SPEAR The *Secure Platform for Enterprise Applications and Services* (SPEAR) provides a cryptographically secure computation platform as a service for cloud applications and services. Its construction is described in the deliverable D21.2 [2].

DAGGER The *Distributed Aggregation and Security Services* (DAGGER) is the key subsystem of SPEAR that represents the middleware for using cryptographically secure computing in cloud applications. It represents a complete secure computation system including the SCS and SCE modules and a number of SCPS modules (as described in D21.2 [2] and further specified in this document). Due to the special knowledge of the underlying cryptographic protocols required to develop SCPSs, these will be developed separately from the remainder of the DAGGER system before being integrated into the DAGGER system.

When the context is clear we may drop the *Secure Computation* prefix from most of these terms.

Chapter 2

Architectural Drivers

This chapter focuses on the different aspects that drive the architecture forward and ensure that it has a place in the world. We describe the different requirements that envision for such an architecture, and also describe the different stakeholders. Based on this, we present a number of use-cases that the architecture should be able to handle. Lastly, we define the quality attributes that this architecture provides.

2.1 Business Requirements

This section lists the identified business goals and requirements, that the management and a board of directors would typically understand. These describe the general motivation behind the architecture and mostly focus on high-level marketing considerations and cost efficiency.

Requirement B1: Bring secure computation closer to being used in practice.

Requirement B2: Enable researchers to save cost implementing, testing and comparing secure computation techniques.

Requirement B3: Allow those in the industry to quickly integrate new secure computation techniques in their secure computation applications.

Requirement B4: Increase reusability of secure computation technique implementations across different products.

2.2 Stakeholders

In the following we identify the stakeholders involved in this architecture, including the different roles they might have as well as the significant goals of these roles. The main focus will be on developers, as the architecture described in this document mainly concerns with the implementation and integration of SMC protocols. We present an overview of stakeholders, roles and goals in Table 2.1, followed by the brief descriptions of the different roles.

Stakeholder	Roles	Significant goals
Application User	Input Party	**Goal 1:** Provide input data to the application. **Goal 2:** Retain complete control over the privacy of their data.
	Result Party	**Goal 3:** Receive resulting output data from the application.
Application Service Provider	Application Developer	**Goal 4:** Write applications utilizing secure computation technologies. **Goal 5:** Achieve sufficient security guarantees for the application. **Goal 6:** Achieve a sufficient performance for the application. **Goal 7:** Minimize vendor and technology lock in.
Secure Technology Provider	Protocol Developer	**Goal 8:** Implement a new secure computation technology. **Goal 9:** Compare with existing implementations of secure computation technologies. **Goal 10:** Minimize implementation overhead. **Goal 11:** Minimize vendor and technology lock in.
	DAGGER Platform Developer	**Goal 12:** Provide application developers access to different kinds of secure computation technologies. **Goal 13:** Enable the execution of implementations of secure computation technologies. **Goal 14:** Provide secure computation technology implementations with needed resources. **Goal 15:** Execute secure computation protocols according to predefined specification.

Table 2.1: Stakeholders with their roles and goals.

Protocol Developer: A protocol developer implements concrete secure computation technologies as protocol suite modules. This is a complex task requiring specialized knowledge about the underlying cryptographic protocols. Often even subtle errors can cause the implementation of a theoretically secure protocol to be completely insecure. For this reason, we imagine that developers filling this role will typically be cryptographers with a deep knowledge of the protocols they are implementing (possibly the inventors of the theoretic protocol).
Furthermore, since developing a protocol suite requires special skill and attention to detail it is preferable if the task can be done independently of a concrete DAGGER system. This should allow the protocol suite developer to focus on the protocol implementation without worrying about the details of the larger system and for the finished protocol suite to be integrated in multiple DAGGER systems.

DAGGER Developer: A DAGGER developer implements and maintains a given DAGGER system. DAGGER developers can be expected to have some knowledge about secure computation technologies in general but not necessarily to have deep knowledge about each technology.

As new or improved secure computation technologies are implemented, the DAGGER developer may want to include these technologies in his DAGGER system. This would involve integrating the new implementation in the form of a protocol suite. They need the integration process to be simple.

Application Developer: An application developer implements and maintains high-level applications that utilize secure computation. Typically, application developers will work in a high-level language using some tool (like a *Domain-Specific Language* (DSL) and a compiler) to translate their secure applications into the SCS of the underlying system. They are expected to have only a superficial understanding of secure computation technologies and the concrete protocols they intend to use. Application developers may, however, be aware of certain special features of the computation techniques that they would want to use.

2.3 Use cases

A use case is a description of a goal-oriented set of interactions between actors and the system. It considers *who* does *what* to the system and for what *purpose*. The main intention is to capture all the different possible interactions with the system such that the entire system is covered. We leave out harmful interaction and malicious behavior. In the following we present a detailed description for the use cases depicted in Figure 2.1.

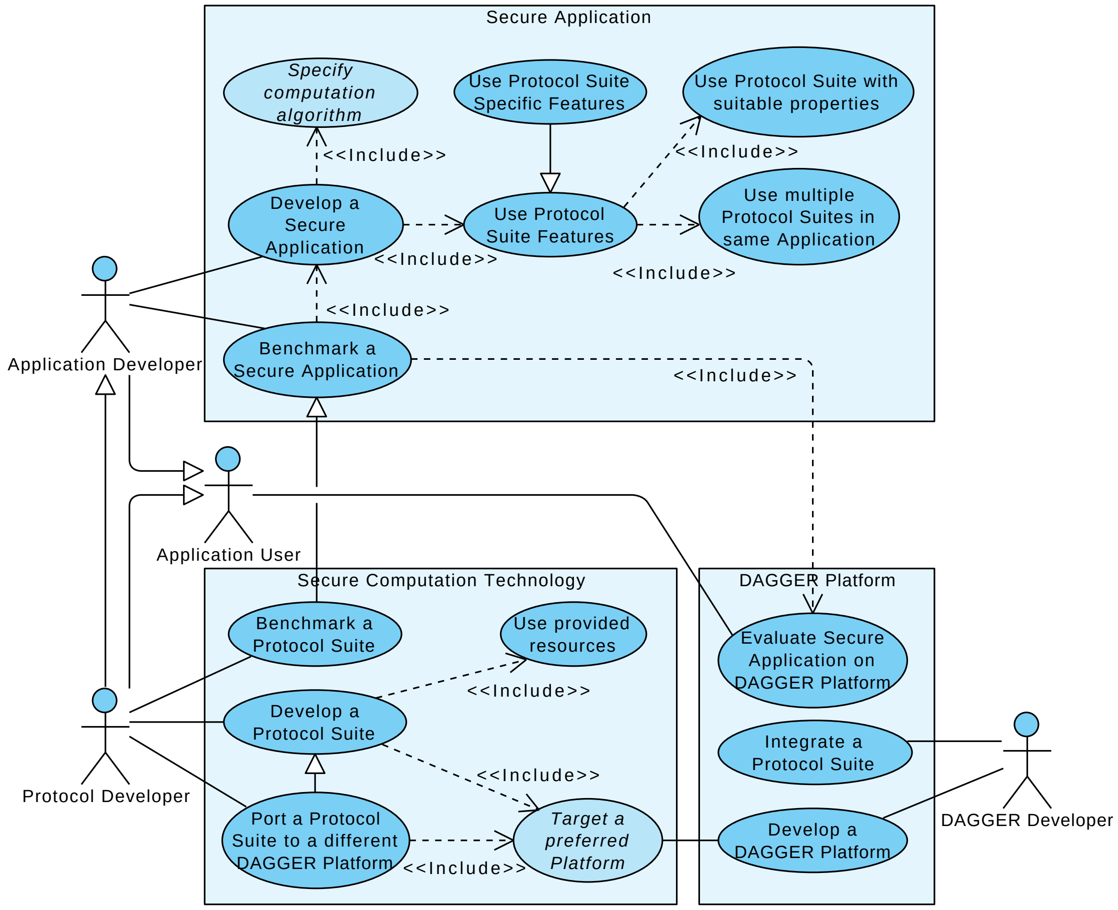

Figure 2.1: The main use cases involving the development and use of Secure Computation Technologies.

Use Case 1:	Develop a Protocol Suite
Actors:	Protocol Developer Jarrod
Brief:	Jarrod has invented a new secure computation technology and wants to implement it as a protocol suite following the DAGGER architecture.
Steps:	1. Jarrod implements the technology following the best suited design for the technology. He can assume that certain common resources such as network interaction is provided by the SCE via a standard interface. This way Jarrod can focus his attention on the specifics of his new technology. 2. Jarrod ensures that the secure computation protocols he wants to expose to the SCE follow the protocol interface. 3. Once Jarrod is done with his implementation he publishes it along with appropriate documentation. DAGGER developers can now start integrating Jarrod's new protocol suite into their systems.

Table 2.2: Use Case 1. Develop a Protocol Suite

Use Case 2:	Develop a Generic Secure Application.
Actors:	Application Developer Michael
Brief:	Michael wants to develop a secure application that computes set intersection on top of a DAGGER system. He does not beforehand know which Protocol Suites are integrated into the DAGGER system he is using. This is configured later when the application needs to be run.
Steps:	1. Michael writes his set intersection application using a high-level language that compiles into a specification the SCE can execute. The code of his application is generic in the sense that it is independent of the Protocol Suite that will eventually run the application. 2. Michael configures the application to use a Protocol Suite that can compute the application he wrote. 3. Michael tests his application by running it using his DAGGER system. He reiterates the process correcting his implementation until he is satisfied with the result. Once working, he could benchmark his application (see the use case "Benchmarking Secure Computation Technologies") and switch protocol suite if needed.

Table 2.3: Use Case 2. Develop a Generic Secure Application.

Use Case 3:	Make Use of Protocol Suite Specific Features.
Actors:	Application Developer Michael
Brief:	Michael has developed an application on top of a DAGGER system using a specific protocol suite (see the general above use case). He has now learned about a special feature in a particular protocol suite which could greatly improve the performance of his application. He wants to alter his application to utilize this feature.
Steps:	1. Michael first reads the documentation for the protocol suite to understand how to utilize the new feature. 2. Michael studies the code of the secure application he has already written and identifies the part where he can use the new feature. 3. Using a high-level language he rewrites the identified part of the application to take advantage of the new feature. This may involve importing protocol-specific packages in his code. As for the remainder of the application, however, code can be left relatively unchanged. 4. As in the above use case Michael configures the application to use the desired Protocol Suite and generates the SCS of his application using some tool. 5. Michael tests his application by running it on the DAGGER system.

Table 2.4: Use Case 3. Make Use of Protocol Suite Specific Features.

Use Case 4:	Create a new DAGGER Secure Computation Engine
Actors:	DAGGER Developer Paige
Brief:	Paige is creating a Secure Computation Engine for a DAGGER system. She wants the SCE to be able to use protocol suites via the interfaces of DAGGER.
Steps:	1. Paige first implements a module to handle common resources that protocol suites expect the SCE to provide. These include components such as network communication and persistent storage. The resource module should expose resources via common interfaces expected by the protocol suites. 2. Paige then writes a VM capable of reading the SCS format of her DAGGER system in order to execute secure computations described in this format. 3. To make protocol suites work in her DAGGER system she writes a translator module. This module takes the instructions of the SCS of her DAGGER system and maps them to concrete protocol calls in the protocol suites. 4. Finally Paige can write a few sample applications to test that the new SCE is working as expected.

Table 2.5: Use Case 4. Create a new DAGGER Secure Computation Engine

Use Case 5:	Integrate a new Protocol Suite
Actors:	DAGGER Developer Anne
Brief:	Anne wants to integrate a new protocol suite into her DAGGER system.
Steps:	1. Anne obtains the protocol suite from the Internet along with the appropriate documentation. 2. Anne studies the documentation to learn the features provided by the protocol suite. 3. If the new protocol suite contains special non-standard features, Anne may want to extend the high-level secure language (and compiler) of her DAGGER system with a module to support these features. She might optionally need to implement a translation module plugin, so that her SCE can successfully translate the SCS of her DAGGER system to calls to these non-standard features of the new protocol suite. 4. Anne adds the integration modules and plugins to her DAGGER system and runs tests to check that the new protocol suite is working correctly.

Table 2.6: Use Case 5. Integrate a new Protocol Suite

Use Case 6:	Benchmarking Secure Computation Technologies.
Actors:	Application Developer Dave
Brief:	Dave is using a DAGGER system that has several integrated protocol suites. He wants to know which of these would be best suited for his auction application. He decides to benchmark the available protocol suites to find out which one performs best at the critical task of comparing two bids.
Steps:	1. Dave starts by writing one or more benchmark application targeting the comparison of two numbers. The comparison may be implemented in multiple ways. 2. To test the performance of a number of protocol suites, Dave configures the test environment to use those protocol suites for the benchmark application(s). 3. Dave can now run the application(s) on his DAGGER test environment while measuring performance for each protocol suite. 4. Dave can now compare the performance data in order to pick the best protocol suite for comparing two numbers.

Table 2.7: Use Case 6. Benchmarking Secure Computation Technologies.

Use Case 7:	Change protocol suite to a different set of properties.
Actors:	Application Developer Erin
Brief:	Erin has developed a secure application using a DAGGER system, and a given protocol suite. She now wants to switch the current protocol suite for another one that offers different properties (e.g. better security, different number of parties). She needs to do this without significant development effort.
Steps:	1. If a protocol suite providing the desired properties is integrated into the DAGGER system and is compatible with the application, Erin needs to take no further action. 2. If no such protocol suite is integrated in the DAGGER system, Erin must integrate one (see the use case “Integrate a new Protocol Suite”) and/or modify the SCS to fit the chosen protocol suite. 3. Erin configures the application to use the new protocol suite and tests the application.

Table 2.8: Use Case 7. Change protocol suite to a different set of properties.

Use Case 8:	Use multiple protocol suites in the same SCS
Actors:	Application Developer Simon
Brief:	Simon intents to use two different protocol suites in the same SCS as that can give him a performance advantage.
Steps:	1. Simon denotes in the SCS which protocol suites the VM should use. This includes when to change from one protocol suite to another. If the high-level language does not support such a conversion instruction, Simon would have to implement this himself. 2. The conversion between representations of the different protocol suites needs to be handled internally by the SCE. In case the conversion cannot take place, a runtime error should occur.

