Realia et naturalia

DISSERTATIONES
MATHEMATICAE
UNIVERSITATIS
TARTUENSIS
83

DAN BOGDANOV

Sharemind: programmable secure computations with practical applications

TARTU ÜLIKOOL · UNIVERSITAS TARTUENSIS · 1632

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
83

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
83

DAN BOGDANOV

Sharemind: programmable secure computations with practical applications

UNIVERSITY OF TARTU
PRESS
1632

Institute of Computer Science, Faculty of Mathematics and Computer Science, University of Tartu, Estonia

Dissertation is accepted for the commencement of the degree of Doctor of Philosophy (PhD) on January 21th, 2013 by the Council of the Institute of Computer Science, University of Tartu.

Supervisor:

Dr. Tech.	Sven Laur University of Tartu Tartu, Estonia

Opponents:

Prof. PhD.	Nigel P. Smart University of Bristol Bristol, United Kingdom
Dr. Ir.	Berry Schoenmakers Eindhoven University of Technology Eindhoven, Netherlands

The public defense will take place on February 28th, 2013 at 16:15 in Liivi 2-403.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu.

ISSN 1024–4212
ISBN 978–9949–32–216–9 (print)
ISBN 978–9949–32–217–6 (PDF)

University of Tartu Press
www.tyk.ee
Order No. 17

Contents

PUBLICATIONS INCLUDED IN THIS THESIS

The publications included in this thesis describe the main results achieved with the Sharemind system developed by the author.

1. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) Proceedings of the 13th European Symposium on Research in Computer Security, ESORICS '08. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008).

2. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. International Journal of Information Security 11(6), 403–418 (2012)

3. Bogdanov, D., Jagomägis, R., Laur, S.: A universal toolkit for cryptographically secure privacy-preserving data mining. In: Chau, M., Wang, G.A., Yue, W.T., Chen, H. (eds.) Proceedings of the Pacific Asia Workshop on Intelligence and Security Informatics, PAISI '12. Lecture Notes in Computer Science, vol. 7299, pp. 112–126. Springer (2012).

PUBLICATIONS NOT INCLUDED IN THIS THESIS

These publications by the author describe various aspects of SHAREMIND, but are not included in this thesis. Some of the results achieved in these works are referred to by the thesis. The list includes both papers and technical reports not yet published as papers.

1. Bogdanov, D., Sassoon, R.: Privacy-preserving collaborative filtering with sharemind. Tech. Rep. T-4-2, Cybernetica AS, Tartu, `http://research.cyber.ee/`. (2008).

2. Bogdanov, D., Talviste, R.: A Comparison of Software Pseudorandom Number Generators. In: Cap, C. (ed.) Proceedings of Third Baltic Conference on Advanced Topics in Telecommunication - BaSoTi 2009. pp. 61–71. Universität Rostock, Wissenschaftsverbund IuK (2009).

3. Bogdanov, D., Laur, S.: The design of a privacy-preserving distributed virtual machine. In: Kaklamanis, C. (ed.) Collection of AEOLUS theoretical findings. Deliverable 1.0.6, pp. 269–280. Published online at `http://aeolus.ceid.upatras.gr/deliverables` (2010).

4. Bogdanov, D., Kamm, L.: Constructing privacy-preserving information systems using secure multiparty computation. Tech. Rep. T-4-13, Cybernetica AS, Tartu, http://research.cyber.ee/. (2011).

5. Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis - (short paper). In: Keromytis, A.D. (ed.) Proceedings of the 16th International Conference on Financial Cryptography and Data Security, FC '12. Lecture Notes in Computer Science, vol. 7397, pp. 57–64. Springer (2012).

OTHER PUBLISHED WORK OF THE AUTHOR

These publications by the author are on various topics in information security.

1. Bogdanov, D., Crispino, M.V., Čyras, V., Lapin, K., Panebarco, M., Zuliani, F.: Virtual World Platform VirtualLife: P2P, Security, Rule of Law and Learning Support. In: Proceedings of 2009 NEM Summit "Towards Future Media Internet". Distributed as an eBook. NEM Initiative (2009).

2. Ahmed, A.S., Bogdanov, D.: A Model for Automatically Evaluating Trust in X.509 Certificates. Tech. Rep. T-4-11, Cybernetica AS, Tartu, http://research.cyber.ee/. (2010).

3. Bogdanov, D., Livenson, I.: VirtualLife: Secure Identity Management in Peer-to-Peer Systems. In: Daras, P., Ibarra, O.M., Akan, O., Bellavista, P., Cao, J., Dressler, F., Ferrari, D., Gerla, M., Kobayashi, H., Palazzo, S., Sahni, S., Shen, X.S., Stan, M., Xiaohua, J., Zomaya, A., Coulson, G. (eds.) Proceedings of the 1st International ICST Conference on User Centric Media, UCM '10. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 40, pp. 181–188. Springer (2010).

ABSTRACT

Information about the health, personal beliefs and wealth of an individual is considered sensitive and special care needs to be taken to ensure its privacy. Organizations processing such data must take precautions to prevent its leakage to unauthorized parties. However, at the same time, both public and private organizations are motivated to share information to make better decisions.

Secure multiparty computation is a cryptographic method for securely processing data among several parties. While the first protocols were proposed in the 1980s, the first practical implementations were developed early this century.

In this thesis, we present SHAREMIND—a complete solution for building data processing applications that use secure multiparty computation. The author is both the designer and implementer of SHAREMIND. The thesis describes secure computation and storage methods, performance measurements, application development techniques and introduces several practical applications.

In the thesis, we describe and analyze the real-world threat model of secure multiparty computation systems. We present a suite of protocols that is highly optimized for secure data collection and processing in this model.

We introduce a secure database design based on secret sharing and describe techniques for querying and updating this database. We show how data in a secret-shared database can be processed using secure multiparty computation.

We present a performance analysis of the SHAREMIND system and its computation protocols. SHAREMIND achieves high performance in laboratory and cloud settings, performing hundreds of thousands of secure operations each second.

We have created two languages for programming the SHAREMIND system—a low-level assembly language and a high-level imperative programming language called SECREC. Both allow the user to combine public and private computations.

Finally, we describe and benchmark several application prototypes and introduce one real-world application of SHAREMIND—a financial information analysis tool. This tool is, to our knowledge, the world's first secure multiparty computation system running on the public Internet.

CHAPTER 1

INTRODUCTION

> Should everyone who self-discloses information lose control over that information forever, and have no say about whether and when the Internet forgets this information? Do we want a future that is forever unforgiving because it is unforgetting? (Viktor Mayer-Schönberger, *Delete: The Virtue of Forgetting in the Digital Age*)

1.1 Why do we need secrets?

Digital computing technology has enabled the collection and processing of information on a global scale. Every organization has the possibility to gather data from its environment and study it in order to learn new ways for achieving its goals. This includes monitoring the behavior of people and companies to find patterns that could help predict their future interactions with the organization.

Retail companies study the shopping habits of customers to increase sales. Financial institutions and insurance providers study people and companies to assess whether they will default on loans or trigger contingencies that lead to insurance payments. Governments analyze their populations to understand trends in migration, employment and welfare.

These actions are driven by our desire for efficiency and survivability. Indeed, by understanding the future, we can adjust our own strategies to reach this future in a more advantageous position than our competition. For example, having the best information about one's surroundings may become an essential requirement of survival for a company.

However, as most companies now understand the value of information, they have become more cautious about sharing it even if such sharing would ultimately benefit them. Such fears are caused by the simple truth that once you disclose data in digital form, it becomes trivially and perfectly copiable and thus the owner of

the data loses control over what else can be done with it. Indeed, attempts to control the use of digital content through techniques like digital rights management have not been very successful.

The problem is also relevant when personal data are considered. People may have many reasons why they would like to have their past actions or statements forgotten. These concerns are typically an afterthought when those past actions or statements have already caused harm in the present. However, with the systematic tracking and profiling of individuals using digital services, potentially harmful information is continuously collected and placed outside the reach of the individuals themselves.

To conclude, the main risk with digital personal information and business data is that initially, they may be collected for a valid purpose. However, as they are stored for an unspecified amount of time and out of the reach of their owners, they can be used for any other purpose without the original owner being able to stop it. In the digital world, the existence of secrets is being justified with the lack of control over data.

1.2 Background and claims of this work

Cryptography is the mathematical science of secrets. Cryptography has given us encryption schemes as a tool for confidentiality, digital signatures as a tool for non-repudiation, message authentication codes and hashing as tools for checking integrity and many other useful primitive operations that are used in digital communications.

However, many of these primitives are final—once you transform data using a cryptographic function, the resulting form usually cannot be meaningfully modified anymore without reversing the transformation. There are exceptions—cryptographic functions whose output is *homomorphic*. Such functions can be used to perform *secure computation*—to manipulate data even when it has been encrypted or digitally signed while preserving the security guarantees offered by the cryptographic primitive.

Secure computation has been researched for at least three decades. Recent developments in the field are becoming more and more efficient to the point that we can use the technology in real-life applications.

Research on secure multiparty computation focuses on the complexity of protocols, the strength of security guarantees and efficient solutions for particular problems. However, the results are rarely taken outside the academical environment to solve real world problems in real world settings with actual stakeholders.

This work approaches secure computation from a more practical standpoint. We claim that secure multiparty computation can be made suitable for use in ev-

eryday applications. More specifically, we make three claims. First, we claim that secure multiparty computation can provide building blocks for creating complex data processing tools without the need to design new protocols for each task. Second, a secure multiparty computation implementation can be made fast enough for processing databases with millions of records on easily obtainable hardware. Third, we claim that secure multiparty computation can be packaged as a tool so that it can successfully be applied by specialists in any field who do not have the training of a cryptographer.

To achieve these goals, we take secure computation all the way from elegant mathematical constructions to a deployment at the customer with their individual wishes and requests. This work focuses on how to deliver secure computation capability to applications, what kind of tools are needed to implement the applications and their business logic, and what kind of processes have to be followed to ensure that the assumptions we bring from the theoretical solutions also hold in the final application.

1.3 Thesis outline and contributions of the author

The author is the designer, architect and implementer of the SHAREMIND secure multiparty computation system. This thesis provides the rationale and decisions that directed the design of SHAREMIND. The author has also developed several tools to measure the performance of SHAREMIND and direct its optimization efforts. This thesis describes how the system is constructed, explains the security guarantees and measures performance. The author developed the secure virtual machine that can execute cryptographic protocols to perform secure computations, integrated this machine with a secure database and created interfaces for end-user applications. The author has designed protocols that enable the secure virtual machine to collect and store data and has actively participated in the design of computation protocols.

The author of the thesis has designed developer tools such as the SECREC programming language to simplify the creation of SHAREMIND applications and has developed the necessary interfaces for integrating these tools into the SHAREMIND system. The programming language interpreters and compilers have been implemented in co-operation with students. Finally, the author has analyzed the real-world deployment issues, including the necessity for software development procedures, user interfaces, maintenance and economic aspects. This analysis is based on prototype applications that the author has designed and implemented with co-authors.

In the following, we introduce the thesis chapter by chapter and describe the author's contribution to different parts of SHAREMIND and the associated tools.

Chapter 2 describes secure computation and provides an overview of general-purpose secure computation implementations. The thesis presents different secure computation paradigms and provides a survey of published implementations of general-purpose secure computation systems.

Chapter 3 introduces the SHAREMIND secure computation system and its design decisions. The thesis describes the design goals and the security model of SHAREMIND secure computation protocols. The work continues with an explanation of the storage model and data management protocols of SHAREMIND and an overview of the software implementation of the system.

The chapter refers to the following papers included in this thesis.

1. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) Proceedings of the 13th European Symposium on Research in Computer Security, ESORICS '08. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008).

   This paper contains the secure computation protocols designed by the author and the co-authors. The author's contributions include the software architecture of the SHAREMIND system, the implementation of the vectorized secure computation protocols, the design and implementation of the networking layer, secure database, controller library, profiling mechanism and performance analysis tools. The author also conducted benchmarking experiments and analyzed performance results.

2. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. International Journal of Information Security 11(6), 403–418 (2012).

   This paper proposes a new set of secure computation protocols that are significantly more efficient than the ones in the original paper [25]. The author collaborated in the design and implementation of the new protocols in the SHAREMIND system. The author's personal contributions include the design of the new secure protocol implementation interface that supports secure batch execution of larger vector operations, the benchmarking and performance analysis of the new protocols in comparison with the old protocols, the design, implementation and benchmarking of the privacy-preserving k-means clustering algorithm.

The chapter also refers to several other works of the author.

1. A preliminary version of the SHAREMIND design was proposed in the author's Master's thesis [20].

2. The overall architecture of SHAREMIND is described in a technical report [24]. The report describes the design of the storage and computation features in SHAREMIND that are designed and implemented by the author.

Chapter 4 is an in-depth discussion on the performance and practical feasibility of secure computations with SHAREMIND. The thesis lists the measurement and analysis methods used to build the performance model of secure computations. Several different aspects of the performance are considered, including computational complexity and communication complexity. The thesis describes experiments that validate the analysis.

Parts of the chapter are based on joint work with students:

1. The performance analysis of SHAREMIND with different pseudorandom generators is joint work with Riivo Talviste [29].

2. The performance analysis of SHAREMIND in different network settings and in a cloud deployment is joint work with Reimo Rebane. The full analysis appears in [106].

Chapter 5 explains the programming model of the SHAREMIND system. The thesis explains the choices that have driven the design of the SHAREMIND programming languages. Two languages—the SHAREMIND assembly language and the SECREC language—have been implemented as joint work of the author and his students. The thesis introduces both languages and gives an overview of their features and use.

The thesis discusses how to ensure that privacy is preserved in SHAREMIND programs and introduces a new analysis tool for detecting privacy leaks in SECREC programs. The thesis also shows how an integrated development environment for the SECREC language can simplify the development of secure applications. The chapter ends with a survey of other secure multiparty computation languages.

Parts of this chapter are based on joint work with students:

1. The design and implementation of the SHAREMIND assembly language interpreter is joint work with Roman Jagomägis [70].

2. The design and implementation of the SECREC language compiler is joint work with Roman Jagomägis [71].

3. The design of the SECREC privacy leak analyzer is joint work with Jaak Ristioja [107].

4. The design and implementation of the SECRECIDE integrated development environment is joint work with Reimo Rebane [105].

Chapter 6 explains how to use the SHAREMIND framework in application development and presents a list of practical secure computation applications that have been built using the SHAREMIND framework. The thesis gives guidance for developing secure applications that make use of secure computing. Several example applications and algorithms are then discussed, including the first real-world application of SHAREMIND.

The chapter gives an overview of the application. The detailed design of data mining applications can be found in the following paper that is also a part of this thesis.

1. Bogdanov, D., Jagomägis, R., Laur, S.: A universal toolkit for cryptographically secure privacy-preserving data mining. In: Chau, M., Wang, G.A., Yue, W.T., Chen, H. (eds.) Proceedings of the Pacific Asia Workshop on Intelligence and Security Informatics, PAISI '12. Lecture Notes in Computer Science, vol. 7299, pp. 112–126. Springer (2012).

   The paper presents SHAREMIND as a universal toolkit for creating privacy-preserving data mining applications based on secure multiparty computation. The author collaborated with the co-authors on the design, implementation and optimization of the secure frequent itemset mining algorithms presented in the paper. The author's personal contributions include the presentation of the generic data mining framework, descriptions of the deployment and optimization options, the comparison of SHAREMIND to other secure computation frameworks. The author also designed and implemented the performance testing tools and performed benchmarking on the SHAREMIND implementations of the algorithms

Parts of this chapter are based on the following joint works:

1. The proposal of using SHAREMIND-style secure computation for frequent itemset mining, association rule mining and collaborative filtering comes from a technical report jointly authored with Richard Sassoon [28].

2. The method for collecting secret-shared data in web applications has been jointly developed with Riivo Talviste. A detailed description of the technique is given in [122].

3. The design, implementation, deployment and maintenance of the first real-world SHAREMIND application is joint work with Riivo Talviste. The technical description of the application appears in [123]. Another study of the application together with end-user feedback appears in [30].

CHAPTER 2

SECURE COMPUTATION IN PRACTICE

2.1 Overview of practical secure computation systems

In this work, we focus on general-purpose secure computation frameworks that can easily be tailored for new algorithms and applications. There are task-specific protocols and implementations, but our goal is to show that general-purpose systems can be made efficient enough for practical use. We also focus on systems designed for data analysis rather than ones specifically designed for a single task like voting or auctions.

The theory of secure computation is significantly older than practice. The concept of secure function evaluation was introduced by Yao in 1982 [129]. The first practical implementation work on garbled circuits was done over twenty years later when Fairplay was introduced in 2004 [95]. Similarly, multiparty solutions based on secret sharing were proposed in 1987 [40, 64, 17], but practically feasible implementations were not demonstrated before 2006 in Denmark [32]. Since then, several secure computation frameworks have been developed. They differ in their goals, security guarantees, efficiency and programming paradigms. Table 2.1 gives a general overview of the published secure computation frameworks. The information has been collected both from public sources and personal communication with the researchers who created the systems. This overview will not focus on security issues, but instead on deployment models and efficiency.

2.2 Introduction to circuits

Circuits are used as a model of computation in digital electronics and computational complexity theory. A circuit is a graph where edges are called *wires* and vertices are called *gates*. Wires carry data values. Gates perform operations on the values coming from *input wires* and put the results on the *output wires*. While

Framework	Project started	Techniques	References
Fairplay	2003, Israel	Yao circuits	[95, 57]
SCET	2004, Denmark	secret sharing	[32]
SHAREMIND	2006, Estonia	secret sharing	[20, 25, 114]
FairplayMP	2006, Israel	Yao circuits + secret sharing	[15, 58]
SMCR	2006, Denmark	secret sharing	[31]
VIFF Passive	2007, Denmark	secret sharing	[61, 127]
VIFF Active	2008, Denmark	secret sharing	[46, 127]
VIFF Paillier	2008, Denmark	homomorphic encryption	[61, 127]
VIFF Orlandi	2008, Denmark	secret sharing, additively homomorphic encryption and additively homomorphic commitments	[49, 127]
SEPIA	2008, Switzerland	secret sharing	[35, 112]
TASTY	2009, Germany	Yao circuits and additively homomorphic encryption	[66, 124]
VMCrypt	2010, USA	Yao circuits	[94]

Table 2.1: General-purpose secure multiparty computation frameworks.

in *Boolean circuits* wires carry bit values, this is not an inherent limitation as we can also construct arithmetic circuits with wires that carry integer data.

