D22.1
State-of-the-Art Analysis

Project number:	609611
Project acronym:	PRACTICE
Project title:	PRACTICE: Privacy-Preserving Computation in the Cloud
Start date of the project:	1st November, 2013
Duration:	36 months
Programme:	FP7/2007-2013

Deliverable type:	Report
Deliverable reference number:	ICT-609611 / D22.1 / 1.1
Activity and Work package contributing to deliverable:	Activity 2 / WP 22
Due date:	April 2014 – M6
Actual submission date:	27th February, 2015

Responsible organisation:	SAP
Editor:	Isabelle Hang
Dissemination level:	PU
Revision:	1.1 (r-2)

Abstract:	This deliverable gives a survey of techniques and tools which might be contributing to realize privacy preserving computations in the cloud.
Keywords:	State-of-the-Art

Editor

Isabelle Hang (SAP)

Contributors (orderd according to beneficiary numbers)

Isabelle Hang (SAP)

Ferdinand Brasser, Niklas Buescher, Stefan Katzenbeisser, Ahmad Sadeghi, Kai Samelin, Thomas Schneider (TUDA)

Jakob Pagter, Janus Dam Nielson, Peter Sebastian Nordholt (ALX)

Kurt Nielsen, Johannes Ulfkjaer Jensen (PAR)

Dan Bogdanov, Roman Jagomägis, Liina Kamm, Jaak Randmets, Jaak Ristioja, Reimo Rebane, Jaak Ristioja, Sander Siim, Riivo Talviste (CYBER)

Manuel Barbosa, Bernardo Portela, Rui Oliveira (INESC Porto)

Stelvio Cimato, Ernesto Damiani (UMIL)

Disclaimer
The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609611.

Executive Summary

The objective of workpackage 22 is the development of software making use of secure computation and provide an easy-to-use tool for joint data analysis applications. This includes the development of new programming languages, libraries, and developer tools as well as the improvement of existing ones. In particular, this workpackage focuses on the development of secure database technologies and secure programming languages and compilers.
Deliverable 22.1. provides a capability analysis of existing secure application frameworks and secure programming languages as described in Task 2.2.1 in the DoW. Therefore, we review the state-of-the-art of existing approaches and describe their capabilities. In addition, we analyze their application scenarios and discuss assumptions and guarantees for their deployments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Workpackage 22 focuses on the improvement of methods for developing software which utilizes secure computation.
The first goal is to create and improve programming languages, libraries, and developer tools to simplify secure software development. The second goal is to develop a user friendly database and application server to use with secure computation and to develop a joint data analysis application.
Deliverable 22.1 serves as an overview of existing work and a state-of-the-art analysis. Therefore, we conduct a survey of existing techniques and tools and evaluate the capabilities of existing applications.
The implementation of secure two- or multi-party computation protocols can be supported by domain specific languages and compilers. Applying these tools helps to efficiently implement these protocols. We summarize existing work focusing on languages and compilers in Chapter 2.
Chapter 3 discusses languages and tools designed to produce formally verified implementations of cryptographic algorithm and protocols.
Confidentiality leaks or data compromises are innate security risks for database providers or cloud providers offering database-as-a-service. In particular, sensitive data demand special protection measures. Using cryptographic methods to protect the confidentiality often leads to limited usability. Therefore, efficient processing of encrypted data is an important requirement. Another problem arises if the data shall be processed by secure two- or multi-party computations. The execution of such computations involves different parties and we cannot assume that all parties are always online to provide their public and private data. Therefore, we need a persistent storage device. We discuss techniques and tools to tackle both problems in Chapter 4.
The utilization of specialized libraries and frameworks can also facilitate the implementation of secure two- or multi-party computation protocols. We discuss relevant libraries and existing frameworks in Chapter 5.
In Chapter 6 and Chapter 7, we discuss different application scenarios and analyze the capabilities of the described techniques and tools.

Chapter 2

Languages and Compilers

2.1 Fairplay and FairplayMP

FAIRPLAY is a system originally developed to support two-party computation [MNPS04] and then extended to FAIRPLAYMP to support multi-party computation [BDNP08]. The systems allow the real execution of secure two or multi-party computation protocols enabling the players to run a joint distributed computation in which each player locally evaluates a garbled boolean circuit, receiving at the end the designated output only [Yao86]. Both FAIRPLAY systems include a high-level language, called the Secure Function Definition Language (SFDL), for specifying a distributed protocol, a compiler that translates the high-level definitions into Boolean circuits described in the Secure Hardware Definition Language (SHDL), and a cryptographic engine for executing the protocols that compute securely the obtained circuits. The basic usage of the FAIRPLAY systems is the following: (1) Users write a program in SFDL , using the SFDL editor (a syntax driven GUI); (2) The program is compiled into a low level representation as a Boolean circuit file in SHDL language; (3) Players perform a joint computation computing separately the circuits and interacting when needed in order to get the desired results.

A difference between FAIRPLAY and FAIRPLAYMP lies in the underlying cryptographic engine. FAIRPLAY implements a two party computation protocol in the manner suggested by Yao, where the two parties evaluate a garbled Boolean circuit and engage in an oblivious transfer (OT) protocol to exchange the input values to the circuit. In the current implementation two variants of OT protocol have been considered, both based on the Diffie-Hellman problem: one is the 1-out-of-2 protocol presented by Bellare and Micali [BM89], the other one is the optimization of the first due to Naor and Pinkas [NP01].

FAIRPLAYMP system is based on the Beaver-Micali-Rogaway protocol which runs in a constant number of communication rounds (eight) regardless of the depth and the size of the Boolean circuit representing the computed function. In the FAIRPLAYMP optimized implementation of the protocol, players are distinguished in different roles, each player possibly having several of these roles: the Input Players (IP) which provide inputs for the computation, the Computation Players (CP) which carry out the construction and the evaluation of the garbled circuit, and the Result Players (RP) which receive the result of the evaluation.

FAIRPLAY implementation are written in Java to support cross-platform portability. FAIRPLAYMP is composed of four main packages: The player package contains implementation of the three different types of players, each one run as a different thread to avoid bottlenecks; the communication package contains simple implementation of the server and client threads which use SSL encryption in a peer to peer communication model, the circuit package holds the Circuit interface and the SHDL Circuit implementation; the utils package includes the rest of the classes, including the implementation of the BGW protocol and of the pseudo-random generator.

2.1.1 SFDL 1.0/2.0

The Secure Function Definition Language (SFDL) is the high level programming language used by FAIRPLAY to specify the function to be evaluated in the form of a computer program. SFDL 2.0 is an extended version of the language SFDL 1.0 used in FAIRPLAY and supports multi-party computation. The programs define what a virtual trusted third party must evaluate, so that when the cryptographic engine runs the compiled form of the program, the participating players can emulate the behaviour of the trusted party.

The SFDL has a C like syntax, supporting primitive types such as boolean, integer, enumerations and composition using arrays and structures. An SFDL program is divided into two main parts: the first one defines data types used in the computation and the input and the output of each player; a second part defines the computation in form of a sequence of functions. Integers may be declared to be of any bit-length, and enumerated types are allocated the minimal number of required bits. Functions receive parameters and return values and may define and use local variables. Pointers and recursion are not allowed; conditional statements are in the form if-then or if-then-else, while for-loop can be used if the loop bound is defined as a constant at compile-time.

In SFDL 1.0 the last function in the program defines the computation of the desired output. The special types AliceInput, AliceOutput, BobInput, BobOutput, must be defined in every program in order to specify input and output types of the two players, while the types Input and Output must be defined as structures to encapsulate the inputs and the resulting outputs. In Figure 2.1 an example program for computing the millionaires problem is presented: Integers are defined to be of 4 bits, input types are integer and output types are Boolean for the two players.

```
1  program Millionaires {
2      type int = Int<4>;     v
3      type AliceInput = int;
4      type BobInput = int;
5      type AliceOutput = Boolean;
6      type BobOutput = Boolean;
7      type Output = struct { AliceOutput alice, BobOutput bob};
8      type Input = struct {AliceInput alice, BobInput bob};
9
10     function Output out(Input inp) {
11         out.alice = inp.alice > inp.bob;
12         out.bob = inp.bob > inp.alice;
13         }
14 }
```

Figure 2.1: The SFDL 1.0 Code for the Millionaires Problem

The SFDL 2.0 version has been defined for the FAIRPLAYMP system supporting the definition of multiple players and introducing some changes in the syntax and in the functionalities. The function defining the computation must be called ”main” and has the names and the types of the players as parameters. Some other changes in the syntax regard the possibility to define global variables, the introduction of generic functions, where the type of the return values depends on the parameters, and the possibility to access specific bits in an already defined integer number. A sample program implementing a second price auction

among five bidders is depicted in Figure 2.2. The program specifies six players, a single Seller, and five Bidders. The Seller has no input, and the output defines the winner and the amount of the winning bid (contained in an integer 8-bit long). Each Bidder defines its bid as input, and outputs a boolean specifying if this bidder won, and the winning price. The computation is executed in the main function, selecting the highest bid and setting the winner and the winning price for each bidder' output.

```
1  program SecondPriceAuction {
2     const nBidders = 5;
3     type Bid = Int<8>;
4     type WinningBidder = Int<3>;
5     type SellerOutput = struct{WinningBidder winner, Bid winningPrice};
6     type Seller = struct{SellerOutput output};
7     type BidderOutput = struct{Boolean win, Bid winningPrice};
8     type Bidder = struct{Bid input, BidderOutput output};
9  function void main(Seller seller, Bidder[nBidders] bidder) {
10    var Bid high = bidder[0].input, Bid second = 0;
11    var WinningBidder winner = 0;
12
13    for(i=1 to nBidders-1) {
14       if(bidder[i].input > high) {
15          winner = i;  second = high;  high = bidder[i].input;
16       } else if(bidder[i].input > second)
17          second = bidder[i].input;
18    }
19    // Setting the result.
20    seller.output.winner = winner;
21    seller.output.winningPrice = second;
22    for(i=0 to nBidders-1) {
23       bidder[i].output.win = (winner == i);
24       bidder[i].output.winningPrice = second;
25    }}}
```

Figure 2.2: The SFDL 2.0 Code for a Second Price Auction among 5 Bidders

Programs written in SFDL are compiled and transformed in SHDL format, obtaining the functions and the inputs and outputs specified as a low level boolean circuit. In the SHDL output file each line specifies a wire in the boolean circuit that can be either an input bit or a Boolean gate with given truth-table and input wires.

2.1.2 Security and Applications

The runtime for FAIRPLAY and FAIRPLAYMP have the same level of security, being both immune to attacks from semi-honest adversaries and relying on the security assumptions made for the underlying cryptographic protocols (the OT and BGW protocols). In FAIRPLAYMP security is guaranteed against a coalition of at most $\lfloor n/2 \rfloor$ corrupt computation players, operating in a semi-honest way.

FAIRPLAY and FAIRPLAYMP systems have not been used in real-life applications, but a numbers of programs are available to solve different multi-party computation problems, such

as the millionaires problem, second price auction, voting (http://www.cs.huji.ac.il/project/Fairplay/home.html). Provided implementations have showed linear dependency of the running time on the size of the resulting Boolean circuit and on the number of computation players.

2.2 SecreC and Sharemind Assembly

SECREC [Jag10] (pronounced as *secrecy*) is a privacy-aware domain-specific programming language for programming secure computations on SHAREMIND 2 [BLW08, Bog13]. The main objective of the language is to simplify the implementation of high-level algorithms that process delicate data. SECREC is independent of any particular secure computation paradigm allowing the developer to focus on privacy-preserving algorithms rather than dealing with the deployment model or the details of underlying cryptographic protocols. The first version of SECREC is designed with the general architecture of SHAREMIND 2 in mind and introduces the distinction between *public* and *private* data processing. The language has later been redesigned for SHAREMIND 3, as described in Section 2.3

Being a C-like language, SECREC is procedural and statically typed, has facilities for structured programming, and allows lexical variable scope and recursion. It also offers basic array processing capabilities and provides syntactic means for pointwise operations on arrays. These design choices were made to simplify the porting of data processing algorithms found in the literature.

The language's type system clearly separates the *public* and *private* execution environments, allowing the developer to explicitly control the flow of data between the environments. This is achieved by introducing the notion of a security type in addition to the traditional data type, and strictly defining the constructs by which the security type of a piece of data can change. The importance of explicit data flow markup becomes paramount when tracking the conversion of private values into public values in order to analyze the program for privacy leaks, thereby minimizing the associated risks.

In SECREC the flow control is restricted to decisions based on public values. The reason for this is efficiency. It is infeasible to hide the whole state space of the secure computation. Each branching statement increases the number of parallel states needed to be secured and maintained, otherwise the security can be compromised by the side channel attacks like timings. Instead, the language focuses on hiding the values of private data and supporting programming patterns that hide the control flow and defeat side channel attacks.

SECREC is a compiled language. Programs written in SECREC are first translated into the SHAREMIND assembly [Jag08], which is then interpreted by the SHAREMIND 2 virtual machine. The SHAREMIND assembly language effectively links the functionality of secure SMC protocols and other supporting operations to mnemonic instructions with parameters. As such, it represents the hardware level of the SHAREMIND 2 platform and provides runtime for SECREC without becoming protocol-specific. The language is rather low-level and allows flow control through conditional and unconditional jumps.

To optimize the programming task even further, SECREC has a built in standard library of additional functionalities for vectors and matrices, as well as database interface and various general utility functions. The latest API documentation for both SECREC and SHAREMIND assembly can be found in the SHAREMIND developer reference [1]. SECREC is part of SHAREMIND SDK described in Section 6.1.

[1] SHAREMIND developer reference - https://sharemind.cyber.ee/developer-reference/

Figure 2.3 presents an example function written in SECREC. It counts the number of occurrences of a private integer value in a private array of integers, which is an interesting subtask in, e.g., histogram computation. The example demonstrates the use of security types, a vectorized pointwise comparison and two utility functions.

```
1 private uint32 count(private uint32[] data, private uint32 value) {
2     public uint32 n; n = vecLength(data);
3     private bool[n] matches = (data == value);
4     private uint32 counter; counter = vecSum(matches);
5     return counter;
6 }
```

Figure 2.3: A Parallelized Counting Function in SECREC.

2.2.1 Applications

SECREC has been used in the following applications.

1. In [BJL12, Jag10], Bogdanov, Jagomägis, and Laur used SHAREMIND 2 and SECREC to implement and evaluate four algorithms for frequent itemset mining (a data mining task). The original algorithms were redesigned specifically for SHAREMIND to take advantage of its properties.

2. In [BTW12, Tal11], Bogdanov, Talviste, and Willemson used SHAREMIND 2 and SECREC to build a secure system for jointly collecting and analyzing financial data for a consortium of Estonian ICT companies.

3. In [KBLV13], Kamm, Bogdanov, Laur, and Vilo showed how to conduct genome-wide association studies without violating privacy of individual donors and without leaking the data to third parties. SHAREMIND 2 and SECREC were used to implement and evaluate core algorithms for GWAS.

4. SECREC was used to implement statistical algorithms in the privacy-preserving income analysis demo application, built to demonstrate multi-party computations on the cloud. https://sharemind.cyber.ee/clouddemo/

2.3 SecreC 2 and Sharemind Bytecode

SECREC 2 [BLR13b, BLR13a] is a domain-specific programming language for specifying privacy-preserving applications on SHAREMIND 3 [BLW08, Bog13] platform. SECREC 2 is an imperative, strongly and statically typed language, with syntax heavily influenced by C++. SECREC 2 is a complete redesign and reimplementation of SECREC (see Section 2.2) for the new version of the SHAREMIND platform. Some of the new features of the language include: various data types, more advanced array programming facilities, basic module system, extensive standard library, parametric polymorphism, operator overloading and support for different kinds of security schemes and the concurrent use of them. The standard library of SECREC 2 is described in Section 5.6.

Syntactically, SECREC 2 is a C-like language, but that's where the similarities end. The language is mainly designed for statistical computations and data mining, and therefore lacks many of the features often found in other C-like languages. For example, input-output is mainly limited to database operations and logging. Instead, the language is focused on a strong security type system and on array operations.

With strong type system it's guaranteed that private information does not leak into public domain without the programmer explicitly stating so, and the focus on arrays allows for otherwise very costly private operations to be performed more efficiently via the use of vectorized operations.

Using a dedicated domain-specific language over existing general purpose one has various advantages. The following design decision make SECREC a unique language:

1. SECREC provides a type system that makes it difficult to implement programs that leak secure information. While the same effect is achievable in general purpose languages it is often via the abuse of the type system or by modifying or extending the language itself.

2. SECREC and the underlying bytecode is designed with remote execution in a multi-party computation setting in mind.

3. SECREC focuses on data-parallel array processing. Sequential execution of multi-party computation protocols is very slow due to the network overhead. For this reason it's almost always better to execute multi-party protocols in parallel whenever possible. SECREC focuses on data-parallel approach instead of task-parallel approach.

4. SECREC allows for concurrent use of different kinds of security schemes. This can be useful as different security schemes have different performance profiles and different security guarantees.

A small code example, that imports a module for additive 3-party protection scheme, performs a private multiplication and publishes the result, can be found in Figure 2.4. Figure 2.5 features SECREC 2 version of a parallelized counting function. Compared to the original SECREC function in Figure 2.3 the new version relies on polymorphism and on the standard library.

```
1 import additive3p;
2 domain pd_a3p additive3p;
3 void main () {
4     pd_a3p int x = 10, y = 20;
5     pd_a3p int z = x * y;
6     publish ("result", z);
7 }
```

Figure 2.4: Small SECREC 2 Example.

2.3.1 Protection Domains

A protection domain kind (PDK) is a set of data representations, algorithms and protocols for storing and computing on protected data. A protection domain (PD) is a set of data that is protected with the same resources and for which there is a well-defined set of algorithms and protocols for computing on that data while keeping the protection. Each PD belongs to a certain

```
1 import stdlib;
2 template <domain D>
3 D uint count (D uint32[[1]] data, D uint32 value) {
4     return sum (data == value);
5 }
```

Figure 2.5: A Parallelized Counting Function in SECREC 2.

PDK and each PDK can have several PDs. Protection domains are a fundamental concept in SECREC, partitioning the techniques and security resources available to the program.

A typical example of a PDK is secret sharing, with implementations for sharing, reconstruction, and arithmetic operations on shared values. A PD in this PDK would specify the actual parties doing the secret sharing, and the number of cooperating parties for reconstruction. Another example of a PDK is a fully homomorphic encryption [Gen10] scheme with operations for encryption, decryption, as well as for addition and multiplication of encrypted values. Here different keys correspond to different PDs. Non-cryptographic methods for implementing PDKs may involve trusted hardware or virtualization.

In SECREC, public computations also form a PDK, containing a single PD called *public*. In general, a PDK has to provide a list of data types it operates with, and functions that operate on them. The cryptographic implementations of these functions are beyond the scope of the application programmer. These functions are made available by the SHAREMIND engine to applications, written in SECREC, through *system calls*. The available functions may be different for different PDKs and data types. It is possible that certain functionality that is provided through a dedicated system call in one PDK is implemented as a SECREC program for another PDK.

In SECREC it is possible to write PD-polymorphic code via C++ template-like syntax. Template domain-arguments can be restricted to a particular PDKs. During function overload resolution restricted polymorphic functions are preferred over unrestricted ones and monomorphic functions are picked over polymorphic ones. For example, in Figure 2.6 a generic sorting function is declared that operates on any PD but also a more efficient implementation for additive 3-party PDK is provided too. The SECREC 2 standard library is built on the concept of providing generic, but potentially slow, implementations that protection domain modules can overload for better performance.

```
1 template <domain D>
2 D int[[1]] sort (D int[[1]] src) { ... }
3 template <domain D : additive3p>
4 D int[[1]] sort (D int[[1]] src) { ... }
```

Figure 2.6: Domain-Polymorphic Sorting Functions in SECREC 2.

2.3.2 Implementation

The SECREC 2 compiler has a fairly standard structure. After performing lexical, syntactic and semantic analysis a polymorphic high-level program is transformed into a simple monomorphic intermediate representation. The intermediate representation does not contain

complicated control-flow structures (such as for-loops and if-statements) or implicit type or domain conversions. All of the optimization passes and program analysis, such as information-flow analysis, are performed on the intermediate representation. Finally, the intermediate representation is transformed into bytecode that can be executed by SHAREMIND.

SHAREMIND bytecode contains a list of system calls that it may invoke during execution and a list of PDs that it uses. During deployment it is checked if all the PDs and system calls (usually provided by the PDs) are available, and the program is rejected if not. System calls are the mechanism for the program to invoke the capabilities of SHAREMIND. For example, all of the multi-party computation protocols are provided by system calls and they are not wired into the bytecode itself. System calls have very small run-time overhead as they are associated with native function calls during deployment.

2.3.3 Applications

SECREC 2 has been used in the following applications.

1. In [KW13], Kamm and Willemson used SHAREMIND 3 and SECREC 2 to build a satellite collision prediction tool that keeps the trajectories of satellites confidential.

2. In [BLT13], Bogdanov, Laud and Talviste used SHAREMIND 3 and SECREC 2 to implement oblivious sorting algorithms using secret sharing. The study evaluates the theoretical performance and discusses the practical implications of the different approaches.

3. In [BKLPV13], the authors present a privacy-preserving statistical analysis toolkit built using SECREC 2.

2.4 Sharemind Protocol Language

Existing secure multi-party computation (SMC) frameworks use different protocol sets for achieving privacy. Several frameworks implement the arithmetic black box (ABB) [DN03], the methods of which are called during the runtime of a privacy-preserving computation by the SMC engine in the order determined by the specification of the computation. For our concern, an ABB is just a set of SMC protocols that are *universally composable* [Can01]. Universal composability is a property that guarantees that the protocols can be composed in any way, either sequentially or in parallel, without losing security guarantees. An ABB must at least contain the methods for linear combination and multiplication of private integers, but it contains more in typical implementations.

The SHAREMIND SMC framework [Bog13] features an exceptionally large ABB. Besides the operations listed above, it also contains comparison, bit extraction, widening, division of arbitrary-width integers [BNTW12], as well as a full set of floating-point [KW13] and fixed-point operations, including the implementations of elementary functions. More protocol sets on top of different SMC methods are planned. More often than not SHAREMIND protocols are specified in a compositional style forming a hierarchy, with more complex protocols invoking simpler ones (multiplication, widening, certain bit-level operations) for certain tasks [BNTW12].

The implementation of protocols for ABB operations is an error-prone and repetitive task. Manual attempts to optimize complex protocols over the composition boundaries is a

laborious task prone to introduce errors and make the library of protocols hard to maintain. Implementation is made more difficult due to the fact that protocols need to work for various different integer widths and many of the language abstractions (such as virtual function calls) entail unacceptable run-time overhead. Most commonly we need to implement protocols for 8-, 16-, 32- and 64-bit integers, but in some cases for 128-bit or even larger integers. The task of building and maintaining implementations of primitive protocols is naturally answered by introducing a domain-specific language (DSL) for specifying them.

The SHAREMIND protocol DSL [LPPR13] allows for protocols to be specified in a manner similar to their write-up in papers on SMC protocols. A different language called SECREC 2 [BLR13b, BLR13a] (also see Section 2.3) is used for specifying the privacy-preserving applications as a composition of these protocols. Having different languages for implementing different levels of the privacy-preserving computation allows for the best suited optimizations to be applied to each level, and improves the user experience by allowing the languages to be tailored for the specific domains. For example, primitive protocols are specified and implemented in a declarative style, but applications built on top of the primitive protocols are specified in imperative style as a sequence of protocol invocations.

2.4.1 Language Description

SHAREMIND protocol DSL is a functional language, mimicking the style of the pseudocode used to present protocols. A program in this language states, which party computes which values from which previously available values. Computations used several times are abstracted as functions.

The language allows to state similar computations performed by different parties only once. Actually, this is the default mode and each variable `x` and constant `c` in the program may denote a separate value for one or more parties, depending on the context. To access the value of `x` at a particular party no. i, one may write `x from` i. In party i's code, the pseudo-numbers `Prev` and `Next` denote the parties no. $(i-1)$ and $(i+1)$ (modulo the number of parties). There are specific syntactic constructs to state that certain computations have to be made only by a subset of parties. Network communication in the DSL is always explicit: it's a type error for a party to use a value directly that it has not defined itself.

The data type system of the DSL is inspired by Cryptol [Lew07], and at its core the type system is classic Damas-Hindley-Milner [DM82] extended with type constraints (or predicates). The language has only one ground type — `bit`— and one data type constructor for arrays and a type constructor for functions. We write `arr`$[\tau, n]$ for a n-element array with elements of type τ. For the sake of conciseness, `uint`$[n]$ is a type synonym for an array of n bits. Size polymorphism allows the protocols to be specified once for any input length. Similarly to Cryptol, types can be refined with equality constraints over type variables.

Figure 2.7 gives the specification for multiplying numbers $u, v \in \mathbb{Z}_{2^{64}}$, additively shared between three parties [BNTW12] (i.e. a private value $x \in \mathbb{Z}_{2^{64}}$ is represented as each party i holding $x_i \in \mathbb{Z}_{2^{64}}$, satisfying $x_1 + x_2 + x_3 \equiv x \pmod{2^{64}}$). The code first specifies that the protocol operates over 3 parties, then defines two functions that are polymorphic over the bit widths, and finally declares one protocol as instantiation of `mult` function to a concrete 64-bit integers. We see the similarity with Algorithm 1 and Algorithm 2 in [BNTW12].

```
1  parties 3
2  def reshare (u: uint[n]): uint[n] = {
3    let r = rnd ()
4        w = u - r + (r from Prev);
5    return w;
6  }
7  def mult (u: uint[n], v: uint[n]): uint[n] = {
8    let u' = reshare (u)
9        v' = reshare (v)
10       w  = u' * v' + u' * (v' from Prev) + (u' from Prev) * v';
11   return reshare (w);
12 }
13 protocol mult(u: uint[64], v: uint[64]): uint[64]
```

Figure 2.7: Multiplication Protocol specified in SHAREMIND Protocol Language

2.4.2 Implementation

The compilation pipeline of the DSL is fairly standard. The compiler performs the following phases: lexical analysis, syntactic analysis, static checking, translation to an intermediate representation, optimizations and translation to C++. Static checks include: data type verification, party type verification and, optionally, security verification. After static checks the high-level code is translated to much simpler low-level code based on system F_ω (λ-calculus with type application and type operators). The low-level representation is evaluated to a normal form which is converted to an arithmetic circuit. The arithmetic circuit is optimized with a separate tool and is converted to C++ code and is finally integrated with SHAREMIND. The generated C++ code uses various facilities, such as network communication primitives, provided by the SHAREMIND framework.

2.5 TASTY and TASTYL

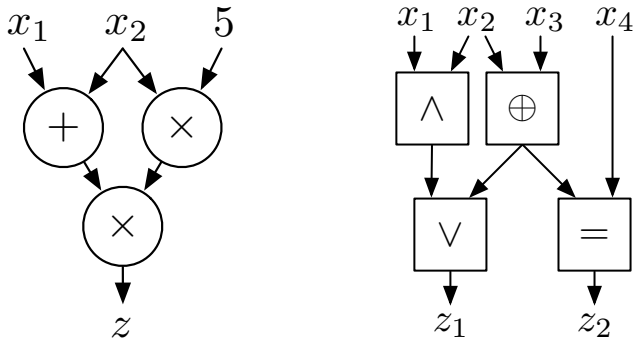
2.5.1 Introduction

TASTY (Tool for Automating Secure Two-partY computations) is a tool suite addressing secure two-party computation in the semi-honest model [HKS+]. TASTY incorporates a compiler, an expressive high-level domain-specific description language named TASTYL (TASTY input language; refer to Sect. 2.5.4 for more details), and the possibility to run benchmarks. TASTY's goal is to speedup the *online-phase* of the evaluation of a function f. The online-phase can be considered more critical: the setup can be done at any point in time, i.e. prior to evaluation of the function f. To achieve the desired speedup, TASTY's main feature allows to compile and evaluate functions not only using Garbled Circuits (GC) [Yao82, Yao86], but also (additively) homomorphic encryption (HE) schemes, e.g. Paillier [Pai99], at the *same time*. However, TASTY is not per se limited to the mentioned primitives: TASTY can easily be extended to use other primitives as well, e.g. fully-homomorphic encryption [Gen10]. Compiled binaries of TASTY are available for download.[2]

[2] https://code.google.com/p/tastyproject/

2.5.2 Background and Ideas

Prior to TASTY, compilers for secure function evaluation were limited to compile protocols to either use GC- *or* HE-schemes. TASTY implements a hybrid-approach, combining GCs *and* HE, as described in [KSS13].[3] More precisely, for each function f to be evaluated, the programer can, using TASTYL, define which parts of f should be computed by GCs or HE. The combination of both approaches results in a considerable speedup of the computation, as some functions cannot be efficiently encoded as either one [HKS+, KSS09]. Namely, arithmetic circuits (See Fig. 2.8(a)) cannot efficiently encode non-linear functions like comparisons, which includes Yao's millionaire problem [Yao82, Yao86], and XORs [KS08]. On the



(a) Arithmetic Circuit (b) Boolean Circuit

Figure 2.8: Arithmetic and Boolean Circuits

other hand, Boolean circuits (See Fig. 2.8(b)) cannot express arithmetic functions like a multiplication efficiently. Currently, the most efficient implementations, including TASTY, use the multiplication algorithm by Karatsuba and Ofman [KO62], one of $\mathcal{O}(3n^{\log_2 3})$, where n is the number of bits of each input number. Their algorithm must still be encoded as a Boolean circuit, which results in an additional blow-up of the resulting circuit's size, and therefore also the time required for its evaluation. TASTY is able to use (additively) HE to evaluate a multiplication gate. However, evaluating this gate requires one additional round of communication [HKS+].

Hybrid Representation and Evaluation.