Table 2.9: Use Case 8. Use multiple protocol suites in the same SCS

Use Case 9:	From a Boolean to an arithmetic model of computation.
Actors:	Application Developer Michelle
Brief:	Michelle has implemented a secure application configured to use a protocol suite which uses a Boolean model of computation. She has now learned that using a protocol suite with an arithmetic model may make her application perform better. Michelle wants to reconfigure her application to use the arithmetic protocol suite to see if this conjecture holds true.
Steps:	1. Michelle reads the documentation of possible arithmetic protocol suites to find one that supports conversion between boolean and arithmetic values. 2. If such a protocol suite does not exist, she must implement the boolean to arithmetic conversion herself or rewrite her application to use only instructions that work for arithmetic values. 3. Then, she can configure the application to use the newly found arithmetic protocol suite and test her application.

Table 2.10: Use Case 9. From a Boolean to an arithmetic model of computation.

2.4 Functional Requirements

2.4.1 Secure Application

Requirement F1: The Secure Application shall be capable of expressing general purpose computation algorithms.

Requirement F2: The Secure Application shall be capable utilizing generic secure operations in algorithms in order to securely process sensitive data.

Requirement F3: The Secure Application shall be capable of utilizing special purpose secure operations that are specific to a particular Secure Computation Technology.

Requirement F4: The Secure Application shall be capable of operating with multiple Secure Computation Technologies simultaneously.

2.4.2 Secure Computation Technology

Requirement F5: The Secure Computation Technology shall be implementable using this architecture.

Requirement F6: The implementations of Secure Computation Technology shall be compatible with a DAGGER Platform supporting this architecture.

Requirement F7: The Secure Computation Technology shall expose a number of invokable operations to a DAGGER Platform.

Requirement F8: The Secure Computation Technology shall be capable of performing continuous recurring tasks (such as preprocessing) in the background while being invoked to execute regular on-demand operations by a DAGGER Platform.

Requirement F9: The Secure Computation Technology shall be capable of storing and accessing intermediate computation results for its internal purposes.

Requirement F10: The Secure Computation Technology shall be capable of using the data representations required for its operations.

Requirement F11: The Secure Computation Technology shall be capable of using network resources required for its operations for secure data transfer between itself and a number of adjacent network nodes.

Requirement F12: The Secure Computation Technology implementations shall be capable of using various specialized secure hardware resources.

Requirement F13: The Secure Computation Technology shall be capable of using data store resources.

2.4.3 DAGGER Platform

Requirement F14: The DAGGER Platform shall be capable of loading Secure Computation Technologies according to a configuration.

Requirement F15: The DAGGER Platform shall enable the loaded Secure Computation Technologies to perform continuous recurring tasks in the background while executing regular on-demand operations.

Requirement F16: The DAGGER Platform shall provide the Secure Computation Technologies with access to secure network resources.

Requirement F17: The DAGGER Platform shall provide the Secure Computation Technologies with access to specialized secure hardware resources.

Requirement F18: The DAGGER Platform shall provide the Secure Computation Technologies with access to data storage resources.

Requirement F19: The DAGGER Platform shall be capable of executing the algorithm instructions as specified by a Secure Application.

Requirement F20: The DAGGER Platform shall be capable of executing the operations of Secure Computation Technologies as specified by a Secure Application.

Requirement F21: The DAGGER Platform shall be capable of providing input data to the operations of Secure Computation Technologies as specified by Secure Application.

Requirement F22: The DAGGER Platform shall be capable of reading output data from the operations of Secure Computation Technologies as specified by Secure Application.

Requirement F23: The DAGGER Platform shall be capable of applying different evaluation strategies when executing the instructions of a Secure Application.

2.5 Quality Attributes

In the following we describe the quality attributes that will be targeted by the architecture design and that a framework implementing this architecture will feature. Quality attributes are essentially the criteria characterizing how well the system must accomplish its functions.

Portability One of our goals (as inferred from Requirement B4, Goal 7, Goal 11 and Goal 12) is that the implementations of Secure Computation protocols must be portable across various DAGGER secure computation systems, so that the protocol developers can cover a larger market. This should be possible to do without significant development effort. We can achieve this by introducing a standardized interface that every SCE would conform to, and that future protocol suites would adopt in order to be easily plugged into different frameworks. This also implicates the need for a standardized interface for abstract resources, as the protocol suite cannot assume that certain implementations of resources (e.g. persistent storage) are available.

Modularity The architecture design must be structured as a set of modules and components, each having its own responsibility. This would allow to organize the development of independent parts efficiently in teams as well as make the overall system easier to understand and maintain. This would definitely contribute to fulfilling the identified business requirements.

To support portability, the framework should allow secure computation technologies to be implemented as a module independent of the concrete secure computation systems using the protocol. I.e., it should be possible to implement new technologies without knowledge of the DAGGER systems using the implementation. This should also make the work easier for protocol developers as they do not have to deal with the inner workings of any concrete DAGGER system. Similarly, the DAGGER systems using the implementations of secure computation technologies should be as independent of the implementation as possible.

Due to the heterogeneousness of the various existing secure computation technologies it may not be possible (or desirable) to completely decouple these two components. Therefore, it may be necessary to do some work to integrate a new secure computation technology into a given DAGGER system. However, the integration should require implementing an isolated module or plugin, and not modifying the DAGGER system.

Flexibility The framework should support the implementation of a variety of SMC protocols with potentially very dissimilar features and structure. For example, some protocols may work with a Boolean model of computation while others work on an arithmetic model, some may have high round complexity while others are constant round, and some protocol suites may have special features that makes certain computations perform particularly well. All these features may call for different design decisions when implementing the protocol. The framework, therefore, needs to be flexible enough to allow developers to implement SMC protocols in the way that best suits the concrete protocol.

Furthermore, since new and improved SMC protocols are frequently being discovered, the flexibility should extend to also allow for types of protocols that may not be known at this point in time. However, guidelines as to how to use the architecture, and providing standard solutions to common problems as well as healthy code patterns are highly advised in order to support code reuse.

Another goal of the flexibility quality is to be able to switch the underlying protocol suite in an application. This should be possible to do at runtime without having to recompile the code.

A third goal is to be able to, at runtime without recompiling, run a different application by providing the SCE with a new SCS.

Performance As performance is one of the largest challenges for SMC it is important that the architecture adds as little performance overhead as possible to the underlying SMC protocols.

As mentioned above, SMC protocols may have special features that can allow high performance optimizations for certain computations or for special protocols. The framework should make it possible for the application developer to take full advantage of these features to increase performance of a given application.

We note that this quality may conflict with the modularity mentioned above, and some appropriate tradeoff must be made. A good architecture would help achieving Goal 6 and Goal 10.

Usability From the point of view of the application developer, working with different underlying protocols should be as seamless as possible. The developer may want to know which underlying protocol he is targeting (e.g., in order to take advantage of special features) but he should be required to gain as little awareness of the concrete protocol as possible. In particular, this should be done in a way so that changing the underlying protocol does require a significant change to the application.

Security While the Secure Computation Technology provides various features allowing to process data securely, it usually does so by building on certain security assumptions. The architecture implementations should, therefore, be capable of adhering to these assumptions and providing the technology with necessary features, such as:

- Secure and/or authenticated communication channels.
- Uniformly distributed randomness from a secure source.
- Secure storage where the protocol suite can store persistent data securely.
- Secure Hardware. If the SCE supports protocol suites where specialized secure hardware is required, there must be an interface towards this component.

Chapter 3

Architecture

This chapter describes the Secure Computation implementation and integration architecture from a number of different viewpoints that have different scopes and angles in order to give the best overall understanding of the design. The architecture is presented at two conceptual level views (Logical and Process views) showing the structure and behavior of the system and two physical level views (Development and Deployment views) showing how the logical structure and processes map into physical components and environments.

3.1 Logical View

3.1.1 Overview

The Logical view describes the system functionality in terms of significant classes, interfaces and relationships between them. This essentially allows us to present the logical static structure of the system, that satisfies the functional requirements.

According to the SPEAR architecture presented in deliverable D21.2 [2], the secure computation services and applications rely on Secure Computation Technologies to protect the data while storing and processing it. The DAGGER Platform-as-a-Service (PaaS) layer of SPEAR enables the use of secure computation capabilities in a programmable fashion. The application business logic algorithm is specified in a Secure Computation Specification (SCS) which is then evaluated by a Secure Computation Engine (SCE) that applies the functionalities (*Protocols*) of various Secure Computation Technology implementations (*Protocol Suites*) to enforce the security of data according to the specification.

We are going to take this process as a basis and give a more in-depth view on the architecture regarding how exactly the Secure Computation Technologies are supposed to be implemented and integrated into the Secure Computation Engine of DAGGER so they can be easily used in an on-demand fashion according to some prescribed specification. The high-level logical view diagram of this architecture is depicted in Figure 3.1.

The high-level view suggests a logical separation of concerns between the three main classes of entities, and defines the points of contact where the dependencies between these entities can and have to be minimized for increased portability, modularity and flexibility of the final system. The Protocol Developers desire that their Protocol Suite implementations are independent from the rest of the system to maximize the portability between such systems and to cover a larger market. The DAGGER Platform Developers would like to have the flexibility of switching between different Protocol Suites in order to provide the Application Developers access to wider range of secure computation technologies. In turn, the Application Developers would benefit

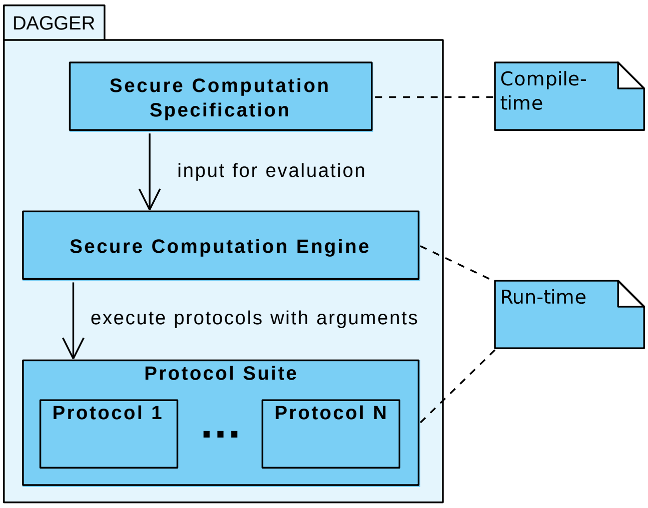

Figure 3.1: A high-level logical view of the architecture.

from being able to easily use the available technology implementations in their applications in a generic way with minimal effort and dependency on technology implementations. Breaking these dependencies would not only remove the technological dependencies, but also simplify the development and maintenance efforts for each entity, as the developers will only need to deal with a very specific part of the system, splitting the responsibilities and functionalities.
This architecture defines the minimal interfaces between the mentioned classes of entities in order to achieve the desired qualities with respect to the integration of Protocol Suites. It also expands on the internal structure of these entities necessary to support all the functionality identified in Section 2.4. Figure 3.2 displays the full class diagram representing all the significant classes and relationships of the final architecture that we are going to describe in detail in the following sections.

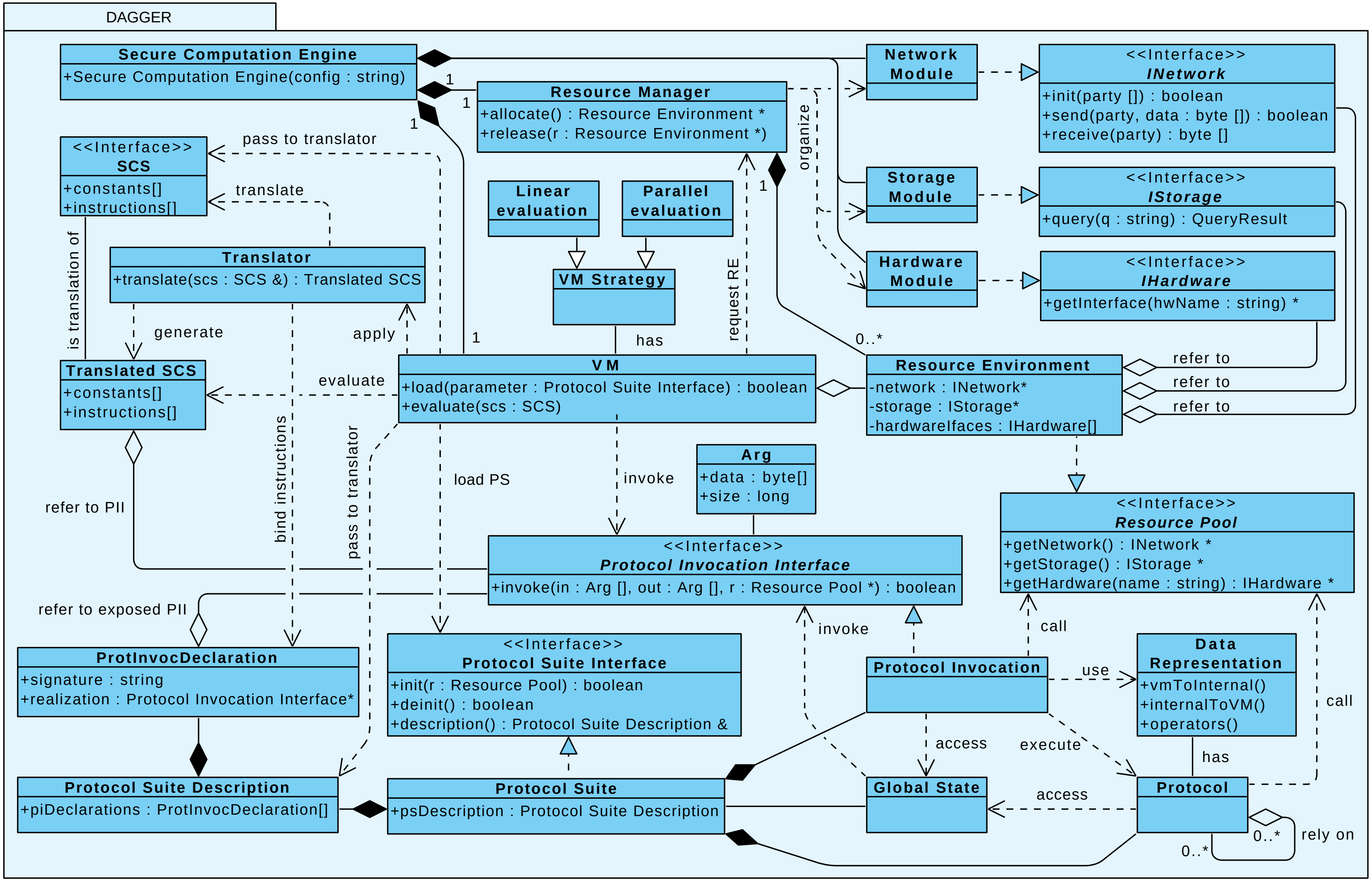

Figure 3.2: The logical view class diagram of an abstract secure computation framework.