Formally, we define a circuit as an directed acyclic graph (G, W) where G is the set of gates and W is the set of wires. A gate $g \in G$ has input wires and output wires. While binary gates with two input wires and one output wire are most popular, gates with an arbitrary number of inputs may be useful in certain applications. A wire $(g_i, g_j) \in W$ represents a connection between an output gate $g_i \in G$ and an input gate $g_j \in G$. Wires that do not originate from a gate within the circuit, provide external input for the whole circuit and output wires that do not lead to a gate within the circuit hold the computational result after the evaluation of the circuit. We currently omit the details on composing of circuits by connecting output wires to the input wires of other circuits. Figure 2.1 gives an example of a simple circuit that computes the greater-than-or-equal function on two one-bit inputs u and v with the result on wire w.

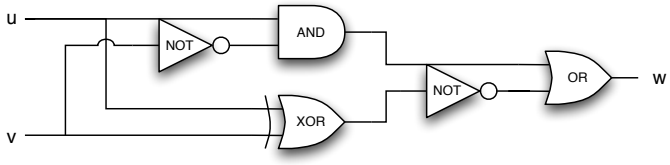


Figure 2.1: A circuit evaluating greater-than-or-equal-to on one-bit inputs.

A circuit is executed as follows. The input values are written on the input wires and gates are executed one-by-one or in parallel. A gate can be executed, if all inputs for a gate are available. During execution, the inputs of a gate are gathered, the function of the gate is evaluated on the inputs and the outputs are written on the respective wires. When all gates are executed, the result can be read from the wires.

2.3 Two-party computation using garbled Boolean circuits

Circuits are useful for secure function evaluation, as their strict mathematical structure provides a clever way to hide the values transferred on the wires during evaluation. This approach was first proposed by Yao in 1982 [129]. The original solution used an interactive protocol for evaluating each gate, but also provided hints on a non-interactive version. Later, the interaction requirement was lowered with the *circuit garbling* approach that requires interactions during the preparation of the garbled circuit, but not during the actual execution. Yao's

techniques have been extensively studied and used in theoretical work and applications alike [64, 108, 76, 82, 14].

The goal of circuit garbling is to build a circuit that computes a function f and does not leak the values on the wires in the process. The core idea behind the garbling process is to transform a circuit by replacing the values on the wires with random bit strings and use a pseudorandom function in a gate to derive the output bit string from the two input bit strings. For efficiency, several implementations have used hash functions (Fairplay [95], FairplayMP [15], VMCrypt [94]), but a symmetric encryption scheme can be used as well. While the evaluation of circuits garbled using hash functions is very efficient, it is hard to give a formal proof of security for such constructions. Usually, an implementation uses a fixed hash function such as SHA-1 that cannot be modeled using a pseudorandom function family. However, the latter is required for a formal proof.

We will now present a modern view on secure function evaluation via circuit garbling. First, we will discuss solutions that provide security against a passive adversary. Assume, that parties $\mathcal{P}_1$ and $\mathcal{P}_2$ have agreed to evaluate a function f and have agreed to the structure of the respective Boolean circuit. The process of evaluating this circuit with two parties is shown in Figure 2.2. First, $\mathcal{P}_1$ constructs the circuit, encodes its inputs and passes it to $\mathcal{P}_2$. $\mathcal{P}_2$ then uses oblivious transfer to learn the garbled versions of its inputs from $\mathcal{P}_1$. After that, $\mathcal{P}_2$ evaluates the circuit and learn the resulting value that it may pass to $\mathcal{P}_1$. This model is implemented by Fairplay and VMCrypt, among other systems.

Protocols based on Boolean circuits can also be extended to more than two parties [12, 98, 47]. One implementation of such protocols is the FairplayMP system [15]. Parties are divided into *input parties* ($\mathcal{IP}$), *computing parties* ($\mathcal{CP}$) and *result parties* ($\mathcal{RP}$). Input parties provide their masked input bits to the result parties and share the mask bits using *threshold secret sharing* between the computing parties. The computing parties garble the circuit in a secure and distributed manner, using the secret-shared inputs and pass it to the result parties. The result parties get the circuit from computing parties and masked inputs from the input parties and evaluate the circuits. An illustration of the FairplayMP model is shown on Figure 2.3.

Evaluating garbled circuits may require a lot of memory, especially if the whole circuit is constructed in advance and stored in computer memory during evaluation. VMCrypt takes steps to streamline the performance of circuit evaluation and reduce the memory footprint [94]. Boolean circuits can be large and their construction and garbling is time-consuming. VMCrypt works in the classical Yao circuit evaluation model, but takes a streaming approach to circuit generation. When a part of a circuit is ready, it is passed to the evaluator and the necessary oblivious transfers are performed. To simplify, VMCrypt streams the circuit gate

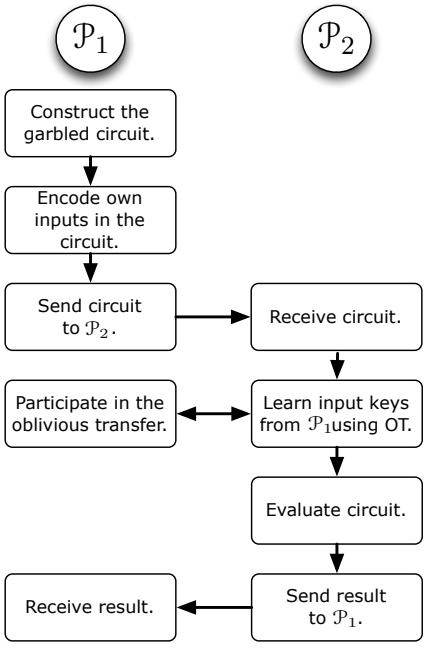


Figure 2.2: Yao style circuit evaluation with two parties.

by gate. This way, the evaluator does not have to wait for the whole circuit to be garbled to start the computation process. The technique has been developed further to evaluate larger circuits [69].

It is also possible to achieve security against malicious adversaries. Such adversaries may, for example, build the wrong kind of circuit or wire the circuit so that it leaks the inputs of the other party. There are various ways of forcing the circuit builder to prove the correct structure of the circuit. The solutions differ in the construction of the proof. For example, one can prove the correct construction of the whole circuit [72], single gates [102] or just generate many circuits and randomly check some of them [90] using the cut-and-choose technique.

2.4 From Boolean circuits to arithmetic circuits

The use of Boolean circuits is practical, as similar methods are used in hardware design. This allows hardware circuit designs to be reused in secure computation. However, the circuit garbling technique adds a computational overhead to each gate. If a single bit is encoded with a pseudorandom function such as a hash function or a symmetric cipher, the runtime representation of the bit is typically at least 80–128 bits long.

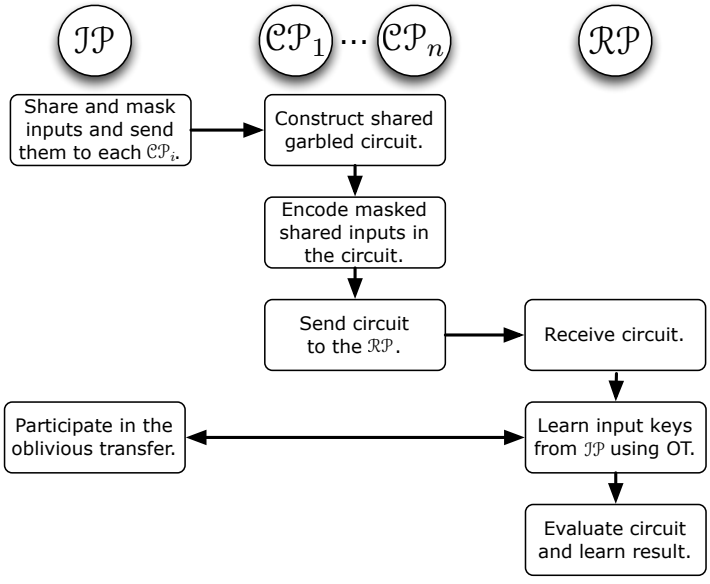


Figure 2.3: Yao style circuit evaluation with multiple parties.

In most garbled circuit systems, a binary gate with a single output wire contains a truth table with four garbled values. This means, that a gate requires 320–512 bits to encode a binary Boolean function and this does not include the memory cost of storing the structure of the circuit. Recent implementations can process circuits with hundreds of millions of gates [69, 82], but the compilers for such circuits still require significant computing power for large circuits.

Some secure computation techniques enable the secure evaluation of arithmetic circuits where one gate processes values larger than a single bit. Figure 2.4 shows an arithmetic circuit that receives integer inputs x, y and a bit value b. The value b is used to choose which of the input integers is output on wire z. Such a circuit is highly useful in secure computation, since it provides a replacement for branching operations. Instead of publishing a secret branching decision we can evaluate both branches and *obliviously select* the correct value to be the final result. Another significant property of this design is that it decreases the secure computation overhead per processed bit.

The example contains gates for addition, subtraction and multiplication. Each gate can process ring or field elements. The obvious benefit of such an approach is the reduction of circuit size and depth. However, we require an efficient construction for secure arithmetic operations, because implementing arithmetic gates using Boolean circuits does not give us the efficiency gain that we are looking for. Fortunately, there are several cryptographic protocols that fit our need. We will

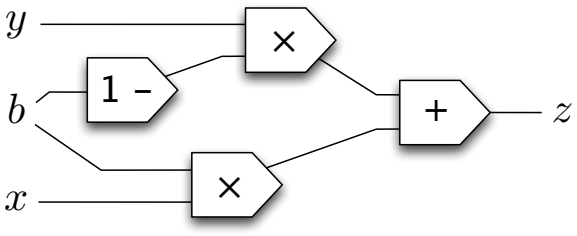


Figure 2.4: The arithmetic oblivious selection circuit.

now describe two such techniques—homomorphic encryption and secret sharing.

2.5 Two-party computation using homomorphic encryption

Homomorphic encryption is a suitable tool for creating secure computation systems in the client-server model. This is thanks to the homomorphic property that allows the encrypted values to be meaningfully modified by manipulating the ciphertext in certain ways. However, if we want to build a secure computation system using homomorphic encryption, we require two additional properties from the encryption scheme—semantic (IND-CPA) security and circuit hiding. Semantic security is an obvious confidentiality requirement. Circuit hiding means that if a ciphertext has been computed by combining two other ciphertexts, the ciphertext leaks no non-trivial information about the plaintexts of combined ciphertexts when decrypted or processed by a computationally unbounded adversary. This is usually achieved by using a special rerandomization procedure.

Notice that malleability is a direct side-effect of the homomorphic property and therefore, a homomorphic encryption scheme cannot be IND-CCA secure.

Consider a client $\mathcal{C}$ and a server $\mathcal{S}$. First, $\mathcal{C}$ generates a keypair for the chosen homomorphic encryption scheme and uses it to encrypt its input data. The public key is made available to $\mathcal{S}$. Now, $\mathcal{C}$ can send its values to $\mathcal{S}$, who uses the homomorphic property of the encryption scheme to run secure operations and evaluate a secure function f on the data. Once the computation is complete, the encrypted result is returned to $\mathcal{C}$ who decrypts it using the private key from the generated keypair.

As practical homomorphic encryption schemes like Paillier [103], Damgård-Jurik [48], Damgård-Geisler-Krøigaard [44, 45] and lifted ElGamal are only additively homomorphic, the server can perform only additions and multiplication with public constants autonomously. For more complex operations like multiplication, the server needs to engage in a specific protocol with the client. For an

example, see [120, 80]. Note that this approach is not limited to a client-server architecture and can be extended to a larger number of parties.

Furthermore, protocols in this model typically achieve security against a passive server $\mathcal{S}$. However, homomorphic encryption is a useful primitive in the construction of protocols that offer some level of protection against a malicious adversary in both two-party [7, 85] and multiparty settings [50].

Homomorphic encryption also has an overhead in its data representation. For example, to achieve practical security with current state of computing hardware, the message space of the Paillier homomorphic encryption scheme must be based on RSA moduli of at least 2048 bits in size. Hence, if we want encode a single value in a single ciphertext, the overhead is even larger than for garbled circuits.

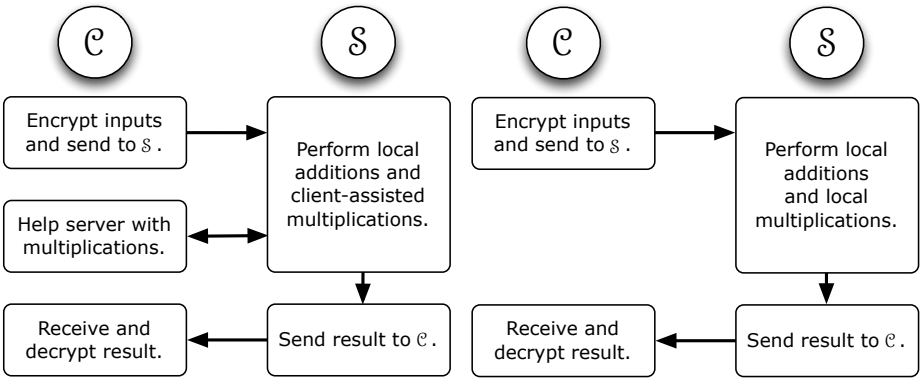
The TASTY [66] and the PaillierRuntime of VIFF [61] are two implementations of secure computation frameworks that use additively homomorphic encryption. The TASTY framework deserves attention for combining additively homomorphic encryption with garbled circuits to balance the benefits of both two-party approaches.

There also exist homomorphic encryption schemes that can perform both additions and multiplications locally. These schemes are called *fully homomorphic*. The first practical and provably secure fully homomorphic encryption (FHE) scheme was proposed by Gentry in 2009 [62]. A proof-of-concept implementation was presented in [63]. However, no practical secure computation system has been constructed yet due to the high resource requirements of the current schemes. To illustrate the overhead of an FHE scheme, consider Gentry's scheme that is based on intractable problems in integer lattices. The encryption scheme uses integers with over 800 000 bits to achieve a scheme-specific medium level of security. The overhead can be reduced through better encoding mechanisms, but these encoding mechanisms need to be designed to preserve the homomorphic property. For an example of such an encoding scheme that is designed for use with FHE, see [118].

Figure 2.5 illustrates how homomorphic encryption can be used to build a client-server secure computation system. The additively and fully homomorphic cases are presented separately for comparison.

2.6 Secure multiparty computation

The solutions discussed up to now have protected each value by using techniques such as encryption. Until now, we have focused on two-party solutions that transform each input value to a single value (with FairplayMP being the exception). We will now consider secure computation techniques based on *secret sharing* [113]. Secret sharing is used to divide a secret value into several pieces called *shares*. Each share looks random to the holder and a predetermined number of shares is



(a) Additively homomorphic encryption (b) Fully homomorphic encryption

Figure 2.5: Secure two-party computation using homomorphic encryption.

required to reconstruct the original value.

More formally, let s be a secret value in the message space $\mathcal{M}$ and $\mathcal{S}$ be the space of secret shares. A *k-out-of-n secret sharing scheme* is a tuple $(\mathsf{Share}, \mathsf{Rec})$ defined as follows:

1. $\mathsf{Share}(s) = (s_1, \ldots, s_n)$ is the randomized sharing function that computes the shares of a secret,

2. $\mathsf{Rec}(s_{i_1}, s_{i_2}, \ldots s_{i_k})$ is the reconstruction function that reconstructs the secret from at least k shares, and

3. having access to any $k-1$ shares from $(s_1, \ldots, s_n)$ gives no information about the value of s, i.e., the probability distribution of $k-1$ shares is independent of s.

In this thesis, we use $[\![x]\!]$ as a shorthand for $\mathsf{Share}(x)$.

Two constructions for secure computation based on secret sharing were published in 1988 [17, 39]. Secure multiparty computation has been continuously improved since then. Implementations differ in security models, number of parties and the used secret sharing schemes and protocols.

We will now describe how secure computation circuits are expressed in secret shared form. As secret sharing distributes every value into several pieces, the computation gates must be able to process data in this form. Figure 2.6 illustrates the concept by presenting the oblivious selection circuit from Figure 2.4 in secret-shared form. On the figure, each wire is separated into three lines, each representing one share of the value being processed. The three wires represent

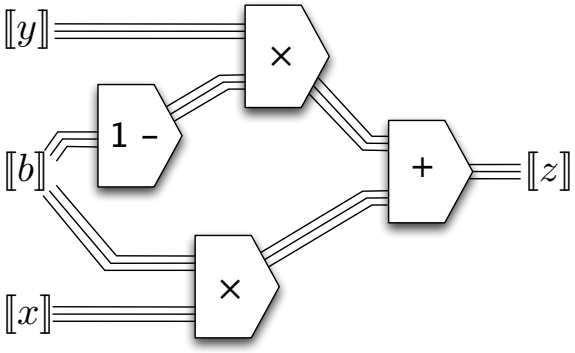


Figure 2.6: The arithmetic oblivious selection circuit with secret sharing and secure multiparty computation among three parties.

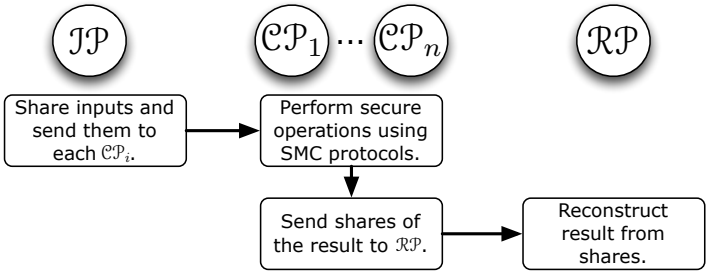


Figure 2.7: General secure multiparty computation using secret sharing.

the three shares of a value. We have chosen three for illustration only, as secure multiparty computation is not limited to three parties.

It follows, that each binary gate needs to be able to process six shares and output three shares representing the output value. The gate itself represents a secure multiparty computation protocol that can compute a secret-shared output from two secret-shared inputs. Intuitively, the protocol must not reconstruct the shares of the inputs as this would compromise the secret values. Instead, it uses distributed computation protocols to compute the result.

We will now describe how secure computation systems based on secret sharing can be deployed. We will use the same party classification for a secret sharing system that was used in the client-server model of the protocols in [47] and in the FairplayMP system [15]. $\mathcal{IP}$ stands for an input party, $\mathcal{CP}$ stands for a computing party and $\mathcal{RP}$ stands for a result party. Figure 2.7 shows the secure computation process with secret sharing.

Input parties use secret sharing on the input data and distribute the shares among the computing parties. Computing parties engage in secure multiparty

computation protocols to evaluate the distributed gates in the function f. Once f is computed, the computing parties send shares of the result to the result parties. In practice, input parties and result parties can be the same entities.