As aforementioned, TASTY implements a hybrid approach that allows to combine GCs and HE. In a nutshell, f can be split into several sub-functions. Each sub-function is then evaluated using the representation which promises the best efficiency. Basically, each participant, denoted $\mathcal{C}$ for the client, and $\mathcal{S}$ for the corresponding server, has its own input x. The server $\mathcal{S}$ then converts x into an encrypted value $[\![x]\!]$ using a HE-scheme. That is, it encrypts its data x under its own public key. For TASTY, the (additive) HE-scheme by Paillier [Pai99], including the performance improvements given in [DJ01], has been implemented. In turn, the client $\mathcal{C}$ converts its own value x into a garbled value $\widetilde{x}$, using GCs. Having $[\![x]\!]$ and $\widetilde{x}$, the function f can then be evaluated using both mechanisms simultaneously. However, evaluating parts of f using different representations requires that one can convert between the representations on-the-fly. TASTY therefore also implements the algorithms required to convert between different representations. We now provide a brief overview how TASTY implements

[3] A preliminary version appeared as [KSS10]

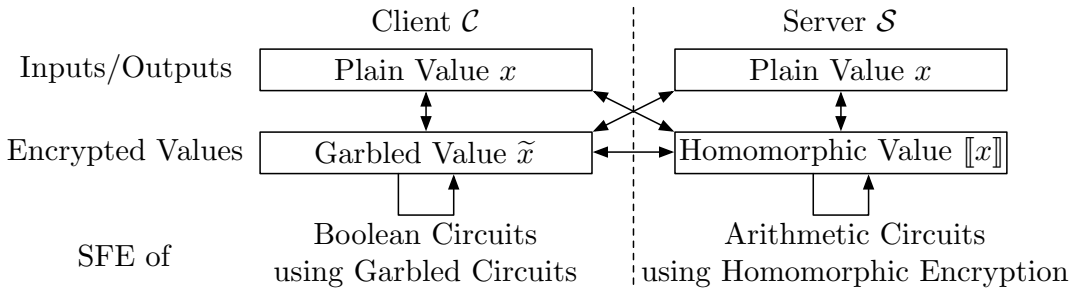


Figure 2.9: TASTY's Hybrid Approach

the conversions, as depicted in Fig. 2.9. A more detailed description of the algorithms, also addressing special cases, can be found in [KSS13].

- *Garbled to Homomorphic.* To convert from a garbled value $\widetilde{x}$ to a homomorphically encrypted value $[\![x]\!]$, $\mathcal{S}$ adds a random mask r. The result is decrypted, and send to $\mathcal{C}$. $\mathcal{C}$ then encrypts it using a HE-scheme. Finally, the random mask r is removed by $\mathcal{S}$.

  There is another method based on bit-wise encryption. Details can be found in [KSS13].

- *Homomorphic to Garbled.* To convert $[\![x]\!]$ into a garbled value $\widetilde{x}$, $\mathcal{S}$ sends an additively blinded value $[\![x+r]\!]$ to $\mathcal{C}$, where r is a random mask. $\mathcal{C}$ then decrypts $[\![x+r]\!]$. This value is encoded into a corresponding garbled value χ. Then, a subtraction circuit subtracts r from χ.

- *Garbled to Plain.* To convert from a garbled value $\widetilde{x}$ to the plain-text x, $\mathcal{S}$ only requires to the send the clear value to $\mathcal{C}$. If the garbled value $\widetilde{x}$ must be converted into a plain value for $\mathcal{S}$, $\mathcal{C}$ only requires to send $\widetilde{x}$ to $\mathcal{S}$ which obtains the plain value by decrypting it.

- *Homomorphic to Plain.* To decrypt $[\![x]\!]$, $\mathcal{C}$ sends $[\![x]\!]$ to $\mathcal{S}$, which performs the decryption and then sends the plain-value back to $\mathcal{C}$. If $\mathcal{S}$ is not allowed to the learn the plain-value, we blind $[\![x]\!]$: $\mathcal{C}$ draws a random r, encrypts it, obtaining $[\![r]\!]$. It then calculates $[\![x+r]\!]$, sends it to $\mathcal{S}$, which then returns $x+r$ to $\mathcal{C}$. $\mathcal{C}$ then unblinds the received value.

As TASTY focuses on semi-honest adversaries, we do not require to prove that conversions were done correctly.

2.5.3 Implementation

TASTY splits the whole protocol execution in several parts. A high-level overview is depicted in Fig. 2.10. We now give a brief explanation of each phase.

1. The server $\mathcal{S}$ and the client $\mathcal{C}$ agree on a protocol specification. This specification is written in TASTYL (See Sect. 2.5.4).

2. Both participants start the protocol execution, which can further be detailed as follows:

   (a) In the "Analyzation Phase", an analysis of the protocol is performed. This includes a correctness check of the protocol description, and the determination which parts of the protocol can already be calculated ahead of time, i.e. during setup.

   (b) In the "Setup Phase", the parts which can be precomputed (determined by the analyzation phase) are now calculated.

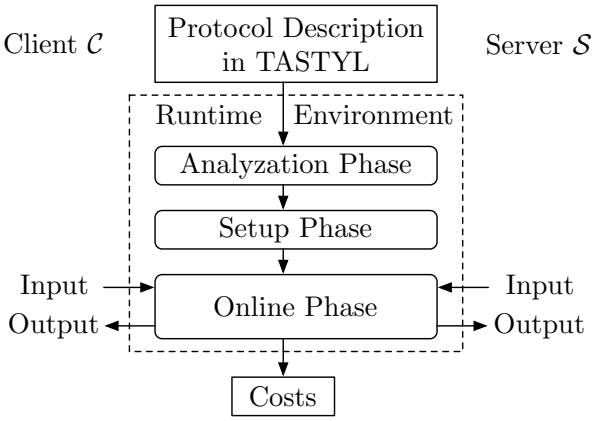


Figure 2.10: Architecture of TASTY

(c) In the "Online Phase", both parties perform the online-part of the protocol. This includes providing their respective inputs, compute de- and encryptions, the OTs, and also evaluating the function f itself.

3. Finally, in the "Cost Phase", TASTY measures the individual costs of the steps above.

Optimizations.

TASTY uses state-of-the-art performance optimizations. This section describes which ones have been implemented for TASTY.

Oblivious Transfer. To decrease the time required in the online-phase, TASTY uses the pre-computable OT introduced in [Bea95]. Beaver's OT allows to shift computational overhead to the setup-phase. This is directly utilized by TASTY. To further decrease the time used in the online-phase, TASTY also deploys OT-Extension, as introduced in [IKNP03]. The remaining required base-OTs, performed in the online-phase, are implemented using the OT-protocol by Naor and Pinkas [NP01].

Garbled Circuits. The implementation of GCs use the Free-XOR technology originating from [KS08]. As its name suggests, free-XOR allows to evaluate XOR-gates nearly for free: the evaluator only requires to perform a XOR on the garbled values given. To construct the circuit, TASTY uses the ideas given in [KSS09], which aim for reducing non-XOR-gates for several functionalities. Moreover, TASTY uses the garbled row-reduction, introduced in [PSSW09], which reduces the size of the garbled table to $2^d - 1$ entries, where d is the fan-in of the non-XOR-gate.

Packing Values. Packing multiple values within the HE-scheme is a natural extension, leading to an additional performance gain. In particular, it is possible to "pack" more than one value, i.e. $x_n, x_{n-1}, \ldots, x_1$, into a single cipher-text $[\![x]\!]$ using Horner's polynomial evaluation scheme. Now set $[\![X]\!] = [\![x_n]\!]$ and for all $0 < i < n$ iteratively set $[\![X]\!] \leftarrow 2^{|x_i|+1}[\![X]\!] \boxplus [\![x_i]\!]$. Here, $[\![x_i]\!]$ is a cipher-text of x_i under the HE-scheme used, while $\boxplus$ denotes the operation to perform the addition on the cipher-texts. $[\![X]\!] = [\![x_n||x_{n_1}|| \ldots ||x_1]\!]$ follows, where $||$ denotes a (uniquely reversible) concatenation.

2.5.4 TASTYL

TASTYL is the description language of TASTY. It is a subset of the Python language. In TASTYL, the programmer specifies a protocol by defining which operations are to be performed on what data. In particular, the programmer specifies a sequence of operations to be carried out on (potentially encrypted) data.

Type System.

The type system of TASTYL, and the operators supported by each type, are depicted in Fig. 2.11. Each variable in TASTY is either a single scalar or a vector $\vec{v} = (v_1, v_2, \ldots, v_n)$ consisting of n scalars. A variable can either be a plain value or can be encrypted (either using HE or GC).

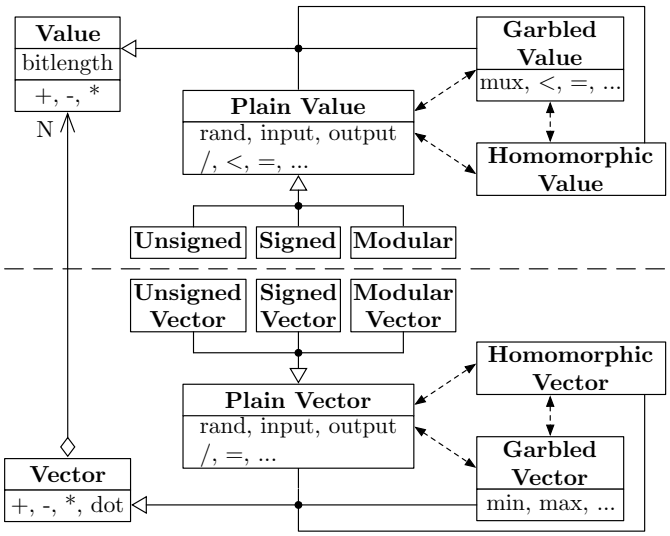


Figure 2.11: TASTY Types and Operators

All values and vectors provide the basic operators for (component-wise) addition, subtraction, and multiplication. Additionally, vectors also provide a method for dot multiplication:

$$\langle \vec{v}, \vec{w} \rangle = \sum_{i=1}^{n} v_i w_i$$

Number Representations. Each `Value` has a bit-length ℓ that represents the number of bits needed for its representation. `Unsigned` are unsigned integer values in the range $[0, 2^{\ell}[$. `Signed` are signed integers in the range $]-2^{\ell-1}, 2^{\ell-1}[$. Finally, `Modular` are elements in the plain-text space of the HE-scheme, e.g. $\mathbb{Z}_n$ for Paillier [Pai99].

In addition to the operations of `Value/Vector`, the plain/encrypted types support further operations and conversions:

Plain Value/Vector. Inputs and outputs of the two parties are `Plain Values/Vectors`. They can be chosen uniformly at random and provide additional operations (integer) division and comparison. A division raises an exception for division by 0 or a non-invertible `Modular` value.

Homomorphic Value/Vector. `Unsigned`, `Signed` and `Modular Values/Vectors` can be converted into and from homomorphically encrypted `Homomorphic Values/Vectors` of a server $\mathcal{S}$. While `Unsigned` and `Modular` values are mapped directly, for `Signed` values, the positive values are mapped to the elements $0, 1, \ldots$ of the plain-text space of the underlying HE-scheme and the negative values to $n-1, n-2, \ldots$, as described in [KSS13]. Addition of two homomorphic, and (dot-) multiplication of a Homomorphic with a `Plain Value/Vector` provided by $\mathcal{S}$ is done non-interactively. (Dot-) multiplication of two homomorphic `Values/Vectors` requires one round of interaction.

Garbled Value/Vector. Unsigned/Signed Plain and Homomorphic Values/Vectors can be converted into and from `Garbled Values/Vectors` of client $\mathcal{C}$. A `Garbled Value` can be compared with another one resulting in a `Garbled Value` of length one bit. This can be used to multiplex (`mux`) between two `Garbled Values`. Similarly, the minimum or maximum value and/or index of the components of a GARBLED VECTOR can be determined as `Garbled Value(s)`, e.g. `min_value` computes the minimum value. For each operation on `Garbled Values/Vectors`, TASTY automatically generates the underlying GC.

Syntax and Example.

We will use the example depicted in Fig. 2.12 to clarify how TASTYL works.

Example. Client $\mathcal{C}$ and server $\mathcal{S}$ have vectors $\vec{v}$ and $\vec{w}$ of $n = 4$ unsigned 32-bit values as inputs. As output, $\mathcal{C}$ obtains $\min_{i=1,..,n}(v_i \cdot w_i)$. The products $\langle v_i \cdot w_i \rangle$ are computed with HE and the minimum with GCs.

This protocol can be directly formulated in TASTYL as shown in Fig. 2.12 and described in the following.

The protocol gets two parties `client` and `server` as inputs to whom the variables used throughout the protocol are bound (details below). At the beginning, two constants $n = 4$ and $\ell = 32$ are defined. Then, the input of $\mathcal{C}$, `client.v`, is defined as an unsigned vector of bit-length ℓ and dimension n, and read from standard input. Similarly, the input of $\mathcal{S}$, `server.w`, is defined and read. Then, $\mathcal{C}$'s input vector `client.v` is converted into a homomorphic vector `server.hv` for $\mathcal{S}$ who multiplies this component-wise with his input vector `server.w` resulting in the homomorphic vector `server.hx`. This homomorphic vector is converted into a garbled vector `client.gx` and its minimum value `client.gmin` is computed. Finally, $\mathcal{C}$ obtains the intended output by decrypting (converting) `client.gmin` into the unsigned value `client.min`.

Type Conversions. Types can be naturally converted into each other by providing them as input to the constructor of the target type, e.g. in Fig. 2.12, the unsigned vector `client.v` is converted into the homomorphic vector `client.hv` via `client.hv=HomomorphicVec(val=client.v)`.

Garbled Bit Manipulations. To allow manipulation of single bits, a Garbled Value `gv` can be converted back and forth into a list of *Garbled Bits* (= Garbled 1-bit Values): `gv[i]` yields the i-th garbled bit of `gv` ($i = 0$ is the least significant bit). Vice versa, a (unsigned) garbled m-bit value `gv` can be constructed from a list of m garbled bits, e.g.

```
# -*- coding: utf-8 -*-
def protocol(client, server):
    N = 4
    L = 32

    # input of client
    client.v = UnsignedVec(bitlen=L, dim=N)
    client.v.input(desc="enter values for v")

    # input of server
    server.w = UnsignedVec(bitlen=L, dim=N)
    server.w.input(desc="enter values for w")

    # convert unsigned to homomorphic vector
    client.hv = HomomorphicVec(val=client.v)
    server.hv <<= client.hv

    # multiply vectors (component-wise)
    server.hx = server.hv * server.w

    # convert homomorphic to garbled vector
    client.gx <<= GarbledVec(val=server.hx)

    # compute minimum value
    client.gmin = client.gx.min_value()

    # convert garbled to unsigned value and output
    client.min = Unsigned(val=client.gmin)
    client.min.output(desc="minimum value")
```

Figure 2.12: Example TASTYL Program

`gv = Garbled(val=[gb0,gb1])`. Additionally, TASTYL allows to select a slice of garbled bits, e.g. `gv[i:i+m]` is a (unsigned) garbled m-bit value consisting of the i-th to $(i+m-1)$-th garbled bit of `gv`.

Send Operator. The send operator `<<=` transfers variables between the parties, e.g. in Fig. 2.12, `hv` is sent from $\mathcal{C}$ to $\mathcal{S}$ with `server.hv <<= client.hv`. When combined with a type conversion, the send operator invokes the corresponding conversion protocol, e.g. in Fig. 2.12, homomorphic vector `hx` held by $\mathcal{S}$ is converted into garbled vector `gx` held by $\mathcal{C}$ with `client.gx <<= GarbledVec(val=server.hx)`.

Binding of Variables. While constants can be declared globally (e.g. `N` and `L` in Fig. 2.12), each variable has to be assigned to one of the parties as an attribute.

Inferring Type and Length Automatically. For each operator, TASTY automatically infers the bit-length and type of the output variables from those of the input variables, such that no

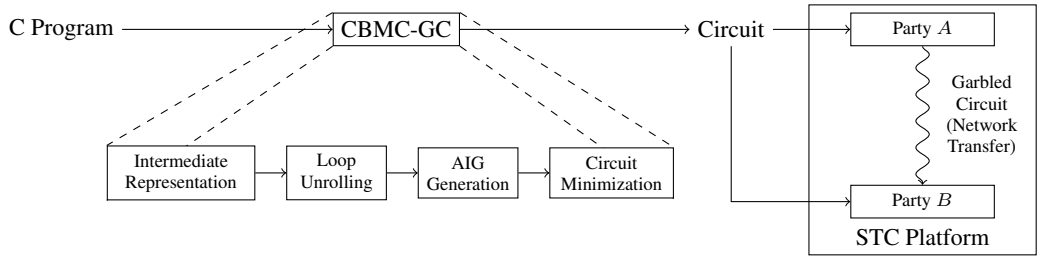


Figure 2.13: STC Tool Chain

overflow occurs. Homomorphic variables raise an exception, if the result does not fit into the plain-text space of the homomorphic crypto-system.

For example, in Fig. 2.12 the component-wise product of two vectors with n components of unsigned `L`-bit values results in the homomorphic vector `server.hx` with n components of unsigned 2ℓ-bit values.

Multiple Outputs. GCs can also have multiple garbled output values written as comma separated list on the left side of the assignment operator, e.g. the garbled minimum value `gv` and its index `gi` can be computed as `(client.gv, client.gi)=client.gx.min_value_index()`.

Circuits from File. TASTY allows secure evaluation of Boolean circuits read from an external file, e.g. circuits generated by the FairplayMP compiler [BDNP08] (See Sect. 2.1). For this, the labels of the input- and output wires of the circuit are mapped to Garbled Values of corresponding bit-length.

2.6 CBMC-GC

One main obstacle for practical application of Secure two-party computation (STC) was the lack of support for general programming languages, as only circuit evaluation [HEKM11] or simplified programming languages [MNPS04] were supported. In [HFKV12] CBMC-GC, the first STC compiler for full ANSI C, was presented by Holzer et al.. The authors argue that the practical application of STC should be viewed as a combination of compiler and security research (cf. Figure 2.13):

(i) *STC compilation*, i.e., the STC compiler translates the source code into a circuit that is optimized towards its use in STC and (ii) *STC interpretation*, i.e., the STC framework evaluates generated circuits in a way that ensures the STC guarantees. Thus, separation of concerns is a crucial step towards broad practical use of STC.

Figure 2.13 shows CBMC-GC in the STC tool chain. CBMC-GC translates a C program into a circuit which is then deployed to the two STC parties A and B. The STC framework is essentially an interpreter for the circuit. In the current implementation, the GC construction proposed in [KS08] is used with optimizations from [KSS09, PSSW09], allowing XOR-gates to be evaluated at essentially no cost.

CBMC-GC solves the millionaires' problem with the source code shown in Figure 2.14: The procedure `millionaires` is a standard C procedure, where only the input and output variables are specifically marked up, designated as input of party A or B (Lines 4 and 5) or as

output (Line 7). But aside this input/output convention, arbitrary C computations are allowed to produce the desired result, in this case a simple comparison (Line 6).

Implementation The compiler CBMC-GC[4] is based on the software verification tool CBMC [CKL04]. Since CBMC is a bounded model checker for ANSI C, it translates any given C program into a Boolean constraint which represents the program behavior at a bit-precise level up to a bounded number of steps. In a nutshell, this capability of CBMC is adapted to provide the circuits needed for STC. The compilation is divided into four steps, where the first two steps are part of the standard CBMC processing and the second two are specific to STC tasks. For more details on the first two compilation steps, please see [HFKV12].

(1) *Intermediate Representation.* The C program gets translated into an intermediate representation—a so-called GOTO program. The only control structures remaining in a GOTO program are guarded GOTOs.

(2) *Loop Unrolling.* Loops and recursive function calls are unrolled up to a specific depth. CBMC-GC tries to compute this depth by a static analysis, but in case of failure, the depth can be specified by the user. After unrolling, we have a loop-free representation of the program.

(3) *AIG Generation.* It remains to translate each program statement into a circuit which encodes the bit-precise semantics of the computation the statement performs. CBMC-GC uses *and-inverter graphs* (AIGs) as an intermediate circuit representation. AIGs are directed acyclic graphs whose nodes represent logical AND gates. The edges of an AIG represent wires between gates. Some of these wires can negate the transmitted signal. Throughout the generation of this intermediate circuit, structural hashing, i.e., the removal of duplicated gates, and constant propagation are performed to keep the resulting circuit small [MCJB05]. CBMC-GC incorporates the ABC framework [MCB06] to generate the intermediate representation.

(4) *Circuit Minimization.* XOR gates are preferable due to their small computation costs and therefore the circuit minimization step tries to maximize the number of XOR gates in the resulting circuit while keeping the overall circuit size small. Here, a repeated pattern based subcircuit rewriting is performed in combination with structural hashing, constant propagation, and a simplified version of SAT-sweeping [Kue04].

```
1 #include <cbmc-gc.h>
2 void millionaires() {
3   int a, b, result;
4   __CBMC_GC_INPUT_A(1, a);
5   __CBMC_GC_INPUT_B(2, b);
6   result = (a > b)?1:0;
7   __CBMC_GC_OUTPUT(3, result);
8 }
```

Figure 2.14: C Code for Yao's Millionaires' Problem.

[4] http://forsyte.at/software/cbmc-gc/

2.7 L1

The tool L1 consists of a programming language and a compiler for secure computation (SC) protocols. It also includes the option to run benchmarks.
L1 is a programming language for cryptographic protocols with the design goal to achieve faster SC protocols. It enables to program special SC protocols potentially mixing techniques from multiple general SC protocols.
The compiler of L1 enables to translate a SC protocol written in L1 into a set of Java programs. The tool L1 enables players in a SC protocol to encode this protocol in the L1 programming language and to compile it with the L1 compiler. The result is a set of java programs, one for each player in the protocol. To run the SC protocol, the players have to execute the Java Program.
The L1 language was designed with two requirements in mind. First, it should enable a developer to easily learn this language. Second, it should enable a developer to quickly and efficiently implement SC protocols.
To meet these goals, the following design decisions were made:
To support the easy adoption of the L1 language by a developer, the grammar of L1 is similar to the well-known programming languages Java and C. As most of developers are either familiar with Java or C, this reduces the initial burden to learn a new programming language.
In addition, the syntax of the language of L1 is reduced to a minimal set of constructs. This reduces the learning effort. However, this minimal set is sufficient to provide all necessary language constructs to implement SC protocols efficiently.
The L1 language contains the following language constructs:

- Variables
- Expressions
- Control Flows
- Functions
- Modules

Listing 2.1 shows an example of source code written in L1 containing all language constructs.

```
1  //modules
2  include "key.l1";
3
4  //self defined functions
5  int newHash(int value) {
6  int hash;
7  ...
8  return hash;
9  }
10
11 //variables
12 int hash;
13
14 //assignment with expression
15 int salt = rand(1000)+1;
```

```
16
17 //player dependent statement
18 1: {
19 //function call
20 startBenchmark("hash");
21 hash = newHash(salt);
22 stopBenchmark("hash");
23
24 //message sending (non-blocking)
25 send(2,hash,"hash_value_from"+id());
26 }
27
28 //message receiving (blocking)
29 2: readInt("hash_value_from_1");
30
31 if (hash%2 == 0)
32 output("odd hash: "+hash);
33
34 //for-loop
35 for (int32 i=0; i<200; i=i+1) <
36 //parallel execution
37 ...
38 >
39
40 //while-loop
41 while (condition) {
42 ...
43 }
```

Listing 2.1: L1 sample

The rest of this section focuses on the language constructs of L1.

2.7.1 Variables

L1 supports variables of various basic data type. These are

- *bool*: This is a signed integer of one bit length representing the boolean keywords true and false as known from Java and many other programming languages.
- *int32*: This is a signed integer of 32 bit length as known from Java and many other programming languages.
- *int*: This is a variable bit length integer to address the needs for multi precision arithmetic. This data type is implemented using the BigInteger library of Java.
- *string*: This is a built-in basic data type behaving like Java string objects, i.e. a variable length array of characters.
- *prvk*: This is assigned to variables containing private keys of public key encryption schemes. Introducing its own basic data type enables the use of operators on that data type, e.g. for derivation of public keys. Furthermore it enables the built-in functions for encryption and decryption to determine the corresponding encryption scheme. The compilers translate variables *prvk* into our own class PrivateKey which is accessible to and extensible by built-in and library functions. It is therefore easy to extend the compiler with new encryption schemes, not yet implemented in L1, that inherit from this class. The additional classes simply need to be integrated into the L1 library and are then readily accessible from the L1 language.

- *pubk*: This is assigned to variables containing the public keys of public key encryption schemes. Clearly it contains only the public part of the cryptographic key necessary for encryption. The compiler translates it into our class PublicKey which in case of extensions also needs to be implemented.

Furthermore, L1 also supports the following composites of a basic datatype:

- *scalar*: single value
- *1-dimensional array*: vector
- *2-dimensional array*: matrix

2.7.2 Expressions

Expressions in L1 consist of operands which may be connected through operators. This is similar to almost all other imperative programming languages, operands can be function calls, variables, or constants. In case of a type mismatch between operands, we implicitly upcast *int32* to *int* and any data type to *string*. For other cases L1 provides built-in functions for explicit type conversion. Most operators and their handling correspond to Java and are directly translated into their Java counterpart by the compiler.

2.7.3 Control Flow

The basic control flow constructs in L1 are quite standard for an imperative programming language and closely adhere to Java. L1 provides the basic constructs *if-else*, *for*, and *while* in combination with basic blocks. The compiler translates them directly into their Java counterpart. One notable difference occurs in case of the index variables in *for* loops. In L1, these variables are pass-by-value. In contrast, these variables are pass-by-reference in Java. This difference is necessary in case the loop spawns multiple threads which all access the index variable. We describe how to spawn multiple threads next.

Parallel Execution

The language L1 offers basic blocks and sequential statements like the Java semantics. In addition, it also offers parallel execution of basic blocks. This is motivated by the fact that SC protocols can be very demanding in terms of computational power [KDSB09]. This challenges the performance of a single CPU. Inspired by previous work [BCD+09a, DK09] which showed that SC protocols can be quite efficiently parallelized capitalizing on the trend to multi-core CPUs, L1 includes the option of parallel execution as we expect that future implementations will need to heavily exploit parallelism since most large-scale SC problems have not yet been tackled due to performance concerns.

L1 offers a unique feature for the definition of parallel code sections. Basic blocks to be execute as a new thread are specified by angle brackets as delimiters (instead of curly brackets). The compiler inserts the necessary instructions into the Java code in order to spawn and execute a new thread containing this basic block. Afterwards, the execution continues and any adjacent parallel basic blocks are spawned and run in parallel threads. L1 will also synchronize those threads before returning to sequential processing.

All parallel threads register with a barrier before executing the L1 basic block. A barrier is a synchronization mechanism that blocks execution until all registered threads have finished. The compiler uses the Java standard library class for barriers. After spawning parallel threads the L1 compiler inserts a call to the barrier to wait until all threads have finished before executing the next sequential statement.

Listing 2.2 shows an example for thread synchronization in L1. Line 1 contains an initial sequential statement that outputs "S1". The body of the *for* loop is a parallel basic block which will spawn and run two parallel threads. One outputs "P1" and one "P2". The last statement is a sequential one again and outputs "S2".

The barrier mechanism of L1 synchronizes the threads, such that the last statement will always be executed last, i.e. the first and last output of the program will always be "S1" and "S2", respectively. The only two possible traces of the program are: S1, P1, P2, S2 and S1, P2, P1, S2.

```
1 output("S1");
2 for (int32 i=1; i<=2; i=i+1)
3 <
4   output("P"+i);
5 >
6 output("S2");
```

Listing 2.2: L1 Parallel Execution

We show the code generated by the L1 compiler in Listing 2.3. First, a new barrier (class *RegistrationBarrier*) is created. Second, an instance of class *ParallelStep* is created in a for loop wrapping the parallel basic block. The barrier receives a call to method *registerThread* passing the instance of the parallel basic block. The barrier increments its counter and then returns to the *ParallelStep* instance calling its method *registeredAtBarrier* signaling successful completion of the registration. In this method a new thread for the parallel basic block is created and started which, after the basic block finished, calls the method *reachedBarrier* of the barrier decreasing its counter.

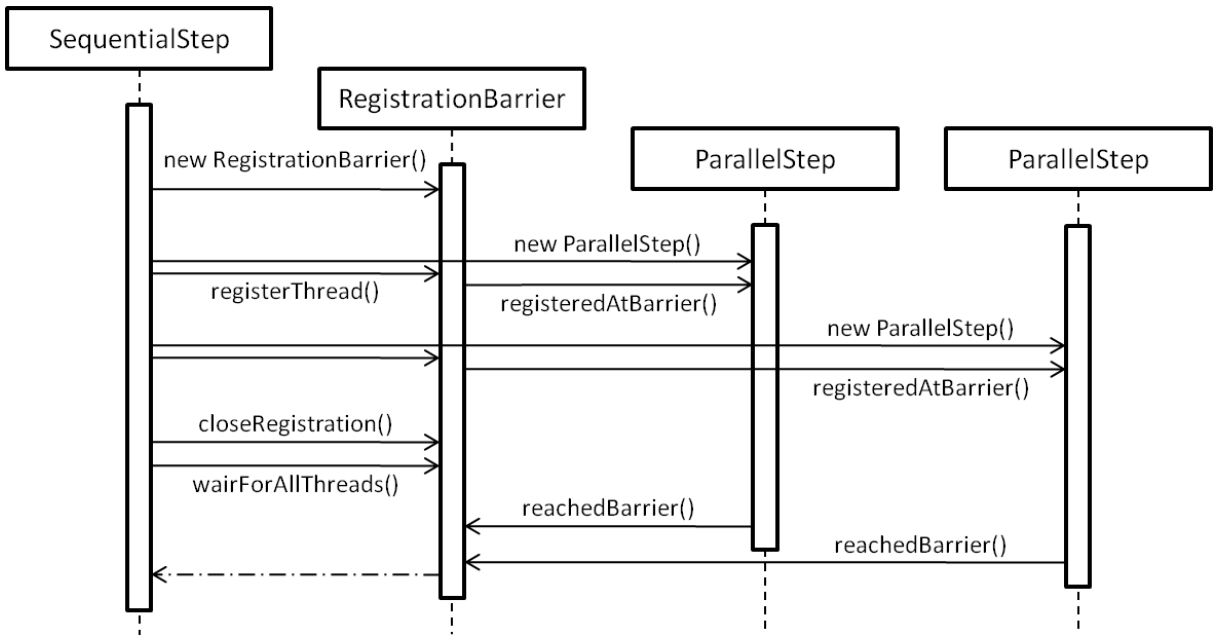


Figure 2.15: Sequence Diagram for Parallel Execution

```
1  public void step() {
2    BuiltInFunctions.output("S1");
3    RegistrationBarrier barrier =
4      new RegistrationBarrier();
5    for (i = 0;i < 2;i=i+1) {
6      barrier.registerThread(
7      new ParallelStep(this, stepThis) {
8        public Integer sum;
9        public Integer j;
10       public Integer i;
11
12       public void init() {
13         sum = DefaultValue.newInteger();
14         j = DefaultValue.newInteger();
15
16         //copy instead of reference
17         i = (Integer) parent.getClass()
18           .getField("i").get(parent);
19       }
20
21       public void step() {
22         for (j = 0;j < BuiltInFunctions
23                .pow(1000, i);j=j+1) {
24           //body
25           sum = sum + 1;
26         }
27         BuiltInFunctions
28           .output("P" + i + ":" + sum);
29       }
30     });
31   }
32   barrier.closeRegistration();
33   barrier.waitForAllThreads();
34
35   BuiltInFunctions.output("S2");
36 }
```

Listing 2.3: L1 sample

The main thread of the sequential program first calls method *closeRegistration* and then waits at the barrier using method *waitForAllThreads*. This method returns when the counter of the barrier is decremented to zero and the sequential program may continue. Figure 2.15 shows the swim lanes diagram for this interaction.