3.1.2 Protocol Suite

A *Secure Computation Protocol Suite* (SCPS) is the component that contains the implementation of a particular secure computation technology and exposes it towards the upper layers of the DAGGER Platform in a generic way. Secure computation is a cryptographic method for evaluating a function on secret inputs while ensuring inputs' privacy. The function is evaluated by following a corresponding cryptographic protocol based on the chosen secure computation technique. The techniques often describe multiple protocols allowing to compute different basic and complex functions. A Protocol Suite is therefore defined by the complete set of *Secure Computation Protocols* implemented for the chosen secure computation technique and made available via the *Protocol Invocation Interface* as described in Section 3.1.4.

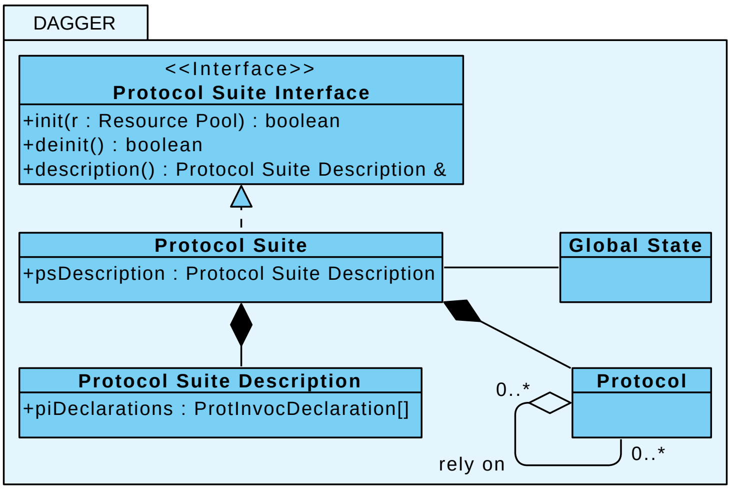

Figure 3.3: A Protocol Suite implementing the Protocol Suite Interface.

A Protocol Suite shall implement the *Protocol Suite Interface* (see in Figure 3.3) that is intended to be used by the DAGGER system to load the Protocol Suite and inspect the Protocols it exposes. The interface has functions for initializing and deinitializing the Protocol Suite as well as getting the *Protocol Suite Description* object owned by the Protocol Suite. The initialization is necessary so the Protocol Suite can create its internal data structures, initialize its global state, perform any one-time tasks and, optionally, start continuous recurring tasks such as precomputation protocols to run in the background. In order to run the tasks, the Protocol Suite can expect to be given access to a pool of certain standard resources (see Section 3.1.11) such as network communication and storage, that are implemented by the DAGGER platform and need not be implemented by the Protocol Developer.

3.1.3 Protocol

A *Secure Computation Protocol* (SCP) manifests a single cryptographic protocol of a single secure computation technique within the Protocol Suite. A Protocol typically represents the secure evaluation of a small function fundamental to the secure computation technology, e.g. addition or multiplication in case of techniques based on arithmetic circuits. However, a Protocol can also represent a more complex function such as sorting, and may rely on the simpler protocols of the same technique. The more complex operations are particularly interesting when the secure computation technology implemented by the Protocol Suite supports special-purpose high performance protocols for the operations that can not be simplified or efficiently realized by composing the generic operations.

The SCPs of a Protocol Suite are meant to be called either by the Protocol Suite internally or, if exposed, by the upper layers in order to compute an algorithm using secure computation. The number of different SCPs a Protocol Suite exposes can vary. A Protocol may use any of the resources provided through the Resource Pool as well as any internal (w.r.t. the Protocol Suite) components such as the Global State to draw Preprocessed Data, or internal Data Representation to work with. In principle a protocol can belong to multiple Protocol Suites and can thus be reused.

3.1.4 Protocol Invocation

A *Protocol Invocation* is the entry point to the execution of Protocol implementations and represents some meaningful operation that can be used by the Protocol Suite internally or exposed to the DAGGER system. When called, the Protocol Invocation is responsible for instantiating and executing the correct protocol(s) to perform the operation it represents. It also passes along all the parameters that the protocol(s) would need to be executed. A single protocol execution can trigger the execution of multiple sub-protocols. Similarly to protocols, the Protocol Invocation can access the internal global state of a Protocol Suite.

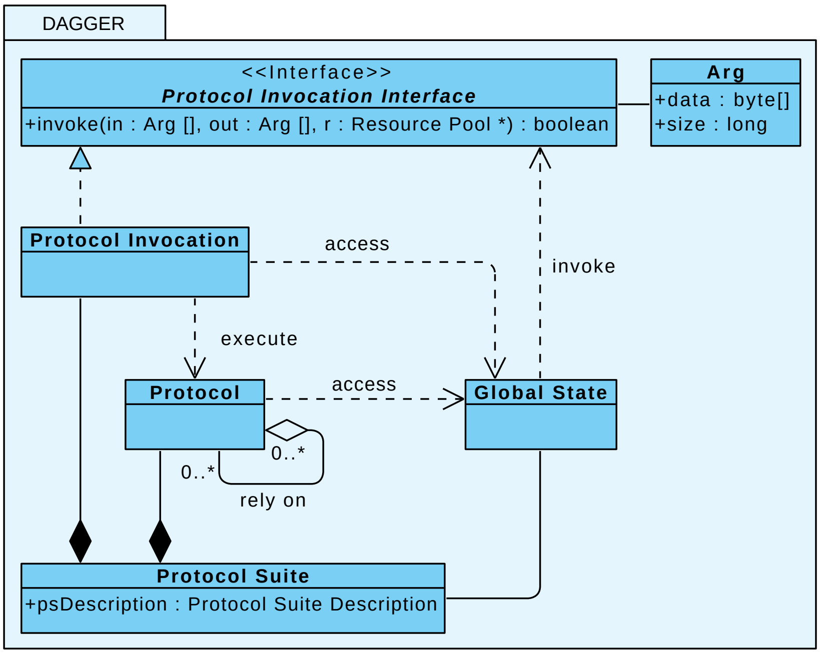

Figure 3.4: The class diagram for Protocol Invocations.

For the Protocol Suites to be both easily portable across DAGGER systems and interchangeable within each DAGGER system, it is desirable that the Protocol Invocations can be accessed using a common format. For this reason, as displayed in Figure 3.4, a Protocol Invocation implements the *Protocol Invocation Interface* (PII), that is generic in nature and is used for providing the protocols with input and output arguments, as well as a reference to preallocated resources that the protocol can reach (see Section 3.1.11 for details on the resources). The arguments are also passed through the interface using generic data types such as byte arrays. While the Protocol Suites can internally use very different and potentially non-standard data representations, the rest of the DAGGER should be agnostic w.r.t. the internal implementation of Protocol Suites. Therefore, a Protocol Invocation must ensure that the input is converted to the internal data representation that the protocols can understand and the outputs are converted back to generic

data type before returning. Once called, the protocol should execute itself, drawing upon the provided arguments and resources as needed. If the interface provides a call-back, the Protocol Invocation should notify this call-back once it is finished with the execution. As the call-back may be optional, we leave it to the actual implementation to decide whether to include it in the interface and what it should be.
The Protocol Invocation Interface should accommodate for different Protocol Suites supporting different sets of protocols, and, possibly, unique special features. So while we define the standard PII to make integration simpler, it may be desirable to allow Protocol Suites to only partially implement it, possibly making up for the lacking functionality in the translation modules (see Section 3.1.10).

3.1.5 Protocol Suite Description

A *Protocol Suite Description* is an array container within a Protocol Suite, that declares the Protocol Invocations exposed by the Protocol Suite. For each Protocol Invocation it contains a *Protocol Invocation Declaration* with a signature string, that uniquely identifies the Protocol Invocation, and a reference to its realization function (can be, e.g., a function pointer or a functor), that implements the Protocol Invocation Interface. The class diagram with respective relationships can be found in Figure 3.5

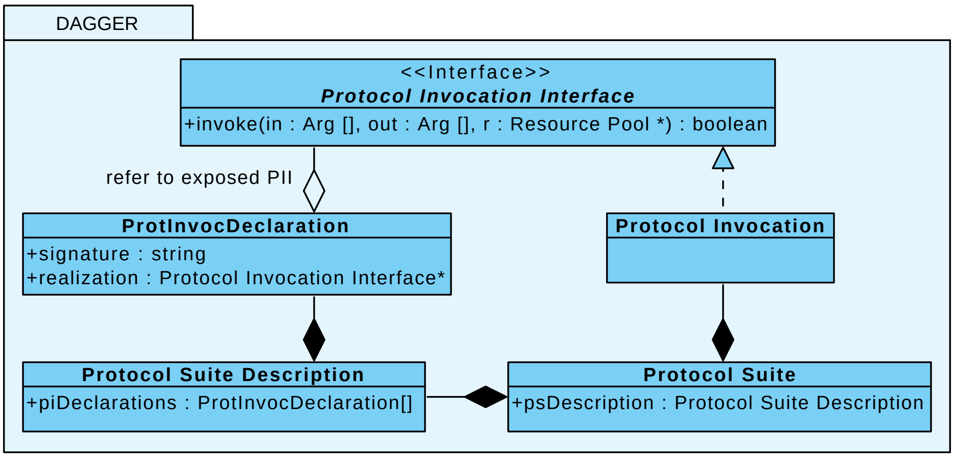

Figure 3.5: The exposure of protocols via Protocol Suite Description.

A DAGGER system can obtain the description from the Protocol Suite by the means of *Protocol Suite Interface* described in Section 3.1.2 and inspect its contents to identify the required operations by their signatures and invoke them by their reference to PII realization.

3.1.6 Data Representation

Secure computation techniques are built on top of various algebraic structures that define a set of possible values and a number of possible operations on it satisfying certain algebraic properties. The techniques use these structures to represent the values of the computation using some internal representation. For example, some Multi Party Computation schemes use finite fields or rings. Hence, the Protocol Suites need to implement their *Data Representation* to work with internally (see Figure 3.6]).

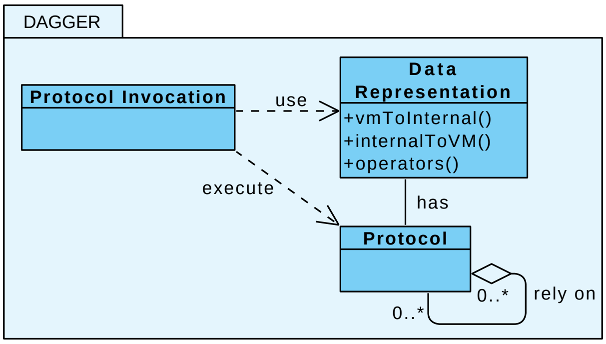

Figure 3.6: The Data Representation and its relationships.

The main responsibilities of data representations are:

- Facilitate the internal representation of data types that Protocols operate with.
- Provide basic operations on these data types.
- Convert the values to and from the generic data representation supported by the Protocol Invocation Interface.

3.1.7 Global State

A *Global State* is generally used for storing necessary data that is globally accessible to various parts within a Protocol Suite, such as the Protocols and Protocol Invocations. A Protocol Suite may also run any recurring operations in the background within the global state. The global state and any tasks related with it are initialized and deinitialized via the Protocol Suite Interface and have access to their own pool of resources.
Although completely internal to the Protocol Suite and not always required to be implemented, this component is essential to point out in the architecture, as it can serve multiple useful purposes. First, some protocols rely on some kind of preprocessed data during the online phase of the computation to save time in the actual execution, and can therefore use the global state to draw additional data while it is continuously precomputed in the background. An example of preprocessed data would be a Beaver triple which can be computed in advance to the actual execution of the protocol (e.g. while idling). Alternatively, the preprocessed data can also be fetched from a Storage resource instead. It all depends on the needs. Second, the global state can be used to cache any global variables or computation data for improved performance. Third, a secure computation technique may require to cache some data for improved security purposes. In any case it is completely up to the Protocol Developer how the global state of a Protocol Suite is implemented and used.

3.1.8 Secure Computation Specification

A *Secure Computation Specification* (SCS) is a low-level representation of a secure application that can be understood and executed by a Secure Computation Engine of the DAGGER platform (described in Section 3.1.9). An SCS basically specifies the instructions of the business logic algorithms, i.e. what exactly must be securely computed in a particular application and which secure operations should be used in the computation, without going into details how these operations actually work. As such, it is typically a product of some high-level language compiler. An Application Developer would assume the existence of a set of callable instructions that can take inputs and return outputs and that the developer can use to compose algorithms. How the secure instructions are implemented is the internal business of Protocol Suites that provide these operations and not the function of the SCS. It is up to a Secure Computation Engine to act as a middleware and evaluate an SCS by executing the correct operation implementations according to the specified instructions.

The format of an SCS is dependent on the design of the particular DAGGER system that will be used for the execution. A portable and flexible approach would be to use a *bytecode* format also known as *portable code*, which is a form of instruction set designed for efficient execution by a software Virtual Machine (interpreter or just-in-time). Bytecode typically encodes algorithms as instruction opcodes, constants and references and can be dynamically loaded during execution. In this case an SCS would be a file (or a set of files) of bytecode instructions that an SCE can load and execute. Compared to direct compilation of algorithms into machine code, a two-staged approach involving translation into bytecode and execution by a VM offers several advantages. Generating bytecode is much easier than machine code as no machine-dependent behavior, including the specifics of secure operations, must be considered. This usually results in a smaller and more portable code. Also, while executing the bytecode, a VM can perform run-time optimizations and checks that are difficult to do in native code. Although the execution speed may be a slight disadvantage, it can be remedied by efficient VM implementation, and is by far not the main bottleneck when it comes to Secure Computation. One could either design his own bytecode format and a respective VM, or use an existing one such as Java bytecode and JVM. Alternatively, it could be possible to have an SCS represented as native object code, but that would likely result in higher dependency on the engine and the actual operation implementations provided by the Protocol Suites, as well as lack in other advantages of the bytecode format. A dynamically loadable object code may also introduce an additional attack vector allowing one to unwillingly or on purpose directly access and tamper with the memory space of the Secure Computation Engine running the object code. This could be the source of security or determinism issues. Because of all that, the bytecode approach is preferable.