This kind of model is implemented in several frameworks. Examples include SHAREMIND, implementations from the SCET [32] and SIMAP projects [31], PassiveRuntime in VIFF [46, 61] and the SEPIA system [35]. SHAREMIND, SEPIA and the VIFF PassiveRuntime are the most similar frameworks, providing similar security guarantees. The VIFF ActiveRuntime and VIFF OrlandiRuntime can withstand more complex forms of corruption. Security models for secure multiparty computation are discussed further in Section 3.3.4.

Remarkably, the VIFF OrlandiRuntime combines secret sharing, additively homomorphic encryption and additively homomorphic commitments to achieve a secure computation method that can remain secure even when the majority of the computing nodes are dishonest. This is different from the other mentioned implementations that only provide security given an honest majority.

2.7 Resource cost estimates

Every secure computation system has its own bottlenecks and this makes good comparative benchmarking a challenging task. In this work, we present an analysis of the practical complexities of the described secure computation paradigms. We have taken an efficient design for each paradigm and analyzed its practical complexities and overheads.

We chose two data types for the analysis—the bit and the 32-bit integer. For both data types, we considered the size of the secure representation, the communication and computational cost of the secure addition and multiplication operations. These choices were made because they allow generic computations. More fine-tuned protocols may exist for specific tasks, but our goal in this work is to create generic, programmable secure computations. We are analyzing the performance of a single operation, thus ignoring protocol-specific parallelization opportunities.

For secure multiparty computation (SMC), we look at the SHAREMIND protocols given in [27]. The protocols are based on additive secret sharing with three parties, so each secret value is shared into three pieces and requires three times as much storage. As the chosen secret sharing scheme is additively homomorphic, addition requires no communication. The cost of the multiplication operation was measured from the SHAREMIND multiplication protocol.

For garbled circuits (GC), we assume an abstract construction with a number of standard features. The implementation follows the standard two-party construction and uses a 128-bit pseudo-random function (PRF) such as the SHA-1 hash function or the AES block cipher. The oblivious transfer primitive is not speci-

fied, but its use is counted in the cost analysis. The constructions for addition and multiplication circuits for 32-bit integers are reasonably optimized. The addition circuit uses 128 XOR and 32 AND gates and the multiplication circuit consists of 1984 XOR gates and 1024 AND gates. We assume, that the XOR gates can be evaluated with no communication [81]. The computation complexity is presented in the amount of randomness needed to generate the keys, the number of PRF calls and number of oblivious transfer calls. The communication complexity is given in bits to transfer the circuit and in the instances of the oblivious transfer (OT) protocol.

For additively homomorphic encryption (HE), we consider the Paillier scheme with plaintexts in the 2-kilobit range and ciphertexts twice as large. We assume, that packing is used in the multiplication protocol to reduce the number of encryption operations and communication. We have deconstructed the encryption and decryption operations into ring operations to improve comparability.

For fully homomorphic encryption (FHE), we generalize over several schemes providing local additions and multiplications. We assume that homomorphic operations are local over a field where the integer representation of each element is more than a million bits large. This achieves the "small" security level of the implementation described in [63]. The current fully homomorphic encryption schemes require a bootstrapping operation after a certain number of operations to control the noise in the ciphertext. This operation is very expensive and is the main limiting factor of the FHE technique. We do not measure it, as different schemes use different methods to perform this task.

The performance results for processing single bits are given in Table 2.2 and Table 2.3. Results for 32-bit integers are given in Table 2.4 and Table 2.5.

	Storage size	XOR communication cost	AND communication cost
SMC	3 bits	none	15 bits
GC	128 bits	640 bits + OT for 1 bit	640 bits + OT for 1 bit
HE	~4 Kbits	none	~8 Kbits
FHE	> 1 Mbit	none	none

Table 2.2: Storage and communication cost of securely processing 1-bit Booleans.

	XOR computation cost	AND computation cost
SMC	1 $\mathbb{Z}_2$ operation	3 bits of randomness + 13 $\mathbb{Z}_2$ operations
GC	512 bits of randomness + 10 PRF calls + OT for 1 bit	512 bits of randomness + 10 PRF calls + OT for 1 bit
HE	1 $\mathbb{Z}_{2^n}$ operation ($n > 4000$)	~12 Kbits of randomness + 10 $\mathbb{Z}_{2^n}$ operations ($n > 4000$)
FHE	1 $\mathbb{Z}_{2^n}$ operation ($n > 10^6$) + control noise as needed	1 $\mathbb{Z}_{2^n}$ operation ($n > 10^6$) + control noise as needed

Table 2.3: Computational complexity of securely processing 1-bit Booleans.

	Storage size	ADD communication cost	MUL communication cost
SMC	96 bits	none	480 bits
GC	4 Kbits	28 Kbits + OT for 32 bits	500 Kbits + OT for 32 bits
HE	~4 Kbits	none	~8 Kbits
FHE	> 1 Mbit	none	none

Table 2.4: Storage and communication cost of securely processing 32-bit integers.

	ADD computation cost	MUL computation cost
SMC	1 $\mathbb{Z}_{2^{32}}$ operation	96 bits of randomness + 13 $\mathbb{Z}_{2^{32}}$ operations
GC	16 Kbits of randomness + 384 PRF calls + OT for 32 bits	250 Kbits of randomness + 9312 PRF calls + OT for 32 bits
HE	1 $\mathbb{Z}_{2^n}$ operation ($n > 4000$)	~12 Kbits of randomness + 10 $\mathbb{Z}_{2^n}$ operations ($n > 4000$)
FHE	1 $\mathbb{Z}_{2^n}$ operation ($n > 10^6$) + control noise as needed	1 $\mathbb{Z}_{2^n}$ operation ($n > 10^6$) + control noise as needed

Table 2.5: Computational complexity of securely processing 32-bit integers.

CHAPTER 3

THE DESIGN OF SHAREMIND

3.1 Design goals and intended purpose

SHAREMIND is designed to be deployed as a distributed secure computation service that can be used for outsourcing data storage and computations. The distributed design is a requirement for using the secret sharing technique to guarantee the confidentiality of data during storage. Figure 3.1 illustrates the general usage model of SHAREMIND.

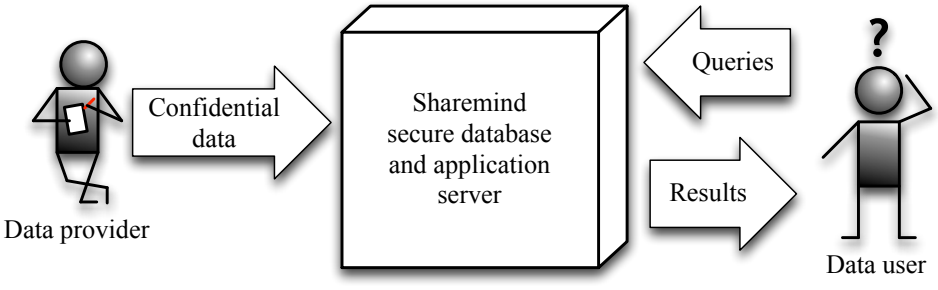


Figure 3.1: The deployment model of a SHAREMIND system.

In the deployment model, *data providers* control the confidential data that *data users* want to analyze. As the data are confidential, data providers cannot simply give it to the data users. In practice, there are various possibilities. For example, if the data providers are individuals, they may hesitate before providing behavioral information for a scientific study. Similarly, companies are reluctant to disclose metrics about their performance, as these can give advantages to a competitor. In a more mixed case, a company cannot share information about its customers because of data protection restrictions.

On the other hand, the data users have an interest in aggregating data to learn

the statistical properties of the attributes or discover patterns. The secure computation technology may let data users analyze information that they previously had no access to. For example, data providers will be more inclined to provide confidential data if they have provable guarantees for their security during storage and computations. This encourages data users to commit resources to deploying secure computation tools such as SHAREMIND.

From a technical standpoint, SHAREMIND is a general secure computation system designed with the following major goals in mind:

1. SHAREMIND will be used in data mining to arrange or outsource the processing of confidential data;

2. SHAREMIND must be sufficiently efficient to be used in practice;

3. SHAREMIND must be usable by non-cryptographers.

These goals have been the motivators behind several features of SHAREMIND that we discuss in this thesis. Also, given that the efficiency of the implementation is a key goal, we prefer techniques that are efficient on current computing hardware.

3.2 Different flavors of privacy

3.2.1 Record-level privacy

We will now look at a number of threats to privacy in typical data analysis scenarios. We consider record-level, source-level, output-level and cryptographic privacy. Most of these goals are are independent, i.e., we can achieve one without satisfying the others.

First, we look at *record-level* privacy. Each record in a database of individuals corresponds to a certain person who wants to prevent anyone from discovering specific values about him or her. For example, no researcher should be able to tell with certainty, whether an individual is a drug addict or not by looking at the respective database record. Record-level privacy is most important in statistical surveys and scenarios where databases are published, e.g. for research purposes.

A classical solution for preserving record-level privacy is the *randomized response technique* used in social studies and data mining [128, 42, 1, 5, 3]. The technique is useful in the application model where individuals submit personal data to a distrusted data collector. In such a survey, the individual will flip a coin before giving an answer to a sensitive question. If the coin comes up heads, the answer is expected to be truthful. Otherwise, a default answer to the question is given.

In the alternative *microdata publishing model*, a trusted data collector discloses a part of its database without revealing sensitive information about individuals. Similarly, this problem has been thoroughly studied in security and data mining communities, see [60] for an overview.

Common methods for ensuring record-level privacy involve randomization in some way. However, the randomization of data values has several weaknesses. First, it requires a trade-off between privacy and accuracy. If we increase noise in the data, we get better privacy, but the quality of global estimates will decrease. Second, the added noise can be cancelled only for some aggregation functions and the randomization method needs to be tailored for that function. Third, privacy is preserved on average and it can be breached by using related background information. These weaknesses and several attacks have been discussed in the literature [121, 93, 75, 99].

3.2.2 Source-level privacy

The idea behind of *source-level* privacy is very similar to record-level privacy. The main difference is in the number of records. In a typical scenario, a data owner wants to limit the amount of information that leaks to others during data processing. Source-level privacy is mostly relevant in settings where the data are split between several organizations who want to collectively analyze them.

Privacy-preserving data aggregation over horizontally and vertically partitioned data are the two most common scenarios studied in this context, see [89, 126, 74]. Notably, only a few published solutions are cryptographically secure. Others can leak significantly more information that the desired output.

The main reason behind solutions without cryptographic security is the perceived inefficiency of cryptographic solutions. Theoretically valid proposals have been overly inefficient for practice and even the computation of simple data mining primitives like scalar products has been a resource-demanding task [120]. The inefficiency of most solutions can be attributed to reliance on slower asymmetric cryptographic primitives like homomorphic encryption.

However, recent developments in secure computation have made the technology much faster and it is now possible to design solutions that rely on cryptographic primitives and provide source-level privacy. SHAREMIND is well-suited for this scenario as the distributed nature of the system is an ideal match for a setting with several data owners.

3.2.3 Output-level privacy

While the previous two types of privacy were concerned about what can be learned by looking at the source data, *output-level* privacy looks directly at the result of

the data analysis procedure. More specifically, we want to know how much the outputs of a data mining procedure leak information about its inputs.

Output-level privacy has been studied mostly in the context of *query auditing* where a trusted database owner can refuse to answer queries in order to protect individuals. Starting from the original problem statement [52], many hardness and impossibility results for naïve solutions have been derived, see [79, 77].

The most rigorous results about output-level privacy can be given in the framework of differential privacy [54]. Differential privacy studies how much a change in an input data record affects the output of a data analysis task. Intuitively, an algorithm achieves output-level privacy, if it can compute the correct aggregation of a set of records so that a single change to the records does not affect the output in a way that would leak the inputs. However, the construction of such algorithms is hard and typical solutions fall back to adding noise to the input and output data. This again reduces the accuracy of the analysis. Furthermore, differential privacy does not easily address the issue of privacy leaks as a result of a large number of repeated queries.

It is important to see that a breach in output-level privacy may constitute breaches to record-level privacy and source-level privacy. This is because breaking output-level privacy means that we learned something about the private inputs that we should not have learned.

3.2.4 Cryptographic privacy

There is a fourth kind of privacy that is orthogonal to the three previous ones. *Cryptographic privacy* guarantees that only the final result of a data analysis task is published. All inputs and intermediate values are protected using encryption, secret sharing or other similar methods. This protection is maintained throughout the secure computation process.

If a data analysis task can be completed with both cryptographic privacy and output-level privacy, then we cannot learn anything substantial about the private inputs and have successfully achieved also record- and source-level privacy.

3.3 The model of a SHAREMIND application

3.3.1 Overview of parties

We will follow a similar notation for parties that was used to describe the parties in FairplayMP. A secure computation system consists of any number of input parties $\mathcal{IP}_1$, $\mathcal{IP}_2$, ..., $\mathcal{IP}_m$, computing parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, ..., $\mathcal{CP}_n$ and result parties $\mathcal{RP}_1$, $\mathcal{RP}_2$, ..., $\mathcal{RP}_r$. In a deployed application, the input parties map to the data providers. Similarly, the result parties map to data users like data analysts.

Computing parties perform secure computations using the SHAREMIND miner server software.

There may also be cases when a single organization fills the roles of several parties. For example, an organization may be both providing data and receiving the results of queries.

3.3.2 Encoding private data

SHAREMIND applications represent data as unsigned integers. We have chosen standard types from programming languages such as 8-bit, 16-bit, 32-bit and 64-bit unsigned integers, because these types are efficiently implemented in modern computing hardware. We also support the boolean type for logical operations. The default data type in the SHAREMIND implementation is the 32-bit integer. Therefore, in our protocols we will use this data type as an example.

SHAREMIND computing parties use secret sharing to securely store confidential data. The mathematical representation of the chosen integer data types is a ring, e.g., $\mathbb{Z}_{2^{32}}$ for the set of 32-bit unsigned integers in the range $0, 1 \ldots, 2^{32} - 1$. The classic Shamir secret sharing scheme [113] is most suitable for protecting elements of fields. Also, the secure multiparty computation protocols relying on the properties of Shamir's secret sharing are not secure on rings like $\mathbb{Z}_{2^{32}}$. We, therefore, use the *additive secret sharing scheme* instead.

We write $x \leftarrow M$ to show that the element x has been uniformly chosen from the set M. To share a value $s \in \mathbb{Z}_{2^{32}}$ among n parties we compute the shares s_1, s_2, ..., s_n as follows:

$$\begin{aligned} s_1 &\leftarrow \mathbb{Z}_{2^{32}} \\ s_2 &\leftarrow \mathbb{Z}_{2^{32}} \\ &\ldots \\ s_{n-1} &\leftarrow \mathbb{Z}_{2^{32}} \\ s_n &= s - s_1 - s_2 - \cdots - s_{n-1} \bmod 2^{32}. \end{aligned} \tag{1}$$

In SHAREMIND, each party will receive one share of every secret value. The original secret can be reconstructed by collecting all the shares of a value and adding them up using the addition operation in the ring. The correctness and efficiency of this secret sharing scheme are trivial, but we will argue about its security, because it helps us later in showing the security of storage and computations.

Theorem 1. *For each secret value $s \in \mathbb{Z}_{2^{32}}$, any subset of $n - 1$ shares of s is uniformly distributed and for any two secret values $u, v \in \mathbb{Z}_{2^{32}}$, their secret*

shared forms are indistinguishable for any coalition of parties holding up to $n-1$ shares.

Proof. According to the secret sharing algorithm (1), the shares s_1, s_2, ..., s_{n-1} are uniformly chosen from $\mathbb{Z}_{2^{32}}$. We will now show that s_2, s_3, ..., s_n are also uniformly distributed. The same argument can be extended for any other $n-1$ different shares. We know that s_2, s_3, ..., s_{n-1} are uniformly distributed and independent.

Let $s' = s_2 + s_3 +, \ldots, +s_{n-1}$ for a fixed s_2, s_3, ..., s_{n-1}. Now, according to algorithm (1),

$$\begin{aligned} s_n &= s - s_1 - (s_2 + \cdots + s_{n-1}) \\ &= (s - s') - s_1 \ . \end{aligned}$$

As s_1 is still uniformly distributed when $s_2, \ldots, s_{n-1}$ are fixed, we get that s_n is uniformly distributed and independent from $s_2, \ldots, s_{n-1}$. □

The uniformity of a single share is a useful property for showing that the shares controlled by a single party reveal no information about the information processed. We will use this property later to argue about the security of a secure database.

3.3.3 The overall threat model

In the most common case, the behavior for all parties in our secure computation system will be implemented in software and will run on standard computing hardware. Messages will be transmitted on public computer networks. Also, in the real world there is no global clock in the system so both communication and processing are asynchronous.

Our main security goal is that the values provided by the input parties remain secret from all other parties. As the data of the input parties will be stored and processed by the computing parties, we need to ensure that they cannot learn anything from the information available to them.

Furthermore, the result parties must not learn anything except for the final results of the secure computation performed by the computing parties. Note, that depending on the used algorithms and provided data, the desired output may leak the input of one or more input parties. This threat is not handled by the threat model in this section. Possible solutions are discussed in Section 5.4.

The environment surrounding the parties is hostile and the adversary has many ways for breaking the security. To illustrate the situation, consider an input party who has been asked to provide private data for computations. From the perspective of this party, the situation looks really bad.

1. The adversary could be reading and modifying the messages exchanged between parties.

2. The adversary could be scheduling communication between the parties, delaying messages in the communication channel.

3. The adversary could be controlling all the other input parties.

4. The adversary could be controlling all the computing parties.

5. The adversary could be controlling all the result parties.

It is evident, that it will be impossible to do anything in this environment of paranoia, so we have to deploy security mechanisms that lower the probability of the listed threats. We will now discuss both cryptographic and administrative methods that reduce the attacking capabilities of the adversary and help us achieve our set security goal.

3.3.4 Reducing the power of the adversary

In our threat model we take the pessimistic view and assume that all adverse behavior is directed by a global adversary that can use any means available to break the security of the system. In a distributed setting, the adversary may corrupt any component or party in the system.

The most common corruption models describe *active* and *passive* corruption. In the active case, the adversary behaves maliciously and can make the party do anything. This includes making any or no computations, sending incorrect values, too much values or no values at all.

In the passive (*honest-but-curious*) case, we assume that the corrupted party follows the protocol but also reports everything that it sees to the adversary, who tries to compute the inputs and outputs of honest parties based on all the available information.