Player-Specific Code

Focusing on SC protocols and especially on multi-party SC protocols, many of these protocols require that all involved players execute the same code but with different data as input. To tackle this requirement, the L1 compiler provides several instances of the Java code – one for each player.

However, some SC protocol work differently. Consider Yao's two-party SC [Yao82, Yao86]. This protocol deviates from the described pattern and requires that different code is executed for different players. Since the player identifier is accessible within L1, the differentiation could be performed at run-time using an *if* statement. Instead we chose for performance reasons to

differentiate at compile time and potentially produce different Java code for each player. A programmer can specify player-specific code sections in the L1 source. A player-specific code section is a statement or basic block prepended by the identifier of the player to execute this code followed by colon. Line 18 of Listing 2.1 shows an example of a player-specific code section.

The interpretation at compile time ensures leaner code at each player that only needs to execute the statements for this player. Furthermore, we feel that it makes the L1 code easier to read and maintain.

2.7.4 Functions

The L1 language uses functions to structure its code. This is also similar to procedural languages. L1 differentiates between two types of functions: The *user-defined functions* are specified and compiled comparable to the C language. Before its first invocation, L1 requires the definition of each user-defined function. We show an example of a function definition in L1 in line 5 of Listing 2.1. *Built-in functions* are programmed in Java and not L1, but can be called from L1 just like any other function. Built-in functions are defined in a Java class BuiltInFunctions of the compiler. The compiler uses reflection in order to import the built-in functions. Using built-in functions the compiler can be extended with new features. Listing 2.4 shows a built-in function for computing the inverse in a field.

```
1 public static BigInteger inv(
2   BigInteger value, BigInteger modulus)
3 {
4   return value.modInverse(modulus);
5 }
```

Listing 2.4: L1 built-in function

Built-in functions can be polymorphic with respect to the parameter type. If the parameter type is `Object`, the compiler creates a function instance for each L1 data type including composites for arrays and matrices.

Many features of the L1 language have been implemented as built-in functions. Two which are particularly worth mentioning are

- messaging
- benchmarking

Messaging Messaging allows the transmission of messages between players enabling the distributed (secure) computation. L1 provides two sub-systems both based on TCP/IP: synchronous and asynchronous.

The built-in functions send and sendSync send messages to other players (line 25 of Listing 2.1). Their parameters are the identifier of the receiving player, a name for the message and its value. If the identifier of the player is 0, the player will broadcast to all other players. The name of the message is a replacement of its address and used by the recipient to retrieve the message in case of asynchronous communication.

The asynchronous send function implements non-blocking behavior (i.e., the next statement in line will be executed immediately). The synchronous sendSync will block the execution

until the message has been acknowledged by all recipients. Synchronous send also supports an optional timeout parameter. Furthermore, L1 also supports buffered sends which bundle several send invocations.

The recipient has a built-in receive read function for every data type. These functions require the message name as a parameter (line 29 of Listing 2.1). Message receiving is always blocking, i.e., the read function will block and wait until the message with the specified name has been received. An optional timeout can be specified as a second parameter or else a default timeout is used.

Benchmarking The design goal of L1 is programming faster SC protocols. Measuring the performance improvement therefore enables verifying whether this goal has been reached.

L1 provides a benchmarking sub-system using built-in functions. Several benchmarks can be measured in parallel. Each benchmark is started by calling the built-in function *startBenchmark* (line 20 of Listing 2.1) and stopped by calling *stopBenchmark* (line 22 of Listing 2.1). Its parameter is the name for this benchmark called a benchmarking section. L1 implicitly takes care of multiple threads by internally appending the thread identifier to the name.

In a benchmarking section the following quantities are captured

- run time (wall clock time)
- number of messages sent or received
- number of bytes sent or received

These correspond to computation and communication complexity in theoretic papers on SC.

2.7.5 Modules

For larger programms written in L1, it might not be sufficient to use only functions to structure the code. Therefore, the L1 language provides modules. Each module is stored as a separate file and can be loaded using the include statement (line 2 of Listing 2.1). This approach is useful to analyze complex SC protocols which are composed of several sub-protocols (e.g. see [Tof07]). Modules allow easily swapping sub-protocols for different implementations and then benchmarking the composed functionality may reveal novel dependencies and side-effects.

Chapter 3

Verification

3.1 Overview

This chapter is dedicated to software development languages and tools that address the problem of producing formally verified implementations of cryptographic algorithms and protocols. Our discussion does not cover general-purpose software verification frameworks, for two reasons. On one hand, covering such a broad range of knowledge would necessarily render this chapter a much less informative enumeration of pointers to tools and techniques. On the other hand, such tools typically address only functional correctness and/or safety properties, which can be seen as a well studied subset of the security requirements we must address in PRACTICE. Instead, we consider domain-specific languages and verification tools for cryptography. We give more emphasis to languages and tools which have been developed and used by project PARTNERS, namely the CAO domain specific language and supporting tools, as well as EasyCrypt and CertiCrypt. Nevertheless, we also provide an overview of third-party developments that may have an impact in the activities of the project by looking at tools that have been recently applied to the verification of secure multi-party computation protocols or their underlying components, such as CryptoVerif, ProVerif and the toolbox developed by David Nowak (which we call the DN Toolbox).

3.2 CAO Language and Tool-Chain

CAO [BMP+12] is a domain specific language (DSL) tailored for the implementation of cryptographic software that was initially developed in the Computer Aided Cryptography Engineering (CACE) project[1]. CAO is an imperative language that supports high-level cryptographic concepts as first-class features, allowing the programmer to focus on implementation aspects that are critical for security and efficiency.

CAO has call-by-value semantics and does not provide any language construct to dynamically allocate memory nor input/output support, as it is targeted at implementing the core components of cryptographic libraries. The native types and operators in the language are highly expressive. The CAO type system includes a set of primitive types: arbitrary precision integers int, bit strings of finite length bits[n] , rings of residue classes modulo an integer mod[n] (intuitively, arithmetic modulo an integer, or a finite field of order n if the modulus is prime) and boolean values bool. Derived types allow the programmer to define more complex abstractions. These include the product construction struct, the generic one-dimensional container vector[n] of T, the algebraic notion of matrix, denoted matrix[i,j] of T, and the construction of an

[1] Work in CAO by the University of Bristol predates the CACE, but the existing tool-chain was designed in this project and later improved and completed by the HASLab group at INESC Porto in the subsequent ENIAC/JU project Secure Memories and Applications Related Technologies (SMART).

extension to a finite field T using a polynomial p(X), denoted mod[T<X>/p(X)]. Algebraic operators are overloaded so that expressions can include integer, ring/finite-field and matrix operations; the natural comparison operators, extended bit-wise operators, boolean operators and a well-defined set of type conversion (cast) operators are also supported. Bit string, vector and matrix access operations are extended with range selection (also known as slicing operations). CAO is strongly typed, and the type system provides a powerful mechanism for implementing templates of cryptographic programs by using *symbolic constants* and a limited form of dependent types. A detailed description of the CAO language, type checking rules and a proof of their soundness can be found in [BMP+12]. Figure 3.1 shows an example of a CAO program corresponding to a (partial) implementation of the AES block cipher.

In addition to a compiler, CAO is supported by two other tools: the CAO interactive interpreter and the CAOVerif tool [BPFV10], a deductive verification tool inspired by the deductive verification feature in the Frama-C platform[2]. We next describe the main characteristics of the CAO compiler and the CAOVerif verification tool since, together, they can be used to generate high-assurance implementations of cryptographic algorithms.

CAO compiler

The CAO compiler is a tool that converts CAO programs into C libraries, i.e., cryptographic components that can then be integrated into more complex software projects. Although at the high-level it appears similar to that of a standard compiler, the architecture of the CAO compiler has been tailored to cater for the widely different scenarios for which cryptographic code may need to be produced, with two main design goals: i. to create a compilation tool that is flexible and configurable enough to permit targeting a wide range of computing platforms, from powerful servers to embedded microcontrollers; and ii. to incorporate, whenever possible, domain-specific transformations and optimizations early on in the compilation process, avoiding platform-specific variants of these transformation stages. One example of this is the generation of indistinguishable operations needed in the deployment of countermeasures against side-channel attacks.

The CAO compiler architecture is logically divided into the classical *front-end*, *middle-end* and *back-end* structure. The front-end parses the input file and produces an abstract representation, or *Abstract Syntax Tree* (AST), which is then checked against the typing rules of the language. This results in an annotated AST which is used in subsequent stages. The most distinctive parts of the compiler are the middle-end and the back-end which we will describe in more detail in the following.

In addition to generating C code, the CAO compiler is also intended to perform meaningful CAO-to-CAO transformations. The middle-end takes the annotated AST and applies a sequence of such transformations towards a CAO format suitable for easy translation to C. The most interesting steps are the following.

Expansion. This optional transformation follows from the fact that most cryptographic algorithms use iterative structures with statically determined bounds. The body of the iteration is unrolled and the loop variables are instantiated.

[2] http://frama-c.com/

```
typedef GF2  := mod[ 2 ];
typedef GF2N := mod[ GF2<X>/X**8 + X**4 + X**3 + X + 1];
typedef GF2V := vector[8] of GF2;
typedef S,K  := matrix[4,4] of GF2N;
typedef Row  := matrix[1,4] of GF2N;
typedef Col  := matrix[4,1] of GF2N;
typedef RowV,ColV := vector[4] of GF2N;

def M : matrix[8,8] of GF2 := { ... };
def C : vector[8]   of GF2 := { ... };
def mix : matrix[4,4] of GF2N := { ... };

def SBox( e : GF2N ) : GF2N {
  def x : GF2N;
  if (e == [0]) { x := [0]; } else {   x := [1] / e; }
  def A : matrix[8,1] of GF2:=(matrix[8,1] of GF2)(GF2V)x;
  def B : GF2V := (GF2V)(M*A);
  return ((GF2N)B) + ((GF2N)C);
}
def SubBytes( s : S ) : S {
  def r : S;
  seq i := 0 to 3
    seq j := 0 to 3 {r[i,j] := SBox( s[i,j] );}
  return r;
}
def SubWord( w : vector[4] of GF2N ) :
  vector[4] of GF2N {
  def r : vector[4] of GF2N;
  seq i := 0 to 3 { r[i] := SBox( w[i] ); }
  return r;
}
def ShiftRows( s : S ) : S
{
  def r : S;
  seq i:= 0 to 3 {r[i,0..3]:=(Row)(((RowV)s[i,0..3])|>i);}
  return r;
}
def MixColumns( s : S ) : S
{
  def r : S;
  seq i := 0 to 3 { r[0..3,i] := mix * s[0..3,i]; }
  return r;
}
def AddRoundKey( s : S, k : K ) : S
{
  def r : S;
  seq i := 0 to 3
    seq j := 0 to 3 { r[i,j] := s[i,j] + k[i,j]; }
  return r;
}
def FullRound( s : S, k : K ) : S
{
    return MixColumns( ShiftRows( SubBytes(s) ) ) + k;
}
def Aes( s : S, keys : vector[11] of K) : S
{
  seq i := 1 to 9 { s := FullRound( s,keys[i] ); }
  return ShiftRows( SubBytes(s) ) + keys[10];
}
```

Figure 3.1: AES implemented in CAO

Evaluation. This transformation evaluates the statically computable expressions, possibly instantiated in the previous step. Operator properties such as idempotence and cancellation are also used to simplify expressions.

Simplification. This transformation is in charge of reducing the mismatch between CAO and C. Compilers that generate assembly code traditionally use an intermediate representation known as three-address code, in which every instruction is in its simpler form with two operand addresses and one result address. The intermediate format used in the CAO compiler shares some of the same principles and it is consistent with the syntax adopted in the construction of the supporting static libraries.

Optimization. At this stage, the *Control Flow Graph* (CFG) of the CAO code is inferred and transformations to and from *Static Single Assignment* (SSA) form are implemented. The compiler's internal API provides a set of functions to manipulate the CFG (and CFG in SSA form), to ease the task of implementing (domain-specific) optimization passes.

Side-channel countermeasures. The CAO compiler incorporates a popular software countermeasure against side-channel attacks [BP05]. The compiler ensures that the code generated for two potentially vulnerable functions (specified by the user) is indistinguishable: both functions execute the same sequence of native CAO operations. To this end, it reorders instructions and, if necessary, introduces dummy operations. The resulting code is kept as efficient as possible by heuristic optimization. This is done after the optimization stage, since this could break the security-critical protection. We note that such countermeasures do not guarantee security against side-channel attacks, but are commonly used to increase the resilience of implementations.

Targeting a language like C poses different challenges than translating code to assembly. One of the reasons for this is that the design space is much larger and the C code can be compiled to very disparate platforms. The CAO compiler tackles this problem using a two-layer approach: the CAO code is translated to a specific C format, which is then linked with a static library where the semantics of the CAO operations is implemented and the data types are defined. This allows adjusting the C data type definitions and the implementation of the operations to the characteristics of the target platform. For each target platform, the back-end takes a configuration file that describes the specific implementation choices adopted for the static library and generates the C code accordingly with the definitions. For example, in the case of variables of a given type use explicit allocation, the compiler will know to call a memory allocation routine. Similarly, if operations over a given type take parameters by reference, then the code generator will make sure the routine receives a pointer to the input parameter.

An important point is that the target platform specification also declares which operations are defined in the static library allowing for incomplete implementations. Therefore, the compilation may fail with an error when the translation is not possible because an operation or data type is not supported. Currently, the CAO compiler offers a generic back-end that targets any computational platform where the well known GNU Multi Precision library[3] (GMP) and Shoup's Number Theory Library[4] (NTL) can be compiled. Another back-end is being

[3] http://gmplib.org/
[4] http://www.shoup.net/ntl/

developed to target ARM platforms via the CompCert verified compiler[5].

CAOVerif

Experience shows [ABPV09, ABPV10] that a tool such as Frama-C has a great potential for verifying a wide variety of security-relevant properties in cryptographic software implementations. However, it is well-known that the intrinsic characteristics of the C language make it a hard target for formal verification, particularly when the goal is to maximize automation. This problem is amplified when the verification target is in the domain of cryptography, because implementations typically explore language constructions that are little used in other application areas, including bit-wise operations, unorthodox control-flow (loop unrolling, single-iteration loops, break statements, etc.), intensive use of macros, etc. The idea behind the construction of the CAO verification tool was to take advantage of the characteristics of this programming language to construct a domain-specific verification tool, allowing for the same generic verification techniques that can be applied over C implementations, simplifying the verification of security-relevant properties, and hopefully providing a higher degree of automation.

The CAO deductive verification tool was inspired by previous work done in languages such as Java and C. These verification platforms, which can be used to prove complex program properties, are usually based on variations of *Hoare logic* [Hoa69, Flo67], whose axioms and inference rules capture the semantics of imperative programming languages. These verification platforms are usually structured around the following components:

Annotation Language Usually the specification (properties that must be proven) are included in the source code by means of program annotations, together with additional annotation intended to facilitate the proof process.

Verification condition generator (VCGen) This is a component that takes the annotated program and generates a set of proof obligations (also known as verification conditions). The validity of these proof obligations will imply that the software is indeed correct with respect to the specification – a consequence of the correctness of the VCGen.

Proof generation Proof obligations are essentially formulas of first-order logic, so that a first-order proof tool (an automatic prover such as Z3 [dMB08], or a proof assistant such as Coq [The11]) is required to construct the proof that the formulas are valid.

The CAO verification tool builds on Frama-C [BFM+08], an extensible framework where static analysis of C programs is provided by a series of plug-ins. Jessie [MM10] is a plug-in that can be used for deductive verification of C programs. Broadly speaking, Jessie performs the translation between an annotated C program and the input language for the Why tool. Why is a VCGen, which then produces a set of proof obligations that can be discharged using a multitude of proof tools. The gwhy graphical front-end, allows monitoring individual verification conditions. This is particularly useful when used in combination with the possibility of exporting the conditions to various proof tools, allows users to first try discharging conditions with one or more automatic provers, leaving the harder conditions to be studied with the help of an interactive proof assistant.

The annotation language (CAO-SL) that is supported by the CAO verification tool is rich

[5] http://compcert.inria.fr/

```
/*@
  @ requires: 0 < c < n;
  @ ensures:  0 < \result < n;
  @*/
def RSAInvShort(k : RSAPrivShort, c : int ) : int
{
      def m : mod[n];

      m := (mod[n]) c;

      m := m ** k.d ;

      return (int) m;
}
```

Figure 3.2: Example of a Simple CAO Program with Contract Annotations

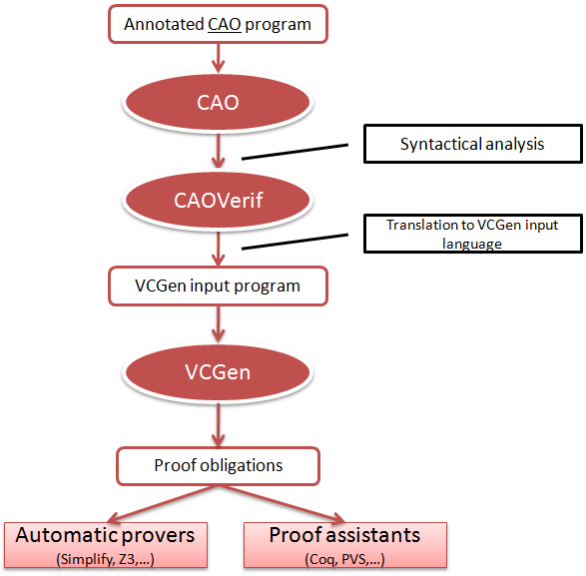


Figure 3.3: CAO Deductive Verification Tool Architecture

enough to formalize arbitrary functional properties of CAO programs. Annotations written in CAO-SL are embedded in comments (so that they are ignored by the CAO compiler) using a special format that is recognised by the verification tool. CAO-SL is strongly inspired by ACSL [BFM+08].

The expressions used in annotations are called *logical expressions* and they correspond to CAO expressions with additional constructs. The semantics of the logical expressions is based on first-order logic. CAO-SL includes the definition of *function contracts* with pre- and postconditions, statement annotations such as assertions and loop variants and invariants, and other annotations commonly used in specification languages. CAO-SL also allows for the declaration of new logic types and functions, as well as predicates and lemmas. A complete description of CAO-SL can be found in [Bar09].

Consider the example of a CAO program shown in Figure 3.2. The developer is allowed to annotate the CAO code with pre-condition and post-condition annotations, i.e., a contract on the function behaviour. The contract consists of a pre-condition (requires) that indicates which conditions should be met by the caller, and a post condition (ensures) that states what is guaranteed after the execution of the function terminates.

The architecture of the CAO deductive verification tool is shown in figure 3.3.

– An annotated CAO program (which can be processed without change by the CAO compiler, since annotations are included in the code as comments) is first checked for syntactic errors and is then translated into the input language of a generic VCGen, using CAOVerif.

– This VCGen is constructed from an existing tool, the Jessie plug-in in the Frama-C framework, with extensions that cover the CAO primitive types and memory model. The proof obligations are generated by running the VCGen on the CAOVerif output.

– The proof obligations can then be checked by some existent automatic prover or proof assistant.

One advantage of this modular architecture allows the enrichment of the annotation language without the necessity of changing the VCGen input language, and vice-versa. Conversely, the VCGen mechanism can be changed without modifying the specification language (in particular it can be interfaced with additional proof assistants and proof tools). Another advantage is that the processing made by the CAOVerif tool can incorporate a memory model translation, into something different from the one inherent to the CAO semantics, which can facilitate the generation/verification of the proof obligations.

3.3 EasyCrypt and CertiCrypt

3.3.1 EasyCrypt

EasyCrypt[6] [BGHZB11] is a tool-assisted framework for verifying the security of cryptographic constructions in the computational model. It is developed by a team that brings together members from IMDEA Software Institute and INRIA Sophia-Antipolis Méditerranée.

Following suggestions by Bellare and Rogaway [BR06] and Halevi [Hal05], *EasyCrypt* adopts a code-based approach, in which cryptographic constructions, security notions and computational assumptions are modelled as probabilistic programs in a core, but extensible, probabilistic programming language with imperative constructs and procedure calls. Procedures can be concrete, in which case they are provided with a body consisting of a command and a return expression, or abstract, in which case only a type signature is provided. The primary purpose of abstract procedures is to model adversaries, but they are also an essential ingredient for compositional reasoning.

Program logics

Reasoning in *EasyCrypt* is supported by two program logics: a probabilistic relational Hoare logic (pRHL), which allows to reason about judgments of the form

$$[c_1 \sim c_2 : \Psi \Longrightarrow \Phi]$$

where c_1 and c_2 are probabilistic programs, and Ψ and Φ are relations on memories, and a probabilistic Hoare logic (pHL), which allows to reason about judgments of the form

$$[c : \Psi \Longrightarrow \Phi] \diamond \delta$$

[6] https://www.easycrypt.info

where c is probabilistic program, Ψ and Φ predicates on memories, δ is a real-valued expression, and $\diamond$ is a comparison operator, i.e. $\leq$, $=$, or $\geq$. The logics are respectively used to establish a formal connection between two games in game-based proofs, and to resolve the probability of an event in a game. As an illustration, consider two programs c_1 and c_2 that are equivalent up to a failure event F in the execution of c_2; the equivalence is formalized in pRHL by the judgment:

$$[c_1 \sim c_2 : \mathsf{true} \Longrightarrow \neg F\langle 2\rangle \Rightarrow \equiv]$$

where $\equiv$ denotes equality of memories, and $\langle 2\rangle$ indicates that the interpretation of F is taken in the output memory of c_2. Moreover, assume that the probability of F in c_2 is upper bounded by some constant δ; this bound can be formalized in pHL by the judgment:

$$[c_2 : \mathsf{true} \Longrightarrow F] \leq \delta$$

EasyCrypt also provides rules to convert pRHL and pHL judgments into probability claims. Using these rules, one obtains that for every event E,

$$\Pr[c_1 : E] - \Pr[c_2 : E] \leq \delta$$

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

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

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

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

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

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

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

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

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

The key to compositional reasoning in *EasyCrypt* relies in its ability to quantify (either universally or existentially) over modules in its ambient logic.

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

Theories might also contain modules, allowing the definition of libraries of standard games depending on abstract algebraic and data structures.

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

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

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

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

clone ElGamalT as InstantiatedElGamalT
  with CG.t <- $\mathbb{Z}^*_5$, CG.g <- $2$, CG.( * ) x y = $x * y$ mod $5$, proving * by smt.
```

Code generation

EasyCrypt includes an extraction mechanism that generates *OCaml* code from functional programs written in *EasyCrypt*, allowing the production of correct-by-construction implementations from an *EasyCrypt* proof. The nature of the source and target languages being

close, the extraction mechanism is simple enough that one can have a high confidence in the generated code.

The presence of abstract types/operators does not prevent the use of the extraction mechanism. Indeed, when encountering an abstract library, stubs for its operators, that have to be filled manually, are generated.

For instance, the *EasyCrypt* distribution provides *OCaml* implementations for most of the algebraic and data structures that have been abstractly formalized in the core libraries, such as fixed-length bitstrings or functional arrays.

Security Claims

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

Verifying a secure computation protocol in *EasyCrypt*

An ongoing collaboration between PRACTICE partners INESC Porto and the *EasyCrypt* development team is currently addressing the ambitious goal of formally verifying an implementation-level formalization of Yao's garbled circuit-based protocol for secure function evaluation.

Various challenges were encountered in this process that will be revenant during the PRACTICE project, namely:

- Secure computation protocols are high-level cryptographic primitives, with complex security proofs. These proofs are typically structured in a layered way, by introducing abstractions and security notions for intermediate cryptographic primitives (in this case, a *garbled scheme* and an *oblivious transfer* protocol), which can themselves require complex security proofs.

- The security definitions used in secure computation protocols are simulation-based and can comprise a quite complex set of rules when describing a *real world* attack scenario and an *ideal world* that must be matched by the protocol under consideration. The module system in *EasyCrypt* is critical to being able to manage the complexity of such specifications.

- Secure computation protocols are often *higher-order* in nature, in the sense that their operation is parametrized by the description of a function to be computed (typically in circuit form). This means that reasoning about the security of the protocol invariably implies considering, e.g., an arbitrary number of gates, and applying proof techniques

that permit composing an arbitrary number of elementary proof steps, e.g., using a hybrid argument.

These challenges placed Yao's protocol out of reach of previous versions of *EasyCrypt*. However the new features that were added to the tool, namely the module system and theory systems, have brought this use case well within the capabilities of the tool. For example, it is now possible to define simulation-based security for a generic encryption primitive in a very simple way, as shown below.

```
module Game_SIM(R:Rand_t, SIM:Sim_SIM_t, ADV:Adv_SIM_t) = {
   fun main(): bool = {
     var query:query_SIM;
     var c:cipher;
     var real, adv:bool;
     var r:rand;

     query = ADV.gen_query();
     real = ${0,1};