There can be different approaches when it comes to specifying which secure operations should be executed. An SCS may be entirely generic, i.e. having instructions independent of any particular Protocol Suite configured and used during execution. While this would increase the portability of a secure application, namely allowing to switch the underlying Protocol Suites without changing the application, it could reduce the flexibility of utilizing technology-specific operations or using multiple different Protocol Suites simultaneously in the same application, which could prove useful in complex applications with specific functionality, security and performance requirements. Alternatively, an SCS can be more Protocol Suite specific, allowing to specify exactly which named operations in the targeted Protocol Suites should be used. The latter would infer that the Protocol Suites whose required operations are exposed would be available to the SCE during SCS execution. One would also have to update the application when changing the underlying Protocol Suites, but this should only require the minimal amount of modifications (e.g. change the operation or Protocol Suite names) and is, therefore, acceptable.

3.1.9 Secure Computation Engine

A *Secure Computation Engine* (SCE) is the main component of the DAGGER platform driving the applications based on secure computation. It provides the software infrastructure necessary to organize and facilitate the successful execution of secure applications utilizing Secure Computation Technologies.

An SCE contains a Virtual Machine (see Section 3.1.12) for loading and evaluating the application business logic in Secure Computation Specifications (see Section 3.1.8). Based on configuration the VM can be extended with a number of Protocol Suites providing the choice of secure operations to be called during evaluation.
Additionally, an SCE provides and manages a number of standard resources required by the Protocol Suites. It contains modules for networking, storage and specialized hardware and makes these available to the secure computation technology implementations in a synchronized manner. Please refer to Section 3.1.11 for more details on resource management.

3.1.10 Translator

The *Translator* is a kind of preprocessor responsible for translating the instructions of a Secure Computation Specification to real references to Protocol Invocation Interface realization functions of corresponding secure operations exposed by the targeted Protocol Suites. The result of this transformation is the *Translated SCS*, an intermediate representation of an SCS that a Virtual Machine (see Section 3.1.12) can directly and efficiently evaluate by calling the bound references to operation implementations, and not having to do the translation during SCS evaluation.
There are multiple ways to approach the translation. It can be performed either at compile time (when compiling the SCS), or at run time (when evaluating the SCS to run the application), or partially at both stages. The exact choice depends on the specifics of the DAGGER platform and its SCS format. Let us take a look at the options.

A) Compile time When developing and compiling an SCS in native object code format, one would rely on special compilation (translation) modules for the targeted Protocol Suites telling exactly which functions need to be called for the supported secure operations. The compiler would generate the correct Protocol Suite specific object code ready for execution. An SCE would dynamically load the application object code expecting that all the functions it depends on are available to the SCE and can be linked. In this case the translation is done completely by the compiler, and besides the standard dynamic linking procedure there is really no other translation required during run time, as the processor will be executing the object code directly as it is. While this option may look simple, it is also the least portable and flexible. For example, in order to change the underlying Protocol Suite, one would have to modify and recompile the whole application. Also, the dynamically linked software typically shares common memory with with the host application, introducing an additional attack vector that makes this option less secure.

B) Run time The SCS application is represented in bytecode format and is completely generic w.r.t the Protocol Suites. This means that each used instruction is generic and would be similar for each Protocol Suite. The exact Protocol Suite to be used is configured when the application is deployed to an SCE. During run time the translator inside an SCE translates the generic SCS instructions to references to realization functions of corresponding secure operations in the configured Protocol Suite. The translator would either assume some standard operation names that the Protocol Suite must support, or rely on Protocol Suite specific modules containing additional translation rules specifying how the generic instructions can be executed using the configured Protocol Suite. As a result a single instruction may be expanded to a set of instructions. The translation modules can be written by either the Protocol Developer or the DAGGER Platform Developer, who have an overview of both the SCS of the targeted DAGGER platform and the integrated

Protocol Suites. This option simplifies the compilation of the SCS, but the run time translation becomes more complex as single SCS instructions may now involve several Protocol Invocations in the Protocol Suites.

C) Compile time & Run time This option is a hybrid combination of the previous two. When developing and compiling an SCS application in the bytecode format, an Application Developer relies on the Protocol Suite specific compiler modules. However, instead of hard coding the exact functions to be called (as this was the case with the object code), the bytecode would specify the string identifiers of required operations. The Protocol Suite specific compiler modules would define the exact names of the supported operations. During run time, the translator would match the instruction string identifiers against the signatures of secure operations exposed by the configured Protocol Suites and make the links in the Translated SCS representation. The VM would then execute the correct realization functions directly by the references in the Translated SCS. This option also allows the use of multiple Protocol Suites in the same application simultaneously.

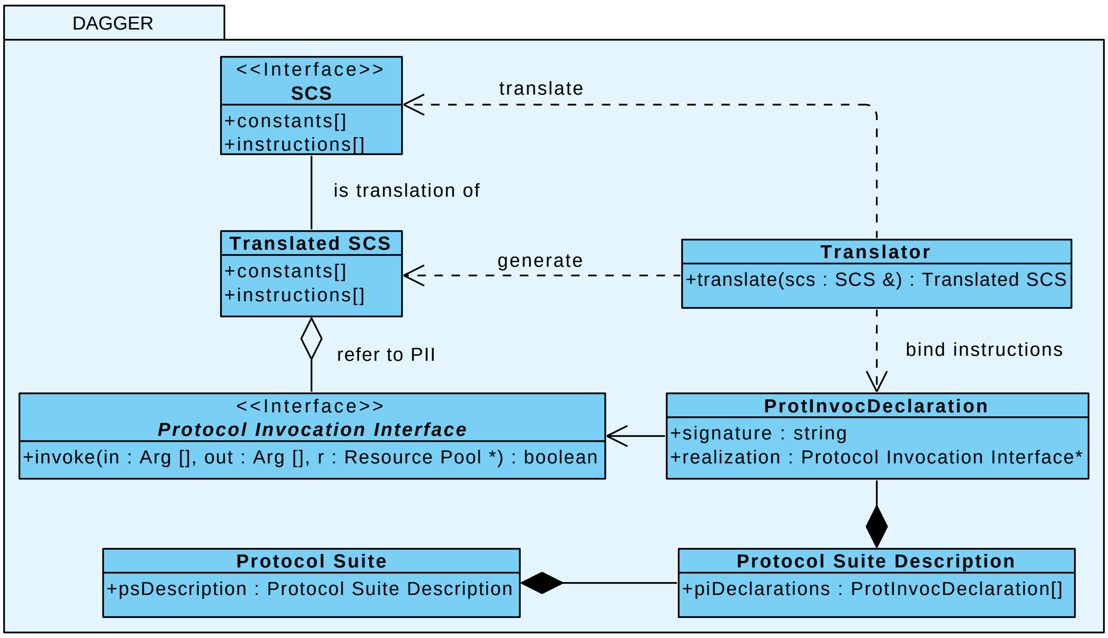

Figure 3.7: The class diagram for translation of SCS.

In this architecture we mainly target the options B and C, as they allow us to better achieve the required quality attributes. Figure 3.7 displays the general class diagram for the translation functionality. The Protocol Suites expose a number of secure operations that can be found by their signature and executed using the PII interface. The run time translator takes an SCS and having access to all the Protocol Invocation Declarations of supported secure operations it can perform the translation of bytecode instructions and generate the Translated SCS representation. In case the SCS compiled correctly, but the operation required by the specification cannot be found in the available Protocol Suites, then the translation process would fail. A successfully translated SCS would indicate that all the required operations are available to the SCE and can be executed.

Regardless of where the translation is done, it is also potentially a point of optimization. Since the translator is aware of Protocol Suites, it can make sure that SCS instructions are translated to the most efficient protocol calls. It can also possibly order protocol calls in such a way that it benefits performance.

3.1.11 Resource Pool

When implementing a Secure Computation Technology in a Protocol Suite, a developer would typically assume the existence of some standard resources (e.g. network) that it can rely on. These resources are generic in nature and can be seen as the lowest common denominator of secure computation technologies. As such, reimplementing these for every Protocol Suite would be an inefficient use of time and effort. Moreover, the resources may have to be managed and synchronized depending on the way the Protocols that rely on these are invoked.

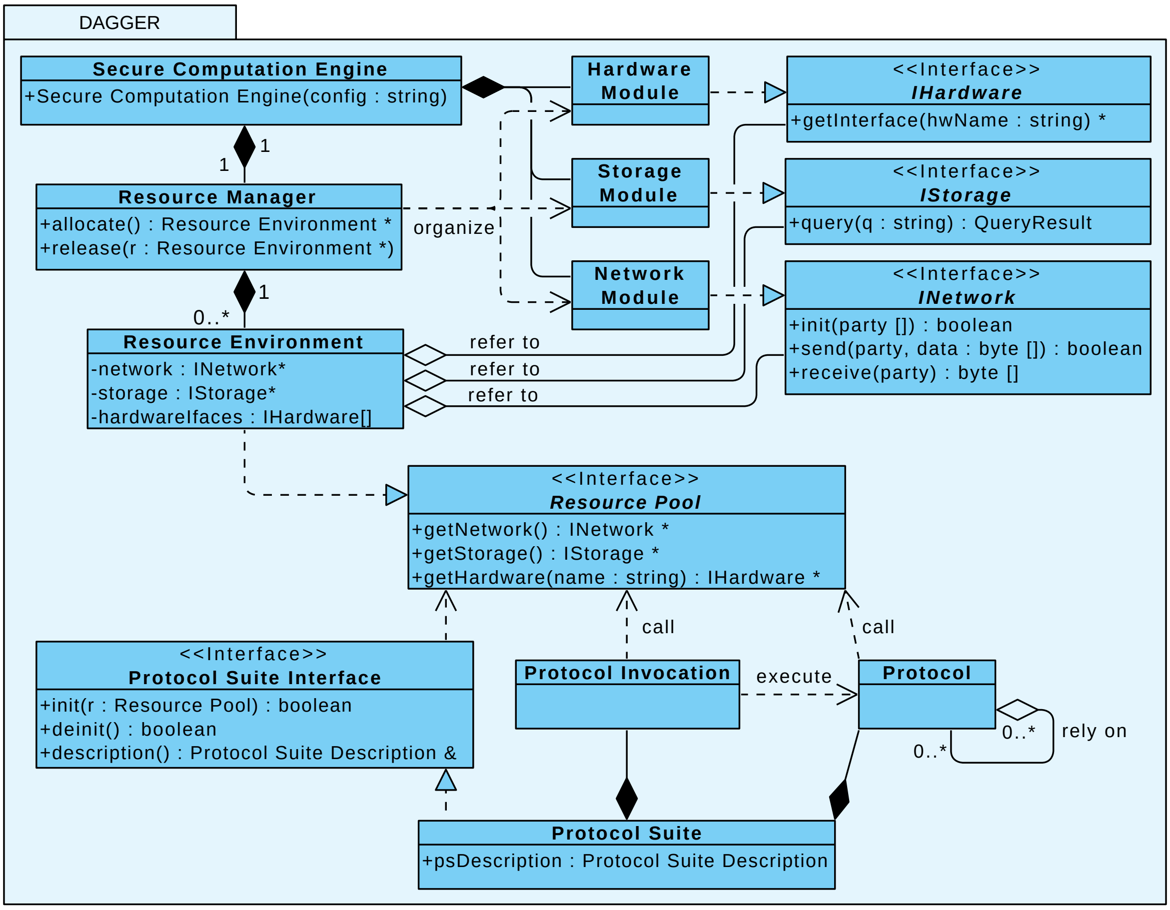

Figure 3.8: The class diagram for resource management.

In order to simplify the development of Protocol Suites and increase their portability, we move the responsibility for implementing the standard resources to the Secure Computation Engine of the DAGGER platform. As displayed in Figure 3.8, an SCE would supply and manage a set of generic resources, and make them accessible to the Protocol Suites in a synchronized manner via an abstract *Resource Pool* interface. This way a Protocol Suite becomes independent from the resource implementations and only focuses on using the required ones to complete its intended functionality.

The resources may be used by multiple resource consumers simultaneously, e.g. multiple threads executing different secure computations. Therefore, the resource consumers shall be capable of using the resources independently, without interfering with other resource consumers. Since the resources may have their internal state or require synchronization, then a Secure Computation Engine should allow allocating a separate *Resource Environment* for each resource consumer by the means of the Resource Manager. The Resource Environment would implement the *Resource Pool* interface and refer to all the resources initialized with a separate state and

synchronization mechanisms. This way each resource consumer would get its own instance of the Resource Environment and be able to access the resources without interfering with other consumers.
The Resource Pool is intended to be passed to the Protocol Suite during its initialization and to the Protocol Invocations during SCS evaluation, so that the implemented technology can function properly. The exact amount of resources that should be managed and exposed by the SCE is debatable, but the following minimum should be available.

Abstract Network (INetwork interface) This interface provides the protocols with access to the network resources and should allow communication with all the other parties in the computation. I.e., it should have simple functionality to send and receive data to and from the parties involved in the secure computation. The network resource should support both communication in plain-text and protected communication (e.g., via SSL) between parties. Slightly more complex communication primitives, such as message broadcasting, could also be supported. The interface should allow initializing the network resource when necessary, and configuring it with the required parties.

Abstract Storage (IStorage interface) The interface provides access to persistent storage where the Protocol Suite can put or fetch data needed during the execution of the protocols. For example, this could be some previously stored preprocessed data that is ready to be loaded at runtime. A generic approach would probably be to allow making SQL-like queries. Although, a simpler key/value interface could also be an option.

Abstract Hardware (IHardware interface) The Resource Pool should allow accessing specialized hardware resources. An SCE may be configured with a number of hardware modules, that allow accessing the hardware in a controlled manner. A Protocol Suite should be able to request a reference to some Abstract Hardware by its name, and then work with its structures and interfaces by assuming that a correct type of hardware is returned.