As an omnipotent adversary can defeat any security measure, we have to combine several methods and build a holistic defense model for a secure computation system. We will use two methods to reduce the power of the adversary. First, we will use established cryptographic techniques to secure the communication channels. Second, we will identify the threats that we cannot or will not solve using cryptographic means and we will either use administrative and legal methods to defeat them or accept the related risks.

Security of the communication channels. We start by securing the communication infrastructure. The use of standard secure channels such as TLS [51] over an Internet protocol like TCP ensures privacy, authenticity, integrity and the correct

order of messages. TLS and TCP require an underlying communication channel with at least some reliability to function properly. However, it is trivial to see that if all messages are dropped by the adversary, then any kind of communication between parties is not possible. In practice, we focus on a scenario where reliable communication links can be deployed and provide fallbacks for the case when they fail temporarily. Therefore, we can assume that it is possible to form secure, point-to-point communication channels between the parties using cryptographic communication protocols.

The current implementation version of SHAREMIND at the time of writing this thesis is SHAREMIND 2. Its network layer is based on UDP networking, so we have to account for a malicious network scheduler that can drop messages or change their order. This corresponds to the asynchronous network model described in several works on secure computation [16, 36, 68]. However, the SHAREMIND 2 network layer uses algorithms to guarantee reliability and message ordering. In a nutshell, messages are retransmitted until they arrive and buffered until they can be delivered in the correct order. If multiple retransmissions fail, the connection is considered to be lost. This means that the actual communication model of SHAREMIND 2 is less asynchronous than the one usually found in the literature.

It is evident, that lost connections will lead to troubles terminating the protocol. Even though the passive adversary model adopted by SHAREMIND does not accommodate for lost connections, we want to provide a solution for real-world applications. In practice, this means that SHAREMIND applications can use *restart points* between individual protocols to reconnect and continue computations.

General techniques against corruption. One should never underestimate the usefulness of organizational and legal measures. Even though secure multiparty computation reduces the need for non-disclosure agreements and penalties in contracts, we must combine cryptographic methods with proper usage procedures to ensure that cryptographic privacy is actually achieved in practice.

For example, organizations deploying secure multiparty computation must ensure that data security controls are in place to prevent the theft of the secret shares available to a single computing party. If such thefts occur at several computing parties, it may be possible to recover the original secrets by combining the shares. Therefore, standard data security controls are a natural complement to secure multiparty computation.

Technologies like secure hardware and secure virtualization are outside the scope of this thesis. Their use may reduce the risk of both insider and outsider attacks on secure computation.

Corruption of input parties. If the adversary controls any input parties, it can

convince them not to enter data or to enter invalid values. In the first case, the privacy of the input party who entered data in good faith, can be compromised, if secure computation outputs a trivial aggregation of the inputs. For example, if just one party enters data, aggregations such as sum or mean will leak the inputs. This constitutes a breach of output-level privacy and all means of countering such a breach also apply to the described case (see Section 3.2).

In the second case, incorrect data provided by other input parties can affect the result of data aggregation. Because the data are private, the computing parties cannot just look at individual values and discard them from the computations. All corrections must be made obliviously, without seeing the data. This requires an oblivious input validation method such as outlier detection. Furthermore, by cleverly selecting the inputs of the dishonest parties, the adversary may be able to learn the outputs of the honest ones. For example, entering zeroes can be equivalent to entering no value at all, if the aggregation function contains a sum.

In both cases, the adversary may need to corrupt several parties and possibly do this over a period of time. This means that the adversary can adaptively corrupt the input parties. However, for our secure goal to have a meaning, we assume, that at least one honest input party remains.

In our model, we consider corrupted input parties a risk that can be mitigated with the clever design of secure computation algorithms. We must reduce the dependence of outputs on exceptional values in the inputs and we have to do it by using oblivious filtering and similar techniques.

For example, we can implement algorithms for detecting exceptional values. The algorithm can output a secret-shared mask that can later be used to exclude exceptional values from processing. Examples on how this can be achieved in a privacy-preserving manner are given in Section 3.4.3. Note that making the output less dependent on the input is also a feature of differential privacy.

It must be noted that both described cases also apply to non-secure computations and general-purpose administrative techniques can be adapted to secure computations as well.

Corruption of computing parties. The greatest risk to privacy comes from the corruption of computing parties, as they store all the confidential data collected from the input parties, albeit in secret-shared form. The properties of the additive secret sharing scheme guarantee that all shares of a secret value are needed for reconstruction. This gives us the first obvious (but necessary) assumption—we cannot let the adversary corrupt all the computing parties.

However, this assumption is not constructive enough and, therefore, we look at two kinds of adversarial models. The *threshold model* sets a limit on how many parties can be corrupted by adversary. The classical results of [17, 39] state that unconditional security against a passive adversary can be achieved with an honest

majority—less than $\frac{n}{2}$ parties of n can be corrupted. To achieve security against an active adversary, the number of corrupted computing parties must be smaller than $\frac{n}{3}$.

The more general adversary structure model describes sets of corrupted parties that the protocol can still tolerate without losing security [67]. The most common adversary structures are $\mathcal{Q}_2$ and $\mathcal{Q}_3$. In the $\mathcal{Q}_2$ structure, no two sets of corrupted parties can form the full set of parties. This corresponds to the passively corrupted minority case in the threshold model. Similarly, in $\mathcal{Q}_3$, no three sets of corrupted parties can form the full set of parties and this has been shown to correspond to the threshold case less than $\frac{n}{3}$ parties can be actively corrupted.

As the additive secret sharing scheme is an n-out-of-n scheme, we cannot prevent actively corrupted parties from modifying the shares during secure computation. Solutions to that include the use of verifiable secret sharing schemes [41, 109] or using a shared MAC to check the shares [18].

It is also possible to achieve security against an active adversary with a dishonest majority, but such systems cannot guarantee the successful termination of the protocol. Furthermore, such constructions need to rely on slower asymmetric cryptographic primitives. Experimental results have shown that going from the passive model to the active model at least doubles the complete running time of a secure computation protocol, that may include a precomputation phase [56].

Precomputation allows a secure computation protocol to perform the majority of the expensive computations in an offline phase. Typically, the offline phase does not depend on the computational task at hand. However, the requirement for precomputing may limit the usability of the protocols in certain scenarios where quick reaction times are needed immediately after starting the secure computation system. Some of the secure computation protocols that are secure against active adversaries and use an offline phase are described in [49, 50].

Note that there are also other security models that provide different security guarantees. Security against active adversaries is a very strong property and comes with a relatively large overhead. However, in some application scenarios, honest parties do not need to detect malicious behavior, if it does not affect their outputs.

In the *covert model*, the privacy of the input party cannot be guaranteed if the adversary cheats. However, a cheating adversary can be detected with a significant probability and, therefore, a rational adversary will not cheat, especially, if the penalty for getting caught is higher than the expected gain [8].

Security models can also control, how much the adversary can learn about the inputs of the honest players. For example, in the *k-leakage model*, the adversary can learn up to k bits of information about the inputs of the honest players [97]. The somewhat stronger *consistent model* ensures that the honest parties can detect malicious behavior that modifies their outputs and actively corrupted parties learn

nothing but their own outputs [86].

We assume static corruption—that is, formally, the adversary can pick the parties to corrupt before the application is started and cannot corrupt any new parties during the execution.

As SHAREMIND tolerates only passive corruption in at most one computing party, we need to handle the residual risk of a computing party not following the protocol. Given that our protocols do not make use of commitments or proofs of inputs, it is easy to see how an actively corrupted party can make the result of the computations incorrect by sending arbitrary messages. Similarly, it is easy to stop computations from completing by not participating in the protocols.

A computing party may also try to modify the messages in the secure computation protocol to break the privacy of the input parties. However, the computing party needs a feedback channel to see how its changes affected the final output. Unless the computing party colludes with the other computing parties and manages to collect all the shares of a computed value, the only way to get such feedback is to analyze the published outputs of the computed function when they are published by the other computing parties.

Such an attack is more successful, if the dependency between the input of the function and the output of the function is stronger. Essentially, the adversary can try to modify the function being computed by modifying its share in the computation. Therefore, SHAREMIND achieves better privacy guarantees, if the computed function performs aggregations of several input values and filters out exceptional values from its inputs. The properties of such functions are discussed further in Section 5.4. Not all useful functions have such properties, so an application using passively secure protocols should also consider this risk.

However, it is not trivial to break the privacy of the input parties, even if a computing party changes messages in the protocols. The protocols of SHAREMIND are designed in such a way that if a single party does not follow the protocol, it cannot extract the contents of the shared database without at least one more computing party disclosing the shares of the affected values.

Even by changing the messages in the protocol, a corrupted party will need access to the published results of the computation to successfully extract the leaked bits. If the application does not provide the computing party with this output, it needs to corrupt another computing party at least passively to learn the shares needed to leak any bits. Therefore, without collusions, SHAREMIND 2 can also maintain privacy even when one computing party does not follow the protocol. Assuming that the protocols are information-theoretically secure, the number of bits leaked about the private inputs in a collusion is limited by the size of the published output. If the protocols are computationally secure, the leaked bits may help us compute many or even all of the bits of the private input.

Of course, if no computations take place, all computing parties need to be corrupted for data to leak, as all three shares are needed for reconstructing the secret values.

Corruption of result parties. The result parties are the ones who send queries to the SHAREMIND system. The main motivation for corrupting a result party is to siphon as much information about the stored secure data and use it to break output-level privacy. Therefore, we assume that the adversary is interested in corrupting the result parties to make malicious queries. We consider that the adversary corrupts result parties adaptively and there is no threshold on how many result parties are corrupted.

The only way the result parties affect the system is through queries and their parameters. Therefore, we need to make sure that SHAREMIND applications are designed so that the queries leak the minimal amount of information to the result party. We provide examples on how to achieve this for some practical applications in Chapter 6.

The model for secure computation protocols. To conclude, we have described potential attacks against secure computation in the cryptographic model. In our chosen model, the adversary can passively corrupt computing parties and control message scheduling in the network connections. In our applications, we have to assume that the input parties and result parties are actively corrupted and provide malicious inputs and malicious queries.

In the next section, we will describe the context in which our protocols are proven secure. Our goal is to achieve a set of protocols that allow efficient, programmable execution of secure computation operations.

3.3.5 The optimal number of computing parties

As the goal of SHAREMIND is to achieve maximum efficiency, we choose to use three computing parties. This is the lowest number for which we can form an honest majority of computing parties that processes secret-shared data. While secret-shared data can be processed with just two parties, these protocols require computationally more expensive cryptographic primitives than what we plan to use.

At the same time, adding more parties will increase the communication complexity of the secure computation protocols and reduce the performance. Therefore, three is the optimal number of parties for countering passively corrupted computing parties.

3.3.6 The case for passive security in SHAREMIND

The SHAREMIND secure computation framework presented in this thesis is based on protocols that are secure against a passive adversary. This means that if a computing party actively tampers with the software implementation, it may be able to break the security guarantees that SHAREMIND offers. Such attacks can be detected or even countered by protocols that are secure against an active adversary. However, such protection requires additional overhead and makes the protocols less efficient.

Taking all this into account, the version of SHAREMIND described in this thesis is most suitable for use in scenarios where multiple parties want to jointly process data, but are prevented from doing so by data protection rules. The parties can set up SHAREMIND and share data without any of them seeing each others inputs.

It must also be noted that SHAREMIND is not only defined by its protocols. It is a full framework for developing secure multiparty computation applications. The server infrastructure, programming language and application model are all suitable for other kinds of protocols. Therefore, SHAREMIND can be updated to use an actively secure suite of protocols and provide the benefits that they bring.

3.3.7 Constructing simulators for secure computation protocols

We will now present the security proof framework for SHAREMIND protocols. This framework has evolved during the development of the system. The first version was published in [25] together with the first set of secure computation protocols. New protocols and an updated model is presented in [27]. This thesis presents these models in more detail.

The security proofs of SHAREMIND protocols are built on the ideal versus real world paradigm. Assume that we want to evaluate the function

$$(y_1, y_2, y_3) = f(x_1, x_2, x_3)$$

so that each computing party $\mathcal{CP}_i$ provides the input x_i and learns the result y_i and nothing else. In the real world, parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ exchange messages to evaluate the function f given the secure multiparty computation protocols. However, we have to account for one corrupted party. Consider an example where $\mathcal{CP}_3$ is passively corrupted. The adversary $\mathcal{A}$ is now controlling $\mathcal{CP}_3$ and has access to all its input and output messages and the local state. However, given the passive adversary assumption, it cannot change the local state or outgoing messages. The real world setting is illustrated in Figure 3.2.

In the ideal world (see Figure 3.3), there is a *trusted third party* F that collects the inputs from the computing parties, evaluates the function f, and returns the results. As $\mathcal{CP}_3$ is corrupted, we handle it differently from the honest parties.

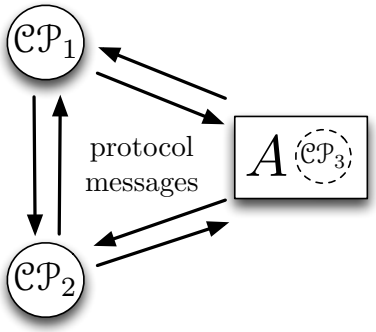


Figure 3.2: SHAREMIND protocols in the real world setting.

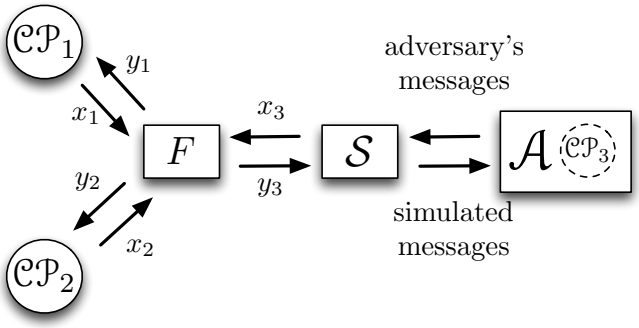


Figure 3.3: SHAREMIND protocols in an ideal world setting.

Our goal is to show that for any real-world attack there exists an attack in the ideal world so that both attacks use roughly the same amount of resources and have roughly the same probability of success. In more detail, an attack should corrupt the same parties, use the same background information and have similar computational and storage complexities. The standard approach to show the existence of an attack in the ideal world is to construct a *simulator* $\mathcal{S}$ that can simulate every protocol run of the real world in the ideal world.

In a typical setting, the simulator in the ideal world interacts with the trusted third party on one side and the adversary $\mathcal{A}$ on the other side. Its goal is to simulate protocol messages to $\mathcal{A}$ so that $\mathcal{A}$ cannot distinguish the ideal world from the real world. If we can design such a simulator, we have shown the *simulatability* of the secure multiparty computation protocol, as everything that the adversary can see, has been derived from either its own input or the output and nothing is learned about the inputs or outputs of other computing parties. For security, we also require that the output of all honest parties is correct and the joint output distribution of all parties coincides in the real world and in the ideal world.

In our setting, the outputs of all computing parties are in the forms of shares. Therefore, no computing party gets the reconstructed output from a protocol. This allows us to simulate protocol messages for the adversary without interacting with a trusted third party. The resulting security property is weaker and simplifies the security analysis of protocols in our model.

We consider a simulation *perfect* when the distributions of the adversary's view in the real world and in the ideal world coincide. The simulator is *non-rewinding*, when it goes through the protocol in a straight line without rewinding the adversary algorithm to an earlier state.

We will now give a more thorough explanation of how simulatability is used in the SHAREMIND model. Every computing party $\mathcal{CP}_i$ has an input ϕ_i. The input consists of the party's input value x_i and any background information that the party has available. We assume, that $\mathcal{CP}_3$ has been corrupted by the adversary $\mathcal{A}$ so the input ϕ_3 is also given to the adversary. We write $\mathcal{S}^{\mathcal{A}}$ to show that the simulator $\mathcal{S}$ has black-box access to the adversary $\mathcal{A}$ in the ideal model. We write $\mathcal{A}^{\mathcal{CP}_1,\mathcal{CP}_2}$ to show the execution of the adversary in the real world and in communication with parties $\mathcal{CP}_1$ and $\mathcal{CP}_2$.

Definition 1. A secure computation protocol is *perfectly simulatable* if there exists an efficient universal non-rewinding simulator $\mathcal{S}$ so that for all adversaries $\mathcal{A}$ and for all inputs ϕ_1, ϕ_2, ϕ_3, the output distribution of the simulator $\mathcal{S}^{\mathcal{A}}$ and of the adversary running in the real world setting $\mathcal{A}^{\mathcal{CP}_1,\mathcal{CP}_2}$ coincide.

Notice that this is not the standard definition of simulatability used in cryptographic proofs. Our version of simulatability is a restricted notion that is more related to the definition of privacy for secure multiparty computation protocols that are secure against a passive adversary.

For completeness, this definition blueprint must be extended by specifying the model for communication, by defining what is the class of efficient computations and so on. Note that the definition is given in the standalone setting where no other computations are performed before, after or during the execution of a protocol.

However, both sequential and parallel composition are important for implementing programmability and vector operations in SHAREMIND. Therefore, we must preserve security under composition. For that, we extend the universal composability framework by Canetti [38] to our notion of simulatability.

Our definition of simulatability does not directly imply security. Simulatability ensures that the adversary gets the expected output, but the honest parties might get different outputs in the real world and in the ideal world. If these outputs are used in followup computations, information may leak to the adversary. Considering this, we will define the security of a secure computation protocol in Section 3.3.8.

The simulatability proofs for SHAREMIND typically follow a similar blueprint. We look at the incoming messages of each computing party and show that these messages are independent from the input shares held by the other computing parties. The latter guarantees that no computing party learns the inputs of others. If the secure computation protocol is symmetrical, we can also simplify the proof by only looking at the incoming messages of one of the computing parties.

This independence is useful for constructing the simulator. The protocols of SHAREMIND often use uniformly chosen values to mask protocol messages and make them look random. This way, the simulator can generate uniformly distributed messages for the adversary without it being able to distinguish the messages provided by the simulator from protocol messages in the real world setting.

Consider V_i as the view of a computing party $\mathcal{CP}_i$. Let $V_i(\phi_1, \phi_2, \phi_3)$ be a program that takes the inputs of the parties and outputs a tuple $(a_1, a_2, \ldots, a_k)$ containing the values that this party sees during the execution of a protocol. This includes the inputs, randomness and received values of $\mathcal{CP}_i$.