     if (!queryValid_SIM query)
        adv = $Dbool.dbool;
     else
     {
       if (real)
       {
         r = R.gen(randfeed query);
         c = enc query r;
       }
       else c = SIM.sim(leak query);
       adv = ADV.get_challenge(c);
     }
     return (real = adv);
   }
 }.
```

The game is parametrized by a real-world adversary and an ideal world simulator; the adversary is either fed a real ciphertext or a simulated one, and it is required to distinguish which in which world it is being executed. This high-level definition can be instantiated with arbitrary encryption-like primitives and, in particular, we have been able to formally verify a proof of security for an implementation of Yao's garbled circuits under this definition of security.

3.3.2 CertiCrypt

CertiCrypt [BGB09] is the *older brother* of *EasyCrypt*. Similarly to the latter, it aims to capture the full generality of code-based game hopping proofs, as initially proposed in [BR06]. However, it takes a more fundamental approach to the problem by rigorously formalizing (in Coq) the semantics of the programming language that is used to express the probabilistic imperative and polynomial-time programs employed in the proofs. Furthermore, this formalization must allow proving that original and transformed programs are equivalent under an appropriate notion of observational equivalence. The *CertiCrypt* architecture incorporates several layers [Bar10]:

- Formalization of probabilistic programs: probability library, libraries of arithmetic and semantics of probabilistic programs.
- Formalization of adversarial model: complexity and termination, usage policies (variables and procedures) and well-formed adversaries.
- Formalization of security definitions: tools to reason about probabilistic programs, semantics-preserving program transformations, observational equivalence and relational logic and game-based lemmas (failure events).

- Tools to reason about mathematics.

At the lowest level is the formalization of the language used to express security games. This is called *pWHILE* and it is an imperative programming language with random assignments, structured datatypes, and procedure calls. The formalization of the semantics of the language explicitly reasons about the cost of running programs, classes of efficient algorithms and adversaries, and asymptotic notions of security. Similarly to *EasyCrypt*, *CertiCrypt* also formalizes a (relational[7]) Hoare Logic and a theory of observational equivalence that permit formalizing security games and game transitions. For example, an admissible adversary is often limited to performing a certain number of calls to an oracle. In *CertiCrypt* this is captured by adding a counter to the security experiment and treating this restriction as a post-condition on its execution. The tool also includes a library of (certified) tactics that deal with many transformations used in code-based proofs.

The general character of this formalization is claimed to provide a unified framework to carry out full proofs: one can reason in the same framework about the behavior of games and adversaries, but also about complex side conditions related to probabilities, number theory, etc. This claim is supported by the results already published based on the use of this tool, which address proofs of significant complexity. Furthermore, and this feature is inherited from using the Coq proof assistant, such proofs can be seen as a certificate (a proof object) that can be checked automatically by a small and trustworthy proof checking engine[8]. Finally, *CertiCrypt* is the only tool for which, to the best of our knowledge, the notion of a Trusted Computing Base has been clearly identified: the authors explicitly identify which parts of a proof in their framework need to be *trusted* in order to obtain assurance that the proof is correct. In addition to the Coq proof assistant core, this includes the underlying formalization of probabilities and the definition of the proof goal itself. Of course, the major drawback of this approach is the fact that in order to take full advantage of its potential one needs significant expertise.

The following results have been obtained with *CertiCrypt*. The works in [BGB09, BGZL11] present the formalizations of proofs for encryption schemes ranging from basic ElGamal to the more complex RSA-OAEP construction, including the Random Oracle heuristic. Signature schemes have been addressed in [BBGO09]. Finally, part of the general theory of ZK-PoK protocols has been formalized in *CertiCrypt* [BHB+10].

Verifying implementations of ZK protocols in CertiCrypt

A recent collaboration [ABB+12] between the *CertiCrypt* developers and PRACTICE partners INESC Porto has lead to the development of a formal verification backend, called ZKCrypt, for a compiler that automatically generates implementations of Zero Knowledge (ZK) protocols.

The operation of this tool is described in Figure 3.4. This depicts the inner structure of the verifying compiler that takes high-level proof goals lG to optimized implementations (top), relying on a verified compiler implemented in *Coq*/*CertiCrypt* (center). Here, full lines denote compilation steps and translation over formalization boundary (i.e. the generation of code that can be fed into formal verification tools), dashed lines denote formal verification guarantees. Rectangular boxes denote code in various (intermediate) languages either stored in files or as data structures in memory. Rounded rectangles represent the main theorems that are generated and formally verified by ZKCrypt and which jointly yield the desired formal correctness and security guarantees.

[7] Here the distinction to classic Hoare Logic is that relations between program states must be generalized to deal with probabilistic behaviors.

[8] The same can be said of course for the DN Toolbox discussed later in this chapter.

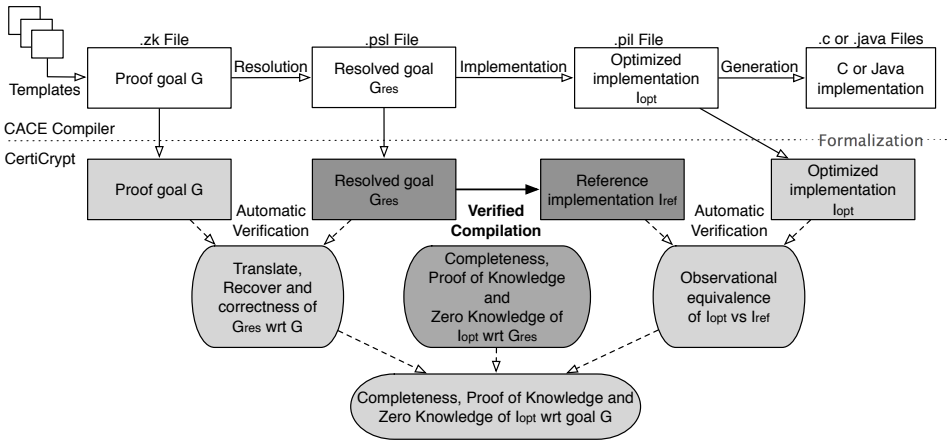


Figure 3.4: ZKCrypt Architecture.

At the top level, ZKCrypt is composed of a chain of compilation components that generates C and Java implementations of ZK-PoKs; these implementations can be turned into executable binaries using general-purpose compilers. These top-level components are an extension of the CACE compiler [ABB+10] with support for user-defined templates and high-level proof goals. At the bottom level, ZKCrypt generates formal proofs in the *CertiCrypt* framework [BGB09]. The compilation component is independent of the verification component.

The main compilation phases in ZKCrypt are the following:

Resolution takes a user-friendly description of proof goal G and outputs an equivalent goal G_{res}, where high-level range restrictions are converted, using standard techniques, into proofs of knowledge of pre-images under homomorphisms; such pre-image proofs are atomic building blocks that correspond to well known concrete instances of ZK-PoK protocols, which can be handled by subsequent compilation phases. The correctness of resolution is captured by a transformation that provably converts ZK-PoK protocols for G_{res} into ZK-PoK protocols for G. The compiler implements both the decomposition and the transformation, and is supported by a proven set of sufficient conditions for correctness and security;

Verified compilation takes a resolved goal G_{res} and outputs a reference implementation I_{ref} in the embedded language of *CertiCrypt*. A once-and-for-all proof of correctness guarantees that this component only produces reference implementations that satisfy the relevant security properties, for all supported input goals. This result hinges a unified treatment of the proof of knowledge property, and a formalization of statistical zero-knowledge;

Implementation takes a resolved goal G_{res} and outputs an optimized implementation I_{opt}. The correctness of this step is established, in the style of verifying compilation, using an equivalence checker proving semantic equivalence between the reference and optimized implementations I_{ref} and I_{opt}.

Generation takes the optimized implementation I_{opt} and produces implementations of the protocol in general-purpose languages. This component in the compiler is the same as that presented in [ABB+10], and is not verified.

Combining the correctness results for each phase yields a proof that the optimized implementation I_{opt} satisfies the security properties of the original high-level goal G. This

approach is the same as verified compilers such as CompCert [Ler06]. As in CompCert, it is convenient to combine certified and certifying compilation, instead of certifying the whole compiler chain.

3.4 Other Tools

3.4.1 DN Toolbox

David Nowak's toolbox [Now07] (DN Toolbox) takes a different approach to the formalization of security proofs. Here the emphasis is *not* on automation, but rather in enabling the writing and checking of actual proofs by cryptographers in a *certified* platform. The toolbox is therefore built on top of the general purpose proof assistant Coq, and great emphasis is put on taming the complexity of the formalization and preserving usability. One important aspect of this approach is that it follows closely the higher level style of proof presentations described in [Sho04], rather than the lower-level code-based style from [BR06]. The justification for this is that it permits avoiding a significant overhead in expressing the semantics of the language in which security experiments are expressed. Instead, the formalization in the DN Toolbox considers games as probability distributions, refining the approach in [Sho04] when extra information is needed to enable mechanical checking.

The framework has as its core the formalization of probabilities and the formalization of the game-based methodology. The probabilistic nature of the framework is formalized with recourse to the definition of a distribution monad, while games are defined as functions returning a distribution. Some automatic tactics to deal with bridging steps are also defined. The framework is supported by the proof of several mathematical results. The model of probabilities and distributions adopted in the DN Toolbox is an important aspect where the formalization is kept lightweight: the tool only considers finite probability distributions implemented as lists of pairs that contain a value and its associated weight (a real number) in the distribution.

Cryptographic games are defined with regards to an event that represents the meaning of breaking the cryptographic scheme. As a result, it is crucial to compute the probability of an event over a distribution. This is achieved by checking for each value in the support of the distribution if the event, modeled as a decidable predicate, is true or not.

The definition of cryptographic schemes involves the use of mathematical structures. The correctness and security of the schemes are dependent on properties of these structures. The framework formalizes such structures, like bit strings and groups, along with its operations and also provides the proof of the corresponding properties. Concerning bit strings, the framework mostly focus on its exclusive-Or operation and the properties that this operation possesses. The group definition is comprised of general properties of groups, such as, associativity or closure, and more specific results like properties of cyclic groups. As expected, the formalization of these results required an extensive development library to support it. To that effect, the development includes several extensions to the standard library by providing lemmas, definitions and tactics that are of general use and can be added to the actual standard library. The most significant extension was the one developed over Coq's lists and Coq's representation of binary integers, while minor additions were done to Coq's library about real numbers, logic, peano arithmetic and mathematical relations.

In this framework, games can be seen as functions that return a distribution. This is used to formalize both the security assumptions, such as Decision Diffie Hellman, and the security notions. For example, the semantic security notion is formalized as a negligible difference

between two games: in the first game the adversary attempts to break the semantic security of the scheme, in which case the boolean output of the game is set to one; and in the second game a random coin is simply sampled and returned. To prove the semantic security of any scheme, the first game needs to be incrementally refined in order to achieve the second trivial game. The change in probability made by these refinement steps must be bounded by a negligible quantity. Because differences between successive games are generally trivial, some automatic tactics are already available that prove these transitions.

The theoretical foundations underlying the toolbox are presented in [Now07, NZ10]. The security proofs for various encryption schemes [Now07, Now08] and pseudo-random generators [ANY09, Now08] have been formalized and checked in this tool-box. More recently, the DN toolbox has been used [ANY12] together with a Coq-based assembly code formal verification framework in order to demonstrate that it is possible to relate security proofs obtained at the specification level with low level implementations.

3.4.2 CryptoVerif

CryptoVerif [BP06] is presented as a tool that brings together the advantages of the symbolic approach and the computational approach to security analysis: it aims to provide a means to automatically construct computational security proofs.

The tool itself if built on top of a formal calculus (inspired by the π-calculus) for probabilistic processes that run in a fixed time t and operate over bit-strings, and which permits formalizing security experiments in a natural way. The probabilistic semantics of CryptoVerif processes allows defining observational equivalence with respect to a concrete measure that captures the distance between the behaviors of a given adversary in two different security experiments. This makes it possible to formalize the notion of a game hop described above. Furthermore, the calculus incorporates arrays as a mechanism to store *views*, i.e. the values taken by variables during process execution. This enables the formalization of various security models and proof techniques such as the Random Oracle heuristic. The first versions of the CryptoVerif tool adopted the asymptotic approach to provable security, but more recent versions allow for concrete security analysis.

The goal of automation is pursued based on the observation that typical game hops involve very simple transformations between adjacent games that can be analyzed at a syntactic level: one can define a set of re-writing rules that permit transforming one game into another one that is observationally equivalent *by construction*. For example, one can define a re-writing rule that states that the generation of a random value at an arbitrary point in the execution of a process, can be moved to an earlier point in the process execution, resulting in an identical behavior. Alternatively, one can formulate a re-writing rule that permits transforming one game into another that will be observationally equivalent under a computational assumption, e.g. under the one-wayness of a specific function. CryptoVerif includes a built-in series of re-writing rules that it may try to automatically apply to the original security experiment, until the adversary's probability of success disappears (this essentially means that the adversary will not be able to influence the output of the final experiment). The prover also has a manual mode, where the user can specify her own re-writing rules: here the tool ensures that the proof is sound, in the sense that the protocol is computationally secure if the underlying computational assumptions hold.

Perhaps the strongest point of the CryptoVerif approach is that it includes the functionality to automatically attempt the creation of sequences of games that constitute valid proofs of security. However, the adopted notion of re-writing rule can be problematic: it is not trivial

to formulate certain transformations used by cryptographers in such a way that CryptoVerif can use it. This means that a hand-made proof argument must be provided to justify the translation of the standard form of some computational assumptions into a syntactic transformation that can be applied by CryptoVerif. Although built on top of a solid theoretical basis, the tool itself is implemented in Ocaml and (to the best of our knowledge) has not been formally verified. This means that the assurance provided by the security analysis must be measured with respect to the trust deposited in the implementation, which is essentially the same state of affairs as that encountered in EasyCrypt.[9].

The first results obtained with the CryptoVerif tool applied to key exchange and authentication protocols [Bla05] and they aimed to demonstrate that computational security proofs (intrinsically computationally sound) could be constructed with the assistance of formal methods. More recent results [BP06, BP10] by the developers of CryptoVerif are achieved by enhancing the tool to allow concrete security proofs and applying it to lower level cryptographic primitives such as signature schemes and key exchange protocols based on assumptions from computational number theory (e.g. Diffie-Hellmann). Efforts to apply CryptoVerif to practical protocols such as Kerberos have also been undertaken [BJST08]. In the final stages of the CACE project, CryptoVerif was evaluated as a possible back-end for a ZK-PoK compiler; the results of this analysis can be found in [Bar11].

3.4.3 ProVerif

ProVerif [Bla01] is an automatic cryptographic verifier implemented in Ocaml that, unlike most existing tool verifiers based on model checking, avoids the problem of the state space explosion without resorting to the limitation of number of runs in the protocol. This is accomplished through a simplistic representation of protocols, and by using an optimized solving algorithm.

In ProVerif, a protocol can be represented by three types of Prolog rules: rules representing the computation abilities of the attacker, facts regarding initial attacker knowledge and rules representing the protocol itself. In order to improve efficiency, additional abstractions are undertaken, such as forgetting the number of times a message appears and replacing that information with a binary value that reflects if the message has ever appeared. Each stage in a protocol can be performed an arbitrary number of times, assuming that the required previous stages have been concluded at least once for the considered principals. This implies that ProVerif does not organize the actions of principals into runs. The solving algorithm can be subdivided in two steps: firstly, the rule base is transformed into a new one, implying the same facts, and afterwards, a depth-first search is conducted, to determine whether a fact can be inferred or not from the rules. This algorithm is similar to an unfolding of the logic program [She92].

Non-termination cases are detected automatically by the verifier. Enforcing termination is possible through limiting the depth of terms. Depth can be defined by the user, such that each term that starts at a depth that exceeds such value is replaced by a new variable. The system remains correct at all times, but precision takes a toll. Optimization is made feasible by: aggregating rules into tuples, automatizing the removal of fruitless rules and hypothesis (when a rule has a conclusion that is already in the hypothesis), and by allowing the user to manually describe facts that will not be derivable. The safety of this last option is reinforced by a final verification after the end of computation, assuring that no wrong conclusions are derivable. Predicate compositions are also available, enabling the possibility to define adversaries with

[9] One can of course check the produced proof by hand.

knowledge from previously compromised sessions.

The system assumes that, if the verifier does not find a flaw in the protocol, then there is no flaw. This implies that the verifier provides real security guarantees, albeit it is possible for false attacks to be presented, resulting from sequences of rule applications that do not correspond to a protocol run. This is a direct consequence to the aforementioned protocol abstraction representation. Attack descriptions are obtainable as the direct sequence of rules that successfully derived the attacker. This notion of secrecy is similar to [AB01, CGG05] and is a weaker assumption than non-interference.

Successful applications of ProVerif include the attack evaluation of protocols such as Needham-Schroeder [BAN89, Low96, NS78, NS87], Denning-Sacco [BAN89, DS81], Otway-Rees [OR87, Pau98], Yahlom [BAN89] and Skeme [Kra96].

Backes et al [BMM10] present an abstraction based on the applied π calculus, along with an automated verification technique for dealing with cryptographic protocols that use multi-party computation as a building block. In theory, it is possible to incorporate this approach into ProVerif's verification technique. The proposed abstraction follows the Universal Composability framework [Can01], associating symbolic abstractions of secure multi-party computations with ideal functionalities. The paper considers static corruptions. This method was experimented in an analysis of the sugar-beet double auction, performed within the SIMAP project [BCD+09b], one of the first large scale applications of secure multi-party computation. More recently, a similar approach was presented in [Bur14].

Secure computation in ProVerif

In 2013, Dreier presented an application of ProVerif various multiparty computation protocols, including Brandt's auction protocol [Dre13]. We present a small example of how this has been done, in order to clarify possible uses of ProVerif within PRACTICE.

In the Brandt protocol, the participating bidders and sellers communicate using a broadcast channel. The protocol uses linear algebra on bit-vector bids to compute a function win_i that returns a vector containing one zero if bidder i was the highest bidder, and random values otherwise. These computations are performed on encrypted bids using homomorphic properties of a distributed n out of n threshold ElGamal encryption. Assuming values $v_1, \ldots, v_n$ (the protocol assumes a finite set of possible prices) and bidders $b_1, \ldots, b_n$, the protocol performs as follows:

1. Each bidder b_i chooses his part of the secret key sks_i, and broadcasts the associated public key pks_i.
2. Each bidder obtains all individual public keys $\{pks_1, \ldots, pks_n\}$ and computes the joint public key pk.
3. Each bidder encrypts its bid $enc(v_i, pk)$ and broadcasts it.
4. Each bidder obtains all encrypted bids $\{enc(v_1, pk), \ldots, enc(v_n, pk)\}$ and computes a partial decryption function (using sks_i), broadcasting the result $res_i \leftarrow win$.
5. Each seller gathers $\{res_1, \ldots, res_n\}$ and decrypts to obtain the winner.

The protocol formalization must encompass the computation of the joint public key (step 3), the bidders' partial decryption function (step 4), and the sellers' process for decrypting results

(step 5). The joint public key is calculated by combining all the individual public keys. This is captured by the following equation:

$$combine(pks_1(sks_1), \ldots, pks_n(sks_n))$$

The partial decryption must receive all encrypted bids and evaluate the maximum, returning a share of the result. The function $tiebreak_i\{v_i\}$ assures that the smallest indexed bidder is the winner, following the original protocol argument. In order to confer some level of indistinguishably, random values $r_1, \ldots, r_n$ are included in the shares. As such, the partial decryption function is captured by:

$$\begin{aligned} &win(enc(v_1, pk, r_1), \ldots, enc(v_n, pk, r_n), sks_i) \\ = &\, share((max_i\{v_i\}, tiebreak_i\{v_i\}), (v_1, \ldots, v_n), pk, sks_i, g(r_1, \ldots, r_n)) \end{aligned}$$

Finally, the decryption of results obtains all shares resulting from win and returns the identity of the winner. This can be captured by the equation:

$$\begin{aligned} &dec(share(m, v_1, pk, sks_1, r_1), \ldots, share(m, v_n, pk, sks_n, r_n)) \\ = &\, m \end{aligned}$$

A ProVerif code that leads to an automatic verification of a given property consists in three major parts: the declarations, the procedure description, and the verification query.

The declaration part may contain types, functions, reductions, equations, etc. A portion of such description is presented in Listing 3.1. This encompasses the protocol verification previously described, namely the `combine` function in line 3, the reduction that allows the sellers to identify the winner `dec` in line 5, and the partial decryption `win` that solves the maximum and returns a share in lines 8 and 9. For this proof, one seller, two different bidders and three different prices were considered, allowing the enumeration of all possible instances on the execution of `win`.

Listing 3.1: Defining functions, reductions and equations

```
1  ...
2  fun win(bitstring, bitstring, shskeys):bitstring.
3  fun combine(shpkeys, shpkeys):shpkey.
4  ...
5  reduc forall m:bitstring, r1:bitstring, r2:bitstring, v:
6        bitstring, k1:shskeys, k2:shskeys; dec(share(m, v,
7        combine(pk(k1), pk(k2)), k1, r1), share(m, v,
8        combine(pk(k1), pk(k2)), k2, r2)) = m.
9  reduc forall x:int; bitToInt(intToBit(x)) = x.
10 ...
11 equation forall pkey:shpkey, r_1:bitstring, r_2:bitstring, skey
12       :shskeys; win(enc(priceToBit(v1), pkey, r_1), enc(
13       priceToBit(v1), pkey, r_2), skey) = share((v1, one),
14       (v1, v1), pkey, skey, f(r_1, r_2)).
15 ...
```

The procedure description executes two bidders and one seller. A single bidder is exemplified in Listing 3.2, and a direct connection can be established between the protocol

execution and the presented code: Step 1) Lines 2-3, Step 2) Lines 5-6, Step 3) Lines 8-11 and Step 4) Lines 13-16. The seller executes by following the code in Listing 3.3, consisting only in receiving shares and compiling the result through the reduction `dec`.

Listing 3.2: Describing the procedure - bidder 1

```
1  let bidder1(v:price, chK:channel, chB:channel, chR:channel) =
2    new shkey:shskeys;
3    out(chK, pk(shkey));
4
5    in(chK, pk2:shpkeys);
6    let pkey = combine(pk(shkey), pk2) in
7
8    new r:bitstring;
9    let bid1 = enc(priceToBit(v), pkey, r) in
10   event bid(v, one);
11   out(chB, bid1);
12
13   in(chB, bid2:bitstring);
14   event recBid(bid2, two);
15   let res = win(bid1, bid2, shkey) in
16   out(chR, res).
```

Listing 3.3: Describing the procedure - seller

```
1  let seller(chR:channel, chW:channel) =
2    in(chR, x:bitstring);
3    in(chR, y:bitstring);
4    let (wbid:price, winner:int) = dec(x, y) in
5    event won(wbid, winner);
6    out(chW, (wbid, winner)).
```

Finally, the verification query describes the security property to validate. For instance, non-repudiation in this protocol implies that, if a bidder b with the value v is concluded to be the winner of the auction, he cannot challenge the validity of such conclusion. This means that the occurrence of an event that describes the winning bid as m with v implies that m bid v, and can be translated into the query:

```
1    query v:price, m:int; event(won(v, m)) ==> event(bid(b, m))
```

ProVerif may now evaluate this protocol implementation for the non-repudiation property. This provides an attack in which an adversary simulates a whole other protocol execution towards the seller, provoking the event `won` without the occurrence of an event `bid`.

From these attack descriptions, the developers of cryptographically secure systems may reach useful conclusions regarding the faulty aspects of their protocols. This specific attack can be interpreted (by humans) as a direct consequence of the protocol lacking authentication, suggesting possible/required improvements.

Chapter 4

Databases

4.1 Sharemind 2 Secure Database

4.1.1 Introduction

For most SHAREMIND applications, it is unreasonable to assume that the data providers are always online to input their data during computations. A method is needed for collecting and storing the data over a longer period of time. This is accomplished by writing the data on a persistent storage device. It also enables processing the data as-needed, in smaller chunks.

One example of an application that would require a database is a survey. In a survey, we collect data in small pieces over a longer period of time. Later, we perform an analysis and generate the results. Alternatively, in an application such as a one-time study, the data could be aggregated at once from existing databases.

4.1.2 Architecture

In the standard deployment setting, SHAREMIND consists of three nodes which perform the multi-party computation protocols. Each of the three nodes maintains a local copy of their databases.

A high-level overview of the database architecture can be seen in Figure 4.1. The database model can be roughly divided into the following parts: the database engine, the transaction service, the SECREC process and the controller application.

The database engine stores the data as tables containing either public or private data. When public data is stored in a database, each of the nodes stores an equivalent copy of the same value. For private data, every node stores their individual share of the private values. The values are accessed through the table identifiers and the appropriate column or row keys. The four basic operations that can be performed on the tables are: delete, insert, read and update.

Applications can have different data usage patterns. A database engine that is suitable for one application may not be appropriate for another. In SHAREMIND both, SQL and NoSQL databases could be used. The database libraries that have been integrated into SHAREMIND are: SQLite 3[1], Tokyo Cabinet[2] and HDF5[3].

The transaction service has the responsibility to synchronize the database operations between the nodes. In an ideal case, all of the database operations should be synchronized - a failure in one node should also be reported in the other nodes. The nodes should also perform appropriate rollback operations on failures to ensure a consistent state for the distributed database. In SHAREMIND 2, only some of the database operations are synchronized. The

[1]SQLite 3 - http://www.sqlite.org/
[2]Tokyo Cabinet - http://fallabs.com/tokyocabinet/
[3]Hierarchical Data Format - http://www.hdfgroup.org/HDF5/

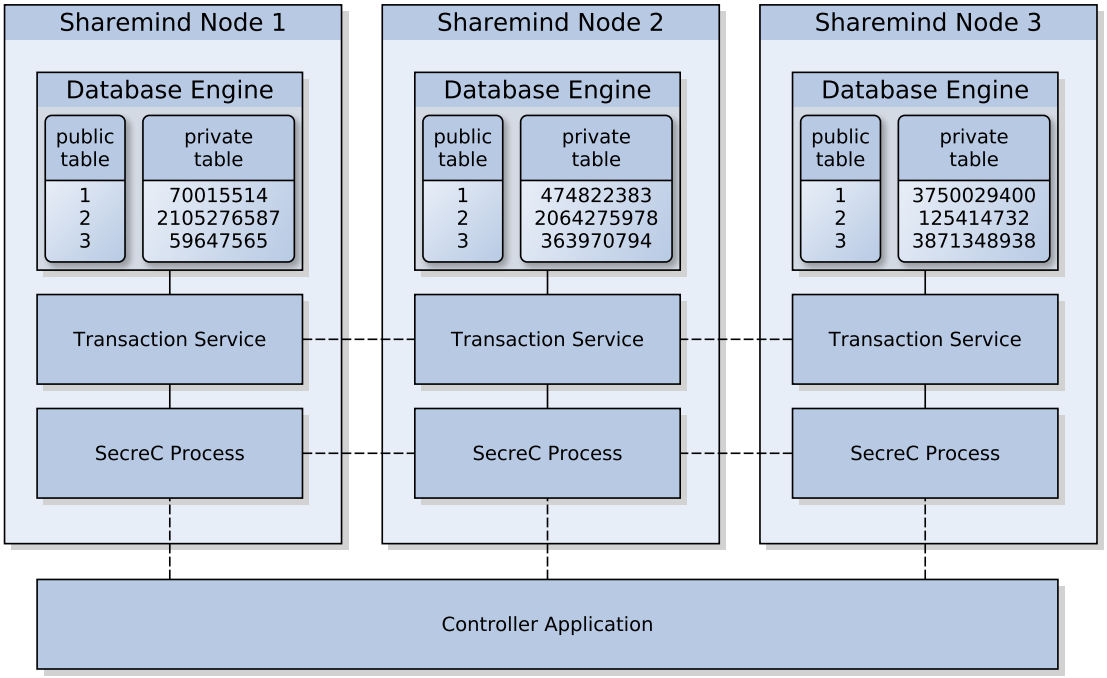


Figure 4.1: High-level Overview of the SHAREMIND 2 Database Model

consistency of the database is ensured by storing additional meta data in the tables. This meta data can be used to restore a consistent state, if required.

The client-side interaction with the SHAREMIND database goes through SECREC procedures. Because most of the data will be stored as secret shared private values, the useful amount of queries that can be performed by the database engine is limited. These queries have to be implemented as SECREC programs, where the private data can be processed using multi-party computation protocols.

The controller applications are responsible for calling the SECREC programs with the right parameters. An example of a SECREC program which creates a public table and inserts some values into it can be seen in Figure 4.2.

4.1.3 Applications

The SHAREMIND 2 database has been used in the following applications.

1. In Bogdanov, Jagomägis & Laur [BJL12], it was used to store the imported data that was later used in evaluating the frequent itemset mining algorithms.

2. In Bogdanov, Talviste & Willemson [BTW12], the database is used to store the financial data summaries sent by members of the Estonian Association of Information Technology and Telecommunications organization.

3. In Kamm, Bogdanov, Laur & Vilo [KBLV13], large database tables of up to 1000×500000 elements were used to store genome data.