3.1.12 Virtual Machine

A *Virtual Machine* (VM) is the functionality of a Secure Computation Engine responsible for executing Secure Computation Specifications in a virtualized environment that abstracts away details of the underlying secure computation technologies and the related software and hardware infrastructure. It provides a generic instruction set for public operations, but can also be extended with additional secure operations exposed by the Protocol Suites. When executing an SCS, the Virtual Machine ensures that correct operation implementations are invoked and that the necessary resources and arguments are made accessible to the operations. The VM-related classes and relationships are depicted in Figure 3.9.
Virtual machines can be implemented in multiple ways. A simpler option would be to have a relatively thin VM, that offloads the execution of SCS instructions (e.g. object/machine code or bytecode) to the host runtime system (e.g. CPU + C runtime application or JVM) powering the SCE itself, and is therefore mainly responsible for dynamic linking and instantiation of SCS, the Protocol Suites and the required resources. A more elaborate VM would implement its own SCS evaluation mechanism, such as a custom bytecode interpreter, that would allow to even better separate the SCS applications from the rest of the infrastructure and have a greater control of the execution process. The essential advantage of bytecode interpreters is that the executed SCS applications are limited to the resources and abstractions provided by the virtual machine, and cannot break out of their virtual environment. This also results in

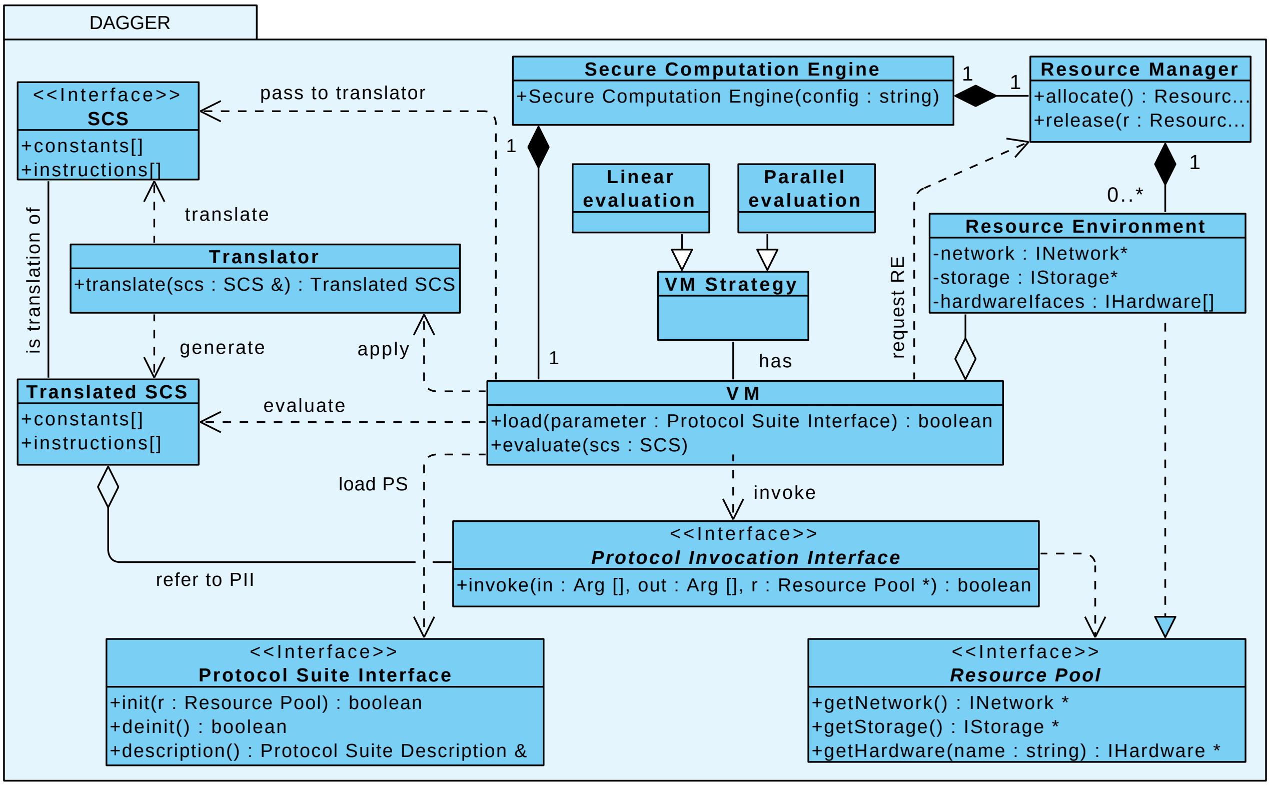

Figure 3.9: The class diagram for the Virtual Machine functionality.

improved portability and security of applications. In this architecture we, therefore, mainly target bytecode-based VM's instead of object code based ones.

One of VM's responsibilities is to load and inspect a number of Protocol Suites via the *Protocol Suite Interface* at the request of the SCE. This allows it to discover new operations that can later be executed via the *Protocol Invocation Interface* while evaluating an SCS. Each Protocol Suite is required to be initialized before the operations can be used, and deinitialized afterwards, so that the internal implementations can gracefully complete their life-cycle.

When working with the Protocol Suites, a Virtual Machine is required to provide them with the required resources (see Section 3.1.11). When necessary, a VM would request a Resource Manager to allocate a set of Resource Environments for it, and then somehow distribute these among the calls to the Protocol Suite Interface and Protocol Invocation Interface via the *Resource Pool* interface. When the resources are no longer required, the VM would notify a Resource Manager to free these resources.

In order to evaluate a bytecode SCS, the Virtual Machine would first need to ensure that all the operations required by the bytecode are supported and available for execution. The VM first uses the Translator (see Section 3.1.10) in conjunction with the descriptions of the loaded Protocol Suites to verify the existence of required operations and bind the opcodes to real references (addresses) of corresponding implementations provided by the Protocol Suites. If this process succeeds, the VM can begin evaluating the Translated SCS and executing the instructions by invoking the bound references to Protocol Invocation Interface implementations with correct arguments and resources. The instructions in the Translated SCS trigger sequences of actions, that produce effects according to the semantics of the instructions.

A VM can use various *strategies* when evaluating the SCS instructions of an application. The strategy (sometimes referred to as execution model) controls the order in which work takes place during execution. This order may be chosen ahead of time, or it can be dynamically

determined as the execution proceeds. The static choices are most often implemented inside a compiler, in which case the order of work is represented by the order according to which instructions are placed into the SCS. The dynamic choices would then be implemented inside the language's runtime system or a VM in our case. In a simple strategy a VM may execute instructions sequentially in the order it receives them. Alternatively, we can think of complex multi-threaded strategies where independent instructions are evaluated in parallel. Strategies may also schedule the execution of SCS instructions (e.g. reorder or mix) in order to optimize performance. An example of such an optimization strategy is a VM that batches independent communication heavy protocol calls in order to save on latency costs.
There is not necessarily one strategy that fits all applications. For example, an application in a high latency setup may require different strategies than than those used in low latency setups. The best strategy may also be dependent on the used PS, as each secure computation technology may differ in which operations perform well. For this reason, it is useful for the VM to be configurable so that a strategy can be chosen to suit the given application. While the choice of strategy could be left to the application developer, it could also be determined automatically by the VM by analyzing the SCS, the available Protocol Suites and possibly even the physical setup. For example the JVM is known to apply techniques such as Just-in-Time (JIT) compilation and adaptive optimization by dynamic recompilation of code hotspots (trace-based JIT).

3.2 Process View

The Process view describes the run-time behavior and interactions of the system classes and objects in processes and threads. This essentially gives us a dynamic perspective of the system. In the following subsections we cover the main processes involved in the architecture.

3.2.1 Loading a Protocol Suite

Protocol Suites are plugged into an SCE via the VM component. Typically, an SCE would be configured with a number of Protocol Suite libraries, and this way know which files exactly need to be loaded. For each PS library, the SCE would invoke a load() function on the VM component, and the VM would then load the passed library in the following main steps.

1. Link the Protocol Suite library and access it via the Protocol Suite Interface.
2. Allocate a Resource Environment for the Protocol Suite via the Resource Manager.
3. Call the initialization function of the Protocol Suite and pass it the allocated Resource Environment via the Resource Pool interface, so that the Protocol Suite can
   (a) configure and access the resources available through the allocated Resource Environment for its operations
   (b) create and initialize the internal data structures required for its operation
   (c) initiate any required internal one-time or continuous tasks in separate threads
4. Inspect the exposed Protocol Suite operations by accessing the Protocol Suite Description.
5. Register all Protocol Invocations declared in Protocol Invocation Declarations.

A more detailed sequence diagram is presented in Figure 3.10, and the respective communication diagram is in Figure 3.11.

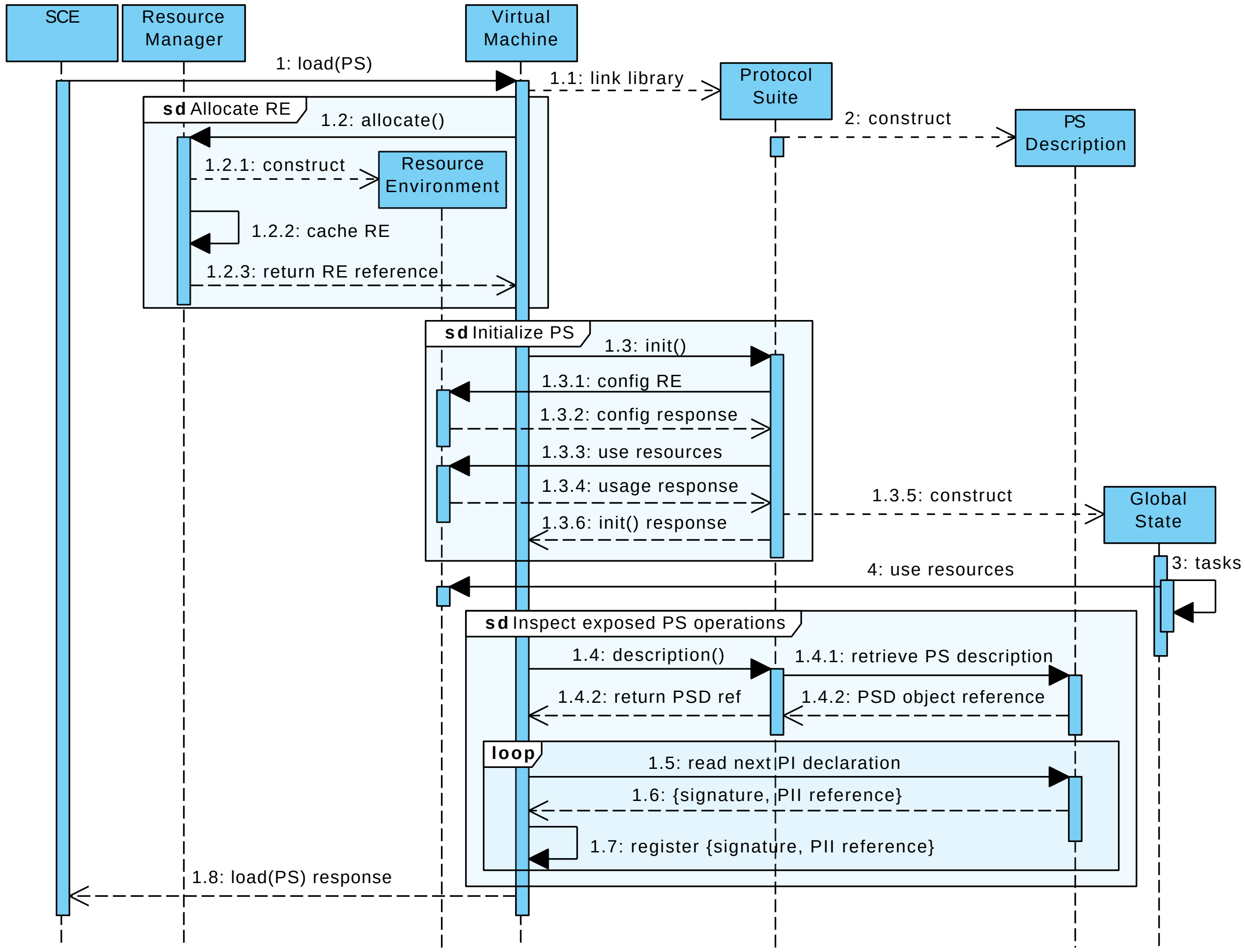

Figure 3.10: The sequence diagram for Loading a Protocol Suite.

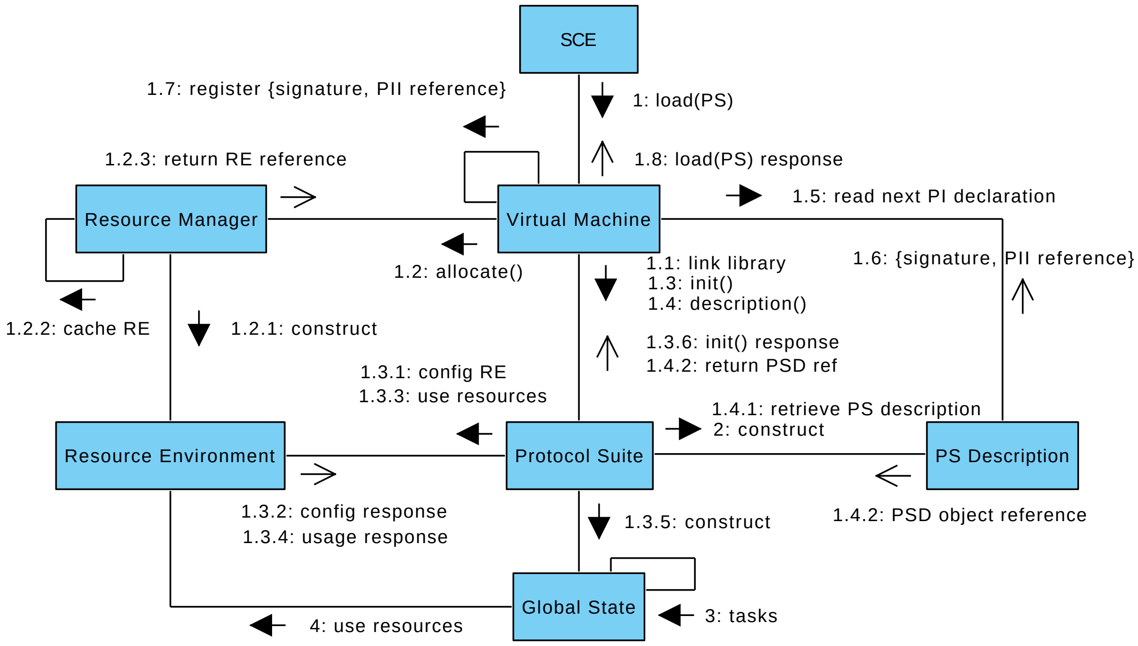

Figure 3.11: The communication diagram for Loading a Protocol Suite.