We write $\langle V_i \rangle$ to denote the distribution of the values $(a_1, a_2, \ldots, a_k)$ output by V_i. Let $V_{i,0}$ denote the view corresponding to the original secure computation protocol. Then we can transform this view to form a sequence of modified views such that

$$\langle V_{i,0} \rangle \equiv \langle V_{i,1} \rangle \equiv \ldots \equiv \langle V_{i,n} \rangle$$

so that the output of the view $V_{i,n}$ does not contain any references to the inputs of the other computing parties. Here, $\langle V_{i,j} \rangle \equiv \langle V_{i,j+1} \rangle$ means that the output distributions of $\langle V_{i,j} \rangle$ and $\langle V_{i,j+1} \rangle$ coincide. Note that such a $V_{i,n}$ performs almost all the duties of a proper simulator, as it takes the inputs of the parties and produces the randomness and messages of the protocol that coincide with the real world view. To build the simulator, we need to use $\mathcal{CP}_i$ together with $V_{i,n}$ to successfully simulate the internal state of $\mathcal{CP}_i$. For this last part we also require that $V_{i,n}$ is an efficient algorithm. That is, every $V_{i,j}$ in the sequence must have roughly the same complexity that the initial game $V_{i,0}$ has[1].

We now show a theorem that we will later use for constructing the sequence of views and building the simulator.

Theorem 2. *Let $\mathcal{G}$ be a finite additive group. Let $(a_1, \ldots, a_j \pm r, \ldots, a_k)$ be the output of V_i so that $a_j \in \mathcal{G}$ and $r \leftarrow \mathcal{G}$ and r is independent from all values a_l. Then*

$$\langle V_i \rangle \equiv \langle V_i[a_j \pm r/r] \rangle ,$$

[1] This is usually formalized by requiring that in asymptotic model, the overhead of $V_{i,j}$ compared to $V_{i,j-1}$ is polynomial in the security parameter of the protocol.

where $V_i[a_j \pm r/r]$ is a program that runs V_i and replaces all occurrences of $a_j \pm r$ with r.

Proof. Let $(a_1, \ldots, a_j \pm r, \ldots, a_k)$ be the output of V_i for a fixed randomness except for the value r. Then, r is uniformly distributed in $\mathcal{G}$ and, therefore, independent from all a_l and so $r \pm a_j$ is also uniformly distributed, as any $f_r(x) := r \pm x$ is a bijective mapping for $\mathcal{G}$. We have shown both distributions to be uniform and proved the claim of the theorem. □

3.3.8 From simulatability to security and composability

For simulatability, we needed to convince the adversary that it is in the real world even when it is really in the ideal world. Simulatability is a weak property, as it does not take into account the outputs of honest computing parties. To achieve security, all honest parties in the real world must get the same output as they would get in the ideal world. For this, we must allow the simulator to also interact with the trusted third party F.

Definition 2. A secure computation protocol is *perfectly secure* if there exists an efficient universal non-rewinding simulator $\mathcal{S}$ so that for all adversaries $\mathcal{A}$ and for all inputs ϕ_1, ϕ_2, ϕ_3, the joint output distribution of all computing parties and the adversary coincide in the real world and in the ideal world. That is, the outputs of $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ and $\mathcal{S}^{\mathcal{A},F}$ in the ideal world coincide with the outputs of $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ and $\mathcal{A}^{\mathcal{CP}_1,\mathcal{CP}_2}$ in the real world.

Similarly to Definition 1, this definition can be adapted for sequential, parallel or concurrent composability by adding the relevant context. We stress that the non-rewinding property is necessary for achieving universal composability. In the following proofs and proof sketches, we will assume that we are working in the synchronous model, where computation is split into well defined rounds.

We will now discuss the *resharing* protocol that helps us achieve both security and universal composability in SHAREMIND protocols. The resharing protocol in Algorithm 1 is used by the computing parties to create a version of a secret-shared value with the same public value but different shares. We also remind the reader that the protocols are defined over integer rings of the form $\mathbb{Z}_{2^n}$.

New uniformly distributed values are injected into the shares to make the output shares of w independent from the shares of the input u. We will now show that the resharing protocol in Algorithm 1 is perfectly secure.

Theorem 3. *Algorithm 1 is perfectly secure in the standalone model with one passively corrupted computing party.*

Algorithm 1: Resharing protocol $[\![w]\!] \leftarrow \mathsf{Reshare}([\![u]\!])$.

Data: Shared value $[\![u]\!]$.

Result: Shared value $[\![w]\!]$ such that $w = u$, all shares w_i are uniformly distributed and u_i and w_j are independent for $i, j = 1, 2, 3$.

1 $\mathcal{CP}_1$ generates an uniformly distributed $r_{12} \leftarrow \mathbb{Z}_{2^n}$.
2 $\mathcal{CP}_2$ generates an uniformly distributed $r_{23} \leftarrow \mathbb{Z}_{2^n}$.
3 $\mathcal{CP}_3$ generates an uniformly distributed $r_{31} \leftarrow \mathbb{Z}_{2^n}$.
4 All values $*_{ij}$ are sent from $\mathcal{CP}_i$ to $\mathcal{CP}_j$.
5 $\mathcal{CP}_1$ computes $w_1 \leftarrow u_1 + r_{12} - r_{31}$.
6 $\mathcal{CP}_2$ computes $w_2 \leftarrow u_2 + r_{23} - r_{12}$.
7 $\mathcal{CP}_3$ computes $w_3 \leftarrow u_3 + r_{31} - r_{23}$.
8 Return $[\![w]\!]$.

Proof. To show correctness of the reconstruction of outputs, we show that $w = u$ as follows:

$$\begin{aligned} w &= w_1 + w_2 + w_3 \\ &= u_1 + r_{12} - r_{31} + u_2 + r_{23} - r_{12} + u_3 + r_{31} - r_{23} \\ &= u_1 + u_2 + u_3 = u\,. \end{aligned}$$

Similarly to the proof of Theorem 2, it is easy to see that any two shares in the triple w_1, w_2, w_3 are uniformly and independently distributed. All values w_i are uniformly distributed as they are of the form $u_j + r - s$ for randomly generated elements r, s. As any two values w_i are computed from independent uniformly distributed values then any two of values w_i, w_j where $i \neq j$ are independent.

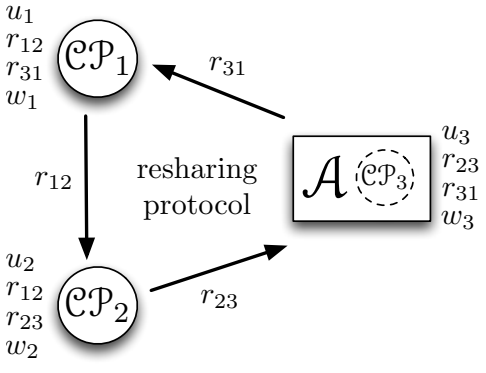


Figure 3.4: Network messages and local values in the resharing protocol.

As the protocol is symmetric, we can choose $\mathcal{CP}_3$ to be the corrupted party. First, we illustrate the exchange of messages in the real world. Figure 3.4 shows

what messages are exchanged and what values are available to the computing parties. The corrupted party $\mathcal{CP}_3$ sends out r_{31} and expects r_{23} in return so that it can compute w_3 as its share of the reshared value u.

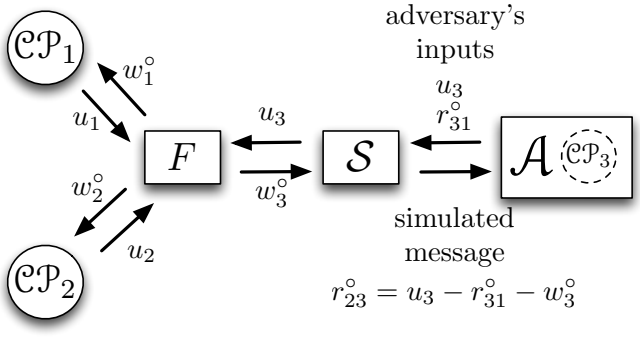


Figure 3.5: Simulator for the resharing protocol.

Figure 3.5 shows the protocol in the ideal world. The trusted party F computes the output shares w_1°, w_2° and w_3° as follows:

$$\begin{aligned} u &\leftarrow u_1 + u_2 + u_3 \\ w_1^{\circ} &\leftarrow \mathbb{Z}_{2^{32}} \\ w_2^{\circ} &\leftarrow \mathbb{Z}_{2^{32}} \\ w_3^{\circ} &\leftarrow u - w_1^{\circ} - w_2^{\circ}. \end{aligned}$$

Each share w_i° is sent to $\mathcal{CP}_i$, except for w_3° that is given to the simulator $\mathcal{S}$. The simulator $\mathcal{S}$ simulates the value r_{23}° by computing

$$r_{23}^{\circ} = u_3 + r_{31}^{\circ} - w_3^{\circ}.$$

The simulator can compute r_{23}° even before it receives r_{31}° from the adversary, because it is providing the adversary with all the randomness and, therefore, it can precompute the value of r_{31}°.

This way, when $\mathcal{A}$ computes its output, it gets the expected value, because

$$\begin{aligned} w_3^{\circ} &\leftarrow u_3 + r_{31}^{\circ} - r_{23}^{\circ} \\ &= u_3 + r_{31}^{\circ} - u_3 - r_{31}^{\circ} + w_3^{\circ} \\ &= w_3^{\circ}. \end{aligned}$$

We need to show that the distribution of the shares is the same in both the ideal and the real world. In the ideal world, w_1° and w_2° are uniformly generated and independent. We already showed that w_1 and w_2 are also uniformly generated and independent in the real world.

To complete the proof, we have to show that the joint distribution of the outputs of honest computing parties and the messages received by the adversary in the real coincide with the matching outputs and messages in the ideal world. This is sufficient, as the randomness and messages received by the adversary uniquely determine its output. According to Algorithm 1, both r_{31} and r_{31}° are uniformly generated. As the following equation holds in both the ideal and real settings,

$$r_{23} = u_3 + r_{31} - w_3 \ ,$$

all received messages are identical. □

We do not give a full universal composability proof here as the corresponding proof would be highly technical and the result can be concluded by combining the characterization of universal composability in the standalone setting. For details, see the discussion in Section 4.3.1 of [37] and Chapter 7 in [84]. In brief, our simulator construction satisfies the universal composability requirements as it is non-rewinding and has black-box access to the adversary.

We will now show how to compose other protocols with the resharing protocol to make any perfectly simulatable protocol secure and universally composable.

We start by providing proof sketches for composing simulatable protocols in the SHAREMIND model. The first step towards universal composability is to show that perfect simulatability is preserved when we compose protocols sequentially or concurrently under certain restrictions. In such a composition, a *top-level* protocol consists of several *sub-protocols*, each of which is an instance of a perfectly simulatable protocol. Several sub-protocols can be instantiated from the same kind of protocol. For example, the top-level protocol can many instances of the multiplication protocol as sub-protocols.

The main restriction is on how we can use the output shares of a sub-protocol of the top-level protocol. Specifically, we can do one of three things:

1. Forget the output shares.
2. Use them as inputs to another perfectly simulatable subprotocol.
3. Use them as an output of the main protocol.

The need for such a restriction is easily shown with an example. Figure 3.6 shows two protocols with the respective ideal functionalities F_1 and F_2. The figure also contains possible implementations for both protocols. Note that in the implementation of the first protocol, $\mathcal{CP}_1$ also sends $r \oplus a$ to $\mathcal{CP}_2$. Both protocols are trivially perfectly simulatable for both computing parties, as all messages can be emulated with random values.

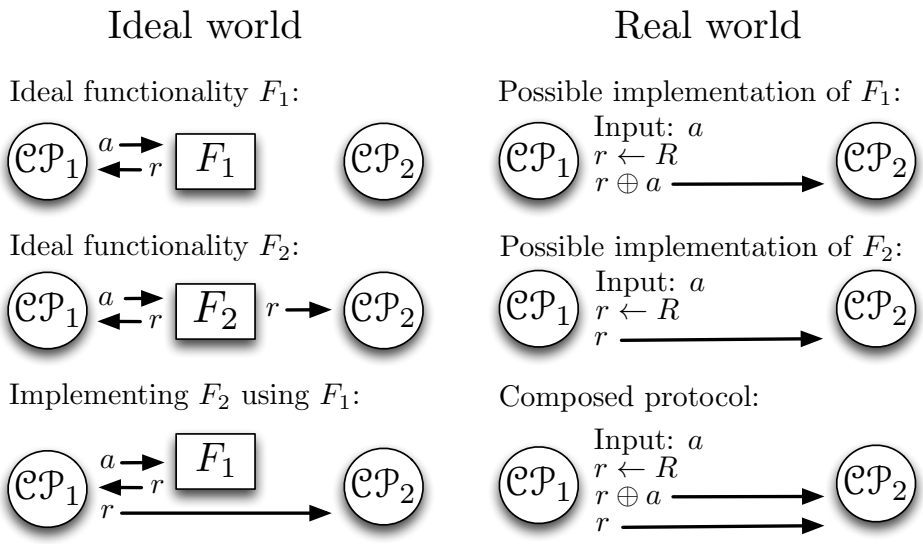


Figure 3.6: A perfectly simulatable protocol and its non-simulatable composition.

Now, consider a third protocol that implements F_2 using the ideal functionality of F_1. The ideal implementation of this third protocol is perfectly simulatable. However, if we substitute the ideal implementation of F_1 for the perfectly simulatable implementation of F_1, the resulting composed protocol is not perfectly simulatable any more. The composed protocol implementation sends both r and $r \oplus a$ to $\mathcal{CP}_2$ and we cannot simulate these messages to $\mathcal{CP}_2$ together.

The problem is that we are using the implementation of F_1 in an environment where the output of F_1 is sent to $\mathcal{CP}_2$ via $\mathcal{CP}_1$. This allows the adversary to learn both the outputs even though we have sub-protocol simulatability proofs for the outputs that $\mathcal{CP}_2$ receives in the sub-protocols. The fact that $\mathcal{CP}_1$ sending $r \oplus a$ to $\mathcal{CP}_2$ invalidates the simulatability and we should therefore restrict how sub-protocol outputs are used.

Theorem 4. *A protocol consisting of several sub-protocols is perfectly simulatable, if the following conditions hold:*

1. *All the sub-protocols are perfectly simulatable.*
2. *The output of each sub-protocol is either the input of another sub-protocol or the output of the main protocol.*
3. *The data dependency graph of sub-protocols is a directed acyclic graph.*

Proof sketch. According to our assumptions, all simulators $\mathcal{S}_i$ of sub-protocols are non-rewinding. Therefore, we can combine them to form a single compound

simulator $\mathcal{S}_*$ that runs the simulators $\mathcal{S}_i$ sequentially or concurrently to provide the adversarial computing party with all the required messages. In the case of dependencies between sub-simulators, outputs from one sub-simulator can be used as inputs for the next sub-simulator. We also compose all the trusted third parties into a single functionality by composing their code. Figure 3.7 illustrates how the new simulator is constructed.

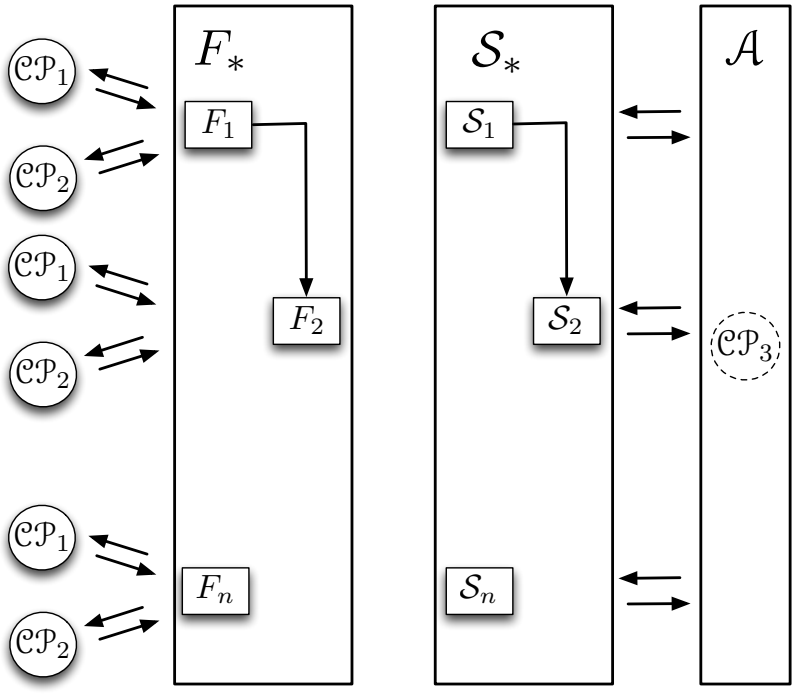


Figure 3.7: Composing several simulators into a single simulator. The line between $\mathcal{S}_1$ and $\mathcal{S}_2$ illustrates the case where the input of sub-protocol 2 is dependent on the output of sub-protocol 1.

As every simulator provides a perfect simulation, the final view of the adversarial computing party is also perfectly simulated. Assume that some messages in the protocol are computed from the output shares of a sub-protocol. If any of these messages are then sent to other computing parties, we cannot properly simulate them for the receiving party. This is because the adversary acquires some information about the shares of the honest parties and this invalidates the security guarantees that are given by the simulatability definition. □

We require that the sub-protocols do not have cyclic dependencies between them as this would make the top-level protocol impossible to execute. This is

because at least one input for one of the sub-protocols would not be available when the protocol is executed.

In the next step we show how to make a perfectly simulatable protocol secure by finishing it with a resharing step such as the one in Algorithm 1.

Theorem 5. *A perfectly simulatable secure computation protocol that is followed by a perfectly secure output resharing is perfectly secure in the standalone model.*

Proof sketch. In this proof sketch we show how to prove the claim for a single output share. The approach is similar for protocols with multiple output shares. Let x_i be the input share for party $\mathcal{CP}_i$. Let $\mathcal{S}_{comp}$ be the simulator for the perfectly simulatable computation protocol and $\mathcal{S}_{reshare}$ be the simulator for the perfectly simulatable resharing protocol. Let z_i be the output share of $\mathcal{S}_{comp}$ for $\mathcal{CP}_i$.

Let F_{comp} be the trusted third party for the computation protocol and $F_{reshare}$ be the trusted third party for the resharing protocol. Note, that $\mathcal{S}_{comp}$ and F_{comp} can be compositions of other simulators and trusted third parties, respectively. Let F be the composition of F_{comp} and $F_{reshare}$ and similarly, let $\mathcal{S}$ be the composition of $\mathcal{S}_{comp}$ and $\mathcal{S}_{reshare}$. This structure is illustrated in Figure 3.8.