```
1  public string tbl; tbl = "name_of_table";
2
3  // Create an empty public table
4  dbTableCreatePublic(tbl);
5
6  // Add columns to the table
7  public uint32[0] empty;
8  dbInsertColumn(tbl, "col0", empty);
9  dbInsertColumn(tbl, "col1", empty);
10 dbInsertColumn(tbl, "col2", empty);
11
12 // Add rows to the table
13 public uint32[3] row;
14 row[0] = 1; row[1] = 2; row[2] = 3;
15 dbInsertRow(tbl, "row0", row);
16 dbInsertRow(tbl, "row1", row);
17 dbInsertRow(tbl, "row2", row);
18
19 /*
20  * |      | col0 | col1 | col2 |
21  * ============================
22  * | row0 |    1 |    2 |    3 |
23  * | row1 |    1 |    2 |    3 |
24  * | row2 |    1 |    2 |    3 |
25  */
```

Figure 4.2: Database Operations in SECREC

4. In the income analysis survey of the Estonian public sector [4], to store the income data.

The database is a part of the SHAREMIND 2 SDK. For more information on the SDK, see Section 6.1.

4.2 Encrypted Query Processing for Business Applications

Deploying the database server of a database backed business application in the cloud provides significant advantages for a company running this business application. Using database-as-a-service, a company need not to invest in hardware as the required capacities can be rented by a cloud provider. This is scalable for the company as it only pays for consumed resources and services.
The purchase of hardware, its maintenance, and the employment of skilled personnel is outsourced to the cloud provider that offers its resources to many companies.
Although this scenario is appealing to the database-as-a-service consumers as well as to the cloud provider, companies are very reluctant to store their data in the cloud and entrust a cloud provider with the hosting.
A contributing aspect to this fact might be the open security issues in the cloud. Gartner names seven weak points which companies have to clarify with a potential vendor [Gar08].

[4] Income Analysis of the Estonian Public Sector - https://sharemind.cyber.ee/clouddemo/

1. Privileged user access: Potentially sensitive data is stored and processed outside the companies trusted boundaries. This is that all physical, logical, and personnel controls are outsourced from the company as a customer to the cloud provider. The cloud provider is responsible to hire very skilled and specialized personnel to maintain its services, but also to check if these employees can be trusted with privileged access to the data of its customers. This poses a severe risk for a company.

2. Regulatory Compliance: Companies have to ensure that a cloud provider undergoes external audits and security certifications on a regular base.

3. Data Location: The companies deploying their data in the cloud do not control where their data is hosted. Moreover, they do not even know where their data is stored and processed. This is a risk in itself but in addition, companies might have the obligation to store and process data in a specific jurisdiction (i.e. a company in the EU is only allowed to store and process sensitive personal data within the boundaries of the EU). A cloud provider must make commitments to obey to these obligations and laws.

4. Data Segregation: As data of all customers are typically stored in a shared environment, the cloud provider has to guarantee that the data set of each customer is properly segregated.

5. Recovery: The cloud provider is responsible to manage potential failures or disasters. This includes to offer data replication and to situate application infrastructure across multiple sites to have the ability to do a complete data restoration.

6. Investigative Support: The cloud provider has to ensure that inappropriate or illegal activity can be investigated properly. This can be hard if logging and data for multiple companies may be co-located and may also be spread across an ever-changing set of hosts and data centers.

7. Long-term Viability: Companies must be able to import their outsourced data in a replacement application in case the service provider goes out of business.

Another severe concern for companies might be the security measures established by a cloud provider. It is the cloud provider's responsibility to select and implement appropriate protection measures to ensure that outside attackers are not able to intrude the database servers in the cloud. Attacks on a database server poses a severe risk to database-backed applications as a successful attack might endanger all data stored on the database. Exploiting a software bug, an outside attacker might gain unauthorized access to the server. Such attacks lead to financial [Mac] as well as reputation [Reu11] losses.
The specific reasons for companies' reluctancy can be manifold, but one significant point is the requirement that they must entrust a cloud provider with their sensitive data. This section is dedicated to the protection of data stored on a database in the cloud. In Subsection 4.2.1, we introduce SAP HANA, a database appliance which can serve as a database back-end in the cloud [Han14]. In Subsection 4.2.2, we discuss encrypted query processing techniques.

4.2.1 SAP HANA

The core of SAP HANA is a database but SAP HANA is more than just a database. It is a so called appliance. This is a combined system of hardware and customized software to run on

this hardware. This appliance is designed to support business scenarios with highly complex analytical processes and transactional operational workloads over typically large data sets. In cooperation with amazon.com, SAP offers the option to deploy SAP HANA in the AWS cloud [Han14]. In the following, we focus on the HANA database as a part of the SAP HANA appliance.
The development of the HANA database was triggered by the key insight that data management requirements for enterprise applications have changed significantly over the years.
Modern enterprise applications have to deal with huge amounts of data. Processing this data does not allow to distinct strictly between a transactional or a analytical pattern. Some applications demand the handling of semi-structured, unstructured, or text data. Others require the efficient representation and processing of graph data. SAP HANA supports these needs with the introduction of a multi-engine processing environment. These engines are:

- a relational engine supporting classical database management functions which is able to process column as well as row oriented relational tables
- a graph engine which can represent and process data graphs
- a text engine allowing text indexing and providing search capabilities

The decision if data is stored on row- or column-wise influences the execution time and depends on the type of application which processes the data. Typically, a table is stored column-wise if

- it contains a large number of columns
- it contains a large number of rows but columnar operations are required (e.g. aggregation, search)
- calculations are executed on single or few columns only
- search functions are processed based on values of a few columns
- most of the columns contain only a few distinct values compared to the number of rows (e.g. only two attribute values -yes and no- allowed). This allows a high data compression rate e.g.a special encoding of the columns.

In contrast, a table is stored row-wise if

- it contains a small number of rows.
- columnar operations are not required.
- the application has to access the complete record.
- the application processes only one single record at a time.
- most of the columns contain distinct values.

SAP HANA exploits recent hardware developments with respect to large amounts of main memory, the number of cores per node, cluster configurations, and storage characteristics like SSD or flash to efficiently increase the performance. It also supports parallel execution and in-memory processing.
These features differentiate SAP HANA from other relational databases [FCP+12].

Name	Surname	Amount
Snow	Jon	314 159 265 358
Stark	Sansa	141 421 356 237
Targaryen	Daenerys	161 803 398 874
Lannister	Tyrion	271 828 182 845

Table 4.1: Excerpt of a simplified customer database table of a bank. This table contains name, surname, and bank account number.

3c 80 06 df c3 bd 0b 30 e3 00 2f f2 77 16 54 75	ff 79 60 c9 fd 5f ef 0c 11 2f 3d 45 b5 c7 e3 29	be 72 68 b2 20 13 c1 92 38 9b e6 59 bc c8 0b f2
85 86 8e d4 2f 4c 9c 71 ad b8 f2 82 4f e5 e8 83	02 e6 06 b9 9a 90 1d 05 3f 42 60 e0 71 b3 3e a1	25 27 af a1 1d b6 d4 a6 16 0d 24 cd 96 4a 29 10
cb 8f d0 4e ca 40 c0 b6 fa 12 b4 b4 48 34 ae 8b	4f 28 9f 60 18 bf 90 8d 25 53 72 96 56 6e 82 3b	65 5c be 60 1b e5 5a 03 4f a5 9f c8 f6 93 5f 54
65 42 67 e4 98 1a 13 99 de ad 44 58 e3 6d 2f 24	f2 e6 4e 83 0b 34 ca 9d d0 1f 23 96 a2 86 a1 8d	c1 f0 b2 32 f1 be 5c 08 d5 47 92 58 1a 9f 5d 47

Table 4.2: Excerpt of the same table now encrypted with AES in ECB mode.

4.2.2 Encrypted Query Processing

Maintaining sensitive data on a server poses a severe risk as 94 % of all data compromises are a result of server side attacks conducted by outside attackers [Ver12]. If an intruder exploits a software vulnerability, she can get access to all data stored on the server.
If data is deployed on a server in an untrusted environment (e.g. the cloud), the data owner might also be afraid of honest-but-curious database administrators or other personnel who has physical access to the server. As they have legitimate access, they can snoop on all stored data. Both kind of attacks can be prevented by encrypting sensitive data: if an attacker accesses the database without knowing the encryption key, she is not able to decrypt and read the data.
Although this approach guarantees the confidentiality of the data, it significantly limits the functionality if the encrypted data cannot be processed. Consider the following example: a bank stores all customer records in a SQL database deployed in the cloud. The customer records contain sensitive information like name, address, bank account number, and amount. Take a look at Table 4.1. This is an excerpt of a customer database which depicts name, surname, and bank account number of four customers. These information are sensitive and should be treated confidential. Therefore, the bank encrypts this table with AES in ECB mode. As this is just an example, we have chosen the string ”abc” as encryption key. The encrypted table is shown in Table 4.2. Consider the case that an employee of the bank needs the bank account number of a certain customer. Therefore, she would like to look up the bank account number in the database. Given that the database is encrypted, there are different approaches to handle this request.
One approach would be to hand over the encryption key to the cloud provider. Then, the cloud provider can decrypt the data and conduct the search. Obviously, the processing of data can be now easily done by the cloud provider, but this reveals all information to the cloud provider and allows a curious database administrator to snoop on data.

Another approach would be to process the query locally on a trusted client. Therefore, the required table has to be downloaded to this trusted client. This prevents the cloud provider from learning the decryption key and therefore from accessing the information, but this is costly in terms of time and computational capacity as the encrypted data has to be downloaded, decrypted, and processed locally. In particular, this approach is not feasible if huge amount of data is involved.
Processing encrypted data without revealing the decryption key to the processing party would protect the confidentiality of data as well as maintain the functionality. Fully homomorphic encryption (FHE) fulfills this requirement as it allows the execution of arbitrary functions over encrypted data. However, the performance of current FHE schemes is still not fully practical [Gen10], [GH11].
Popa et al. introduces an intermediate design to process encrypted data without relying on FHE [PRZB11]. They rely on three key idea to enable the processing over encrypted queries:

1. SQL aware encryption strategies: As the language SQL consists of a well-defined set of primitive functions e.g. equality check or aggregation, one can adapt existing encryption schemes and execute SQL queries over encrypted data.

2. Adjustable query-based encryption: The data is encrypted in so called onion encryption layers where the weakest encryption schemes are the most inner layers which are then encrypted with other encryption schemes. Thereby, not all possible encryption schemes are revealed a priori but only if the query execution requires an onion adjustment to a specific encryption scheme.

3. Chain encryption keys to user passwords: A user's password is rooted to a set of keys necessary to access the different encryption schemes.

This work is further enhanced with optimizations which allow the processing of complex queries as proposed in the TPC-H benchmark [TKMZ13].
Consider again the running example of the bank storing its customer database in the cloud. The encryption and the processing over encrypted data protects the confidentiality of sensitive customer information against snooping database administrators and outside attackers. However, the bank also wants to enforce a strict privacy policy when processing sensitive data. Each employee is responsible for a certain set of customers and only allowed to access these records. The employees are not allowed to access the data of other customers. In contrast, their supervisor is allowed to access all customer data.
Using the encryption strategies described in [PRZB11] and applying additional policy enforcement is not feasible. If the policy enforcement is done on client side e.g. on the client controlled by a user, the user can circumvent the policy enforcement. As the user knows the common decryption key of all data and controls the device which enforces the policy, she can manipulate the device and circumvent the policy.
If the policy enforcement is done on server side e.g. defining and applying authorization views, this can be circumvented if user and database administrator are willing to cooperate. As the user knows the decryption key of all data and the database administrator has access to all data, this collaboration leads to the compromise of those data accessible by the database administrator. In addition, it would also be possible that a erroneous authorization view definition grants access to more data records to a user than intended.
Therefore, the decryption key of data should be only known to users who are allowed to access the data. Handling different keys when processing encrypted data is an open problem.

Chapter 5

Libraries and APIs

5.1 VIFF

5.1.1 Introduction

The Virtual Ideal Functionality Framework (VIFF) is a software library written in Python for implementing multi-party computation protocols [VIF, DGKN09] . The development of VIFF started in 2007, originating out of another research project called SIMAP (short for Secure Information Management and Processing) carried out at the University of Aarhus, in collaboration with the University of Copenhagen and other industrial partners.

The main characteristics of VIFF are the *asynchronous execution* together with the possibility to have *automatic parallel scheduling*, and high degree of *modularity* and *composability*.

VIFF provides asynchronous communication so that players distributed across the network keep on running their portion of code in parallel whenever possible, blocking the execution only when explicitly required. This approach enables rapid prototyping of SMC protocols for realistic applications, when communication occur over the network, since the programmers do not have to cope with protocol rounds, or use explicitly multi-threading and calculate the timing of concurrent operations.

VIFF asynchronous execution relies on a Python framework called Twisted, which makes it possible to create and deal with functions, which can be evaluated only when the corresponding data are available. Functions may return *deferred* results, results that at some time will materialize, and which an event handler is attached to (*callback*), so that the corresponding actions will be executed when the results are available.

VIFF is composed of different modules. The main module is the `viff.runtime`, responsible for sharing the inputs, handling the communication, and running the computations. This module contains the `Runtime` and `Share` classes, whose instances are present at each party site running the protocols, and are used to perform the calculations, and exchange data with the other parties, through `ShareExchanger` objects, as depicted in Figure 5.1. The `viff.field` module contains implementations of finite fields, whose elements can be manipulated through basic operators (addition, multiplication, etc) using overloaded operators.

The runtime object provides the basis for the protocol execution and operates on shares that are asynchronous in the sense that they promise to get a value at some point in the future. Shares overload the basic arithmetic operations so that $x = a + b$ will create a new share x, which will eventually contain the sum of a and b, since the operations simply call back to that runtime. Most operations in VIFF are symmetric, meaning that all parties play an equal role as in the case of binary operations like addition and multiplication where all n parties jointly share x and y and wish to compute a shared representation of $x * y$. VIFF supports also asymmetric operations like secret sharing of input values, where one party provides the input and the other

parties receive the corresponding shares. In symmetric operations, each party will execute the same code and the runtime hides the needed network communication. The execution is done in parallel on the communicating parties machines and a simple line of code like $z = x * y$ is needed to multiply x and y and store the result in z, regardless of the fact that the variables are `Share` objects, and that the multiplication requires several rounds of communication over the network.

5.1.2 VIFF Security and Runtime Modules

VIFF security is based on some assumptions on the numbers and the power of the adversary:

- The adversary can only corrupt up to a certain threshold of the total number of players.
- The adversary is computationally bounded: The protocols implemented in VIFF rely on certain computational hardness assumptions, and therefore only polynomial time adversaries are allowed.
- The adversary can be passive or active, depending on the protocol; passive adversaries can only monitor the network traffic, but they are required to follow the protocol, while active adversaries can deviate from the protocol in arbitrary ways.

VIFF includes a collection of runtimes, each one implementing a certain set of MPC protocols in a given security model:

- **PassiveRuntime** module implements passive secure multiparty computation with threshold $t < n = 2$, being the protocol based on Shamir secret sharings and pseudo-random secret sharing [CDI05].
- **ActiveRuntime** module implements active security with threshold $t < n = 3$, being the protocol based on either hyperinvertible matrices [DGKN09] or pseudo-random secret sharing [CDI05].
- **PaillierRuntime** module implements passive two-party computation based on the homomorphic Paillier cryptosystem [Pai99].
- **OrlandiRuntime** module implements active security and self-trust using the techniques described in [NO09].

5.1.3 Applications

VIFF has been used for several realistic applications. One of the most mentioned application of SMC and VIFF is the Nordic Sugar application, where a double auction took place and the production rights for several thousand tons of sugar-beets were traded. This computation solved a real problem in Denmark, where the production of sugar-beet is managed by sugar-beet contracts, determining the quantity of sugar-beet that a farmer is allowed to produce. Traditionally, sugar-beet contracts are traded between individual pairs of farmers, even it is known that a centralized computation, impossible in the real world for conflicting interests and lack of trust between the parties, would increase the overall profit. In January 2008 the first large scale secure multiparty computation was carried out, in the SIMAP research project [BCD+09b], and in 2009 the same computation was successfully repeated using VIFF and involving three players: Nordic Sugar, the Danish sugar company, DKS, the consolidation of

Danish sugarbeet farmers, and Partisia, a Danish company specialized in secure multiparty solutions.

VIFF has also been used to implement a protocol by Boneh and Franklin for generating RSA keys in a distributed fashion [BF01], ensuring that the private key is never available in the clear to any party and an attacker must break into all machines to learn the private key; the parties can decrypt messages encrypted under the public key, just using their shares of the private key.

Another application of VIFF has been the computation of a secret shared AES encrypted ciphertext of a (possibly) secret shared plaintext with a (possibly) secret shared key. The encryption of 128-bit block using a 128-bit secret shared AES key has been executed using three parties sharing elements over GF(256) (using the `viff.aes` module).

Finally, a distributed voting system has been implemented in VIFF, where the votes are stored in secret shared form among three parties, to avoid storing votes in a unique machine, that could become a single point of failure. The voters compute and encrypt the shares on their own machine, and then send the encrypted shares to a database using the public key of the computation server performing the actual multiparty computation.

5.2 FRESCO

5.2.1 Introduction

FRESCO is a Java framework for efficient secure computation that is being jointly developed by The Alexandra Institute and Aarhus University. The goal of the FRESCO framework is to support the implementation of secure computation applications, and to make it easy to experiment with and compare different approaches to secure computation. To this end the framework is designed to be modular so that various components involved in a secure computation can be replaced and reused. These components include such things as

- Underlying secure computation protocols.
- Circuit construction and evaluation strategies.
- Network communication strategies.

A FRESCO application consists of two main parts: a circuit description of the function to be securely evaluated, and a run-time system that evaluates the circuit according to some underlying protocol for secure computation. Below we describe these two parts in a little more detail.

5.2.2 FRESCO Circuit Description

In FRESCO functions to be securely evaluated are described as circuits. In order to decouple the circuit description from the underlying protocol, the circuits are abstract in that they are not explicitly taken to be e.g. boolean or arithmetic circuits. The framework supplies a library of interfaces for basic circuits, such as circuits computing arithmetic and boolean operations. The application programmer can combine these basic circuits into a generic circuit that computes whatever function she desires. It is then up to the implementer of the run-time system to provide implementations of the circuits for the basic operations.

An interesting feature of the FRESCO framework is the way the circuits are represented: To define a circuit the application programmer implements an interface called a *gate producer*. At run-time a gate producer provides the gates of the corresponding circuit for evaluation by the run-time system. Therefore the circuit does not need to be explicitly stored and read from disk. Instead gates can be generated on-the-fly as they are needed for evaluation. This allows for a very succinct circuit description which is much more space efficient than writing down an explicit circuit. This in turn means that secure computation with FRESCO can scale very well even for huge circuits.

5.2.3 FRESCO Run-Time Systems

Run-time systems in FRESCO specify how circuits are evaluated, and are thus highly dependent on the underlying protocol for secure computation that they support. The run-time system must define the notion of a *gate* used by the protocol and how each gate type is to be evaluated. There is no restriction that a gate must implement specific arithmetic or boolean operations. In fact a gate is simply seen as an unit of computation that requires at most a single round of communication. From the gates it provides a run-time system also provides implementations of (at least a subset of) the basic circuits described above.

Additionally a run-time system may provide a number of strategies for gate evaluation and network communication. Such strategies may control how gates are scheduled for evaluation, whether they are evaluated sequentially or in parallel an many other aspects of the evaluation.

Currently run-time systems written for FRESCO includes support for the following protocols for secure computation

- The TinyOT protocol by Nielsen *et al.* for actively secure two-party computation based on boolean circuits [NNOB12].
- The Bedoza protocol by Bendlin *et al.* for actively secure multi-party computation based on arithmetic circuits [BDOZ11].
- The Spdz protocol by Damgård *et al.* for actively and covertly secure multi-party computation based on arithmetic circuits [DPSZ12, DKL+13].
- The protocol by Gennaro *et al.* for passively secure multi-party computation based on arithmetic circuits [GRR98].
- The protocol by Katz and Malka for passively secure private function evaluation based on boolean circuits [KM11].

We note that the idea of having interchangeable run-time systems is inspired by VIFF described in Section 5.1.

5.2.4 Applications Using FRESCO

As of now only a few application has been written using FRESCO. The most noteworthy application so far is the benchmarking application described in Section **??**.

5.3 SCAPI

5.3.1 Introduction

SCAPI is an open-source general library written in Java aimed to provide a general platform of cryptographic primitives for secure computation implementations [EFLL12]. The implementation has been provided in Java for different reasons: the portability of the language to different platforms and devices, the availability of integrated development tools, the existence of already implemented cryptographic libraries, and the capability of integrating existing applications and libraries written in native code (using the JNI framework).

Indeed, the main design principles of the SCAPI library are flexibility, extendibility, efficiency and ease of use. Flexibility is achieved since the protocols written in SCAPI rely on low level primitives and sub-protocols that can be easily exchanged and replaced, allowing the comparisons on the efficiency of different provided implementations, and the adaptability to different devices, where computational power maybe different. SCAPI library is easily extensible, meaning that new implementations of primitives and sub-protocols can be added and used by already defined protocols. Efficiency is achieved since low-level primitives written in native code can be easily included in the library. Finally, ease of use, means that SCAPI has been explicitly designed to support general secure computation protocols, so that all the tools and the documentation are available to support the users.

A first consequence of SCAPI flexibility and extendibility is that the security level of an implemented primitive can also be differently selected by a given application. This is done by defining a hierarchy of security-level interfaces for the different cryptographic primitives. As an example, an application can be developed relying on a general encryption scheme, and then the security level of the scheme can be chosen to require CPA or CCA security. The mechanism is that the application instantiating the protocol will not be able to pass the constructor primitives that do not provide the necessary security guarantees.

5.3.2 Layers

The SCAPI library is divided into three layers. The first layer consists of low-level primitives (discrete log groups, pseudo-random functions, pseudo-random permutations, hash, universal hash, etc.) providing the wrapping of code coming from different libraries and languages into a unified format, available for the other layers. Interfaces provided by this layer offer different levels of abstractions, allowing the selection, for example, of a general pseudo-random function of an instance, such as AES or an implementation of AES for a given device.

The second layer contains non-interactive schemes (symmetric and asymmetric encryption, MACs, digital signatures). SCAPI provides the implementations of asymmetric encryption schemes such as RSA-OAEP, El-Gamal (over any discrete log group), Cramer-Shoup, some obtained from other libraries (Bouncy Castle and from Crypto++), some implemented from scratch.

Finally, the third layer contains interactive protocols and schemes that are commonly used in secure computation: oblivious transfer protocols, with security in the presence of semi-honest and malicious adversary; commitment schemes including Pedersen, ElGamal, and hash based and equivocal; sigma protocols, including also the possibility to operate AND and OR of multiple statements, and transformation to zero-knowledge; garbled circuits, implementing the basic Yao construction and more optimized ones using the free XOR technique; miscellaneous protocols, including tossing a single bit or a string.

In addition to these three main layers, there is an orthogonal communication layer, enabling the possibility of setting up communication channels and sending messages among two or more players. The channels are generated using the Java built-in mechanism for serializing objects, through the conversion of any object in a stream of bytes containing all the information to recreate the object at destination.

5.4 Sharemind 2

5.4.1 Introduction

SHAREMIND 2 [BLW08, BNTW12, Bog13] is a secure service platform for data collection and analysis. Designed as a distributed secure database and application server, it is capable of collecting, storing and processing confidential data without compromising the privacy of individual records. The main motivators behind its design were the following major goals:

1. *Functionality* - It must be able to securely process confidential data;
2. *Efficiency* - It must be efficient enough to be used in practice;
3. *Usability* - It must be usable by non-cryptographers.

At its core, SHAREMIND 2 uses *secure multiparty computation* technology to achieve the necessary cryptographic security in data storage and computations. More specifically, it is based on the 3-party *additive secret sharing scheme* in the ring of 32-bit integers, i.e., a secret $s \in \mathbb{Z}_{2^{32}}$ is split into three random shares $s_1, s_2, s_3 \in \mathbb{Z}_{2^{32}}$ such that $s_1 + s_2 + s_3 \equiv s \pmod{2^{32}}$. In this particular implementation the computation protocols are provably secure in the *honest-but-curious* security model with no more than one passively corrupted party.