3.2.2 Translation of SCS

Translation of SCS is initiated by the VM and performed by the Translator. The Translator has access to the VM's indexes of supported native and secure operations. With this knowledge, the Translator's translate() function is invoked by the VM with the raw SCS as an argument. The Translated SCS structure, that the VM would understand, is then computed by performing the following main steps.

1. Take as input: SCS, Protocol Invocation declarations
2. Translate the SCS format to a Translated SCS format that the VM understands.
   (a) Bind each non-secure instruction in the SCS to the corresponding native VM operations. Indicate translation failure if a non-secure instruction is not supported by the VM.
   (b) Bind each secure instruction in the SCS to the Protocol Invocations according to their declarations. Indicate translation failure if no Protocol Invocation could be mapped to an SCS instruction.
3. Optimize the Translated SCS.
4. Output the Translated SCS.

A more detailed sequence diagram can be found in Figure 3.12, and the respective communication diagram is displayed in Figure 3.13.

3.2.3 Evaluation of SCS

Secure Computation Specifications are evaluated by the VM on request of the SCE. Before evaluation, a number of desired Protocol Suites are first loaded as described in Section 3.2.1. Then, to start the evaluation process, the SCE would call the evaluate() method on the VM and pass a raw SCS as an argument. The VM would begin evaluation by translating the SCS to a more understandable and efficient form, as described in Section 3.2.2. Assuming that the SCS could be successfully translated and all the required Protocol Invocations are available for execution, the VM can now begin the process of executing instructions by applying some evaluation strategy. Exactly how the evaluation itself is performed is implementation-specific and out of the scope of this deliverable, and we, therefore, do not cover this topic in depth. We will only briefly touch on the two most typical strategies.

Linear The simplest way to evaluate an SCS is linear execution of instructions in a single thread. This involves 3 steps: 1) Take next instruction 2) Pass arguments to it 3) Execute. This process continues until either an error occurs or there are no more instructions to execute. This strategy only requires a single Resource Pool to evaluate an SCS.

Parallel A potentially more scalable variant would be to execute instructions in parallel. This could be done by running multiple linear threads, that compute independent branches of the Translated SCS. These branches could either be specified by an SCS, or determined automatically by analyzing SCS code and grouping together independent sections. This option requires a separate instance of Resource Pool for each execution thread, so the concurrently executed instruction implementations do not interfere with each other and use resources in synchronized manner. This also applies to running background tasks while evaluating an SCS using either option.

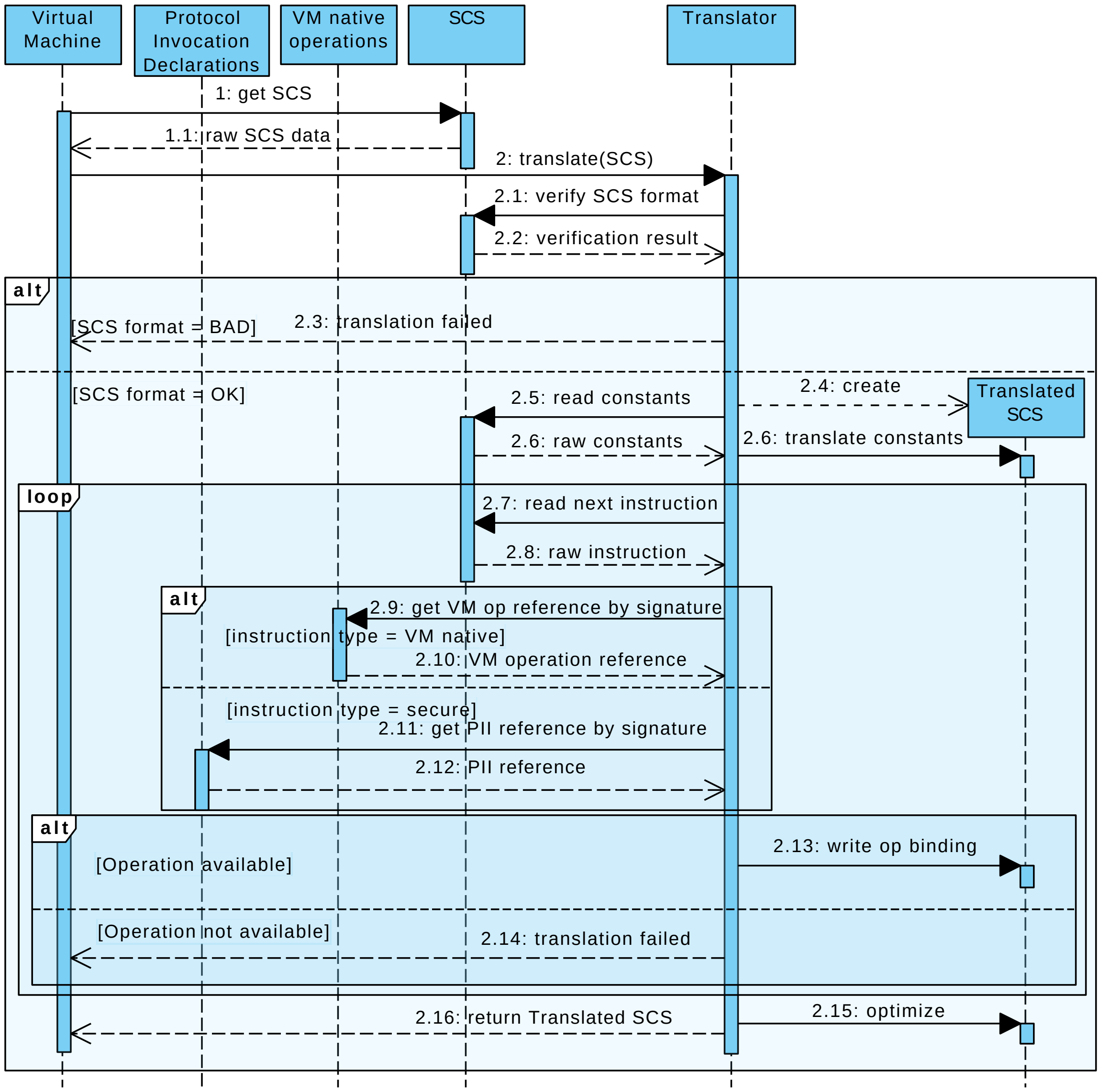

Figure 3.12: The sequence diagram for Translation of SCS.

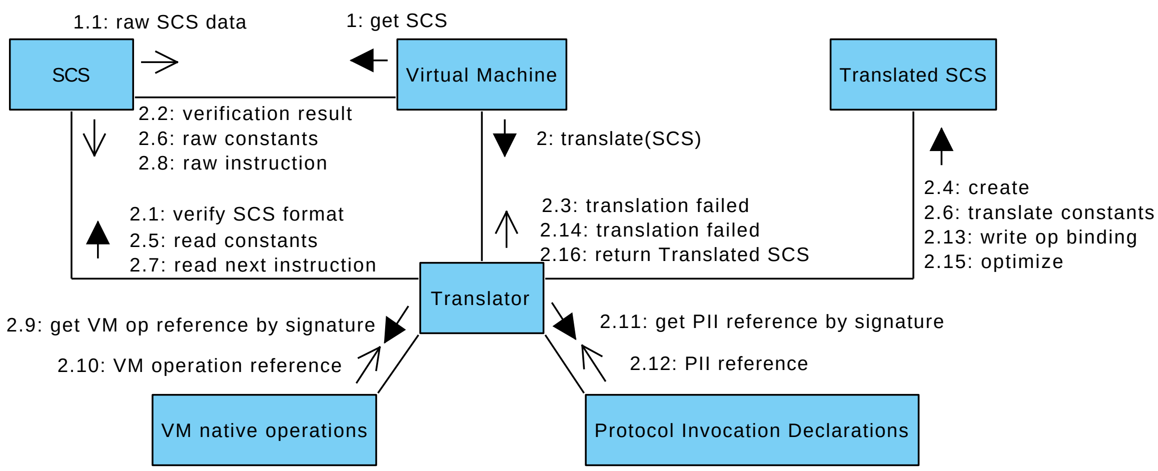

Figure 3.13: The communication diagram for Translation of SCS.

3.2.4 Invocation of a Protocol

While evaluating an SCS, some instructions may trigger the VM to invoke operations provided by the Protocol Suite. These calls are designed to execute the necessary protocol implementations inside the Protocol Suite libraries. In the following we detail the process of invoking a protocol via PII.

1. VM reaches an instruction ordering an invocation of a Protocol X with a set of arguments.
2. The respective Protocol Invocation function has already been bound to the instruction in the process of Translation of the SCS, meaning that the necessary functionality should be available to be executed. The signature of a Protocol Invocation is predefined and only needs to be invoked with correct arguments.
3. VM prepares the input and output data arguments to be passed to the Protocol according to instruction arguments.
4. VM prepares the reference to the Resource Pool that the Protocol Invocation should be called with. VM uses the Resource Pool of the active evaluation thread allocated for the Translated SCS.
5. VM now calls the Protocol Invocation with the prepared data and Resource Pool arguments.
6. The Protocol Invocation parses the input data arguments using the internal Data Representation from the format understandable to VM to the format understandable to the Protocols to be executed. It also prepares the output arguments possibly in a similar way.
7. The Protocol Invocation executes one or more internal protocols with references to the input and output data containers in correct data representation. The protocols draw on the provided resources as needed.
8. The Protocol Invocation stores the final output data to its output arguments.
9. The Protocol Invocation indicates success or some kind of failure if such occurred.
10. VM operates with the output arguments of the finished instruction according to the next instructions of the Translated SCS.

A more detailed sequence diagram is displayed in Figure 3.14, and the respective communication diagram is presented in Figure 3.15.

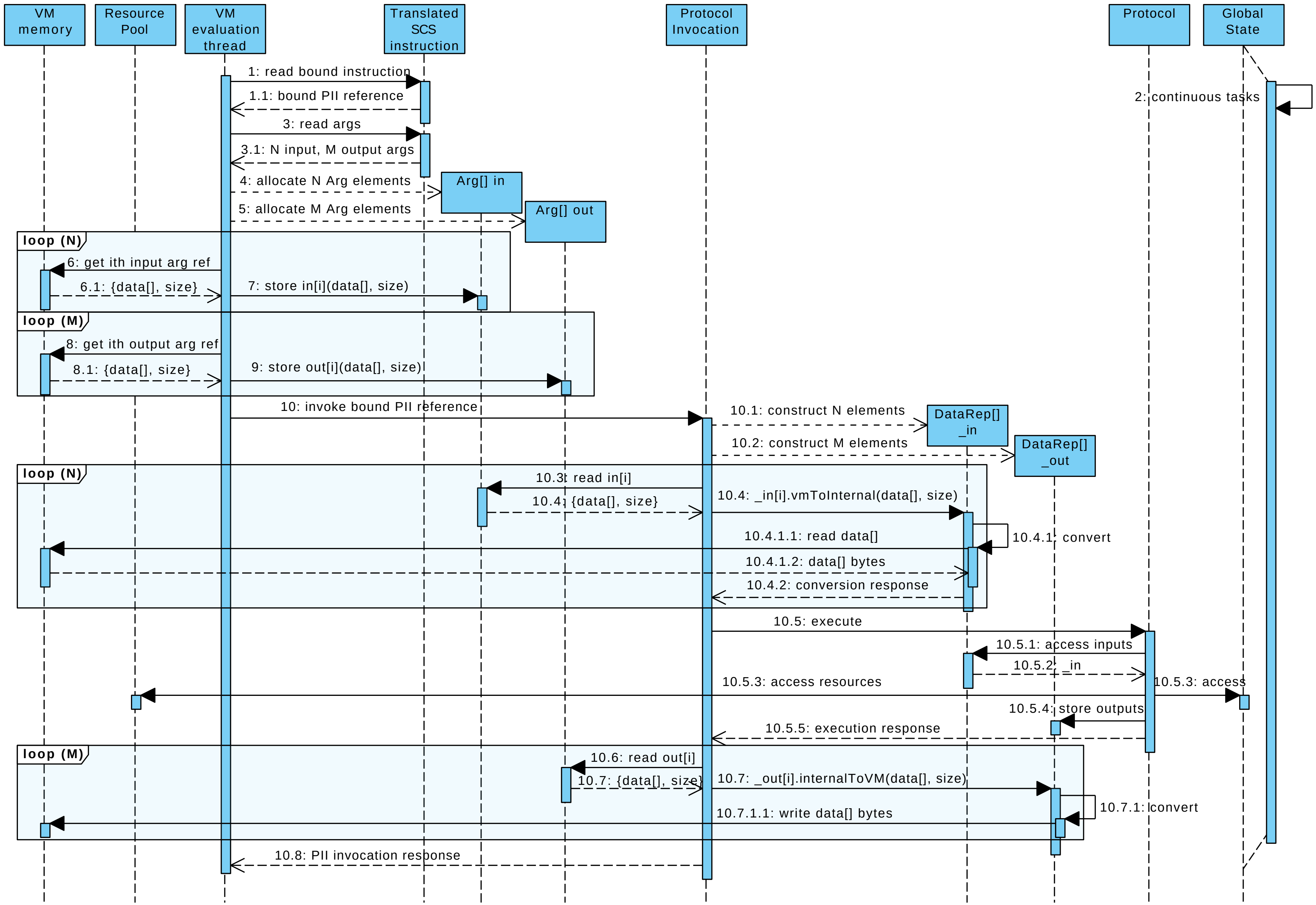

Figure 3.14: The sequence diagram for Invocation of a Protocol.