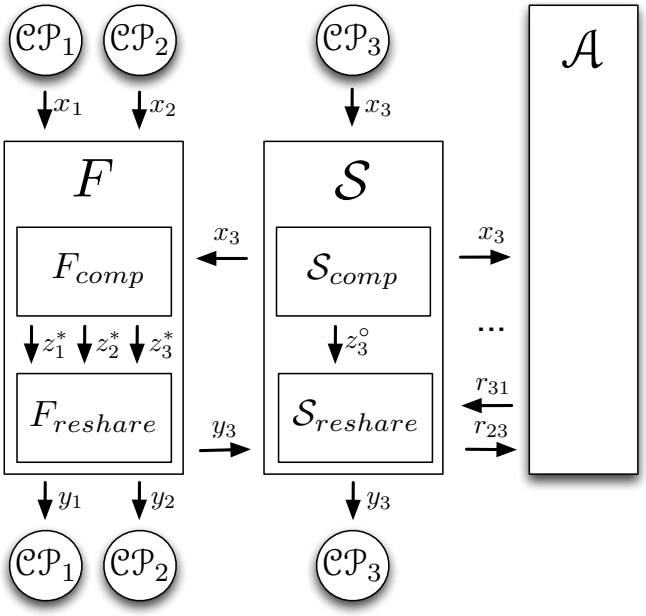


Figure 3.8: The simulator for a universally composable secure computation protocol

The simulator $\mathcal{S}$ learns x_3 from $\mathcal{CP}_3$ and passes it to the adversary $\mathcal{A}$. The simulator uses $\mathcal{S}_{comp}$ to generate all messages of the secure computation protocol including the output share z_3° that is passed to the resharing simulator $\mathcal{S}_{reshare}$.

Let us observe the state of all parties after the completion of the secure computation protocol and the corresponding simulator. In the real world, the computing parties have their output shares z_1, z_2 and z_3 from the secure computation protocol. The adversary has its internal state σ. In the ideal world, the trusted third party has not finished and the honest parties have not received their output shares. However, $\mathcal{S}$ has computed the output share z_3° for $\mathcal{CP}_3$ and the adversary has its internal state σ°. As the secure computation is perfectly simulatable, the distribution of (z_3, σ) coincides with the distribution of $(z_3^\circ, \sigma^\circ)$.

To complete the proof, we show how perfectly secure resharing ensure that the joint output distribution of all parties coincides in the real and ideal world. This directly follows from the proof of Theorem 3, as the input values of the resharing protocol and $F_{reshare}$ represent the same value $z = z_1 + z_2 + z_3$ in both the real and ideal world and the honest parties discard the values z_1 and z_2. Recall, that this is achieved by computing the message r_{23} from r_{31} and the simulated output share z_3°. □

It may seem that resharing at the end of every protocol is wasteful. Especially, if the protocol is used as a sub-protocol and the intermediate results will never be published outside the computing parties. In some cases, we may *inline* resharing into the computation protocol to lower the round count of the composed protocol.

However, such inlining requires that we run the first round of the resharing protocol in parallel with the secure computation protocol and not after it as required by Theorem 5. This requires a slightly different construction for the simulator. The main challenge of such a simulator is that the first round of the resharing protocol has to be simulated without access to the output shares of the secure computation protocol. We sketch the construction of such a simulator in Theorem 6.

Theorem 6. *A perfectly simulatable secure computation protocol that runs a perfectly secure and universally composable output resharing protocol in parallel and outputs the reshared output, is perfectly secure and universally composable.*

Proof sketch. We follow the construction of Theorem 5 with one important difference. We want to run the resharing simulator in parallel with the secure computation protocol, but the inputs of the first are technically dependent on the outputs of the second.

The simulator resolves this issue by internally running the simulator construction from Theorem 5 to learn the output share z_3° of the computing protocol. It then uses z_3° and the output y_3 from the trusted third party F to start the concurrent simulation of the secure computation protocol and the secure resharing protocol with the adversary. The simulator can do this, as $\mathcal{CP}_3$ is passively corrupted and $\mathcal{S}$ can clone and run the code of $\mathcal{S}_{comp}$ with the same randomness to get the same output. Note that this does not mean that we are rewinding $\mathcal{S}_{comp}$.

Algorithm 2: A protocol illustrating asynchronous communication

Data: $\mathcal{CP}_1$ has an input a, $\mathcal{CP}_2$ has an input b.
Result: $\mathcal{CP}_3$ learns c so that $c == b$.

Round 1:
  $r \leftarrow \mathbb{Z}_{2^{32}}$
  $\mathcal{CP}_1$ computes $a' \leftarrow a + r$
  $\mathcal{CP}_1$ sends a' to $\mathcal{CP}_2$
  $\mathcal{CP}_1$ sends a' to $\mathcal{CP}_3$
Round 2:
  $\mathcal{CP}_2$ computes $b' \leftarrow b + a'$
  $\mathcal{CP}_2$ sends b' to $\mathcal{CP}_3$
Round 3:
  $\mathcal{CP}_3$ computes $c \leftarrow b' - a'$

The argument for the joint output distribution of all parties is exactly the same as in Theorem 5. □

The proof can also be done by relying only on the definition of universal composability, provided that we have proved the security of the resharing protocol in contexts where the first round of resharing can be run before the resharing protocol gets its inputs.

The last remaining task is to resolve the issue of malicious network scheduling. As explained in Section 3.3.4, we expect that our communication channel implementation guarantees the reliability and ordering of messages on a single channel. Still, the lack of a central clock leaves us with the chance that a party receives values from another party earlier than anticipated.

Consider the protocol in Algorithm 2. In the implementation of a distributed system, it may happen that the latency on the channel from $\mathcal{CP}_1$ to $\mathcal{CP}_3$ is much higher than the latency between $\mathcal{CP}_1$ and $\mathcal{CP}_2$ or $\mathcal{CP}_2$ and $\mathcal{CP}_3$. We cannot, therefore, rule out that $\mathcal{CP}_3$ receives b' before it receives a' and our simulator must be capable of simulating either option to the adversary.

Fortunately, there is a simple result that helps us build protocols that are secure even in the presence of a malicious scheduler.

Theorem 7. *If the values of protocol messages that are sent out by a party do not depend on the order in which this party receives protocol messages, then a protocol executing on a network with malicious scheduling is as secure as a protocol executing in a synchronous network.*

Proof. An adversarial scheduler can change the order in which messages arrive at a party. However, if corruption is passive and static, then this does not change

the view of the adversary as values received by the adversarial parties remain the same.

To complete the proof, we need to show, how to construct a simulator that can simulate the messages to the adversary in any order chosen by the malicious scheduler. We start with the simulator $\mathcal{S}$ that can simulate the messages in a synchronized model. In this setting, the adversary observes the simulated values at the end. According to the assumption, the outgoing messages of our protocol do not depend on the order of incoming messages. Therefore, we can simply reuse $\mathcal{S}$ to simulate and release the messages in the order specified by the malicious scheduling given by the adversary. □

We have now shown how to prove the security of secure multiparty computation protocols in the SHAREMIND model. For examples on protocols that can be proven secure using this approach, see Section 3.5. For proofs of the SHAREMIND protocols, we refer the reader to the papers included in this thesis [25, 27].

3.3.9 Guidelines for designing secure protocols for SHAREMIND

The described proof model establishes some requirements that must be followed to simplify the security proofs of SHAREMIND protocols. The general blueprint remains the same—to show security, we need to show that the protocol is correct and perfectly simulatable in the synchronous communication model. After that, we add a resharing step and gain security and universal composability.

Given the discussion on asynchronous communication, it is also necessary to construct the protocols in such a way that outgoing messages do not depend on the order of arriving messages. This way the simulator can simulate these messages in all situations.

There are several strategies for choosing when to reshare results. First, one can reshare after every atomic operation. Then it will always be safe to publish results, but this makes every computational operation a bit less efficient. The second option is to skip resharing during computation and do it only when shares are going to be published. This can improve performance, but makes the system's designer responsible for resharing the values in all possible cases where shares might be published. In the current SHAREMIND implementations, all computation protocols that exchange messages perform resharing at the end of the protocol.

Until now, we have focused on achieving information-theoretic security for the secure computation protocols in SHAREMIND. However, the real-world implementation of SHAREMIND uses computational primitives such as a pseudo-random number generator and an encrypted secure channel. It is possible to implement the SHAREMIND protocols using non-computational replacements, such

as random number generators that use environmental noise and secure channels secured using physical means or even one-time pad.

Computationally secure primitives can also be used as an optimization technique. For example, consider the work in [87].

3.4 Secure storage in SHAREMIND

3.4.1 Design goals for secure storage

It is not reasonable to assume that all data providers are online at the same time to give SHAREMIND the input data necessary for performing computations. Indeed, data may need to be collected over a period of time, and if for large databases the collection process can take a significant amount of time. Therefore, we need to provide SHAREMIND with a means for securely storing data.

We propose the use of a database within the secure computation system. Since a secure computer is capable of representing data securely during computations, we will use the same capability for persistent storage. The SHAREMIND system uses a shared database where each computing party stores a single share of each value in the database. The most important use-cases for this database are:

1. secure storage of collected data;
2. retrieval of stored data for use in secure computations;
3. secure storage of data output by secure computations and;
4. removal of data when it is no longer needed.

3.4.2 The structure of secret-shared databases

Most data processing applications work on structured data—tuples, matrices and key-value pairs. For example, relational database systems contain tables consisting of tuples of equal length. Non-relational databases use a variety of different structures with key-value storage being the most popular. We discuss how to adapt these database paradigms to secret-shared storage.

The construction of secret-shared relational database is straightforward. A typical relational database table contains tuples where each value corresponds to an attribute. To convert such a table into a secret-shared form, we use secret sharing on each value stored in the table and store the resulting shares in new tables that replicate the structure of the original. If a database contains many tables, the procedure is repeated for each table. In an application, all input parties

enter their data into such databases by using secret sharing for all input values and sending one share to each database.

We use an example to illustrate this construction. SHAREMIND uses three computing parties and therefore all data are shared into three shares. Each computing party stores a copy of the database, but instead of storing the original value, it stores a share of that value. Such a database is illustrated in Figure 3.9. The figure shows how each value x_i in the original tuples is shared into three shares—x_{i1}, x_{i2} and x_{i3} and the shares are stored in the servers' databases.

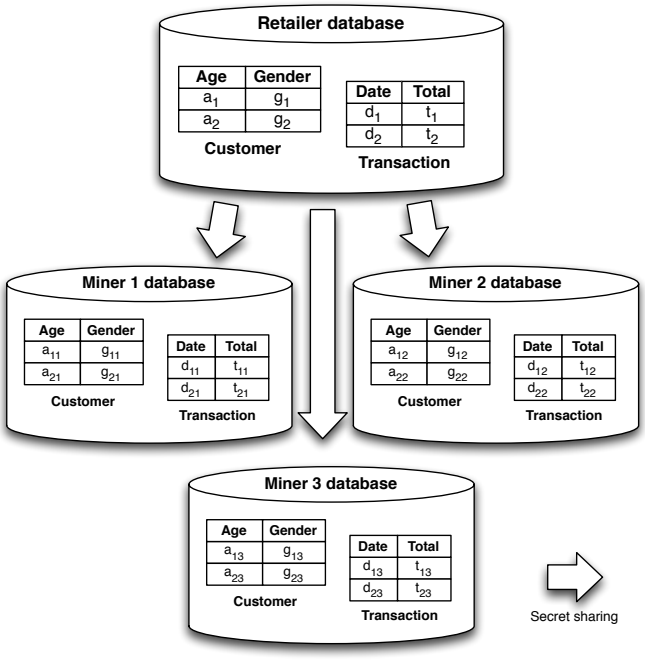


Figure 3.9: Using secret sharing on a relational database.

The main advantage of a secret-shared database is its high level of confidentiality. It is nearly impossible for an individual computing party storing the data to learn anything about the values provided by the input parties. This is possible because of the security of the secret sharing scheme.

There are other options for securely storing data, such as encryption. However, processing encrypted data requires the use of homomorphic encryption schemes that are less efficient than systems based on secret sharing.

Privacy-preserving storage has also been studied using statistical methods. For example, data perturbation methods are a standard solution for protecting anonymized data against reidentification. The k-anonymization technique was proposed for privacy-preserving microdata releases [110]. The idea is to partition

the tuples of a published database into equivalence classes so that the (quasi)-identifiers in each class are indistinguishable from each other and each class contains at least k tuples. The approach has been improved by introducing ℓ-diversity [92] and t-closeness [88].

However, all these approaches require that database values are generalized and this makes randomization less accurate. Furthermore, data perturbation is not a good guarantee against reidentification, as has been demonstrated by several high-profile attacks [10, 99]. Secret sharing takes a fundamentally different approach to protecting data and does not require value to be modified while still allowing the data to be efficiently processed.

We can also transform non-relational key-value stores into secret-shared form. For that, we will use secret sharing on all the values in the input database and store them in a replicated structure on the three SHAREMIND miners.

We will now provide an easy-to-use formal model of the database for use in our protocol descriptions. We consider a simple case where the database of a SHAREMIND application is represented as an $n \times m$ matrix $\mathcal{D} = (d_{ij})$, where $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$ and $d_{ij} \in \mathbb{Z}_{2^{32}}$. We also use individual rows in this matrix as tuples in the form $\mathcal{T} = (t_i)$, for $i \in \{1, \ldots, n\}$.

3.4.3 Manipulating secret-shared databases

A secret shared database is somewhat different from standard databases when it comes to queries. As the raw values are not accessible, we cannot use standard techniques for filtering out individual records from the database.

While we can use secure comparison operations to obliviously evaluate filtering conditions on secret-shared data, we cannot use this information to reduce the amount of data to process. Identifying filtered database records in the physical memory requires that we publish their location and this is an obvious privacy leak. Instead, we need a way for applying filters on the data without leaking information about which records matched the filtering condition. We will now describe some basic approaches to processing data in a secret-shared database.

Structural addressing. We can use the name or index of a database column to load all the values of that column. For example, if we want to compute the average age from a table of person records, each computing party extracts the respective column of shares from $\mathcal{D}$ and uses a summation protocol and a division to compute the average.

Similarly, we can load individual rows of the table by their index. This is useful when we need to process each record individually. Also, by combining the column and row addressing, we can address individual values by their location in the table.

Hybrid databases. In practice, we do not have to make all the columns in a table contain shares. Instead, we can use *hybrid* tables where public values and shared values coexist. Public values are replicated across all three computing parties and stored along the shared values. Sufficient metadata must be available for separating columns with shared values from those with public values.

In such a database, we can use public columns as keys to the records. Consider a data collection application, that generates a unique identifier for each entered record and returns it to the provider of the data. For added security, this identifier could be digitally signed. This data provider now has the ability to update the data record using this identifier. By providing a new secret shared record and requesting a computing party to identify and replace the old record using the public identifier, the original private input is not leaked to the computing party. It is basically replacing some random-looking values with others, based on a random identifier with no inherent meaning.

If secret-shared data is linked to any public identifier, the computing party may be able to identify the related real-world individual. However, it still cannot understand the values that the individual has provided.

Oblivious data access. It is also possible to keep the data access patterns secret by using techniques similar to *oblivious RAM* [65] and *PIR-writing* [33]. We will now describe specific constructions for *oblivious database queries* on secret-shared data. The blueprint gives a general construction for performing filtered queries on secret-shared data. It does not depend on the particular set of arithmetic and comparison primtives used. The basic protocols used in SHAREMIND are given in Section 3.5.

Given a condition expression like "person is male", it is possible to secure compare all values of the gender column to the constant representing the male gender and get a secret-shared boolean result. Each such result is stored as a zero or one in a *mask vector* that can be used in further processing to include or exclude certain values from processing. For example, if we want to find the average income of male individuals in a database, we take the following steps.

1. Get the number of records in the database table as n.
2. Load the gender attribute from the database table by extracting the respective column from $\mathcal{D}$ into a vector $\vec{g} = (g_1, \ldots, g_n)$.
3. Assuming, that the male gender is encoded using the value one, compute the mask vector $\vec{m}$ using secure computation:

$$m_i = \begin{cases} 1, & \text{if } g_i = 1 \\ 0, & \text{otherwise.} \end{cases}$$

4. Load the income attribute from the database by extracting the respective column from $\mathcal{D}$ into a vector $\vec{d}$.

5. Filter out the males by multiplying the vectors $\vec{m}$ and $\vec{d}$ elementwise using secure computation and storing the result in the vector $\vec{f}$. Note that the incomes of non-male individuals will be replaced with zeroes. The result will be

$$\vec{f_i} = \begin{cases} d_i, & \text{if } m_i = 1 \\ 0, & \text{otherwise.} \end{cases}$$

6. Compute the average by evaluating

$$avg = \frac{f_1 + f_2 + \cdots + f_n}{m_1 + m_2 + \cdots + m_n}$$

using secure computation.

7. Publish the computed average avg.

Alternatively, we can publish the sums computed in step 6 and perform the division using public computations. While this allows us to skip a costly private division operation, it leaks the number of male individuals represented in the database.

Oblivious queries are rather simple to construct and use for data retrieval. They can also be used for updating data. Consider an example where we want to increase the wage of everyone who has worked in the company for at least five years by ten percent. We take the following steps.

1. Get the number of records in the database table as n.

2. Load the work experience attribute from the database table by extracting the respective column from $\mathcal{D}$ into a vector $\vec{w} = (w_1, \ldots, w_n)$.

3. We find the mask vector $\vec{m}$ using secure computation:

$$m_i = \begin{cases} 1, & \text{if } w_i > 5 \\ 0, & \text{otherwise.} \end{cases}$$

4. Load the wages attribute from the database by extracting the respective column from $\mathcal{D}$ into a vector $\vec{d}$.

5. At each location where the mask contained a 1, the wage will be increased by 10%. The value remains the same in other locations.

$$d'_i = d_i + 0.1 d_i m_i$$

6. Store the new wages vector in $\mathcal{D}$.

The proposed solution can be more efficient than oblivious RAM, because we do not need to shuffle the vectors or the database. For more details on oblivious database queries in the SHAREMIND model see [87].

3.4.4 A protocol for data collection

Input parties form a secure channel to each computing party and use additive secret sharing over the selected ring of integers to distribute their input values into shares. One share is sent to each computing party over a secure channel. Note that this requires the input parties to have access to a good source of randomness.

It may occur, that the input party does not have access to a good randomness generation mechanism at the protocol implementation level. For instance, this is the case if the data collection protocol is implemented in a current web browser using JavaScript. The web browser can acquire randomness to form secure channels with a web server, but does not make this service available to the JavaScript virtual machine.