SHAREMIND 2 can be programmed to perform various secure computations, thus enabling the development and execution of custom data processing applications. Its protocol suite is universally composable, allowing the basic secure operations to be composed sequentially to form programs, and in parallel to achieve efficient SIMD (single instruction, multiple data) operations on vectors. SHAREMIND 2 implements a distributed virtual machine that provides the consistent instruction set for accessing secure computational resources, while abstracting away most of the low-level protocol implementation details. The secure computation algorithms can be specified either in the low-level SHAREMIND assembly language interpreted directly by the virtual machine, or in the high-level privacy-aware programming language called SECREC. For detailed information on the languages see Section 2.2.

SHAREMIND 2 is accompanied with a software development kit (see Section 6.1). It includes developer tools such as the SECREC development environment, tools for profiling and debugging, as well as examples and documentation. The SDK is available for download on the official SHAREMIND website. [1]

5.4.2 Deployment Model

SHAREMIND 2 is designed to be deployed as a distributed secure computation service that can be used for privacy-preserving data storage and analysis. In the deployment model (see Figure 5.2), there are three different kinds of actors involved in the interactions:

[1] https://sharemind.cyber.ee

1. ***Input parties*** control the confidential data that *result parties* want to analyze, and use secret-sharing to distribute the data among the *computing parties*.

2. ***Computing parties*** collect the confidential data from *input parties* and perform secure computations on the data by request of the *result parties*.

3. ***Result parties*** request secure computations for data analysis from the *computing parties* and receive aggregated results in return.

In terms of SHAREMIND 2, both the input and the result parties act as service clients and are considered to be the *controllers*, as they control the performed computations by making queries to the computing parties. The computing parties, on the other hand, embody the data processing nodes (also referred to as *miners*) of the SHAREMIND 2 application server, hosting the service and running the business logic of data processing applications on top of secure multi party computation primitives. The number of input and result parties is not limited, but the number of computing parties is defined by the underlying computation protocols.

The controller clients use a special *controller library* to communicate with the SHAREMIND 2 system. The library allows the clients to send queries with public and private data arguments to the computing parties, automatically applying secret-sharing on the private data before sending and after receiving it.

The deployed SHAREMIND 2 system consists of three computing parties, each running an instance of the miner server software. The miners are interconnected with secure communication channels and function as a whole. Given that SHAREMIND 2 is designed by the same principles as a database and application server, it can also have multiple users and multiple databases. Each user is assigned its own separate session and queries are executed within that session. In most scenarios, we want to restrict the clients to accessing only the databases and algorithms that they are allowed to query.

Collecting secure data in a database simplifies the deployment of SHAREMIND 2 as different input parties can upload their data independently from each other and over a longer period of time. Once the data collection is complete, we can start running secure computation algorithms to analyze the data. The database layer used for storing the data is described in Section 4.1.

5.4.3 Computational Capabilities

The sole protocol suite of SHAREMIND 2 covers basic arithmetic and comparison on integers. All operations are designed to be performed pointwise on vectors of inputs. Both unary and binary operations are supported. Table 5.1 gives an overview of the protocols that have been implemented on SHAREMIND 2 and refers to the papers that describe them in more detail. Each protocol can take inputs in private vector form and produces a private scalar or vector value.

The boolean data type is supported by emulating shares in $\mathbb{Z}_2$ on top of $\mathbb{Z}_{2^{32}}$ integers. Logic operations on booleans are composed from the addition and multiplication operations on 1-bit integers. These operations correspond to the *exclusive or* and *conjunction* operators on boolean values.

Operands	Operation	Reference
private $\vec{u} \in \mathbb{Z}_{2^n}$ private $\vec{v} \in \mathbb{Z}_{2^n}$	Addition	[BLW08]
	Multiplication	[BNTW12]
	Equality	[BNTW12]
	Greater-than	[BNTW12]
	Division	[BNTW12]
	Remainder computation	[BNTW12]
private $\vec{u} \in \mathbb{Z}_{2^n}$ public $\vec{v} \in \mathbb{Z}_{2^n}$	Multiplication	[BNTW12]
	Division	[BNTW12]
	Remainder computation	[BNTW12]
private $\vec{u} \in \mathbb{Z}_{2^n}$ public $\vec{v} \in \mathbb{Z}_{2^n}$	Shifting bits left v places	[BNTW12]
	Shifting bits right v places	[BNTW12]
private $\vec{u} \in \mathbb{Z}_2$	Conversion of shares from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$	[BNTW12]
private $\vec{u} \in \mathbb{Z}_{2^n}$	Random element shuffling	[LWZ11]

Table 5.1: The Secure Computation Protocols of SHAREMIND 2.

5.5 Sharemind 3

5.5.1 Introduction

SHAREMIND 3 is a complete redesign and reimplementation of the SHAREMIND 2 secure service platform described in Section 5.4. Despite being capable of efficient secure computations using the *additive secret sharing scheme*, SHAREMIND 2 remained too centered around this technique, lacking the flexibility necessary to support other existing and emerging secure computation techniques. The new architecture of SHAREMIND 3 features a more abstract and modular design, independent from any particular computation paradigm. It also focuses on improving the overall performance and robustness of the system, and addresses several design issues identified in the previous version.

The most notable innovation in the SHAREMIND 3 platform is support for the arbitrary data protection techniques both in storage and computations. This is achieved by introducing the *protection domain* concept. A protection domain protects a set of data stored and processed with the same resources using a certain kind of technology. The particular data protection technology used, including its data representations, algorithms and protocols for storing and computing on protected data define the *protection domain kind*. For example, Shamir's secret sharing and fully homomorphic encryption schemes are protection domain kinds. In a sense, a protection domain can be viewed as a black box that instantiates a certain protection domain kind. There can be any number of protection domains deployed simultaneously, each defined by a *protection domain kind* and its *configuration*. In general, the data stored and processed within the protection domain is inaccessible to anyone outside of it, unless a special *declassification* operation is performed to make the particular data values public. The data can be exchanged between the different protection domains, if there exists a *reclassification* algorithm for conversion of data between the kinds of protection domains involved. Protection domains are also discussed in Section 2.3.1.

The new concept has been integrated throughout the entire programming and execution pipeline of SHAREMIND 3. Protection domains are now fundamental part of the SECREC 2 language, a new version of SECREC designed for programming secure computations on the

SHAREMIND 3 platform and described in more detail in Section 2.3. The SHAREMIND 3 server, on the other hand, implements this concept in runtime. An improved networking layer adds support for any number of participating parties in a deployed system as opposed to three parties in case of SHAREMIND 2, enabling the integration of any centralized or distributed secure computation schemes. The server also incorporates a more general and efficient virtual machine, that executes the bytecode of compiled SECREC 2 applications, supports more data types and any number of operations available to it in form of internal or external *system calls*. Protection domain kinds are loaded from dynamic libraries with a well-defined C interface, and the operations supported by them are linked directly into the virtual machine as external system calls. The loaded protection domain kinds are then instantiated as protection domains according to server's configuration. From this point the SHAREMIND 3 server can handle the client queries in parallel processes and execute the respective secure computation algorithms utilizing the involved protection domains.

The developers of protection domain kinds are given a C-level interface, that they must adhere to when describing and implementing the operations supported by their secure computation scheme on various data types. The interface is general enough not to set any restrictions on the internal implementations of data representations and operations. Hence, the operations may even use specialized hardware, if necessary. Among other things, the interface also provides the means for network communication between the participating parties of the secure computation scheme being implemented.

5.5.2 Deployment Model

From the high-level perspective SHAREMIND 3 continues to follow the database and application server design. Multiple clients query the SHAREMIND 3 system to perform the agreed upon computations and get results in return. Within the application server each client is assigned a dedicated session, used to isolate and sanction the client queries and associated computations. From the low-level detailed perspective, however, the deployment model of SHAREMIND 3 is more general and flexible compared to the previous version described in Section 5.4.2.

As depicted on Figure 5.3, the number of SHAREMIND nodes in the deployed system is no more limited to three, as it was the case in SHAREMIND 2. The system now consists of one or more SHAREMIND nodes, fully connected with authenticated encrypted channels.

Additionally, SHAREMIND 3 performs its computations within the protection domains model, and can operate multiple protection domains simultaneously. Each protection domain configured in the deployed system is somehow represented on all the SHAREMIND nodes of the system. The public protection domain is special in a sense, that it is powered by the public virtual machine always present on all the SHAREMIND nodes and aims at handling the public data and computations without hiding these. The other protection domains, on the other hand, are loaded from external implementations and mainly aim at protecting the data.

Depending on its kind each protection domain requires a number of parties for the underlying computation scheme to work. These parties are responsible for the actual computations performed within the protection domain. In the deployment stage the roles of these parties are distributed among the SHAREMIND nodes according to the configuration of a protection domain, and these nodes are then considered to be the *computing nodes* of that protection domain. The rest of the SHAREMIND nodes not participating in the secure computations of that particular protection domain, act as *proxy nodes* for the protection domain, handling the data declassification operations and proxying the data into the public domain. This is necessary, as the technological and topological differences of various protection domains

cause their declassification algorithms to differ as well, requiring each protection domain to handle such operations separately on all SHAREMIND nodes.

Table 5.2 shows an imaginary example deployment of SHAREMIND 3 with several protection domains combined, one of them being the public protection domain, two of them using different kinds of 3 and 4-party MPC, one using 2-party Yao Garbled Circuits and another two using 1-party fully homomorphic encryption (FHE). All the protection domains are configured with names (e.g. public, pd_*), that can be used by the virtual machine to refer to these domains when executing the bytecode. The computing nodes of respective protection domains are denoted by C_i and are distributed among the five SHAREMIND nodes N_i. The proxy nodes of protection domains are denoted by P.

PDK	PD	N_1	N_2	N_3	N_4	N_5
Public virtual machine	public	C_1	C_2	C_3	C_4	C_5
3p MPC, additive, passive	pd_a3p	P	C_1	C_2	P	C_3
4p MPC, Shamir, active	pd_s4a	C_1	P	C_2	C_3	C_4
2p Yao Garbled Circuits	pd_gc	C_1	P	P	P	C_2
FHE	pd_fhe1	P	P	P	C_1	P
FHE	pd_fhe2	P	C_1	P	P	P

Table 5.2: An Example Setup of SHAREMIND 3 with several Protection Domains combined.

The deployment of the client applications mostly involves configuring the locations of the SHAREMIND nodes, their public keys and the available protection domains. The clients also load the dynamic controller modules (the smaller counterparts of similar modules on the server side) containing the implementations of data classification and declassification functionality for the kinds of protection domains available on the server side. The logic necessary to communicate with SHAREMIND 3 system is handled by the controller library, that loads the configuration and the controller modules, and provides the users an interface for making the queries to the system.

5.5.3 Computational Capabilities

Security and computational capabilities of SHAREMIND 3 largely depend on the loaded protection domains and their kinds. The users can therefore choose which underlying secure computation method suits them best. In the following we describe the protection domain kinds currently implemented for SHAREMIND 3.

- **Public virtual machine** controls the public execution flow and powers the public protection domain in SHAREMIND 3, allowing to store and process data publicly. The VM supports signed and unsigned integers (8, 16, 32 and 64 bit) and floating point values (32 and 64 bit), as well as heap manipulation functionality. The booleans and public strings are simulated types on the SECREC 2 level.

- **additive3pp** is the 3-party MPC protocol suite based on additive secret-sharing in the passive model. It is in fact an extended version of SHAREMIND 2 protocol suite described in Table 5.1. The supported data types include booleans, signed and unsigned integers (8 to 64 bit), floating point values (32 and 64 bit) and xor-shared strings. Table 5.3 summarizes various operation classes implemented on these types.

- **additive2pp** is the 2-party MPC protocol suite based on additive secret-sharing and additively homomorphic Paillier cryptosystem in the passive model. It supports arithmetic on 32-bit integers. [PBS12]
- **additive2pa & additive2pa_sym** are the 2-party MPC protocol suites similar to additive2pp, but achieve active security by protecting the shares with MACs. Both support arithmetic on 32-bit integers. [Pul13]

Operation class	Integers	Floats	Booleans	Strings
Arithmetic	✓	✓		
Bitwise operations			✓	✓
Comparisons	✓	✓	✓	✓
Elementary functions *sqrt, sin, ln, exp, erf, etc.*		✓		
Special functions *min, max, sign, abs, etc.*	✓	✓		
Statistical operations *descriptive, distributions, testing*	✓	✓		
String operations *known & bounded-length strings, AES, CRC*				✓
Type conversions	✓	✓	✓	✓
Table operations *shuffle, sort, oblivious choice & lookup, join*	✓	✓	✓	✓

Table 5.3: Operation Classes Supported by the additive3pp PDK on Various Data types.

5.6 SecreC 2 Standard Library

5.6.1 Introduction

The SECREC 2 standard library is a collection of privacy-preserving data processing primitives for the SECREC 2 programming language [BLR13a, BLR13b] developed by Cybernetica. We described SECREC 2 in Section 2.3.

The standard library consists of a number of modules that can be re-used in SECREC 2 programs. The modules contain algorithms for various purposes, e.g., working with arrays and tables, performing statistical analysis or gaining access to features specific to a protection domain (see Section 2.3.1 for an explanation on protection domains).

5.6.2 The Design Principles of the Standard Library

The SECREC 2 standard library is designed to reduce the development time of applications that make use of secure computation by providing the developer with a set of often-used functions. The standard library provides both generic implementations of algorithms and also

their optimized counterparts that run only on specific protection domains along with helper functions that work on public data.

Using the standard libary will also help developers create more efficient applications. The heuristics for optimizing a SECREC program have subtle differences compared to regular programming languages due to the underlying implementation of secure computation protocols, which benefit greatly from vectorization in terms of efficiency. Thus, the developer can always rely on the standard library to have the most efficient implementation. Furthermore, if a standard library function is replaced with a more efficient version in the future, all applications using that standard library will become more efficient after a recompilation.

The standard library is implemented as a set of SECREC modules that are structured as follows.

1. For each protection domain kind, there is one specific module that declares the protection domain kind itself so that it could be used in the application. This module also defines the primitive operations for that protection domain. All functions in this module are specific to the protection domain and the implementations are polymorphic over that protection domain kind. The implementation also makes heavy use of system calls to leverage the protocol-level optimizations available. For example, `additive3pp` is a module that defines the additive3pp protection domain kind and operations like sum, min, max etc.

2. Some modules contain functions that are generic over many protection domain kinds. These include vector and matrix operations like dot product and matrix multiplication, oblivious lookup and update functions and many more. They are implemented polymorphically over any protection domain (including the public one). These functions are implemented in SECREC and rely on the standard library functions they use to provide efficient protocols for each particular protection domain. For example, the `oblivious` module provides oblivious choice functions that are generic to secure protection domains.

3. Generic secure functions, like the ones mentioned in the previous item, can sometimes be optimized for specific types on specific protection domains. For these cases, we provide a specialized module that overloads the generic protection domain polymorphic version with an optimized version for some special cases. These versions use template specialization to ensure that the function only works in certain protection domain kinds and uses system calls to call the relevant protocols. For example, the `a3p_oblivious` module provides some specific oblivious choice protocols for floating point values.

Each module is also commented in a way that allows the Doxygen tool to automatically generate reference documentation to aid the programmer.

5.6.3 Library modules

Currently, the core library consists of 14 modules. See Table 5.4 for an overview of the modules. The modules are packaged together with the SECREC compiler. For a description of protocols and algorithms available within the core library, see [BNTW12, KW13, BLT13, LTW13, LWZ11].

The statistical analysis modules have been developed in the EU FP7 project UaESMC [2]. These modules can perform a range of descriptive statistical analyses, calculate various

[2] UaESMC–Usable and Efficient Secure Multi-Party Computation, contract no. FP7-284731. `http://www.usable-security.eu/`)

Module name	LoC	Module description
Protection domain modules		
`additive3pp`	2225	Declares the additive3pp protection domain kind and provides primitive functions for it.
`xor3pp`	566	Provides primitive functions on bitwise types in the additive3pp domain.
PDK-polymorphic modules		
`matrix`	1127	Provides domain-polymorphic vector and matrix operations.
`oblivious`	1800	Provides domain-polymorphic oblivious vector and matrix access operations.
`profiling`	35	Provides interfaces to the built-in profiling mechanism in SHAREMIND 3.
`stdlib`	1748	Provides domain-polymorphic primitive operations and comfortability functions for printing vectors and matrices.
PDK-specialized modules		
`a3p_bloom`	197	Implements the Bloom filter on the additive3pp protection domain.
`a3p_matrix`	2815	Overloads some operations in the `matrix` module with versions optimized for the additive3pp protection domain.
`a3p_oblivious`	55	Overloads some operations in the `oblivious` module with versions optimized for the additive3pp protection domain.
`a3p_random`	332	Provides randomness generation, vector and matrix shuffling for the additive3pp protection domain.
`a3p_sort`	284	Provides sorting in the additive3pp protection domain.
`x3p_aes`	276	Provides AES key generation, schedule and encryption in the additive3pp domain on bitwise data types.
`x3p_join`	127	Provides matrix joining in the additive3pp domain on bitwise data types.
`x3p_string`	984	Provides string operations in the additive3pp domain on bitwise data types.

Table 5.4: Core Modules of the SECREC 2 Standard Library, with Numbers of Lines of Code.

aggregations and perform statistical testing. The algorithms in the modules are described in [BKLPV13].

Module name	LoC	Module description
stat_summary	444	Functions for calculating summary statistics, with private filter support.
stat_distribution	164	Functions for evaluating the distribution of data, with private filter support.
stat_testing	392	Functions for performing statistical tests, with private filter support.
stat_common	278	Common sub-functions used by the rest of the library.

Table 5.5: Statistical Modules of the SECREC 2 Standard Library, with Numbers of Lines of Code.

5.6.4 Documentation and Unit Tests

The SECREC standard library is documented inline, using the Doxygen [3] tool. The generated HTML documentation describes all the modules and also teaches the basics of SECREC programming. The general language reference is documented using specific code files, that contain only the documentation. The documentation is also bundled together with the SECREC 2 compiler.

There is a test suite, covering a majority of the standard library with various correctness tests. It runs the privacy-preserving functions with normal and extreme functions and evaluates the correctness of the results. The tests are bundled together with the SECREC 2 standard library.

5.6.5 Practical Use

The SECREC 2 standard library has been used in the following applications.

In [KW13], Kamm et al used SHAREMIND 3 and SECREC 2 to build a satellite collision prediction that keeps the trajectories of satellites confidential. The application uses the following key features of the library: floating point arithmetic, floating point elementary functions, vector operations and matrix operations.

The European project UaESMC is building a privacy-preserving statistical toolkit using the SECREC 2 standard library. One of the first results is the statistical module in the standard library that is based on the core architecture (see Section 5.6.3). Further tools are in development and pilot project is under way that will use these statistical tools for the linking and analysis of state databases in Estonia. The statistical functions use the following key features of the library: floating point arithmetic, floating point elementary functions, oblivious shuffling and oblivious sorting.

Finally, in the PRACTICE project, Cybernetica decided to get some preliminary experience with holding private surveys and conducted an employee satisfaction study using both the core library and the statistical functions. The study and analysis was completed successfully in January 2014.

[3] Doxygen - http://www.stack.nl/ dimitri/doxygen/

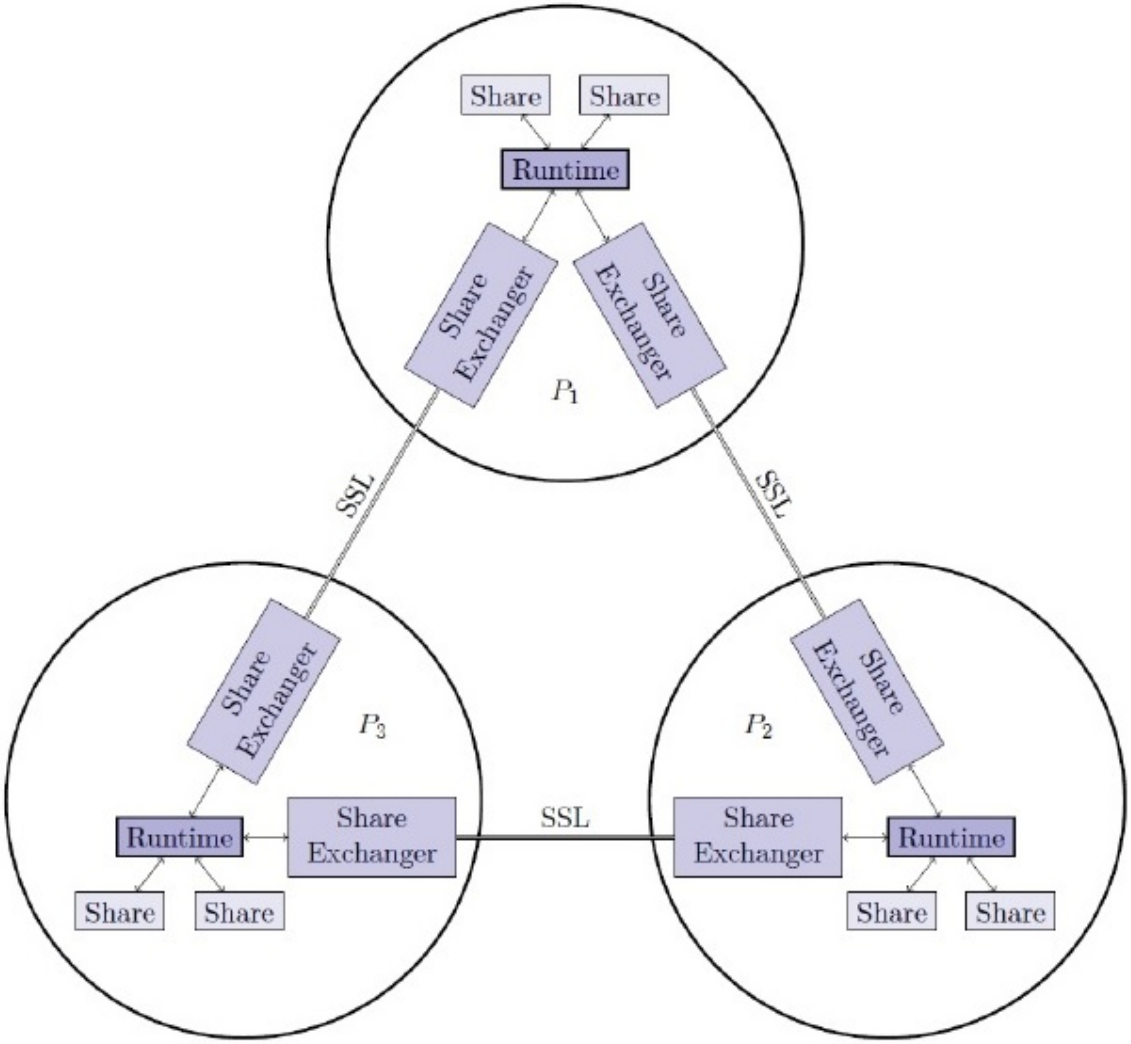


Figure 5.1: The VIFF Runtime.

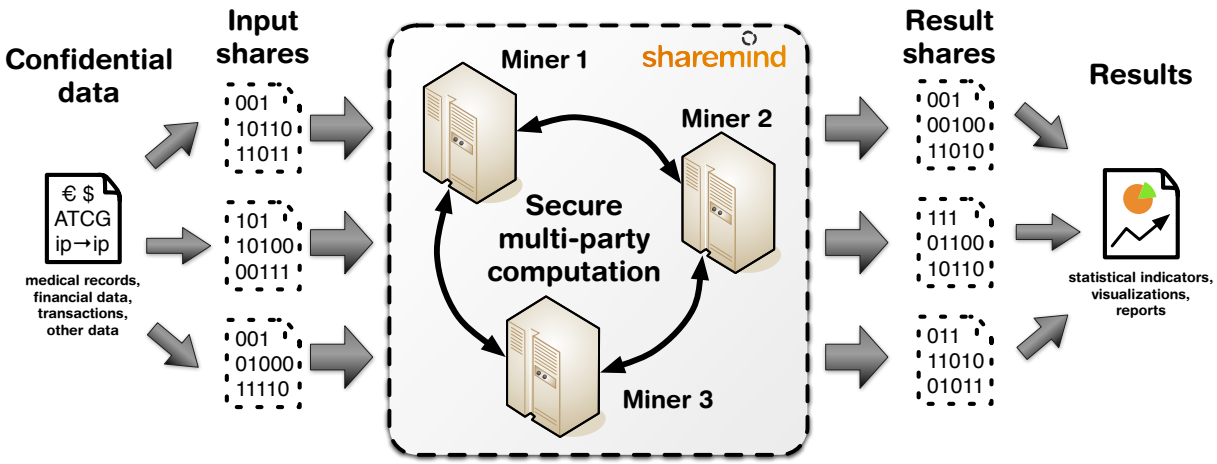


Figure 5.2: The SHAREMIND 2 Deployment Model.

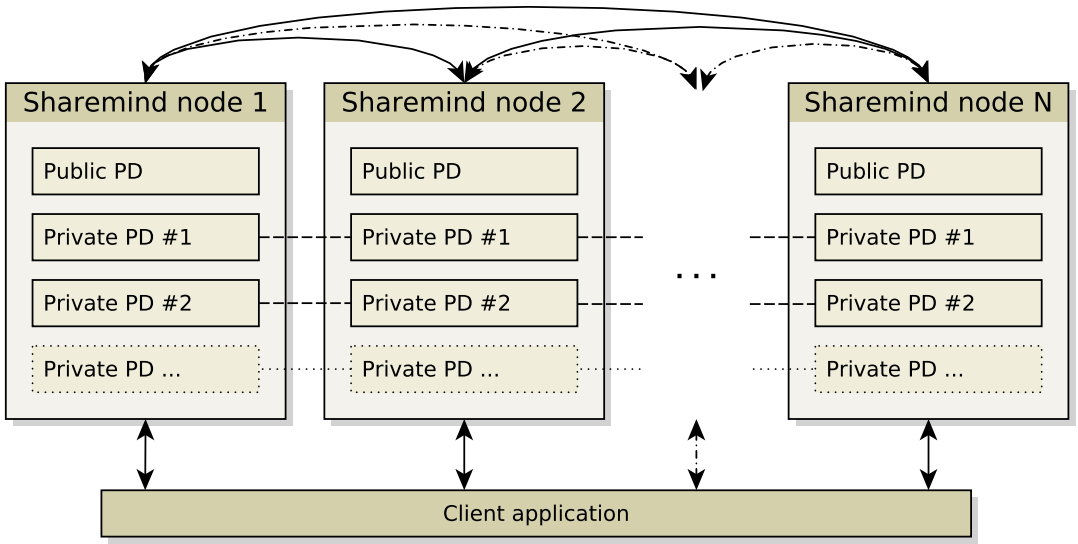


Figure 5.3: The General SHAREMIND 3 Deployment Model with Protection Domains.

Chapter 6

Integrated Tools

6.1 The Sharemind 2 Software Development Kit

6.1.1 Introduction

The SHAREMIND SDK is a software bundle with the aim to support the development of SHAREMIND applications. It includes a number of tools, examples and documentation. The main tools it offers are: DEVMINER, SECRECIDE [Reb10] and the SECREC compiler. Documentation on the basic usage and the API is given for the SECREC and the SHAREMIND assembly language. For an overview on these languages, see Section 2.2. Example controller applications and SECREC programs are given as a quick tutorial.

6.1.2 Tools

DEVMINER is a graphical development application in which three SHAREMIND nodes run in the same process. It provides an easy way to set up a basic SHAREMIND test environment for running and debugging SECREC programs. Figure 6.1 shows the user interface for the DEVMINER tool after starting the nodes. The output for each of the nodes is shown in a separate section.

SECRECIDE is an integrated development environment (IDE) for developing SECREC programs for SHAREMIND applications. The main features it provides are: project management, syntax highlighting and debugging of compiled SECREC programs. Syntax highlighting is provided for both, the SECREC and the SHAREMIND assembly language. Figure 6.2 illustrates some of the features. It shows an open project with multiple SECREC code files. The active code file is ready to be compiled and run.

The SECREC compiler is included as a command line tool and can be used directly. However, for easier usage it is also integrated into SECRECIDE. For details on the implementation of the compiler see [Jag10].

Examples of SECREC programs and controller applications are given from simple data insertion to complex data analysis applications.

6.1.3 Usage

The first version of the SHAREMIND SDK was released in December, 2010. It was packaged as an installer for the Windows operating system. In the following years, the SHAREMIND tools were improved and updates were released. In 2013, a Linux virtual machine version of the SDK was released because a platform independent SDK was requested. This virtual machine uses the open virtualization format (OVF) and runs a 64-bit Debian Linux operating system. Basic Linux tools and helper scripts are provided in addition to the standard SDK tools.

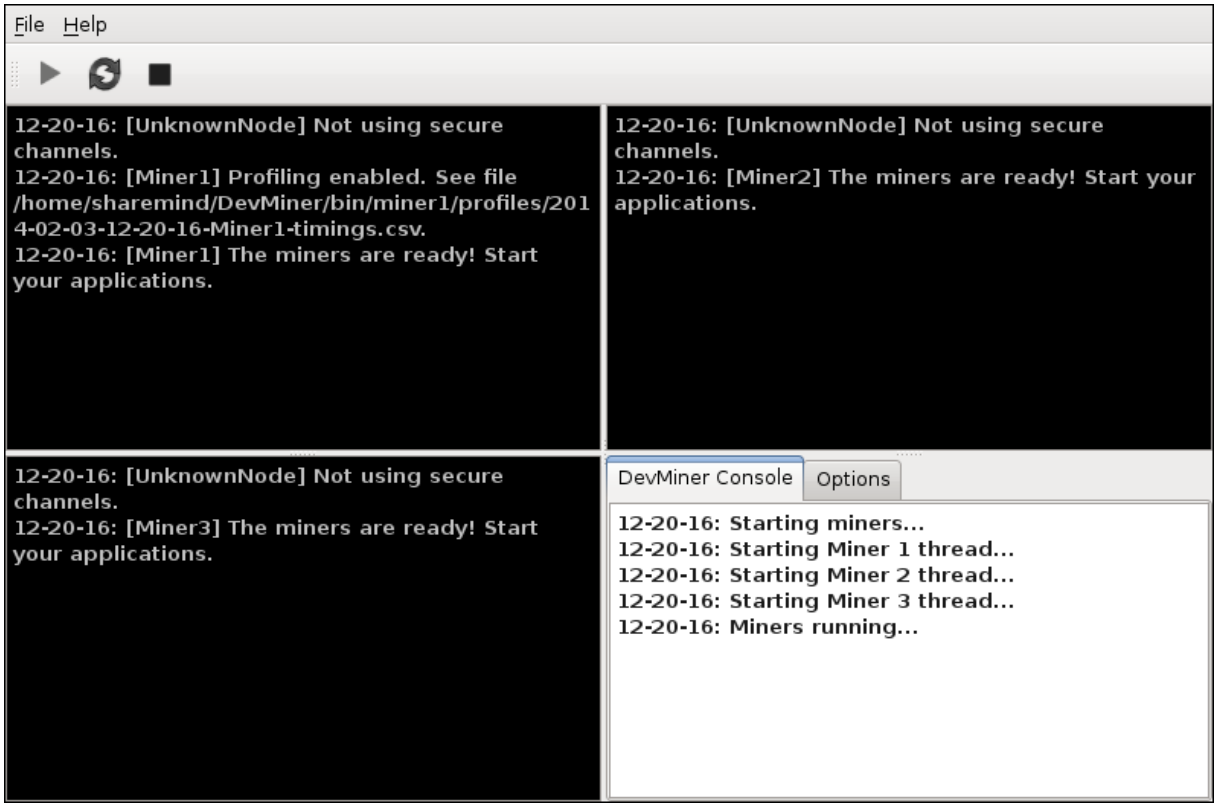


Figure 6.1: Interface of the DEVMINER Tool

The SDK has been useful in introducing SHAREMIND to a wide range of interested parties. Since the first release of the SDK, the Windows installer and the virtual machine have been downloaded hundreds of times. Support for the SDK has been requested from multiple universities in different countries, including: United States of America, Sweden, Iran and the People's Republic of China.

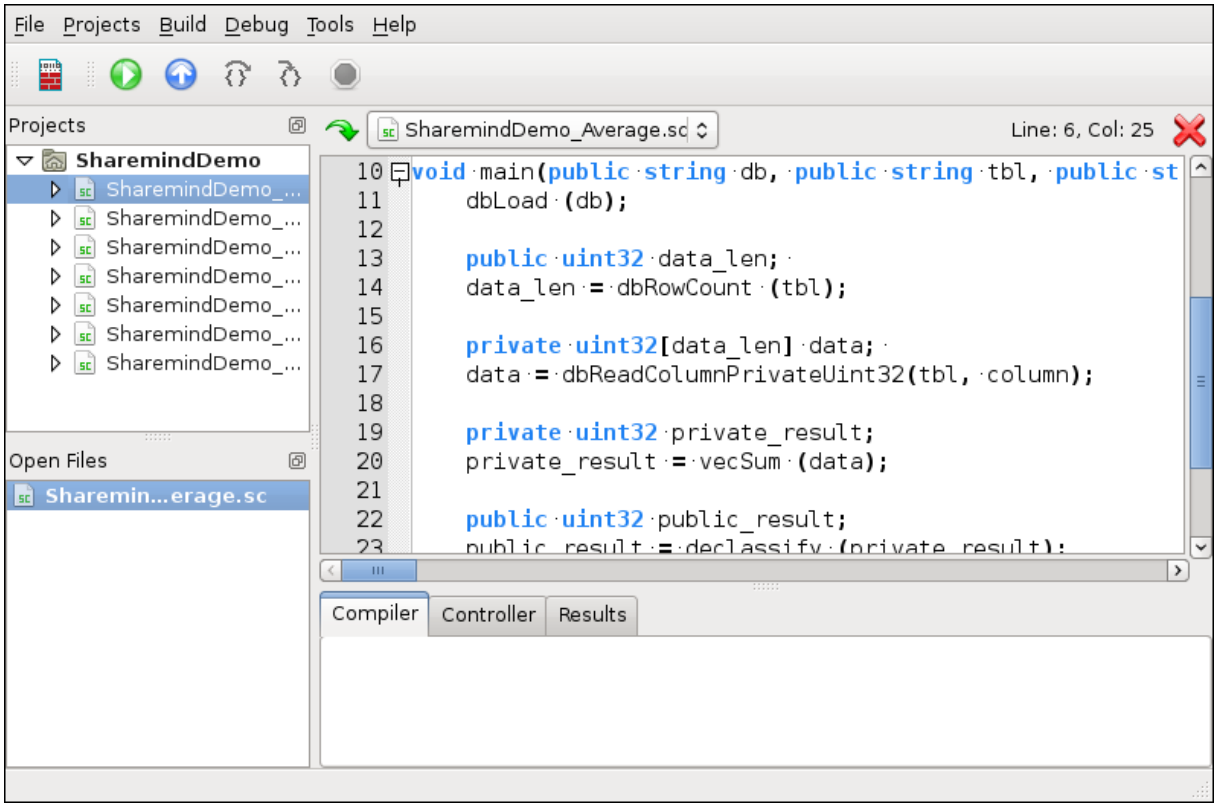


Figure 6.2: Interface of the SECRECIDE Tool

Chapter 7

Applications

7.1 Auction as-a-service

An auction is a set of rules (a protocol) that dictates exactly how information is shared and trades executed. This way a well-designed auction may address traditional market failures in decentralized markets as well as failures in governmentally controlled markets. The use of auction has increased tremendously in almost all sectors of the economy, two examples are the use of double auctions on commodity exchanges as well as some financial markets and the use of uniform price auctions for selling treasury bills or for settling interest rates in lending markets. The use and design of auction is constantly developing to address more complex market situations or to become part of automated trading systems. For a general introduction to auction in theory and practice see e.g. [Rot02, Mil04, Kle04].

An auction is a good example of a service that requires a secure digital infrastructure offered by secure multiparty computation. First, it always requires high integrity to avoid any manipulation of the auction rules (the protocol). Second, it often requires high confidentiality as most auctions has a component of sealed information. Third, it often requires high availability as many auction markets are part of a larger timely economic system.

7.1.1 Implementations

There are several prototypes that demonstrate how secure computation can be used to safeguard auctions. Secure multiparty computation is also used commercially in the two auction markets describe below.

- The Contract Exchange is used to reallocate production rights among Danish Sugar beet growers[1]. The case is described in details in several papers see e.g. [BCD+09b].The implemented auction design is a so-called double auction which is widely used e.g. for exchanging both physical and financial commodities see e.g. [Kle04, GD98].

- Energiauktion.dk is an automated electricity broker. The market for electricity has been liberalized and competitive prices are settled at the large power exchanges, however the endusers do not meet these competitive prices unless they continuously switch to the most competitive electricity suppliers. This is not the case for the vast majority of private consumers and small and medium sized enterprises (SMEs), which is the target group for this service[2].

[1] The solution is developed and operated by Partisia

[2] The solution is developed and operated by Partisia and supplied as Software-as-a-Service to the Danish energy broker www.energiauktion.dk.

The solution is basically a trustworthy ”middleman” that lowers the barriers and the costs of switching suppliers in a competitive sound way. The auction design is a simple first price sealed bid auction and secure computing is used to keep the power suppliers price-bids sealed at all time. The bidders (a legal entity with many employees) should be able to seamlessly delegate access to the encrypted bids across users and devices, this will be solved by the key management solution described in Section 7.4.

7.2 Confidential benchmarking

In general terms, benchmarking is the process of comparing the performance/activities of one unit against that of the best practices. Typically, one of two different so-called frontier evaluation techniques are used; the parametric Stochastic Frontier Analysis (SFA) or the non-parametric Data Envelopment Analysis (DEA). Both are explorative data analyses and relative performance evaluation techniques that support advanced benchmarking. The basic idea is that a ”decision making unit” transform multiple inputs to multiple outputs subject to technological constraints. Instead of benchmarking against engineering standards or statistical average performances, frontier evaluation evaluates the performance against that of the best performing peer units. For a general introduction see e.g. [BO11, CcLs94].

Benchmarking is widely used to generate insight e.g. about the variation in performance or the productivity development in a given sector. It is also widely used for planning e.g. for setting and keeping the right targets in business intelligence systems such as the so-called Balanced Scorecard, see e.g. [BBN06]. Finally, it is used for motivation e.g. for regulating natural monopolies through so-called yardstick competition , see e.g. [BN08, NT07].