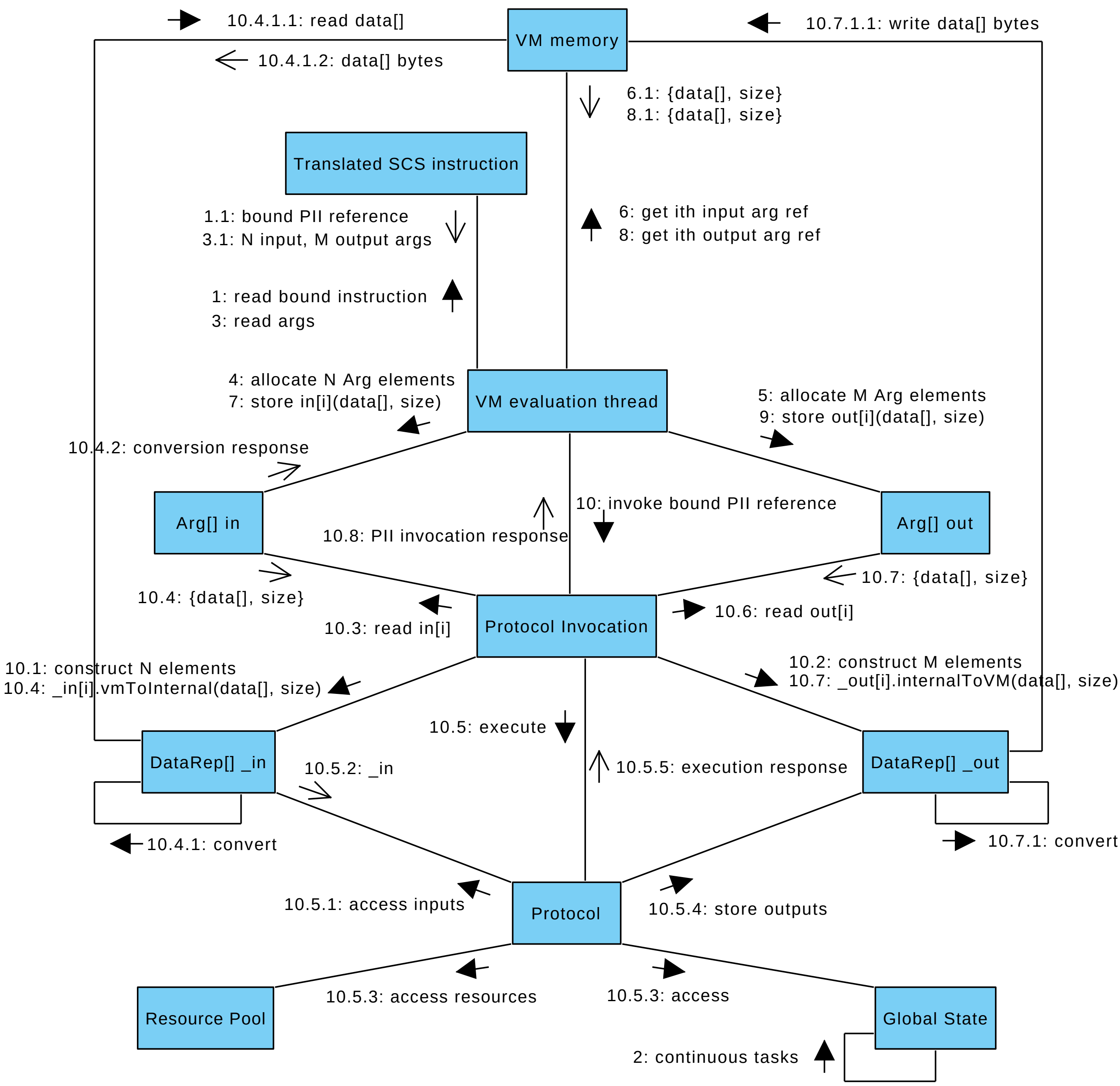

Figure 3.15: The communication diagram for Invocation of a Protocol.

3.3 Development View

3.3.1 Overview

The Development view describes how the logical structure of the system maps into physical development artifacts and is targeted towards the developers who implement the architecture. From the perspective of the developer, the software is partitioned in modules, that can be implemented and tested separately from the rest of the system. This view shows the different modules and gives an idea as to what is needed for the different pieces to fit together. The system is decomposed into layers, subsystems and architecturally significant components that implement the logical classes and interfaces. We begin by discussing the layers, the rules that

govern the inclusion of components to each layer, and the boundaries between layers. For each layer we give an overview of its contents. Finally, we show how the components are organized in the development environment in terms of packages and artifacts.
This architecture models a subset of the general SPEAR architecture presented in Deliverable D21.2 [2], and covers its layers related to the integration of secure computation into DAGGER. Having a good overview of the logical structure of the designed system, we can name the three distinct layers involved in the integration of secure computation, and define their responsibilities.

Secure Computation Application This layer deals with specification and compilation of business logic algorithms, that utilize secure computation technologies. This layer can include Secure Languages & Compilers as well as their direct artifacts, i.e. Secure Computation Specification.

Secure Computation Engine This layer involves components responsible for powering the execution of applications in the upper layer and making it possible to use the technologies in the bottom layer.

Secure Computation Technology This layer contains the components implementing the actual cryptographic Secure Computation Technologies, and is responsible for providing the upper layer with secure operations based on these technologies. The technologies are encapsulated in Protocol Suites.

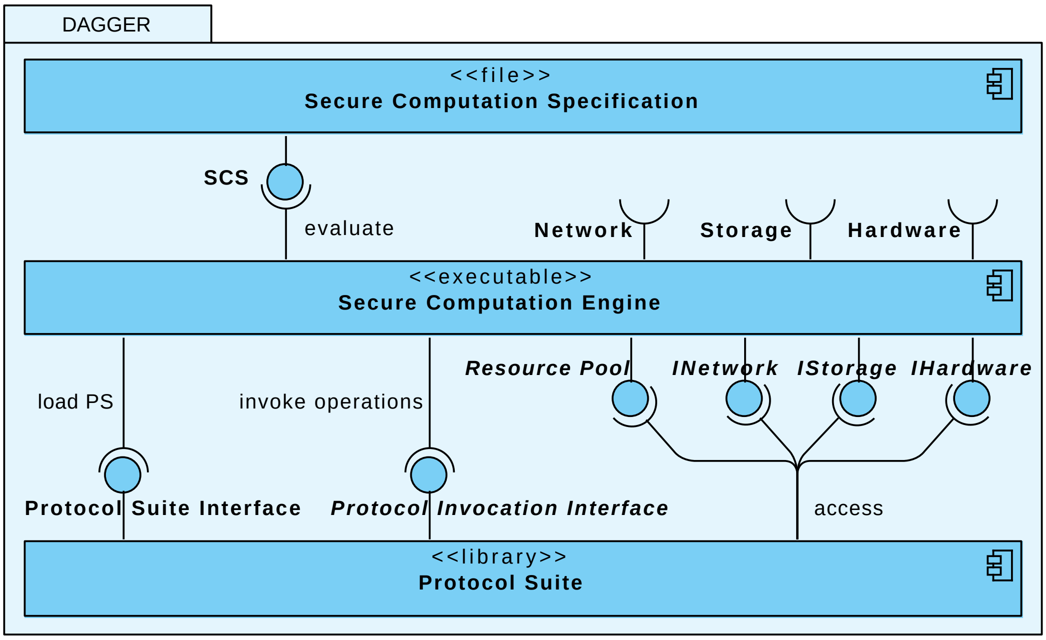

Figure 3.16: The layered overview of the architecture.

Figure 3.16 displays the main components representing the three layers and shows the boundaries where they interact. A Secure Computation Specification is a file that describes business logic algorithms in the format defined by the SCS interface. The Secure Computation Engine is an executable that can read the files in the SCS format and evaluate their contents. The SCE relies on the secure operations that Protocol Suite libraries implement and provide. SCE loads the libraries via the Protocol Suite Interface and invokes the operations via the Protocol Invocation Interface. The Protocol Suite libraries, in turn, rely on resources provided by the SCE via the Resource Pool interface. Each resource has its own interface. The following subsections will give more detail on the internal subsystems of the bottom two complex components based on the logical structure.

3.3.2 Secure Computation Engine

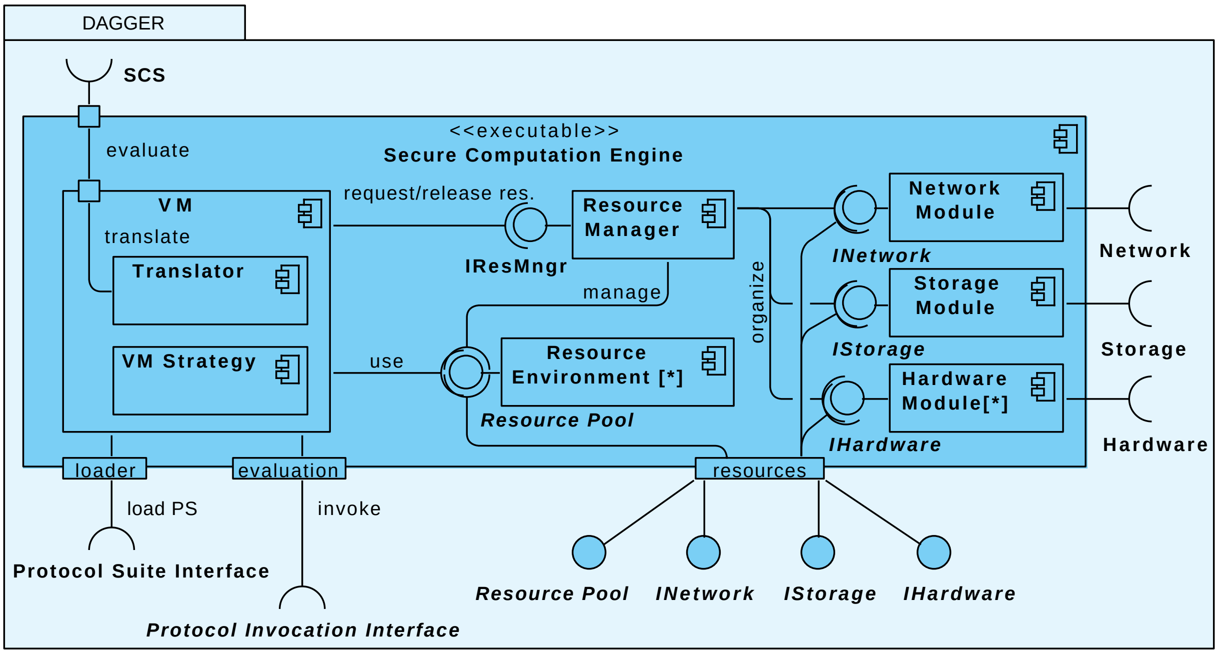

Figure 3.17: The component overview of SCE.

As described above, the SCE drives the execution of a secure application described in the SCS by invoking the appropriate Protocol Suite operations and managing various resources. It consists of the following components (also depicted in Figure 3.17).

Network Module This component is responsible for managing the secure network communication channels and transferring data between network nodes. It is capable of creating separate network resources that implement the abstract INetwork interface.

Storage Module This component is responsible for storing and retrieving data from the supported type of storage in a synchronized manner. It is capable of creating separate storage resources that can use an external synchronization mechanism for over-the-network transactions and implement the abstract IStorage interface.

Hardware Module This component is responsible for accessing secure hardware in a synchronized manner. It uses drivers to communicate with the hardware.

Resource Environment This component represents a collection of common resources that can be used internally by a single resource consumer (e.g. a single invocation of Protocol Suite operation) without interfering with other similar consumers. It implements the Resource Pool interface.

Resource Manager This component is responsible for managing Resource Environments (i.e. constructing and destroying them by request). It organizes the allocation of the separate resource structures (including by the different modules) necessary to construct the Resource Environments.

Translator This is a VM component that translates the SCS instructions to invokable Protocol Suite operations and as a result generates an intermediate representation of SCS (the Translated SCS) that is understandable to the VM.

Virtual Machine This component is responsible for executing the secure application according to the SCS. A VM uses the translator to translate the SCS to a form that the VM can understand, and to direct SCS instructions to the correct Protocol Suite operations. The VM may be augmented with various strategies for evaluating the SCS.

3.3.3 Protocol Suite

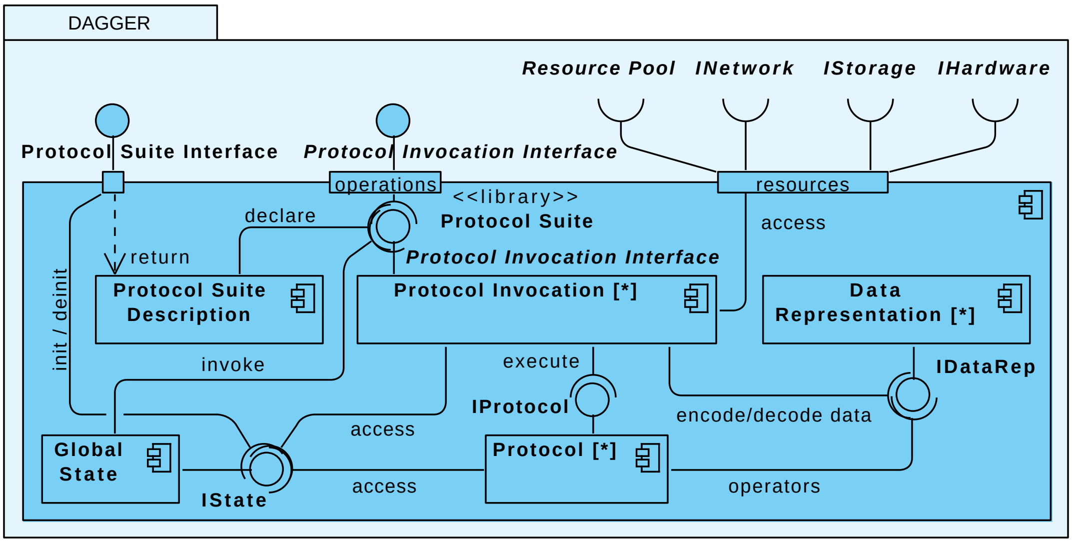

Figure 3.18: The component overview of Protocol Suites.

A Protocol Suite is a library implementing a single Secure Computation Technology such as a SMC scheme or a secure hardware based approach. The library implements the Protocol Suite Interface allowing one to initialize and access its operations. A Protocol Suite consists of the following components (also depicted in Figure 3.18).

Data Representation This is a representation of data values and the supported operations on these values that secure computation technologies use internally.

Global State When initialized, this component is responsible for running background tasks internal for the Protocol Suite, and acting as a cache storage for Protocols.