In such cases, we can still assume that the input party can access a pseudo-random generator $\mathsf{PRG} : \{0,1\}^k \rightarrow \{0,1\}^n$. Then, it can use the protocol in Algorithm 3 to generate the randomness needed for securely performing secret sharing on the private inputs. This protocol lets the computing parties generate secure random values and send it to the input party. The input party uses the XOR operation to combine the bit strings and uses the result as a seed to PRG. This way, none of the computing parties knows the randomness, as long as they do not all collude.

Algorithm 4 shows how to securely collect data represented as 32-bit unsigned integer values. The protocol assumes that a relational database with tables in the form of matrices is used. It is trivial to extend this protocol to perform more complex data management operations, like storing full records, adding new rows or columns to the table, or even use a different database paradigm.

Given that SHAREMIND is designed using the same principles as a database and application server, it can also have multiple users and multiple databases. In most scenarios, we want to restrict users to accessing only the databases and algorithms that they are allowed to query.

Algorithm 3: Protocol for providing secure random values to a input party.

Data: $\mathcal{IP}$ has the size of required random bits n.
Result: $\mathcal{IP}$ has an n-bit random bitstring s.

$\mathcal{CP}_i$:
  $r_i \leftarrow \{0,1\}^k$
  Send r_i to $\mathcal{IP}$.
$\mathcal{IP}$:
  $r = r_1 \oplus r_2 \oplus r_3$
  $s = \mathsf{PRG}(r)$

Algorithm 4: Protocol for securely collecting a 32-bit unsigned integer.

Data: $\mathcal{IP}$ holds a private value $s \in \mathbb{Z}_{2^{32}}$, a database table name $table$ and indices x, y.
Result: A shared value $[\![s]\!]$ is stored by parties $\mathcal{CP}_1$, $\mathcal{CP}_2$ and $\mathcal{CP}_3$ in database table $table$ in column x and row y.

$\mathcal{IP}$:
  $s_1 \leftarrow \mathbb{Z}_{2^{32}}$
  $s_2 \leftarrow \mathbb{Z}_{2^{32}}$
  $s_3 \leftarrow s - s_1 - s_2 \bmod 2^{32}$
  Send $(table, x, y, s_1)$ to $\mathcal{CP}_1$
  Send $(table, x, y, s_2)$ to $\mathcal{CP}_2$
  Send $(table, x, y, s_3)$ to $\mathcal{CP}_3$
$\mathcal{CP}_i$:
  Each $\mathcal{CP}_i$ looks up its local database matrix $\mathcal{D}$ for table $table$.
  $\mathcal{D}_{x,y} \leftarrow s_i$

Applications based on SHAREMIND can use access control [2] to enforce any access policies. Each computing party can authenticate the input party and determine its access rights to the particular table before storing the values.

We note that a real-world deployment of a secret-shared database shares issues with any distributed database. For example, we need to consider the situation where multiple input parties send secret-shared data simultaneously. During communication, the order of the shares may be switched by the network and they will arrive at the computing parties in a different order. This will corrupt the values in the database as wrong sets of shares will be used in processing. Therefore, if several input parties want to add a row or a column to the secure database, then these transactions must be executed in the same order by all computing parties to ensure that the databases remain consistent among them.

One such example is illustrated in Figure 3.10. In this figure, $\mathcal{IP}_1$ is sharing

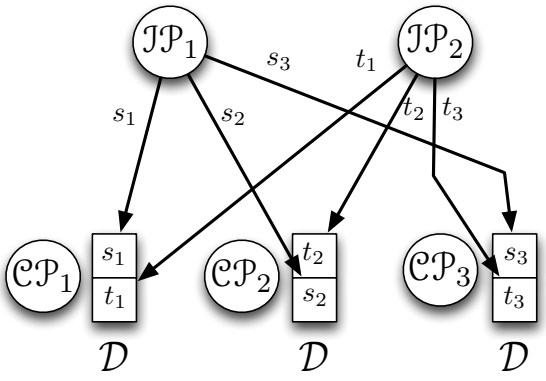


Figure 3.10: Example of a possible database inconsistency.

a value s and $\mathcal{IP}_2$ is sharing a value t. The value is appended as a new record in the databases. However, since the share of $\mathcal{IP}_2$ reaches $\mathcal{CP}_2$ before the share from $\mathcal{IP}_1$ arrives, it is stored in its place. Now, if we reconstruct the records of the database, both will give us corrupted random values.

The solution is for the computing parties to maintain distributed queues of transactions and jointly decide on the order in which they are executed. Each input party will be assigned a session identifier that will be used to fix the order of transactions. We can use standard consensus protocols, e.g., one of the Paxos protocols [83], to achieve this goal.

Even if we can ensure the consistency of the secret-shared database, the variation in the order of database records means that the protocol in Algorithm 4 violates the precondition of Theorem 7 (security of SHAREMIND protocols in an asynchronous network) by having messages arrive in any order. The main security risk is that the order, in which input parties send their messages, may change the output of the computed function. The most obvious example of such a function is the computation of the mean of the first ten collected values.

We describe three possible ways for reducing the risk. First, this risk does not endanger algorithms that do not depend on the order of records in the database. For example, if we aggregate every value in a database column and publish the sum, it will be the same regardless of the order of the records. The second option is to use public identifiers in the data collection protocol and during data lookup. Examples include addressing a record in the database by its row and column and addressing it by a publicly accessible key column, creating a key-value database. If all lookups in such a key-value database are performed using the key column, then the order in which the values arrived will make no difference.

Finally, if our algorithm is dependent on the order of messages in the database and it does not make sense to use a key-value database, we can randomly reorder

the values in the database to remove the dependency between record order and the output of the computation. Note that this reordering does not have to be oblivious. It is enough if the ordering remains uniform even if the adversary can choose any initial ordering.

There is also an alternative way for collecting data into a secret-shared database. Instead of forming direct connections with the computing parties, the input parties encrypt each share with the public key of the target computing party and send the encrypted shares to a proxy server that transfers the shares to the computing parties at a later time. Such a solution can be used when the input parties do not have the capability of forming direct secure channels with the computing parties. A similar solution was used in practice in the Danish sugar beet auction [31].

Furthermore, if it is possible to set up a public key infrastructure that includes all the input parties, the data collection protocol can also include the signing of the shares by the input parties. This allows the miners to use the digital signatures for authenticating the input parties during the processing of input shares. This is critical when we want to use the alternative data collection method employing a proxy server, because the SHAREMIND secure computation servers need a way for checking the source of the input shares that the proxy server is passing on.

Collecting secure data in a database simplifies the deployment of SHAREMIND as different input parties can upload their data independently from each other and over a longer time period. Once data collection is complete, we can start running secure computation algorithms.

3.5 Protocols for secure computation

3.5.1 The general secure computation process

Once the data are collected, a result party may send a query to the computing parties over secure channels. The query contains the following components:

1. the name of the algorithm to run,
2. public query parameters, and
3. private query parameters.

The computing parties determine the protocols to run based on the name of the algorithm. Public query parameters contain information that does not have to be secret (e.g., the name of the database table) and are sent without secret sharing. Private query parameters are secret shared. This allows the result party to hide query parameters from the computing parties.

Algorithm 5 gives the general protocol for running queries on the computing parties. The query is sent to each computing party who looks up the algorithms

Algorithm 5: General protocol for running queries on computing parties.

Data: $\mathcal{RP}$ holds a query q. Computing parties have the algorithms and data to process q.

Result: $\mathcal{RP}$ receives the result r of query q run on the data held by the computing parties.

$\mathcal{RP}$:
  Secret-share all private parameters and send the query q to each $\mathcal{CP}_i$.
$\mathcal{CP}_i$:
  Find and prepare the algorithm needed to run q.
  Store the parameters given in q. Load the databases needed to run q.
  For each operation in the algorithm:
    Prepare inputs for the computation protocol.
    Run the protocol.
    Store outputs of the protocol.
  Prepare the result share r_i.
  Send r_i to $\mathcal{RP}$
$\mathcal{RP}$:
  Compute $r \leftarrow r_1 + r_2 + r_3$.

and data for completing the computations. Because of universal composability, the computing parties can schedule computing protocols in any order. Some form of runtime storage can be used for storing intermediate result shares. The final result shares are sent to the result party, who reconstructs the result.

The use of secure computation protocols allows the computing parties to evaluate the required function on the shares of the input data without reconstructing the original values. However, the computing parties also need to process public values such as public parameters and control flow constants. For this, each computing party replicates the necessary public operations. Public operations do not touch the shares of secret data so they compose trivially with secure operations. For more details, see Chapter 5.

3.5.2 Protocols for addition and multiplication

The most important protocols in a secure computation system are the basic arithmetic protocols for addition and multiplication. This is so, because often, other protocols can be composed from addition and multiplication. In this section, we introduce the protocols that SHAREMIND uses for these operations.

Addition on SHAREMIND is trivial thanks to the additively homomorphic property of the additive secret sharing scheme. To learn the sum of two shared values, we just need to add the shares at each computing party. The complete

Algorithm 6: SHAREMIND protocol for secure addition

Data: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold a shared value $[\![w]\!]$ so that $w = u + v$.
1 Each $\mathcal{CP}_i$ computes $w_i \leftarrow u_i + v_i$

protocol is given in Algorithm 6. Addition is a local protocol that does not need communication with other parties.

As additive secret sharing is not multiplicatively homomorphic, we need a more complex approach for multiplying secret shared values. The product of two secret-shared values, $[\![u]\!]$ and $[\![v]\!]$ can be expressed as follows:

$$\begin{aligned} uv = & (u_1 + u_2 + u_3)(v_1 + v_2 + v_3) \\ = & u_1v_1 + u_1v_2 + u_1v_3 + \\ & u_2v_1 + u_2v_2 + u_2v_3 + \\ & u_3v_1 + u_3v_2 + u_3v_3. \end{aligned}$$

Each computing party $\mathcal{CP}_i$ can autonomously compute the value u_iv_i from this sum. However, the computing parties need to exchange information about the shares to be able to complete sum. At the same time, the parties must keep the shares confidential to protect the original secrets. Our protocol resolves the issue by creating a temporary resharing of the input secrets so that some new shares can then be exchanged between the computing parties. Basically, we temporarily transform the single-share encoding that SHAREMIND typically uses into replicated secret sharing, where each party holds two shares of a value. For this, we use the resharing protocol given in Algorithm 1 in Section 3.3.8. The protocol ends with another resharing step that ensures composability. The complete protocol is given in Algorithm 7.

The security of the addition protocol is trivial to show, as no messages are exchanged. The security proof for the multiplication protocols is given in [27].

3.5.3 Protocols for comparison

The most important computation operations beside basic arithmetic are the operators for equality and greater-than (or less-than) comparison. Both are required for filtering and making other decisions based on data. Greater-than comparison is also an important primitive for implementing privacy-preserving sorting.

While addition and multiplication are basic algebraic operations, comparison operators work on the bit level of values. In equality, we want to know if all the bits of two values are equal. In greater-than comparison, we want to know in

Algorithm 7: SHAREMIND protocol for secure multiplication

Data: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold a shared value $[\![w]\!]$ so that $w = uv$.

$[\![u']\!] \leftarrow \mathsf{Reshare}([\![u]\!])$
$[\![v']\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
$\mathcal{CP}_1$ sends u'_1 and v'_1 to $\mathcal{CP}_2$.
$\mathcal{CP}_2$ sends u'_2 and v'_2 to $\mathcal{CP}_3$.
$\mathcal{CP}_3$ sends u'_3 and v'_3 to $\mathcal{CP}_1$.
$\mathcal{CP}_1$ computes $w'_1 \leftarrow u'_1v'_1 + u'_1v'_3 + u'_3v'_1$.
$\mathcal{CP}_2$ computes $w'_2 \leftarrow u'_2v'_2 + u'_2v'_1 + u'_1v'_2$.
$\mathcal{CP}_3$ computes $w'_3 \leftarrow u'_3v'_3 + u'_3v'_2 + u'_2v'_3$.
$[\![w]\!] \leftarrow \mathsf{Reshare}([\![w']\!])$.

Algorithm 8: SHAREMIND protocol for secure equality comparison

Data: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold a shared value $[\![w]\!]$ so that $w = 1$ if and only if $u = v$. Otherwise, $w = 0$.

$\mathcal{CP}_1$ generates random $r_2 \leftarrow \mathbb{Z}_{2^n}$ and computes $r_3 \leftarrow (u_1 - v_1) - r_2$.
$\mathcal{CP}_1$ sends r_i to $\mathcal{CP}_i$ $(i = 2, 3)$.
$\mathcal{CP}_i$ computes $e_i = (u_i - v_i) + r_i$ $(i = 2, 3)$.
$\mathcal{CP}_1$ sets $\vec{p}_1 \leftarrow 2^n - 1 = 111\ldots1$.
$\mathcal{CP}_2$ sets $\vec{p}_2 \leftarrow e_2$.
$\mathcal{CP}_3$ sets $\vec{p}_3 \leftarrow (0 - e_3)$.
$[\![w']\!] \leftarrow \mathsf{BitConj}([\![\vec{p}]\!])$.
$[\![w]\!] \leftarrow \mathsf{Reshare}([\![w']\!])$.

which value the bit representation difference occurs first. These observations have inspired the design of the following protocols.

The equality of two values can be determined by computing the difference of the two values and making sure that all bits in this difference are zeroes. Our protocol achieves this privately by computing the conjunction of all bits in the difference. The protocol is given in Algorithm 8. The protocol contains an optimization that temporarily reshares the input between two miners among the three. The details on this approach and the bit conjunction sub-protocol $\mathsf{BitConj}()$ are omitted here, see [27] for the details and the security proofs.

The greater-than comparison protocols in SHAREMIND are all based on the observation that we learn the relation between two values by extracting the highest bit from their difference. Note that this approach is restricted to cases where we

Algorithm 9: SHAREMIND protocol for secure greater-than comparison

Data: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, $\mathcal{CP}_3$ hold a shared value $[\![w]\!]$ so that $w = 1$ if and only if $u > v$ (according to the restrictions defined above). Otherwise, $w = 0$.

1 $\mathcal{CP}_i$ computes $d_i \leftarrow u_i - v_i$.
2 $[\![w']\!] \leftarrow \mathsf{ShiftRight}([\![\vec{d}]\!], 31)$.
3 $[\![w]\!] \leftarrow \mathsf{Reshare}([\![w']\!])$.

interpret the highest bit of a value as the sign bit. This way, comparison on the 32-bit unsigned integers of SHAREMIND is effectively a 31-bit unsigned comparison or 32-bit signed comparison.

SHAREMIND extracts the highest bit by using a right shift protocol. The highest bit of the difference is shifted right so that it becomes the lowest bit of the value. This value can then be flipped to turn a greater-than protocol into a less-than-or-equal protocol. It can also be multiplied with the input values to build a simple minimum or maximum value computation protocol. The outline of the comparison protocol is shown in Algorithm 9. The bit shift right protocol is detailed in [27]. The given protocol works for 32-bit inputs. For other input sizes, the shift size parameter will need to be changed accordingly.

3.5.4 The secure computation capabilities of SHAREMIND

The current protocol suite of SHAREMIND covers basic arithmetic on integers. All operations are designed to be performed pointwise on vectors of inputs. Both unary and binary operations are supported. Table 3.1 gives an overview of the protocols that have been implemented on SHAREMIND 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 can be supported using shares in $\mathbb{Z}_2$. Logic operations on booleans can be composed from the addition and multiplication operations on 1-bit integers. These operations correspond to the exclusive or and disjunction operators on boolean values. All the protocols are implemented in the SHAREMIND version 2. The performance of SHAREMIND protocols is analyzed in Chapter 4.

3.6 Notes on the design of SHAREMIND protocols

Work on the SHAREMIND system started in 2006. The first preliminary version—the SHAREMIND version 1—became operational in 2007. The implementation

Operands	Operation	Reference
private $\vec{u} \in \mathbb{Z}_{2^n}$ private $\vec{v} \in \mathbb{Z}_{2^n}$	Addition	[25]
	Multiplication	[27]
	Equality	[27]
	Greater-than	[27]
	Division	[27]
	Remainder computation	[27]
private $\vec{u} \in \mathbb{Z}_{2^n}$ public $\vec{v} \in \mathbb{Z}_{2^n}$	Multiplication	[27]
	Division	[27]
	Remainder computation	[27]
private $\vec{u} \in \mathbb{Z}_{2^n}$ public $\vec{v} \in \mathbb{Z}_{2^n}$	Shifting bits left v places	[27]
	Shifting bits right v places	[27]
private $\vec{u} \in \mathbb{Z}_2$	Conversion of shares from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$	[27]
private $\vec{u} \in \mathbb{Z}_{2^n}$	Random element shuffling	[87]

Table 3.1: The most efficient known secure protocols in the SHAREMIND model.

contained protocols for secure addition, multiplication, comparison and bit extraction. The addition protocol was derived naturally from the additively homomorphic property of the secret sharing scheme. Multiplication was achieved by extending an atomic protocol proposed by Du and Atallah [53] which is based on the standard multiplication triple solution [11]. The other protocols were implemented as compositions of the addition and multiplication protocols. All the protocols were shown to be universally composable in the passive security model. An optimized version of this work was benchmarked and the SHAREMIND system was introduced to the wider scientific community at the ESORICS 2008 conference [25].

In the following years, protocol development continued and, based on a new approach to multiplication, a new protocol set was designed. The existing arithmetic operation protocols were rewritten from scratch and new ones were proposed for bit shifting, division and remainder calculation. The original protocol suite was extensible to a number of computing parties different than three, given that a multiplication operation exists. The new protocol suite was significantly more efficient, but this was achieved by limiting the protocols to three computing parties. SHAREMIND version 2 was completed in 2010. For more details on the protocol suite in SHAREMIND 2, see [27].

More high-level protocols for database operations like oblivious selection, filtering, shuffling and sorting were built on the secure arithmetic protocols and they were published in [73, 87].

Work on the SHAREMIND protocol suite is an ongoing effort. Our goal is to add support for more operations on integers and also start supporting other data types such as fractional numbers. Furthermore, we plan to extend the SHAREMIND secure computation paradigm to different numbers of computing parties. This work will be done for future versions of SHAREMIND.

3.7 The software implementation of SHAREMIND

SHAREMIND is implemented in the form of two software components. The first component is the SHAREMIND server (also called *miner* server). It performs the duties of a computing party by establishing secure channels to other servers, executing secure computation protocols and providing services to the input parties and result parties.

The interfaces for the input and result parties are implemented as *controller* applications using the controller library. The controller library provides an interface for sending data and issuing requests to the SHAREMIND miner server. The controller library performs automatic secret sharing and hides the cryptographic details of secure computation from the developer and the user. Similarly, the results are reconstructed from shares and comprehensible values are returned to the user. SHAREMIND is modular regarding the size of the shares. It can work equally well with shares of any size that the protocol suite supports.