Benchmarking clearly involves data sharing where different decision making units share data that best describes the performance of the units. This involves typically a third party (e.g. a consultant) to confidentially handle the data and run the analysis. Secure multi-party computation may constitute this third party and enhance the control and data protection. As in the case described below, SMC allow the data owners to keep control of the shared data while making it available for benchmarking.

7.2.1 Implementations

Secure multiparty computation is currently used in a controlled R&D experimental setup for benchmarking of commercial bank customers across Danish banks. In this application, benchmarking evaluates economic efficiency of the commercial customers and functions as a complement to traditional credit rating. The value-added comes from a richer data foundation (confidential data sharing) and confidential benchmarking that provide insight in to the individual commercial customers as well as the possibility to explore how exposed a given bank relative to the various subsets of the shared data.

The ideas and this initial demonstration software has been developed in the now ended research project COBE (Confidential Benchmarking). Jointly with the research center CFEM.dk we will in Practice continue the work and develop a prototype for benchmarking, ideally in collaboration with banks or other users. The applied benchmarking technique is the DEA approach which can be formulated as a LP-problem, which is solved by MPC based Simplex[3].

[3]This implementation is at an early stage and will be developed further within Practice and CFEM.dk.

7.3 Confidential Data Sharing Tool

7.3.1 Introduction

In the simplest set up, secure multi-party computation can be used to protect a single data set. Consider a scenario where the owner of a large data set containing sensitive data is interested in analyzing this data but has no competence to carry out the analysis itself. However, because of the sensitive nature of the data, the data set cannot be shared with professional analysts or researchers. Here, secure multi-party computation provides an excellent mechanism, as it can sandbox the user into a controlled learning environment where just a certain number of queries are enabled. Similarly, the user can protect its query parameters so that the database owner will not know, what were the inputs to the query.

However, this case extends naturally to other data sharing scenarios, especially when there are multiple data owners who want to combine their knowledge to learn more. Secure multi-party computation also allows the parties to control that the data they share with each other can not be abused after sharing. In fact, secure computation lets the data owners keep control even after making data available to others.

Secure computation also enables such applications when the data owners do not have significant ICT infrastructure. The guarantees of secure computation enable such data owners to use cloud computing services to host the data sharing services [BK13]. Even further, we foresee the emergence of secure cloud-based data sharing services and we plan to develop the technology for that in PRACTICE.

7.3.2 Implementations

Combining data from various sources can be done on databases with the same structure through concatenation. In this case, the computations are usually similar to the computations each party can do on its own data set. However, combining the data sets allows to cover a greater area of the domain and thus get more meaningful results. There are several prototypes that demonstrate how secure computation can be used to jointly analyse similarly structured data.

- Cybernetica developed an experimental web-based secure data sharing service for a meeting of the Steering Board of the European Cloud Partnership in July 2013[4]. The Income analysis of the Estonian public sector demonstration combines income information from the ministries and larger municipalities of Estonia onto a cloud-hosted secure analysis service. The application can securely analyze income data collected from several sources. The demonstrator is hosted on standard public cloud servers. During the demo, the three cloud-hosted secure computation servers were controlled by two government agencies and one private sector company. At the publication of this document, the demo is hosted at `https://sharemind.cyber.ee/clouddemo/`.

- A special case here is a scenario where each input party only has a single data record to collaborate the shared database. This was the case in the privacy-preserving financial analysis application for the Estonian Association of Information Technology

[4]Report from the meeting of the Steering Board of the European Cloud Partnership hosted by President Ilves – `http://president.ee/en/media/press-releases/9234-picture-news-president-ilves-met-with-the-european-commissioner-for-the-digital-agenda-neelie-kroes/`

and Telecommunications where each party submitted only its financial indicators for the given period [BTW12, Tal11].

- In [KBLV13], the authors show how several biobanks can pool together gene expression data from their patients and perform a genome-wide association study on the combined patient cohort.

Sometimes, a collaboration among organizations brings together parties with different kinds of knowledge. In this case, their databases have different structures, but some common fields that can be used for linking. In common database management systems (DBMS) rows from such distinct data sets are linked together based on values in one or more *key columns*, e.g. social security number or other unique identifier. In SMC systems, one can use oblivious database join that is similar to SQL equi-join, but works on secret shared values [LTW13]. Since there are so few secure linking implementations available, we can currently report on just one prototype.

- The PRIST [5] project where income data from Tax Office is linked with education information from the Estonian Education Information System to analyze return of investment of IT curriculums. Using SMC and oblivious database join operation means that the Statistics Board does not have to compile a new database with highly sensitive data.

7.4 Key Management

Today it is not uncommon for a person to have multiple computing devices (PC, laptop, phone, tablet) both for professional and personal usage. The use of multiple devices naturally leads to a desire of having the same content available on the different devices and many successful solutions currently exists to synchronize the content among devices, examples include DropBox, Box, various Google and Apple products and similar. In many situations it is a requirement that the confidentiality of the data is ensured when the data is at rest on the device. This requirement comes from simple things like consumers or companies wanting to prevent compromising their privacy and identity to more involved cooperate responsibilities when processing data using mobile devices. A multitude of solutions exists which provides some level of assurance of the confidentiality of the data, ranging from password protection (DropBox, Crashplan are examples), to use of AES encryption on top of other commercial offerings (BoxCrypto is an example).

This puts the burden of distributing the password or encryption key among the different devices on the user. In the case of password protected resources like DropBox or Crashplan the burden seems initially reasonable. However, if enough entropy is to be embedded in the password to be able to produce a sufficiently strong encryption key, then the length begins to reach a size (say 15 to 20 random characters) where it is cumbersome, error prone, and plain impractical to enter it using anything but a keyboard. Copying the key to a media (e.g. USB) is not possible for a number of devices as many popular devices does not support USB. Emailing the key or using e.g. DropBox is in most cases a security liability and not allowed for good reasons.

[5] "Privacy-preserving statistical studies on linked databases", funded by the European Regional Development Fund from the sub-measure "Supporting the development of the R&D of information and communication technology" through the Archimedes Foundation.

A solution is to use a centralized keyserver, where all ones keys can be stored and retrieved by each provisioned device. However, this server must either be hosted inhouse or hosted by a trusted provider. The later is not a viable or legal possible solution in many cases. The former solution eliminates the advantages of the cloud e.g. cheap hosting.

Another solution is to use SMC to delegate the trust to multiple cloud providers, and in this way achieve the required security properties while enjoying some of the benefits of the Cloud. A service based on SMC would enjoy availability as the service can be reached through the internet and does not require one to be on a specific local network. The service would be more practical as only one username and password needs to be entered in each device. The password would not be used to generate cryptographic keys and can thus be of shorter length and so easier to type than otherwise. The service would store cryptographically strong encryption keys which can be downloaded by enabled application as needed. This will lead to strong security guaranties. The service can leverage the benefits of the Cloud like reduced host costs and elasticity to a certain degree. It is clear that the benefits are reduced as the service needs to be distributed among more than one provider of cloud infrastructure.

As for any SMC application a solution based on SMC requires the trust to be distributed. The trust can be distributed in a number of different ways. One setup could be to allow all but one cloud provider to be malicious. This will provide very strong security guaranties to the confidentiality of the stored key material. However, it might also introduce some high costs in terms of availability or performance. Another trust model, is the threshold model. In this model the confidentiality of the key material is ensured as long as at least some majority of the cloud providers are honest. This model may offer some additional flexibility in tuning the availability and performance parameters on the expense of less strong trust guaranties. Which model to choose depends on the actual requirements for confidentiality, availability, and performance.

7.5 MobiShare

MobiShare is an example of a key management scheme that can be used to enable secure data sharing in the cloud. It addresses the problem that users desire to store their data in the cloud encrypted to protect their privacy but also want to share it. Users could either download the data and share it locally, decrypt their data in the cloud before sharing it or share the encrypted data in the cloud and give their encryption key to the recipient of the data. The first solution is unfavourable because it is impractical in mobile scenarios and for large data. The second solution would reveal the user's key to the cloud, violating her privacy. In the third solution the user does not have to reveal his key to the cloud but to the recipient of the data.

MobiShare is an application that allows secure file transfer in the cloud neither requiring to download the data before transferring it nor revealing the own encryption key to anyone. This is achieved by encrypting every file in the cloud individually with an separate key. Therefore, the key for one file can be revealed without jeopardizing the confidentiality of the remaining files. When two users want to share a file, the encrypted file is copied in the cloud from the storage of the sender to the storage of the receiver. Then the corresponding encryption key is transferred between the users. In case of MobiShare NFC (Near Field Communication) is used to establish a common secret which in turn is used for the key transfer. The NFC technology provides wireless data transfer over a very short distance (a few centimetre). Thus the trust model of MobiShare leverages the existing trust relationship between two users who know and trust each other. The users visually authenticate each other before physically tapping their mobile devices together to initiate the key exchange. In other words, the mobile phones act as trust anchors in

the MobiShare architecture.

The main challenge in the the MobiShare design is to realize the key management in an efficient way. This is achieved by using two staged encryption. In the first stage each file is encrypted with an individual key. To make the management of these keys easy they are stored together with the file (the file and the key are enveloped). But, of cause, these *file keys* themselves must be encrypted before transferring to the cloud. All file keys are encrypted with an *user key*. This key is stored locally with the user and used whenever the he wants to access a file key. When a file key should be shared it key gets downloaded, decrypted locally and then re-encrypted with a *shared key*. The shared key is established between the MobiShare users via NFC (more precisely, with Diffie-Hellman key exchange). Then the re-encrypted file key gets transferred to the share storage area making is available to the receiver.

A rather technical challenge in the MobiShare design is the transfer of the encrypted files in the cloud between the users private storages, as the users may use different providers. To address this problem MobiShare provides an on-demand shared storage in the cloud. This shared storage can act as an interconnection between the different interfaces for file transfer offered by different cloud providers. Figure 7.1 shows the steps used in MobiShare from the perspective of the users.

1. Both users connect to their personal storages using their mobile devices. Views of the personal files are rendered on both mobile devices (in the upper half of the display).

2. The users tap their mobile devices together to make a NFC key exchange. A temporary and secure shared storage area is created in the cloud, while views of the shared storage area appear on both mobile devices (the yellow areas).

3. The sender drags a logical file link from her personal cloud storage view (red area) to the shared storage area view (yellow area). In the cloud a file transfer from the senders personal storage to the shared storage area is performed, while a logical link to the file appears appears in the receiver's view of the shared storage area.

4. The receiver drags the logical link from his shared storage view to his personal storage view (green area). In the cloud the file is transferred from the shared storage to the receiver's personal storage.

During the transfer in the cloud the file always stays encrypted (in the senders personal storage, in the shared storage area and in the receivers personal storage).

7.6 Statistical Analysis Tool for Confidential Data

In the project UaESMC[6], interviews were conducted with 25 stakeholders with different professional backgrounds. The interviewees pointed to different subjects that they thought could benefit from the use of secure multi-party computation. Among these, the statistical analysis topic was often mentioned [BKLPV13].

As creating a custom tailored application for each study would be too time-consuming and expensive, it is reasonable to make a statistics framework that offers the most commonly used functions and even some simpler statistical tests. The database can be set up and with it this query framework and the data analyst can start work immediately. Interviewees often expressed

[6]UaESMC: project no. FP7-284731 in the EU FP7-ICT FET Open scheme

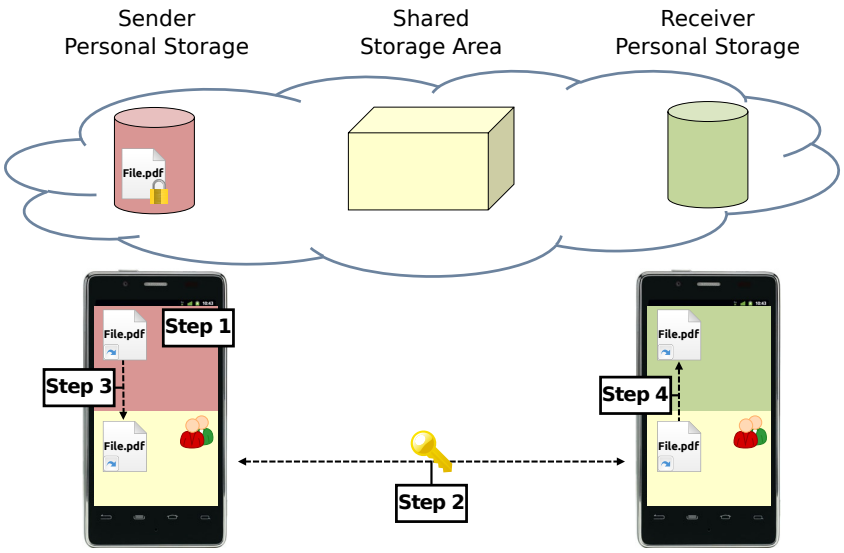


Figure 7.1: MobiShare Application Design Overview

concern when they heard that it is not possible for them to see the data when it has been stored in a privacy-preserving manner. This framework gives the analyst a general overview of the data and its properties and allows him or her to make preliminary tests to ascertain which complex tests are necessary.

Ideally, this framework would be similar to what other widely used statistics packages such as SAS, SPSS, and GNU R offer. This would provide the analyst with not only a possibility to easily access the data but also have a familiar intuitive interface.

The UaESMC project is implementing a prototype of such a tool. The statistics suite currently includes data import from files in comma separated value (CSV) format, calculation of different statistics, different methods for hypothesis testing and a language interpreter similar to GNU R. The tool has not yet been used in practical applications, but there is a plan to use it in the PRIST project described in Section 7.3.

The statistical tool prototype is based on the SHAREMIND application server and uses the SECREC 2 standard library. Currently, it can perform the following statistical analyses:

- Cutting data from a dataset based on a filter;
- One dimensional frequency table calculation for discrete and continuous data (can be visualised as a histogram);
- Two dimensional frequency table calculation for discrete data (can be visualised as a heatmap);
- Quantile calculation;
- Five-number summary, including minimum, maximum, median, lower and upper quartiles (can be visualised as a box-plot);
- Simple outlier detection based on quantiles;
- Hoare's selection algorithm;
- Student's t-test;

- Paired t-test;
- Wilcoxon rank sum test;
- Wilcoxon signed rank test;
- χ^2 test for the general case and an optimised version of the χ^2 test for two classes.

7.7 Supply Chain Management

The project secureSCM [7] implemented and analyzed a use case applying secure multi party computations for a supply chain.
Consider the following problem. Different companies collaborate as members of a supply chain. They can cash in if they share sensitive data to compute certain calculations. Although the financial benefit might be significant, they are reluctant as the confidentiality is compromised by revealing sensitive information.
This compromise can lead to future disadvantages. Revealing sensitive information to competitors or business partners might weaken the bargaining position of company. If a company reveals important business data to a competitor, then the competitor can easily adapt his own offering. If a business partner knows key figures of a company, this knowledge weakens the company's position in business deals. Therefore, companies are usually very reluctant to provide access to sensitive information.
To solve this problem, the supply chain collaborators use secure multi party computation which avoid the revealing of sensitive information but enables to realize gain.
This section gives an overview of the use case in Subsection 7.7.1, defines the model in Subsection 7.7.2, and presents an approach to securely solve a linear programming problem in Subsection 7.7.3.

7.7.1 Use Case

Supply chains can be considered as networks of geographically dispersed facilities. In the facilities, raw materials, intermediate, or finished products are produced, tested, modified, and stored. These facilities are connected by transportation links. The facilities may be operated by organizational units (planning domains) within a company, or individual companies such as manufactures of finished products, suppliers of intermediate products, vendors, logistics service providers, and customers. Therefore supply chains can be described as inter-organizational systems with a multiple number of independent planning domains, each responsible for a specific set of facilities or links of the supply chain.
While the physical and institutional configuration of the supply chain will be determined on a strategic planning level (supply chain configuration), tactical decisions related to production, transportation, and inventory quantities are major tasks of supply chain master planning. The objective of supply chain master planning is to create an aggregated SC-wide master plan defining production, transportation, and inventory quantities for every node and link of the entire supply chain on a medium term basis of 12 to 24 months. We assume that a 4th Party Logistics Provider (4PL) adopts the role of a central planning unit. The configuration of the supply chain is illustrated in Figure 7.2. We focus on a supply chain of the company Avio. Avio

[7] securescm: project no. FP7-213531 in the EU FP7-ICT FET Open scheme

S.p.A was an italian-based company with business units in the aeronautical and space sector. Their business areas included the design, development, and manufacture of accessory drive trains and power transmissions, low-pressure turbines, combustors, afterburners, subsystems for civil and military aeronautical engines. They also participated in the design, development, and manufacture of propulsion systems for space launchers and tactical missiles and the development, integration, and production of the new European launcher Vega. The branch of the company providing this use case is now part of the company Avio Aero.
The structure of the supply chain is depicted in Figure 7.3. It represents a part of the supply chain for shroud nozzles that Avio manufactures and delivers to its customers (engine manufactorers). In the standard operating mode, Avio procures four components from different suppliers that are delivered to a raw materials and component warehouse. Depending on the production schedules, these components are delivered to manufacturing plant 1, where they are pre assembled. They are then shipped to a different warehouse (warehouse 2 for semi-finished products). From there they are delivered to a different manufacturing plant where final assembly, testing and quality control are performed. All completed components are stored in another one of Avio's warehouses that manages all shipments to the customers. These customers are program leader and other companies that order the shroud nozzle as a spare part.
In general, this part of the supply chain is characterized by very stable production schedules that are determined well in advance according to the program leaders schedule for aircrafts. Sometimes, however, spare part orders are placed and the program leader may, on short notice, change its own production schedule. This may lead to an upsurge of demand which cannot be easily handled by the standard supply chain depicted previously. In the current setting, Avio has no means to quickly change schedules and obtain additional parts from its standard suppliers. Therefore, to cover additional demand and to live up to very high service level requirements of its customers, Avio employs a contract manufacturer (outsourcing partner) who supplies, on short notice, the semi-finished shroud nozzle. When receiving orders from Avio, this manufacturer is responsible for procuring the necessary components and for assembling the shroud nozzle. One exception is the support shroud (component 4) that Avio provides to the contract manufacturer. The contract manufacturer supplies directly to Avios semi-finished component warehouse and charges a high price premium (that exceeds the cost of the standard supply chain by an order of magnitude).
Since outsourced manufacturing is extremely expensive, Avio would like to implement a system that enables them to quickly reschedule production in order to accommodate short term demand peaks. So far, the main obstacle to the implementation of such a system is that negotiations with and synchronization across the component suppliers takes too long. Avio has no information about supplier capacity and cannot quickly place orders at the suppliers. One reason for this is the fact that the suppliers are not willing to share detailed and real-time information about their capacity utilization and their ability to provide additional components.
To remedy this problem Avio wants to implement a rapid deployment tool that utilizes short term capacity information from their suppliers to quickly adapt supply and manufacturing schedules as soon as unanticipated demand arrives. To protect the suppliers sensitive and private capacity data, this rapid deployment tool will rely on secure multi-party computation. The basic concept of the rapid deployment tool is illustrated in Figure 7.4. To determine the new supply and production schedule Avio needs to solve an optimization problem with the objective to minimize the total cost (including supply, manufacturing, inventory and backordering cost) while respecting all relevant capacity constraints and fulfilling overall customer demand.

7.7.2 The Model

We define the underlying optimization problem as follows:

Indices/Index Sets

- $i \in \{1, \ldots, 10\}$: Index of nodes in the standard supply chain with $i \in \{1, \ldots, 4\}$ supplier locations, $i = 5$ warehouse, $i = 6$ production plant 1, $i = 7$ warehouse 2, $i = 8$ production plant 2, $i = 9$ warehouse 3, and $i = 10$ customer locations
- $k \in \{1, \ldots, 4\}$: Index of components
- $t \in \{1, \ldots T\}$: Index of time periods

Input Variables

- c_p^k: Unit purchasing cost of component k
- c^{sf}: Unit costs of manufacturing semi-finished goods
- c^{fg}: Unit costs of manufacturing finished goods
- c_o^{sf}: Unit costs of semi-finished product from contract manufacturer
- c_h^k: Unit inventory holding costs of component k
- c_h^{sf}: Unit inventory holding costs of semi-finished goods
- c_h^{fg}: Unit inventory holding costs of finished goods
- c^{back}: Unit backorder cost
- d_t: Demand in period t
- cap_i: Capacities of manufacturing plants i with $i = \{6, 8\}$
- cap_o: Capacity of contract manufacturer
- $\widehat{x}_{t,i,5}^k$: Scheduled procurement quantities of component k from supplier i with $i \in \{1, \ldots, 4\}$ in period t before rescheduling
- ϵ_t^k: Additional capacity of component k in period t (private parameters of suppliers)

Output Variables

- $x_{t,i,5}^k$: Shipping quantities after rescheduling of component k from supplier i with $i \in \{1, \ldots, 4\}$ to warehouse 5 in period t
- $x_{t,5,o}^4$: Shipping quantities of component 4 from warehouse 1 denoted as $i = 5$ to contract manufacturer in period t
- $x_{t,i,i+1}^{sf}$: Shipping quantities of semi-finished products in period t with $i \in \{6, 7\}$
- $x_{t,o,7}^{sf}$: Shipping quantities of semi-finished products from contract manufacturer to warehouse 2 denoted as $i = 7$ in period t

- $x^{fg}_{t,i,i+1}$: Shipping quantities of finished products in period t with $i \in \{8, 9\}$
- $y^{sf}_{t,6}$: Production quantity of semi-finished products at plant 1 denoted as $i = 6$ in period t
- $y^{sf}_{t,o}$: Production quantity of semi-finished products by contract manufacturer in period t
- $y^{fg}_{t,8}$: Production quantity of semi-finished products at plant 2 denoted as $i = 8$ in period t
- $inv^{k}_{t,5}$: Inventory of component k in warehouse 1 denoted as $i = 5$ in period t
- $inv^{sf}_{t,7}$: Inventory of semi-finished products in warehouse 2 denoted as $i = 7$ in period t
- $inv^{fg}_{t,9}$: Inventory of finished goods in warehouse 3 denoted as $i = 9$ in period t
- b_t: Backorder quantity in period t

The optimal supply, production, inventory, and shipping quantities of Avio is the solution of a linear programming problem. The objective function is

$$\begin{aligned}\min C = &\sum_{t=0}^{T}\sum_{k=1}^{4} c^{k}_{p} x^{k}_{t,k,5} + \sum_{t=0}^{T} c^{sf}_{o} x^{sf}_{t,o,7} \\ &+ \sum_{t=0}^{T} c^{sf} y^{sf}_{t,6} + \sum_{t=0}^{T} c^{fg} y^{fg}_{t,8} \\ &+ \sum_{k=1}^{4}\sum_{t=0}^{T} c^{k}_{h} inv^{k}_{t,5} + \sum_{t=0}^{T} c^{sf}_{h} inv^{sf}_{t,7} \\ &+ \sum_{t=0}^{T} c^{fg}_{h} inv^{fg}_{t,9} + \sum_{t=0}^{T} c^{back} b_t\end{aligned} \tag{7.1}$$

and has the following flow, capacity, and non-negativity constraints.
The flow constraints are as follows:

$$x^{fg}_{t,9,10} + b_t = d_t + \sum_{\tau=0}^{t-1} b_\tau \text{ with } t = 0, \ldots, T \tag{7.2}$$

$$inv^{fg}_{t,9} = inv^{fg}_{t-1,9} + x^{fg}_{t,8,9} - x^{fg}_{t,9,10} \text{ with } t = 1, \ldots, T \tag{7.3}$$

$$x^{sf}_{t,7,8} = y^{fg}_{t,8} = x^{fg}_{t,8,9} \text{ with } t = 0, \ldots, T \tag{7.4}$$

$$inv^{sf}_{t,7} = inv^{sf}_{t-1,7} + x^{sf}_{t,6,7} + x^{sf}_{t,o,7} - x^{sf}_{t,7,8} \text{ with } t = 1, \ldots, T \tag{7.5}$$

$$x^{k}_{t,5,6} = y^{sf}_{t,6} = x^{sf}_{t,6,7} \text{ with } t = 0, \ldots, T \text{ and } k = 1, \ldots, 4 \tag{7.6}$$

$$inv^{k}_{t,5} = inv^{k}_{t-1,5} + x^{k}_{t,k,5} - x^{k}_{t,5,6} - x^{k}_{t,5,o} \text{ with } t = 1, \ldots, T \text{ and } k = 1, \ldots, 4 \tag{7.7}$$

$$x^{k}_{t,5,o} = y^{sf}_{t,o} = x^{sf}_{t,o,7} \text{ with } t = 0, \ldots T \tag{7.8}$$

The capacity constraints are as follows:

$$y_{t,8}^{fg} \leq cap_8 \text{ with } t = 0, \ldots, T \tag{7.9}$$

$$y_{t,6}^{sf} \leq cap_6 \text{ with } t = 0, \ldots, T \tag{7.10}$$

$$y_{t,o}^{sf} \leq cap_o \text{ with } t = 0, \ldots, T \tag{7.11}$$

$$x_{t,k,5}^{k} \leq \widehat{x}_{t,k,5}^{k} + \epsilon_t^k \text{ with } t = 1, \ldots, T \text{ and } k = 1, \ldots, 4 \tag{7.12}$$

The non-negativity constraints are as follows:

$$x_{t,k,5}^{k}, x_{t,5,6}^{k} \geq 0 \text{ with } t = 0, \ldots T \text{ and } k = 1, \ldots, 4 \tag{7.13}$$

$$x_{t,7,8}^{sf}, x_{t,o,7}^{sf} \geq 0 \text{ with } t = 0, \ldots T \tag{7.14}$$

$$x_{t,8,9}^{fg}, x_{t,9,10}^{fg} \geq 0 \text{ with } t = 0, \ldots T \tag{7.15}$$

$$y_{t,6}^{sf}, y_{t,8}^{fg}, y_{t,o}^{sf} \geq 0 \text{ with } t = 0, \ldots T \tag{7.16}$$

This linear program will provide the optimal values under the given constraints. The private parameter is the additional capacity of the suppliers denoted as ϵ_t^k. All other parameters are not private.

7.7.3 Securely Solving Linear Programming Problems

We developed a new approach of securely solving linear programs. Rather than letting all parties perform the whole secure computation for the linear program, we transform the original Linear Program (LP) into a second LP. The idea is to let one party solve the second LP locally using a non-cryptographic solver. We then transform back the results of the second LP to those of the original LP. This is done using a cryptographic protocol. The designated player solves the second LP and the second LP does not reveal more information from the original LP than what can be considered practically acceptable (i.e., just some bits of billions). In the following, we present the details of this transformation protocol.

Transforming the Protocol

Our approach significantly improves over [Vai09], which proposes the following transformation. Let

$$\begin{aligned} &\min c^T x \\ &s.t. Mx \leq b \\ &x \geq 0 \end{aligned} \tag{7.17}$$

be a linear program with n variables and m constraints. If we chose a random positive monomial matrix Q, we write

$$\begin{aligned} &\min c^Q(Q^{-1}x) \\ &s.t. MQ(Q^{-1}x) \leq b \\ &(Q^{-1}x) \geq 0 \end{aligned} \tag{7.18}$$

or with $\widehat{M} = MQ$, $y = Q^{-1}x$, and $c'^T = c^T Q$ as

$$\begin{aligned} &\min c'^T y \\ &s.t. \widehat{M}y \leq b \\ &y \geq 0 \end{aligned} \tag{7.19}$$

This transformation is not sufficient as it leaves b completely unprotected. We will show how to improve the security, protect b, and adapt this approach to obtain a protocol suitable for general multi-party scenarios.
Our transformation requires the input

$$\begin{aligned} &\min c'^T x \\ &s.t. M_1 x = b_1 \\ &s.t. M_2 x \leq b_2 \\ &y \geq 0 \end{aligned} \tag{7.20}$$