Protocol This component manifests a single cryptographic protocol of a single secure computation technique. While it typically represents a single basic operation supported by the technique, one could also compose the protocols to create complex operations. Protocols use the Data Representation to work with data values. The component has access to Global State when executed.

Protocol Invocation This component represents an entry point of an operation based on a basic or composite Protocol. It implements the Protocol Invocation Interface and relies on the resources provided via Resource Pool interface. The component has access to the Global State during execution.

Protocol Suite Description This component represents a collection of operations exposed by the Protocol Suite to the external systems so they can be invoked. Each operation is separately declared by a string signature and a reference to relevant Protocol Invocation.

3.3.4 Protocol Development Kit

Although a Protocol Suite is to be integrated into a DAGGER system, we envision that this integration can be performed independently from the concrete SCE using the Protocol Suite. The main reasons for that are flexibility of use, and portability. In order to break the dependency we have defined a set of interfaces between the Protocol Suite and an SCE, as it was discussed in Section 3.1. Altogether, these interfaces define the *Protocol Development Kit*, which is a core library that a Protocol Developer would need in order to implement a Protocol Suite and, therefore, assume to be fixed. The DAGGER platform developer would also use this library to access any Protocol Suite built using the same development kit. We herein list the interfaces that comprise the Protocol Development Kit.

PS Interfaces:

- Protocol Suite Interface
  - Protocol Suite Description
  - Protocol Invocation Declaration
- Protocol Invocation Interface
  - Argument

Resource Interfaces:

- Resource Pool
- INetwork
- IStorage
- IHardware

The library consists of two sets of interfaces. A Protocol Suite would implement the *PS Interfaces* and depend on the *Resource Interfaces* to access resources. An SCE would implement the *Resource Interfaces* and depend on *PS Interfaces* to access operations provided by Protocol Suites. This way, by implementing the same library the SCE and the Protocol Suite implementations become compatible and able to interact with each other.
In this abstract architecture we only describe the general structure and the idea behind these interfaces from a relatively high-level perspective. A more concrete specification (API reference) of an implementation based on this architecture and its interfaces can be found in the deliverable D14.2 [1].

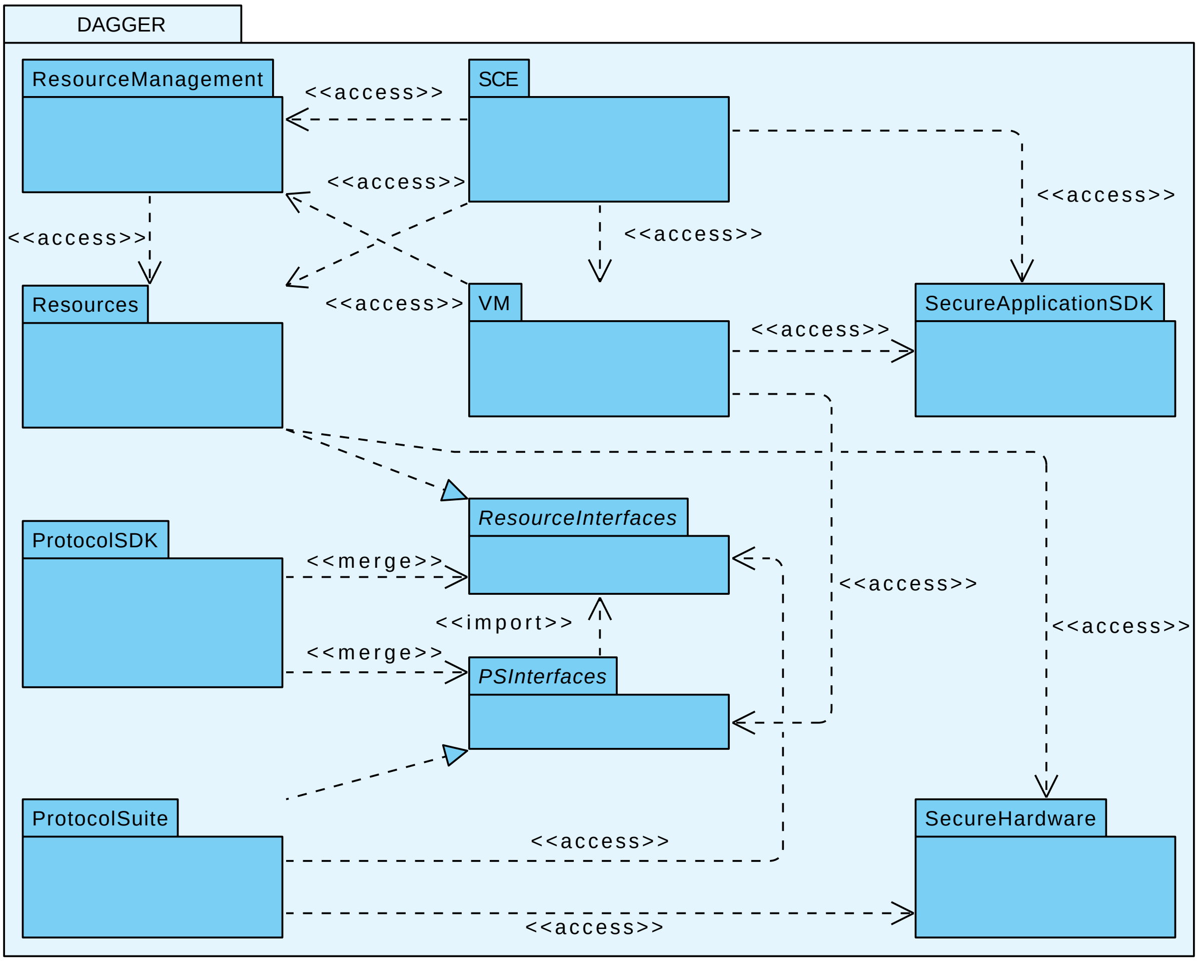

Figure 3.19: The package overview of the architecture.

3.3.5 Software Packages

In the development environment the software is typically distributed into packages that contain source code. Figure 3.19 contains the UML package diagram that gives an overview of such packages and their dependencies. Since this is an abstract architecture, the suggested package structure is likely to be somewhat different in the real implementation of a DAGGER system. Here follows the brief description of the packages.

ProtocolSDK the central package that represents the library described in Section 3.3.4 and merges two sub-packages, ResourceInterfaces and PSInterfaces, containing the relevant interfaces respectively.

SecureApplicationSDK contains the definition of the SCS format.

SecureHardware contains the interfaces and drivers necessary to work with specialized secure hardware.

Resources contains the resource (Network, Storage, Hardware) module components that somehow implement the ResourceInterfaces package. Depends on the SecureHardware to access the hardware via its interfaces.

ResourceManagement contains the Resource Manager and Resource Environment components. Depends on the Resources package as the Resource Manager accesses the resource modules to create the necessary structures for the Resource Environment.

VM contains the implementation of the VM, Translator, Translated SCS and VM Strategy. Depends on the SecureApplication package to work with the SCS format. Depends on the PSInterfaces to access Protocol Suite internal functions. Depends on the ResourceManagement package to work with the Resource Environment components.

SCE contains the implementation of the SCE component, that builds the service around the Resources, ResourceManagement, VM and SecureApplication packages.

ProtocolSuite contains the implementation of a Protocol Suite and all its related components. These include, but are not limited to: Protocol, Protocol Invocation, Data Representation, Global State and an instance of Protocol Suite Description. The implementation is based on the interfaces provided by the PSInterfaces package. It also depends on the ResourceInterfaces package to access the resources.

3.4 Deployment View

The deployment view maps the development artifacts onto physical environments such as hardware. It informs about the physical setup: which nodes exist, what runs on those nodes and how are they connected. Due to the abstract nature of this arcitechture, we cannot describe precisely how the nodes are connected (i.e. which technology is used). Hence, we only depict which nodes must interact. The deployment view for this architecture is presented in Figure 3.20.

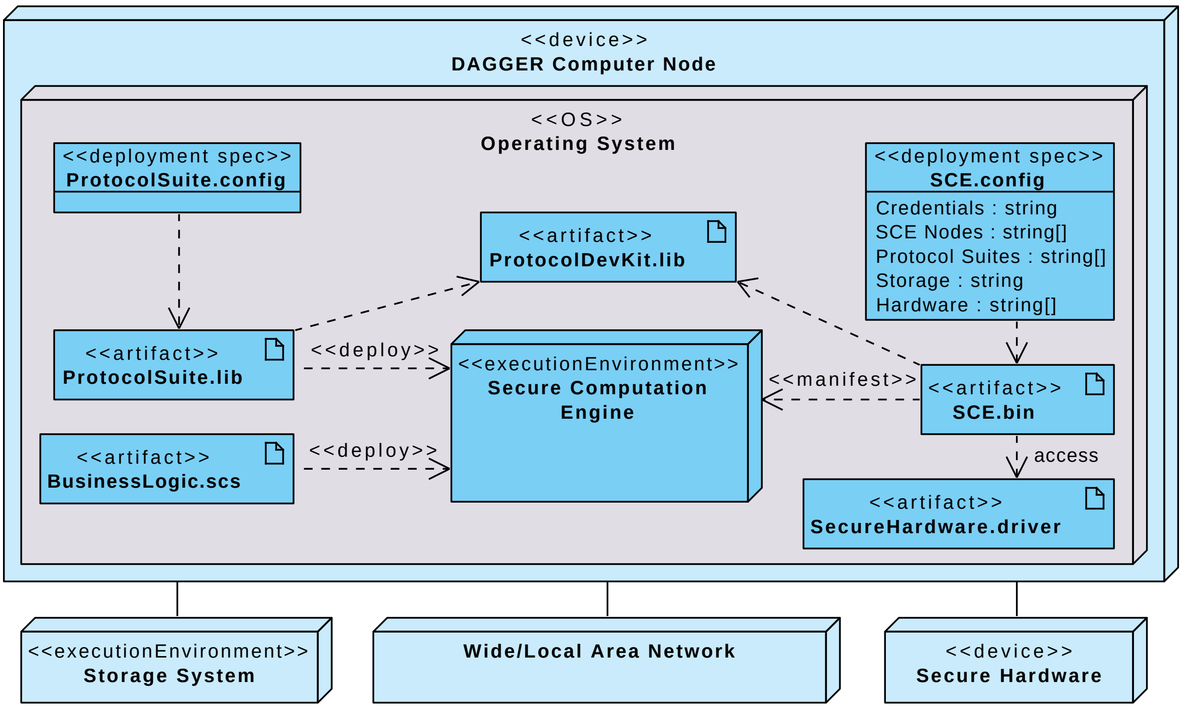

Figure 3.20: Deployment diagram.

SCE.bin This artifact is compiled from the SCE package and together with its configuration file (SCE.config) deployed on the operating system. When executed, this artifact manifests the Secure Computation Engine node.

SCE.config This configuration file represents the deployment specification for SCE.bin artifact. It describes settings such as the credentials of the SCE, the other SCE nodes that this node communicates with, the Protocol Suites this node should use, and what storage and hardware modules to load.

ProtocolSuite.lib This artifact is compiled from the SecureComputation package and together with its configuration file (ProtocolSuite.config) deployed to the Secure Computation Engine execution environment. When loaded, it provides the SCE with secure operations.

ProtocolSuite.config This configuration file represents the deployment specification for the ProtocolSuite.lib artifact. It contains secure computation technology specific settings.

ProtocolDevKit.lib This artifact is compiled from the ProtocolDevKit package and is required by the ProtocolSuite.lib and SCE.bin to communicate with each other. This artifact may be optional if compiled into the other two artifacts.

BusinessLogic.scs This is a compiled version of an SCS containing some business logic algorithms. It is deployed to the Secure Computation Engine to be evaluated.

Storage System A node responsible for persistent storage (e.g. database) and used by the Secure Computation Engine node. There is a requirement that the Storage System instance is only accessible to an SCE instance owned by the same owner. It doesn't have to be physically on a different server, but it is depicted like that since it can easily be done that way. It has advantages in a cloud setup where the database is often separated from the application server.

SecureHardware.driver This artifact is compiled from the SecureHardware package and is required by the SCE.bin to work with the hardware. Deployed to the OS together with SCE.bin.

Secure Hardware This is a component which in its nature must be a separate device. It should be connected to the DAGGER computer device. Its existence is optional, if not configured in the SCE.config and not required by any of loaded ProtocolSuite.lib artifacts.

DAGGER Computer Node Most of the described artifacts are deployed to an OS running on a DAGGER Computer Node. As described in Deliverable D21.1 [3], the secure computation techniques, that are implemented by ProtocolSuite.lib artifacts, vary from centralized (e.g. FHE, Trusted Hardware) to distributed (e.g. GC, MPC). For this reason there can be multiple DAGGER Nodes communicating with each other over the network. The exact network topology depends on the computation technology used.

Chapter 4

Conclusion

In this document we have designed an abstract architecture for a generic secure computation platform, that allows to implement secure computation techniques in a way that makes them easily pluggable into the DAGGER systems and secure applications. We have achieved a good level of portability, modularity and ease of use by breaking dependencies between the main layers and ensuring a reasonable separation of concerns. Aside from the defined interfaces and the suggested component structure, the architecture does not set any limitations on how the interfaces are actually implemented. This gives the flexibility to implement secure computation technology and the secure applications in the most suitable manner.

This architecture will benefit the research community and the industry in general, as it helps bringing Secure Computation Technology closer to being used in practice. Protocol developers will be able to spend less effort prototyping their cryptographic computation primitives while doing so independently of everything else, and still being able to cover larger market by staying compatible with the computation systems built with this architecture in mind. Application developers will be able to adapt their businesses to needs and stay competitive by being able to easily switch between various secure computation technologies without significant development effort, and as a result offer better value for their customers. DAGGER developers will be able to provide a wider variety of options to businesses as well, as more technologies become compatible with their platforms.

Finally, to demonstrate the potential of this architecture, a more concrete framework implementation based on this architecture is specified in the deliverable D14.2 [1]. The framework will support the integration of secure computation techinques from different partners into DAGGER. This architecture also complements the general SPEAR architecture for building secure computation services and applications on the cloud, as described in the deliverable D21.2 [2].

Bibliography

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

[2] Florian Hahn, Peter Sebastian Nordholt, Kasper Damgaard, Roman Jagomägis, and Reimo Rebane. PRACTICE Deliverable D21.2: Unified Architecture for Programmable Secure Computations, 2015.

[3] 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.