We use a positive monomial matrix Q to hide the value c, a strictly positive diagonal matrix S, and a positive vector r to hide the value x. It is

$$\begin{aligned} &\min c'^T Qz \\ &M_1 xQz = b_1 + M_1 Qr \\ &M_2 xQz \leq b_2 + M_2 Qr \\ &Sz \geq Sr \end{aligned} \tag{7.21}$$

for $z = Q^{-1}x + r$. We define $c'^T = c^T Q$. We also define

$$M' = \begin{pmatrix} M_1 Q & 0 \\ M_2 Q & 2A \\ -S & \end{pmatrix} \tag{7.22}$$

with A a permutation matrix representing slack-variables. It is

$$b' = \begin{pmatrix} b_1 + M_1 Qr \\ b_2 + M_2 Qr \\ -Sr \end{pmatrix} \tag{7.23}$$

We rewrite the linear program as follows:

$$\begin{aligned} &\min c_s'^T z_s \\ &s.t. M' z_s = b \\ &z_s \geq 0 \end{aligned} \tag{7.24}$$

with c'_s is c' with added zeros for the slack-variables and z_s is the vector z with added slack-variables.
We hide the content of matrix M' and vector b'. Therefore, we use a non-singular matrix P with

$$M'' = PM'$$

and

$$b'' = P'b'$$

It is

$$\begin{aligned} &\min c_s'^T z \\ &s.t. M'' z_s = b'' \\ &z_s \geq 0 \end{aligned} \tag{7.25}$$

With $z = Q^{-1}x + r$, we can calculate x as $x = Q(z - r)$.

7.7.4 Applying the Protocol

In Supply Chain Optimization, the data required for the linear program is distributed among the participating companies. These companies do not trust (or do not want to trust) each other and thus do not want to exchange the input values. Under these circumstances, it is not apparent how to use our transformation.
We have to address the following issues:

1. How to assemble the data necessary to set up the LP without compromising privacy. The necessary data are M_1, M_2, b_1, b_2, and c.

2. How to jointly choose the random vectors and matrices necessary for the transformation. The necessary data are P, Q, S, A, and r. We also have to ensure that no party knows or learns them as that would allow them to undo the transformation.

3. How to apply the transformation to obtain the values c'_s , M'', and b'' without revealing any other information.

We use Secure Multi-Party computations to solve this problem. More precisely, we will use secure computations as proposed by [BOGW88] based on Shamir-shared values with a security threshold as we focus on the Semi-Honest-Model.
They allow us to share values among the parties in a way that the knowledge of less than k of p shares does not reveal any information about the secret value. Yet these shares can be used to make computations like additions or multiplications on the secret values. After finishing the computations, we can put the shares of the result together and reconstruct it - whereas all input and intermediate values remain secret (see example in Figure 4 in Chapter 2). Once we have shares of all necessary data M_1, M_2, b_1, b_2, c, P, Q, S, and r, we can use secure computations to perform the transformation.
We now discuss how to get these shares. To understand how the data for M_1, M_2, b_1, b_2, and c can be assembled, we have to keep in mind the structure of the LP. Each variable $x^n_{i,k,t}$, $\text{inv}^n_{i,k,t}$, and $\text{inv}^m_{i,k,t}$ is data of one party which is represented as node $k \in K_i$ in the supply chain master planning model. This party knows all relevant data concerning these variables (costs, capacities etc.). The transport variables $y^n_{i,k,k+1,t}$ are jointly owned by the parties denoted as nodes $k \in K_i$ and $k + 1 \in K_i$, i.e. both parties know the relevant data.
The parties can use the following protocol to assemble the matrices:

1. Each party sets up $M_{1,i}$ as part of M_1, $M_{2,i}$ as part of M_2, $b_{1,i}$ as part of b_1, $b_{2,i}$ as part of b_2, and c_i as part of c. In the matrices, each party sets only those values concerning its own variables and its outbound transport variables (or alternatively its inbound variables, but not both). All other values are set to 0.

2. The matrices are shared using secret sharing.

3. The parties jointly calculate the following computations using secure computations on the shares.

$$\begin{aligned} M_1 &= \sum_{i=0}^{p-1} M_{1,i} \\ M_2 &= \sum_{i=0}^{p-1} M_{2,i} \\ b_1 &= \sum_{i=0}^{p-1} b_{1,i} \\ b_2 &= \sum_{i=0}^{p-1} b_{2,i} \\ c &= \sum_{i=0}^{p-1} c_i \end{aligned} \tag{7.26}$$

Each entry in M_1 , M_2, b_1, b_2, and c has at most one non-zero summand due to the structure of the LP and the entire data is correctly merged. Therefore, the involved party require some knowledge which is not publicly known: the structure of the supply chain with the number of nodes involved in the different stages.
Moreover, we have to find an efficient way to choose the random matrices and vectors jointly by all involved parties p. The choice should be fair and secure, i.e. even the knowledge of $p-1$ input values should not leak any information about the resulting values. This is to ensure that no participating company has enough information to undo the transformation and to obtain private values. We will now discuss each type of random matrix and vector we use in detail.
We start with the choice of a random permutation matrix such as matrix A. This can easily be chosen jointly with the following steps:

1. Each party chooses his random permutation matrix A_i as part of A.

2. The matrices are shared among the parties.

3. The parties compute $A = \prod_{i=0}^{p-1} A_i$ using secure computations on the shares.

This is obviously fair and secure since even when knowing $p-1$ permutations, all output values remain possible.
We continue with the explanation how to jointly choose a random vector, in this case the random vector r. To chose a random vector the parties execute the following protocol:

1. Each party chooses a random vector r_i as part of r.

2. The vectors are shared among the parties.

3. The parties compute $r = \sum_{i=0}^{p-1} r_i$ using secure computations on the shares.

If the addition is done within a finite field (as it is the case for Shamir-shared values), the resulting value is fair as all p input values are needed to obtain any information about r . We continue with the explanation how to choose a "real" random matrix such as P and follow the same approach as for the random vector r.

1. Each party chooses a random matrix P_i.
2. The matrices are shared among all parties.
3. The parties compute $P = \sum_{i=0}^{p-1} P_i$ using secure computations in a finite field.

This has the advantage of being fair, secure, and very fast as additions do not require communication between the parties. However, we need P to be invertible. We can either argue that random matrices are almost surely invertible (and accepting a small risk that the transformation might not result in correct results) or we can calculate P as follows:

1. Each party chooses a random matrix P_i.
2. The matrices are shared among all parties.
3. The parties compute $P = \prod_{i=0}^{p-1} P_i$ using secure computations on the shares.

As the product of two invertible matrices is invertible, this guarantees that P is invertible too as long as no overflows take place. Unfortunately, this implies that - in the absence of overflows - we cannot argue that this choice is fair. However, this is still a relatively secure trade-off .
We continue with the explanation how to jointly choose a random monomial or diagonal matrix such as Q. A monomial matrix can be written as the product of a permutation matrix and a diagonal matrix. It is

$$Q = AD \tag{7.27}$$

The permutation matrix A can be chosen as described above, the diagonal matrix D can be interpreted as a random vector and chosen accordingly.
We close this subsection with the summarization of our transformation protocol. Figure 7.5 illustrates this protocol. Combining all introduced algorithms together, this results in the following protocol steps.

1. Each party chooses a random invertible matrix P_i, a random positive monomial matrix Q_i, a random positive diagonal matrix S_i , a random permutation matrix A_i, and a random positive vector r_i.
2. Each party securely shares its parts of M_1, M_2, b_1, b_2, and c. Each party also shares its parts of P_i, Q_i, S_i, A_i, and r_i.
3. Each party follows the described protocols. Each party performs secure computations to compute P, Q, S, A, and r.
4. Each party assembles M_1, M_2, b_1, b_2, and c.

5. Each party executes the transformations and computes

$$M'' = P \begin{pmatrix} M_1 Qr & 0 \\ M_2 Qr & 2A \\ -S & \end{pmatrix} \tag{7.28}$$

$$b'' = P \begin{pmatrix} b_1 + M_1 Qr \\ b_2 + M_2 Qr \\ -Sr \end{pmatrix} \tag{7.29}$$

$$c_s'^T = \begin{pmatrix} c^T Q & 0 & \dots & 0 \end{pmatrix} \tag{7.30}$$

6. The values $c_s'^T$, M'', and b'' are reconstructed from the shares and passed into the cloud.
7. The following LP is solved in the cloud.

$$\begin{aligned} &\min c_s'^T z_s \\ &s.t. M'' z_s = b'' \\ &z_s \geq 0 \end{aligned} \tag{7.31}$$

8. The solutions z are shared securely.
9. The value $x = Q(z - r)$ is securely distributed computed.
10. The respective owners receive their output values x.

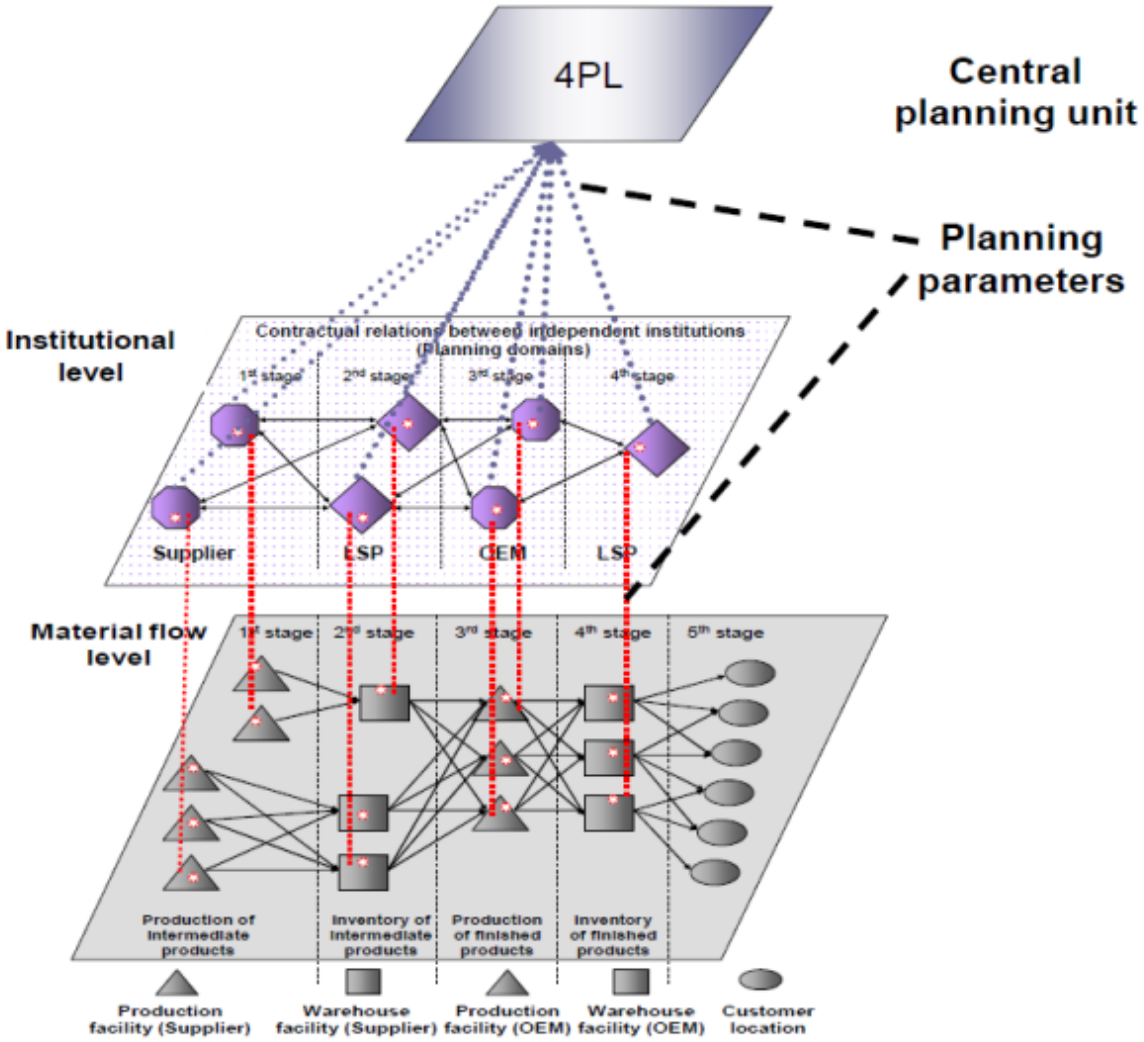


Figure 7.2: Configuration of a Supply Chain with Centralized Coordination through a Central Planning Unit (4PL)

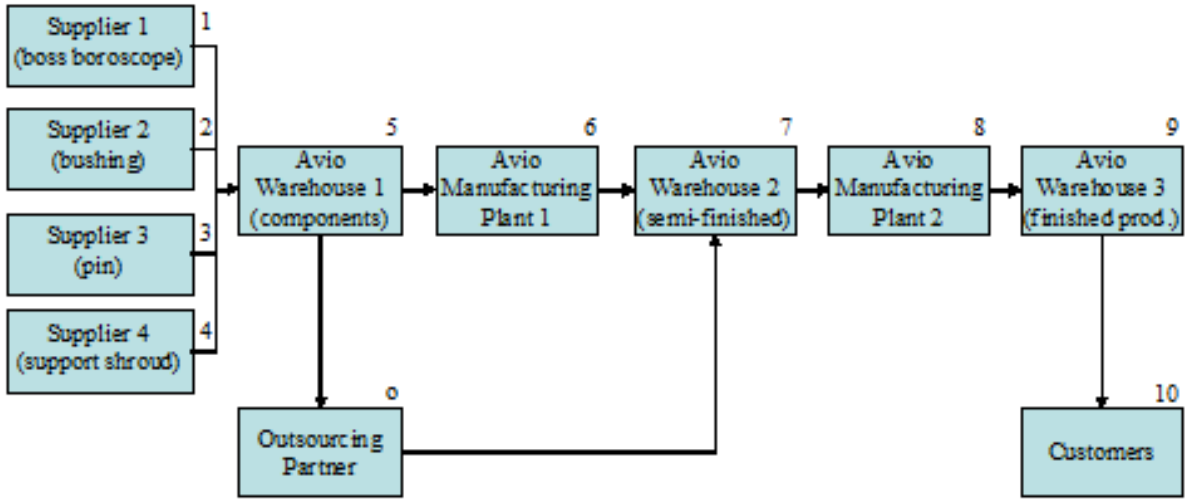


Figure 7.3: Avio's Shroud Nozzle Supply Chain

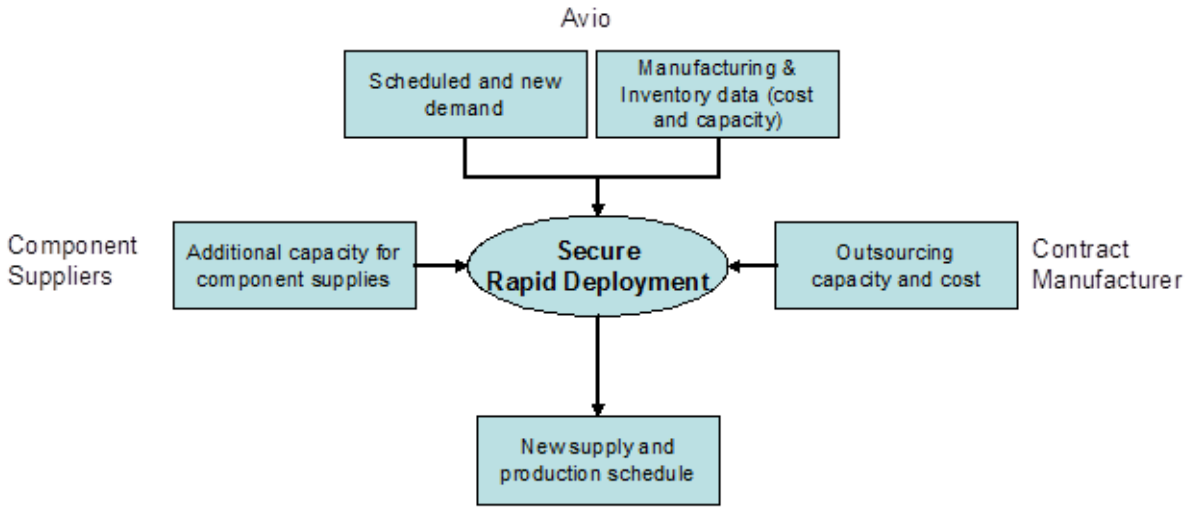


Figure 7.4: Secure Rapid Deployment Tool

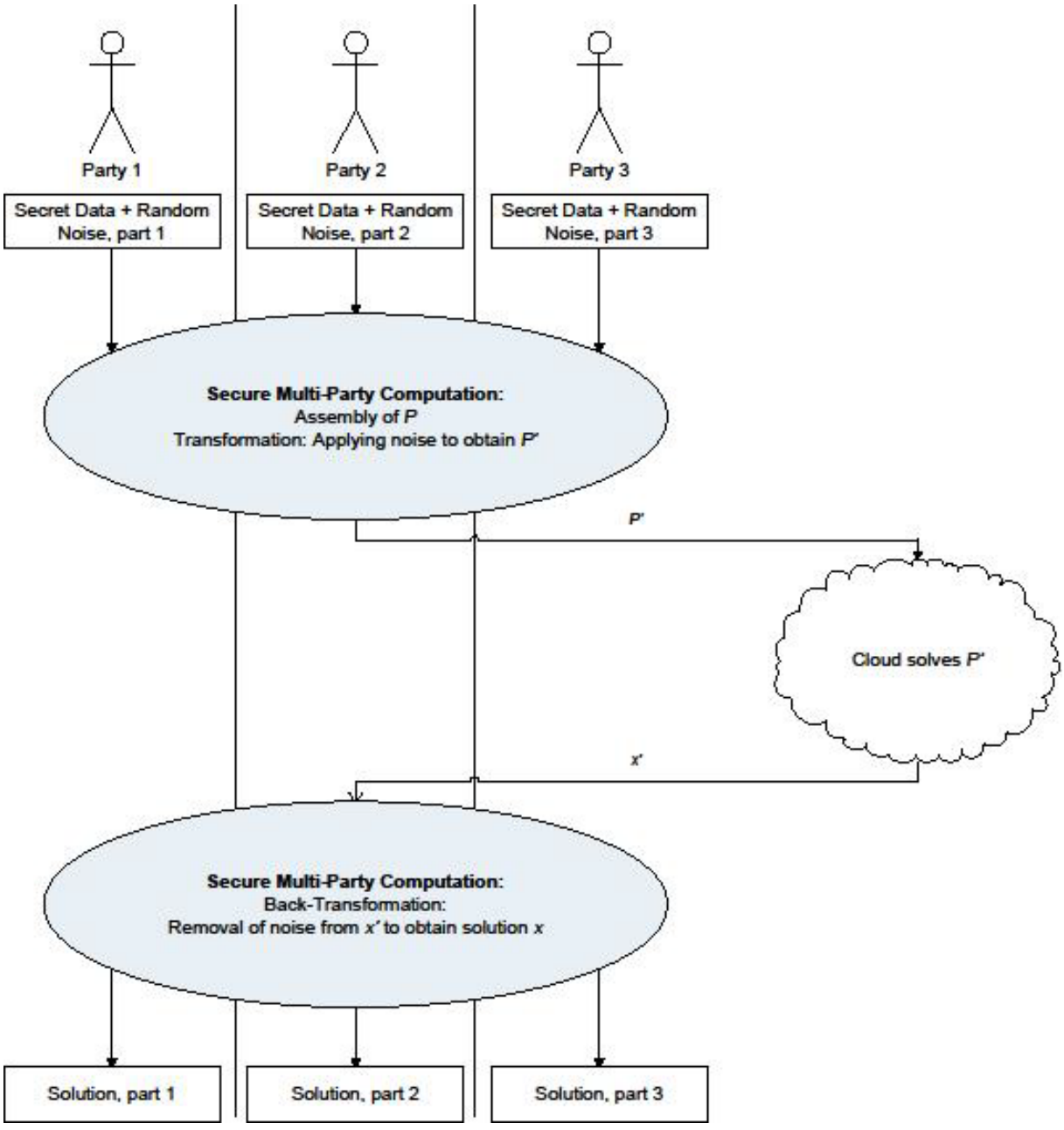


Figure 7.5: The Transformation Protocol

Chapter 8

Summary

Tools for developing secure computation applications This deliverable provides an overview of the state-of-the-art tools that help develop secure computation tools. This field is highly dynamic and every year new and improved tools are proposed. PRACTICE partners have been continually contributing to this area. Our analysis documents the status from the beginning of the PRACTICE project and collects tools of all varieties-including programming languages, libraries of reusable functionalities and support mechanisms like verifiers and databases. Most of the tools still have an academic maturity level, but we also identified one early version of an integrated toolkit with an end user community.

We present various programming languages to specify different parts of a secure computing system. For example, the SFDL language used by the Fairplay family of secure computing systems follows the design of the VHDL hardware specification language. Similar goals are achieved by the CBMC-GC compiler that takes standard ANSI C code as input and provides circuit designs similarly to Fairplay. Both tools have limitations concerning the size of the functions (to be computed) they can express. More complex applications can be presented using higher level languages that may be embedded languages or compiled languages. Examples include TASTYL, a Python-based embedded domain-specific language (DSL) used by the TASTY system. Similarly, the L1 language is based on Java and has a Java runtime as well. Both are suited for expressing various secure computing protocols.

The Sharemind runtimes employs a two-level approach. The Sharemind protocol language is a functional language used to express efficient primitive operations that are later orchestrated into algorithms and business processes using the C-like SecreC (secrecy) language. Sharemind also has a basic integrated development environment with a reusable standard library.

Developers who prefer to use secure computation as a library to enhance their existing applications can use libraries like VIFF (based on Python), FRESCO and SCAPI (both based on Java).

Even though there is a multitude of tools, there is no overview or integration path between them. Also, today's developers lack a clear understanding of which tools are best for certain jobs. The PRACTICE project will establish maturity of selected tools and also show their suitability in various application architectures that can be used in real world cloud setting.

Support tools for developers There are a number of tools that improve the experience of developing an application that supports secure computation. First, we consider verification tools that help the developer to formally verify certain security and privacy guarantees of an algorithm or protocol. These tools include CAO, CertiCrypt, EasyCrypt, DN Toolbox, CryptoVerif and ProVerif. All of them are built on strong formal foundations and provide automatic or semi-automatic proofs of security. Their main challenge lies in the complexity of using them successfully in larger applications and real-world settings. Here, the identified goal of the PRACTICE is certainly to integrate the verification tools better with other tools.

It is evident that secure computation applications work on data and the data has to be stored when it is collected or uploaded. This capability was demonstrated within SAP's HANA database that will receive improved secure computation capabilities in PRACTICE. Furthermore, the Sharemind secure computation system has existing support for relational databases and is planned to get improved support for both relational and for non-relational database backends.

Examples of tool use in applications We also discuss existing secure computation applications to analyze their tool usage, or whether they can be reused as tools. We carefully analyzed some specific applications, such as a secure auction engine and a financial benchmarking system. Both can be repurposed for different customers, but have a fixed use-case.

We also are working on several applications targeted to a more general user base. One of them allows a user to secretly share their key without any single party learning anything about the key - a useful scenario in the cloud. The other - MobiShare - allows secure cloud-assisted file transfer using mobile phones as trust anchors.

Two enterprise scenarios were also considered. First, privacy-preserving data integration and analysis. While the example application is designed to analyze the job market of a country from its tax and education records, the used statistical tool is very generic and usable in many data sharing settings. Second, a secure solution for supply chain management was described.

Within PRACTICE, we will demonstrate the how to upgrade these tools such that some of these applications, within the project scope, can be implemented significantly more securely without requiring developers to have much cryptographic knowledge.

Bibliography

[AB01] Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. In *Foundations of Software Science and Computation Structures*, pages 25–41. Springer, 2001.

[ABB+10] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, volume 6345 of *Lecture Notes in Computer Science*, pages 151–167. Springer, 2010.

[ABB+12] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 488–500. ACM, 2012.

[ABPV09] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Verifying cryptographic software correctness with respect to reference implementations. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5825 of *LNCS*, pages 37–52. Springer, 2009.

[ABPV10] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Deductive verification of cryptographic software. *ISSE*, 6(3):203–218, 2010.

[ANY09] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal cryptographic proofs: the case of bbs. *ECEASST*, 23, 2009.

[ANY12] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The case of BBS. *Sci. Comput. Program.*, 77(10-11):1058–1074, 2012.

[BAN89] Michael Burrows, Martin Abadi, and Roger M Needham. A logic of authentication. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 426(1871):233–271, 1989.

[Bar09] Manuel Barbosa. CACE Deliverable D5.2: formal specification language definitions and security policy extensions, 2009. Available from http://www.cace-project.eu.

[Bar10] Gilles Barthe. Certicrypt: Formal proofs for computational cryptography, 2010. Invited talk at Computational and Symbolic Proofs of Security (CosyProofs) 2010, http://www.lsv.ens-cachan.fr/Events/Cosyproofs10/.

[Bar11] Manuel Barbosa. CACE Deliverable D5.5: tools and documentation on integration with theoretical security validation tools, 2011. Available from http://www.cace-project.eu.

[BBGO09] Santiago Zanella Béguelin, Gilles Barthe, Benjamin Grégoire, and Federico Olmedo. Formally certifying the security of digital signature schemes. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 237–250. IEEE Computer Society, 2009.

[BBN06] Peter Bogetoft, Jens-Martin Bramsen, and Kurt Nielsen. Balanced benchmarking. *International Journal of Business Performance Management*, 2006.

[BCD+09a] Peter Bogetoft, Dan Lund Christensen, Ivan Damgard, Martin Geisler, Thomas Jakobsen, Mikkel Kroigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *13th International Conference on Financial Cryptography and Data Security*, 2009.

[BCD+09b] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 257–266. ACM, 2008.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 1995.

[BF01] Dan Boneh and Matthew Franklin. Efficient generation of shared rsa keys. *J. ACM*, 48(4):702–722, 2001.

[BFM+08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specfication Language*. CEA LIST and INRIA, 2008.

[BGB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101. ACM, 2009.

[BGHZB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Heidelberg, 2011. Springer.

[BGZL11] Gilles Barthe, Benjamin Grégoire, Santiago Zanella Béguelin, and Yassine Lakhnech. Beyond provable security. verifiable ind-cca security of oaep. In *Topics in Cryptology – CT-RSA 2011, The Cryptographers' Track at the RSA Conference 2011*, Lecture Notes in Computer Science. Springer, 2011.

[BHB+10] Gilles Barthe, Daniel Hedin, Santiago Zanella Beguelin, Benjamin Gregoire, and Sylvain Heraud. A machine-checked formalization of sigma-protocols. *Computer Security Foundations Symposium, IEEE*, 0:246–260, 2010.

[BJL12] Dan Bogdanov, Roman Jagomägis, and Sven Laur. A Universal Toolkit for Cryptographically Secure Privacy-Preserving Data Mining. In Michael Chau, G. Alan Wang, Wei Thoo Yue, and Hsinchun Chen, editors, *Intelligence and Security Informatics - Pacific Asia Workshop, PAISI'12, Kuala Lumpur, Malaysia, May 29, 2012. Proceedings*, volume 7299 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2012.

[BJST08] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 87–99, Tokyo, Japan, March 2008. ACM.

[BK13] Dan Bogdanov and Aivo Kalu. Pushing back the rain—how to create trustworthy services in the cloud. *ISACA Journal*, (3):49–51, 2013.

[BKLPV13] Dan Bogdanov, Liina Kamm, Sven Laur, and Pille Pruulmann-Vengerfeldt. Secure multi-party data analysis: end user validation and practical experiments. Cryptology ePrint Archive, Report 2013/826, 2013. http://eprint.iacr.org/.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop, IEEE*, pages 0082–0082. IEEE Computer Society, 2001.

[Bla05] Bruno Blanchet. A computationally sound mechanized prover for security protocols. Cryptology ePrint Archive, Report 2005/401, 2005. http://eprint.iacr.org/.

[BLR13a] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic language for privacy-preserving applications. In *Proceedings of the First ACM Workshop on Language Support for Privacy-enhancing Technologies*, PETShop '13, pages 23–26, New York, NY, USA, 2013. ACM.

[BLR13b] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-Polymorphic Programming of Privacy-Preserving Applications. Cryptology ePrint Archive, Report 2013/371, 2013. http://eprint.iacr.org/.

[BLT13] Dan Bogdanov, Sven Laur, and Riivo Talviste. Oblivious Sorting of Secret-Shared Data. Technical Report T-4-19, Cybernetica, http://research.cyber.ee/, 2013.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In Sushil Jajodia and Javier Lopez, editors, *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer Berlin / Heidelberg, 2008.

[BM89] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and spplications. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 547–557. Springer, 1989.

[BMM10] Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. In *FSTTCS*, volume 8, pages 352–363, 2010.

[BMP+12] M. Barbosa, A. Moss, D. Page, N. Rodrigues, and P. Silva. Type checking cryptography implementations. In *FSEN'11*, volume 7141 of *LNCS*, pages 316–334. Springer, 2012.

[BN08] Peter Bogetoft and Kurt Nielsen. DEA based auctions. *European Journal of Operational Research*, 184:685–700, 2008.

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[BO11] Peter Bogetoft and Lars Otto. *Benchmarking with DEA, SFA, and R*. SPRINGER, New York, 2011.

[Bog13] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BP05] Manuel Barbosa and Dan Page. On the automatic construction of indistinguishable operations. In *Cryptography And Coding*, volume 3796 of *LNCS*, pages 233–247. Springer, 2005.

[BP06] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In Cynthia Dwork, editor, *CRYPTO'06*, volume 4117 of *Lecture Notes on Computer Science*, pages 537–554, Santa Barbara, CA, August 2006. Springer Verlag.

[BP10] Bruno Blanchet and David Pointcheval. The computational and decisional Diffie-Hellman assumptions in CryptoVerif. In *Workshop on Formal and Computational Cryptography (FCC 2010)*, Edimburgh, United Kingdom, July 2010. To appear.

[BPFV10] Manuel Barbosa, J Pinto, J-C Filliâtre, and Bárbara Vieira. A deductive verification platform for cryptographic software. *Electronic Communications of the EASST*, 33, 2010.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, Heidelberg, 2006. Springer.

[BTW12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis (short paper). In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security. FC'12*, pages 57–64, 2012.

[Bur14] Sergiu Bursuc. Formal multi-party computation, jan 2014.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CcLs94] A Charnes, William W cooper, AY Lewin, and Lawrence M seiford. *Data Envelopment Analysis: Theory, Methodology and Application*. Kluwer Academic Publishers, 1994.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.

[CGG05] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127 – 155, 2005.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In Kwangjo Kim, editor, *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[DK09] Rafael Deitos and Florian Kerschbaum. Improving practical performance on secure and private collaborative linear programming. In *DEXA Workshops*, pages 122–126, 2009.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[dMB08] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin - Heidelberg, 2008. 10.1007 978-3-540-78800-3-24.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[Dre13] Jannik Dreier. *Formal Verification of Voting and Auction Protocols*. PhD thesis, Université Joseph Fourier, 2013.

[DS81] Dorothy E Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[EFLL12] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.

[FCP+12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christoph Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45 – 51, January 2012.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proc. Sympos. Appl. Math., Vol. XIX*, pages 19–32. Amer. Math. Soc., Providence, R.I., 1967.

[Gar08] Gartner. Seven cloud-computing security risks. http://www.infoworld.com/d/security-central/gartner-seven-cloud-computing-security-risks-853, July 2008.

[GD98] Steven Gjerstad and John Dickhaut. Price formation in double auctions. *Games and Economic Behavior*, 22(1):1–29, 1998.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, March 2010.

[GH11] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Sciences*, pages 129–148. Springer, 2011.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 101–111. ACM, 1998.

[Hal05] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[Han14] Amazon webservices and sap. http://aws.amazon.com/sap/, 2014.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 772–783. ACM, 2012.

[HKS+] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 451–462. ACM.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[Jag08] Roman Jagomägis. A programming language for creating privacy-preserving applications. Bachelor's thesis. University of Tartu, 2008.

[Jag10] Roman Jagomägis. SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[KBLV13] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, 2013.

[KDSB09] Florian Kerschbaum, Daniel Dahlmeier, Axel Schröpfer, and Debmalya Biswas. On the practical importance of communication complexity for secure multi-party computation protocols. In *24th ACM Symposium on Applied Computing*, 2009.

[Kle04] Paul Klemperer. *Auctions: Theory and Practice*. Princeton University Press, 2004.

[KM11] Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 556–571. Springer, 2011.

[KO62] Anatoly Karatsuba and Yuri Petrovich Ofman. Multiplication of many-digital numbers by automatic computers. *SSSR Academy of Sciences*, 145:293–294, 1962.

[Kra96] Hugo Krawczyk. Skeme: A versatile secure key exchange mechanism for internet. In *Network and Distributed System Security, 1996., Proceedings of the Symposium on*, pages 114–127. IEEE, 1996.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *Cryptology*

and Network Security, 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings, volume 5888 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2009.

[KSS10] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. From dust to dawn: Practically efficient two-party secure function evaluation protocols and their modular design. *IACR Cryptology ePrint Archive*, 2010:79, 2010.

[KSS13] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 21(2):283–315, 2013.

[Kue04] Andreas Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 50–57. IEEE, 2004.

[KW13] Liina Kamm and Jan Willemson. Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. Cryptology ePrint Archive, Report 2013/850, 2013. http://eprint.iacr.org/.

[Ler06] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL 06*, pages 42–54. ACM Press, 2006.

[Lew07] Jeff Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In Peng Ning, Vijay Atluri, Virgil D. Gligor, and Heiko Mantel, editors, *Proceedings of the 2007 ACM workshop on Formal methods in security engineering, FMSE 2007, Fairfax, VA, USA, November 2, 2007*, page 41. ACM, 2007.

[Low96] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996.

[LPPR13] Peeter Laud, Alisa Pankova, Martin Pettai, and Jaak Randmets. Specifying sharemind's arithmetic black box. In *Proceedings of the First ACM Workshop on Language Support for Privacy-enhancing Technologies*, PETShop '13, pages 19–22, New York, NY, USA, 2013. ACM.

[LTW13] Sven Laur, Riivo Talviste, and Jan Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security*, volume 7954 of *LNCS*, pages 84–101. Springer, 2013.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.

[Mac] Greg MacSweeney. The top 9 most costly financial services data breaches.

[MCB06] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 532–535. ACM, 2006.

[MCJB05] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K Brayton. Fraigs: A unifying representation for logic synthesis and verification. Technical report, ERL Technical Report, 2005.

[Mil04] Paul Milgrom. *Putting Auction Theory to Work*. Cambridge Univ Pr, 2004.

[MM10] Claude Marché and Yannick Moy. *Jessie Plugin Tutorial*. INRIA, 2010.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302. USENIX, 2004.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In Omer Reingold, editor, *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

[Now07] David Nowak. A framework for game-based security proofs. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *Information and Communications Security*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333. Springer Berlin / Heidelberg, 2007.

[Now08] David Nowak. On formal verification of arithmetic-based cryptographic primitives. In Pil Joong Lee and Jung Hee Cheon, editors, *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*, volume 5461 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2008.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In S. Rao Kosaraju, editor, *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*, pages 448–457. ACM/SIAM, 2001.

[NS78] Roger M Needham and Michael D Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[NS87] R M Needham and M D Schroeder. Authentication revisited. *SIGOPS Oper. Syst. Rev.*, 21(1):7–7, January 1987.

[NT07] Kurt Nielsen and T Toft. Secure Relative Performance Scheme. *Proc. of Workshop on Internet and Network Economics 2007, LNCS 4858*, 2007.

[NZ10] David Nowak and Yu Zhang. A calculus for game-based security proofs. In Swee-Huay Heng and Kaoru Kurosawa, editors, *Provable Security - 4th International Conference, ProvSec 2010, Malacca, Malaysia, October 13-15, 2010. Proceedings*, volume 6402 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2010.

[OR87] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *ACM SIGOPS Operating Systems Review*, 21(1):8–10, 1987.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[Pau98] Lawrence C Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6(1):85–128, 1998.

[PBS12] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. The design and implementation of a two-party protocol suite for Sharemind 3. Technical Report T-4-17, Cybernetica, http://research.cyber.ee/, 2012.

[PRZB11] Raluca Ada Poppa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Proctecting confidentiality with encrypted query processing. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85 – 100, New York, NY, USA, 2011. ACM.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[Pul13] Pille Pullonen. Actively Secure Two-Party Computation: Efficient Beaver Triple Generation. Master's thesis, Institute of Computer Science, University of Tartu, 2013.

[Reb10] Reimo Rebane. An integrated development environment for the SecreC programming language. Bachelor's thesis. University of Tartu, 2010.

[Reu11] Reuters. Sony suffers second major user data theft. http://www.reuters.com/article/2011/05/02/us-sony-idUSTRE73R0Q320110502, May 2011.

[Rot02] A Roth. The economist as engineer: Game theory, experimentation, and computation as tools for design economics. *Econometrica*, 2002.

[She92] John C. Shepherdson. Unfold/fold transformations of logic programs. *Mathematical Structures in Computer Science*, 2(02):143–157, 1992.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs, Nov. 30 2004. Manuscript, http://www.shoup.net/papers/games.pdf.

[Tal11] Riivo Talviste. Deploying secure multiparty computation for joint data analysis—a case study. Master's thesis, Institute of Computer Science, University of Tartu, 2011.

[The11] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, April 1, 2011. http://coq.inria.fr/refman/.

[TKMZ13] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 289–300. VLDB Endowment, 2013.

[Tof07] Tomas Toft. *Primitives and Applications for Multi-party Computation*. PhD dissertation, University of Aarhus, Denmark, BRICS, Department of Computer Science, 2007.

[Vai09] Jaideep Vaidya. Privacy-preserving linear programming. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, volume 5867 of *Lecture Notes in Computer Science*, pages 2002–2007. Springer, 2009.

[Ver12] Verizon. 2012 data breach investigations report. 2012. http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2012-ebk_en_xg.pdf.

[VIF] VIFF Development Team. VIFF, the virtual ideal functionality framework. http://viff.dk/.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.