SHAREMIND is implemented in C++ for efficiency reasons. SHAREMIND 2 uses the RakNet library [104] for networking because of its low overhead. Boost libraries [34] are used for cross-platform threading and configuration processing.

Various database libraries are used to implement the database interfaces. The default database in SHAREMIND 2 is based on SQLite [119]. SHAREMIND supports the Tokyo Cabinet database [125] and any ODBC-compatible database using the ODBC connector libraries. Finally, the built-in profiling and performance analytics tools use the libcsv library [43].

We discuss the tools and applications associated with the SHAREMIND system in the following chapters.

CHAPTER 4

PRACTICAL PERFORMANCE OF SHAREMIND

4.1 The complexity and performance of SHAREMIND

SHAREMIND supports a range of secure computation operations. All the protocols described in Section 3.5 listed in Table 3.1 in Section 3.5.4 are implemented using a number of core protocols. Table 4.1 lists their round and communication complexities.

The communication and round complexities of secure computation protocols depend on the bit length of the data element n and its logarithm $\ell = \log_2 n$. Integer division depends also on precision parameters n' and m computed using the error calculation of Goldschmidt division [27]. For 32-bit unsigned integers, $n = 32$, $\ell = 5$, $n' = 37$ and $m = 254$. Secure addition is a local operation and therefore takes no rounds and requires no communication.

The performance of SHAREMIND has been measured and analyzed in several works. The performance of the original protocols was given in [25]. The benchmarks in this paper were conducted on a distributed systems research cluster where each machine contained a dual-core CPU, 2 GB of RAM and exchanged information over a gigabit network.

SHAREMIND 2 uses protocols described in [27]. The paper also contains a more detailed performance analysis. The original protocols from [25] were compared to the new ones. Both protocol suites were benchmarked on a dedicated experimental cluster consisting of three servers where each machine had 24 CPU cores and 48 GB of RAM. Also, each machine had a direct gigabit Ethernet interface to other nodes and a 100 Mbit interface to the public internet. This setup was used to show that the new protocols are significantly more efficient and robust than the earlier ones.

It is easy to see that this setup is ideal for SHAREMIND and a separate analysis

	SHAREMIND 2 [27]		SHAREMIND 1 [25]	
Protocol name	**Rounds**	**Data bits**	**Rounds**	**Data bits**
Addition	0	0	0	0
Multiplication	1	$15n$	3	$24n$
Cast $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$	2	$5n+4$	4	$39n$
Equality	$\ell+2$	$22n+6$	$\ell+9$	$87n^2-18n+2\ell(n-1)^2$
Bit shift right	$\ell+3$	$12(\ell+4)n+16$	—	—
Bit extraction	$\ell+3$	$5n^2+12(\ell+1)n$	$\ell+8$	$63n^2+6n+2\ell(n-1)^2$
Division with public divisor	$\ell+4$	$(108+30\ell)n+18$	—	—
Division with private divisor	$4\ell+9$	$2mn+6m\ell+39\ell n+35\ell n'+126n+32n'+24$	—	—

Table 4.1: Complexities of core SHAREMIND protocols [25, 27].

must be conducted in a setting with more common network configurations. A more detailed analysis of that setting is described in Section 4.3.4.

We benchmarked protocols in SISD (single operation, single data) and SIMD (single operation, multiple data) modes. For the SIMD case, we present the best speed that was achieved. All speeds are presented in operations per second. Table 4.2 contains the performance results.

4.2 Benchmarking methodology

4.2.1 The built-in protocol profiler

The SHAREMIND implementation contains several built-in features for measuring the performance of protocols. The protocol profiler traces the execution of each secure computation protocol and SHAREMIND assembly script and measures the time spent on executing an assembly program, that consists of:

1. time spent on executing a secure computation protocol, consisting of
   (a) time spent on generating randomness,
   (b) time spent on database operations,
   (c) time spent on sending messages,
   (d) time spent on receiving messages,

		SHAREMIND 2 [27]		SHAREMIND 1 [25]
		SISD	SIMD	SIMD
Private u private v	Addition	10^5 ops	$5.9 \cdot 10^7$ ops	$5.9 \cdot 10^7$ ops
	Multiplication	39 ops	$5.6 \cdot 10^5$ ops	$2.9 \cdot 10^5$ ops
	Equality	10 ops	$2.0 \cdot 10^5$ ops	$4.5 \cdot 10^2$ ops
	Comparison	8 ops	$6.4 \cdot 10^4$ ops	$2.2 \cdot 10^2$ ops
	Division	2 ops	$1.7 \cdot 10^3$ ops	—
Private u public v	Multiplication	10^5 ops	$1.8 \cdot 10^8$ ops	$1.8 \cdot 10^8$ ops
	Division	8 ops	$1.9 \cdot 10^3$ ops	—
Private u	Cast $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$	65 ops	$1.2 \cdot 10^6$ ops	$5.6 \cdot 10^4$ ops
Private u	Bit extraction	9 ops	$1.9 \cdot 10^4$ ops	$7 \cdot 10^2$ ops

Table 4.2: Speed of SHAREMIND in secure operations per second (ops) [27, 25].

(e) time spent on waiting for messages to arrive,

2. time spent on code interpretation.

If the profiler is enabled on a SHAREMIND server, it caches the measured timings in memory and stores them in a file in a background process. The profiler collects measurements according to the hierarchy presented in the list—each measured execution section is assigned a parent section. Once the measured task is complete, we can store the section durations and their hierarchy in a file and use a separate tool to output the timing breakdown of a script or a protocol.

We chose a hierarchical profiling method to be able to analyze the performance of a script with different granularity levels. For example, if we want to know the bottlenecks of an algorithm, we can analyze the timings on the level of individual operations and discard the runtime breakdown of protocol execution. However, if we are optimizing a protocol and want to know the time spent on network delays, we can discard the higher level and focus on a single protocol and its breakdown.

Each section can also be tagged with metadata, like the size of inputs to the secure computation protocol or the size of the vector to send on a network. This information is preserved in the post-processing phase and allows us to analyze large benchmarks with ease.

4.2.2 Benchmarking tools

There are specific tools for benchmarking protocols and assembly programs. The `OperationBenchmark` tool is used to coordinate performance tests for secure computation protocols. The user provides the tool with the name of the secure

operation to test and the number of iterations for testing each input size. It is also possible to specify the range of input sizes to use and the order of experiments by input size—ascending, descending or random.

The range specification controls which input sizes are tested. For example, the user can request that the secure multiplication operation is benchmarked with input sizes ranging from 100 000 to 1 000 000 with 100 000-element increments.

Given this information, the `OperationBenchmark` tool will generate the list of experiments to perform. This list contains one experiment for each iteration for each input vector size. These experiments can then be performed in ascending order from the smaller vector sizes to the larger ones or in reverse.

Optionally, the order can be randomized to reduce any effects that may result from a sorted ordering, such as the ones resulting from the flow control algorithms of the underlying networking layer. Furthermore, there is an additional option to "warm up" the machine by performing a number of large secure multiplications. This ensures that the flow control algorithm has converged on an estimate for the maximum speed and network throughput has stabilized. In our experiments, we always apply a warmup period and randomize the order of the experiments.

Experiments are coordinated by a result party that requests that the computing parties generate random inputs and then perform the experiment. The `OperationBenchmark` tool also measures the execution time on the client side. However, this timing also includes the round trip time between the result party and the computing parties that is not present in the results of profiling. In our benchmarking, we use the profiling results, because in algorithms, operations are run one after the other with no interactions with the result party.

There is a separate tool called `ScriptBenchmark` that is used to measure the execution of an assembly program. The tool sends the program name to the computing nodes that execute the program and profile it. `ScriptBenchmark` also measures execution time on the client side.

Client-side measurements are more justified in this case, as queries in most real applications consist of requests exactly like this. Therefore, the measurements of `ScriptBenchmark` are a good indication of query response times in real applications. Profiling results can also be used during development to understand the breakdown of the execution time between individual secure operations.

4.3 Performance analysis

4.3.1 SHAREMIND protocol execution pipeline

Before we analyze how different resources affect the performance of secure protocols on SHAREMIND, we describe how protocols perform computations and exchange messages. We focus on how a protocol running on one SHAREMIND

miner server sends a message to a protocol running on another miner server. Once the protocol instance in one miner server sends a message, it needs to be passed through the SHAREMIND network layer, encrypted and sent over the public network, then decrypted and passed to the protocol instance in the destination miner server.

SHAREMIND is capable of SIMD operations that are performed by a single protocol. Whenever such a protocol would process a value, it processes a vector instead. This means that SHAREMIND can pack the messages of several secure computation protocols into a single network message and reduce the networking overhead We will give more details on the effectiveness of this optimization in Section 4.3.3.

On the other hand, if an application uses a vector that is very large, it needlessly increases the memory usage and also the risk of failure in the networking layer. Therefore, we introduce a configurable batching parameter b. All protocols are implemented so that input data will be processed in batches with a maximum size b. For example, in a pointwise multiplication with 100 million values, if the batching parameter $b = 100000$, SHAREMIND will slice 100000-element pieces from the inputs and will run the protocol in 1000 batches.

As SHAREMIND supports multiple parallel users, it also has to manage parallel protocol instances and share the network layer among the users' sessions. The protocols of each user session run in a separate thread. If they want to send or receive messages, they inform the main thread of the miner server. The miner thread talks to the networking thread every few milliseconds to send messages and pick up everything that has arrived.

When messages are passed to the network thread, it encrypts them and buffers a copy for retransmissions. The messages are then sent out as UDP datagrams. Similarly, when messages arrive from the network, they are decrypted, acknowledgments are sent and they are buffered until the miner server thread requests them.

Most of the operations in sending are asynchronous (sending is non-blocking). On the other hand, receiving is a blocking operation so the protocol thread simply waits until the responses arrive. In the future, we will consider making message receiving more asynchronous as well, as it may improve efficiency. However, this will require a review of the protocols to make sure that the values exchanged on the network are not affected by the scheduling of network packets.

This design may not provide the lowest latency, but it can robustly handle large vectors. The networking pipeline has been optimized for performing large SIMD operations, as SHAREMIND is more efficient with them. Figure 4.1 illustrates the whole pipeline between miner servers A and B.

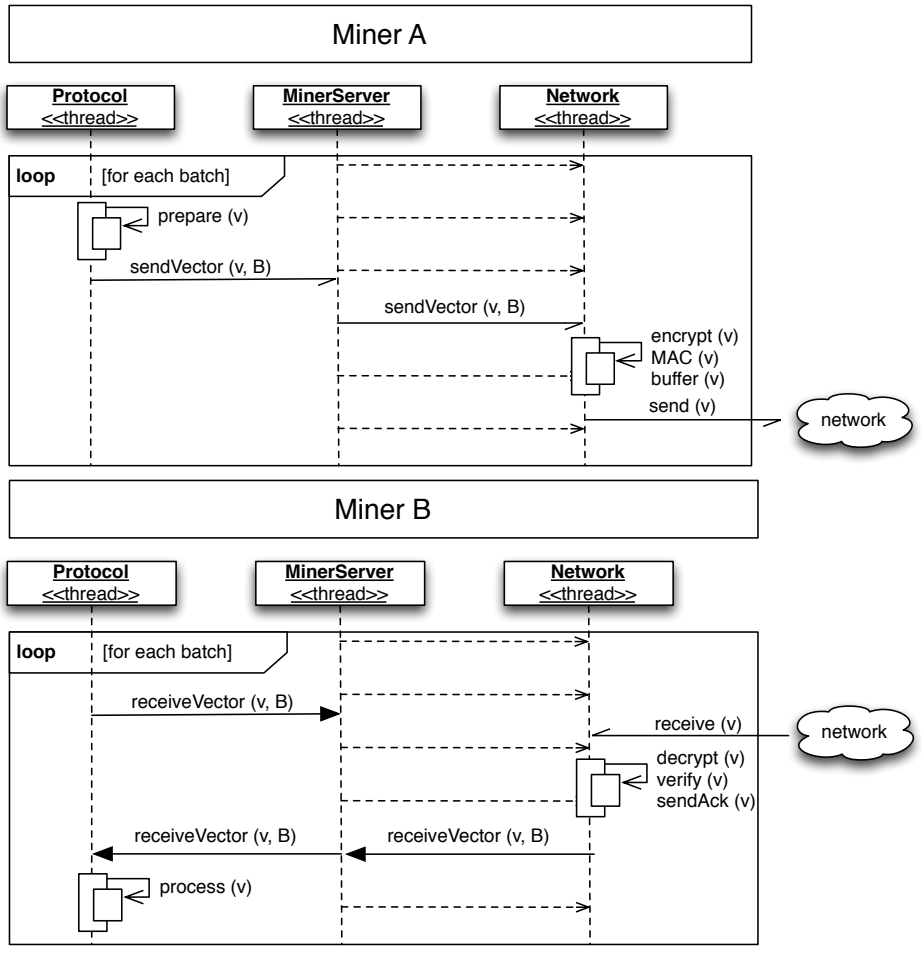


Figure 4.1: Message exchange during SHAREMIND protocol execution

4.3.2 The importance of processor speed

SHAREMIND is based on very simple ring arithmetics and, therefore, does not require heavy-duty cryptographic primitives to operate. However, there are two important cryptographic primitives that use up processing power in SHAREMIND. The first is the suite of encryption schemes that enable the secure channels in the system. The second is the pseudorandom generator that enables secret sharing and provides randomness for all protocols.

Modern secure channel implementations are considered efficient enough for use in mainstream applications. Secure channels functionality in SHAREMIND 2 is provided by the RakNet library. RakNet creates a secure channel with authenticated encryption using 256-bit elliptic curve key agreement, a key derivation func-

tion based on Skein [117], the ChaCha stream cipher [19] and a MAC based on HMAC-MD5. ChaCha is not a standard stream cipher, but known attacks against it are not yet in a practically feasible range [9]. Similarly, while the MD5 hash function is no longer considered collision resistant and thus not secure in the context of digital signing, HMAC-MD5 is not dependent on its collision resistance properties [13].

The performance cost of secure channels in the SHAREMIND system has not been studied in detail. The current assumption is that as RakNet is mostly used in real-time computer games, its secure channels must be efficient enough for performance-critical applications. We also plan to experiment with more standard implementations of secure channels such as TLS in future SHAREMIND versions. TLS was considered inefficient at the time when SHAREMIND was started. However, recent developments such as the hardware-implemented AES-NI instruction set have made TLS significantly faster and it should be evaluated for inclusion in future versions of SHAREMIND.

SHAREMIND protocols require significant amounts of randomness. In fact, the amount of randomness consumed by a protocol is roughly the same as its communication cost. Therefore, we need a source of randomness with a high output rate. For practical reasons, we currently use pseudorandom generators in the implementation.

A preliminary analysis of the effects that the type of the source of randomness has on secure computation speed has been conducted in [29]. We experimented with several pseudorandom generators and then profiled SHAREMIND with two of them. The first is an ANSI X9.17 Appendix C compliant randomness generator based on the AES block cipher and the second is the SNOW 2 stream cipher [55].

Our profiling showed, that with the AES-based generator, SHAREMIND spends a significant amount of time generating randomness for the protocols. By substituting the generator for one based on SNOW 2, the randomness generation part in the execution profiles was reduced from a few seconds to a few milliseconds. For this reason, SHAREMIND 2 uses the SNOW 2 stream cipher as its source of randomness. However, the pseudorandom generator is modular and can be replaced with other implementations. Most importantly, the hardware-accelerated AES-NI instruction set could be used to build an efficient randomness generator that is faster than the current software implementation based on SNOW 2.

4.3.3 The importance of parallelization

Assuming that we have a fast random number generator and efficient secure channels, the next major bottleneck in SHAREMIND is network throughput. The more complex the secure operation, the more rounds and communication it uses. In each protocol round, several messages are exchanged among the SHAREMIND

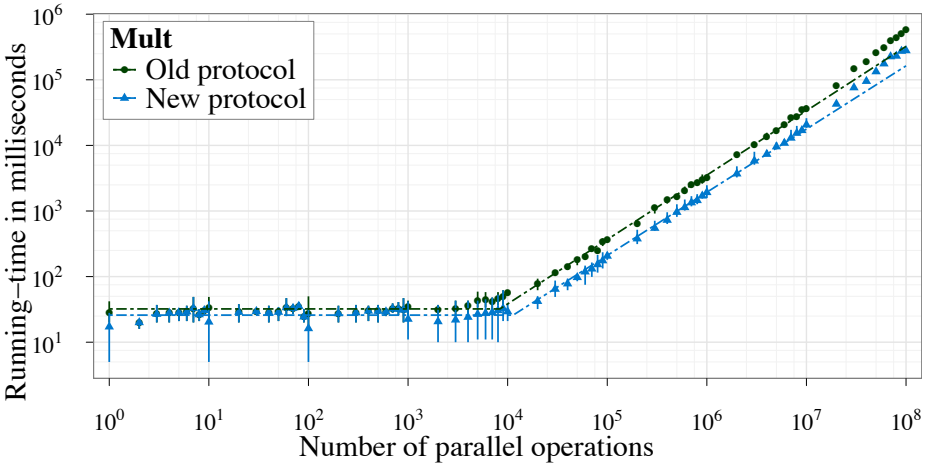


Figure 4.2: Running time of secure multiplication depending on the input vector size [27].

computing nodes. As sending each message has a similar overhead, it makes sense to try to use each message to process several values in parallel.

SHAREMIND implements parallelization by allowing an operation to take any number of inputs (or input pairs, in the case of a binary operation). The protocols perform all operations in parallel and package the values of several secure operations into a single network message. We also remind the reader that such parallel composition is secure thanks to the universal composability property of SHAREMIND protocols.

Figure 4.2 is based on actual experiments and shows that the running time of secure multiplication does not grow significantly with the size of inputs, until a *saturation point* is reached. At that point, the running time starts to grow linearly with the size of inputs. On the figure, this is represented with two linear fits.

Figure 4.3 presents the same data from another angle. It plots the amortized cost of a single operation depending on the size of the inputs. The lines show the same significant speedup up to the saturation point. Table 4.3 shows similar speedup factors for various SHAREMIND 2 protocols. For each protocol we show the most efficient input size and the speedup achieved by performing operations with that input size.

Given the promise of large performance gains, it makes sense to set the batch size parameter for a protocol to be not much higher than its saturation point. This ensures that we are using the best parallelization, but also that we are not wasting memory on huge vectors. However, the batch size must not be too small, as slicing the input vectors for each batch will then reduce the efficiency of the protocol.

It follows, that the main optimization goal of a data processing algorithm run-

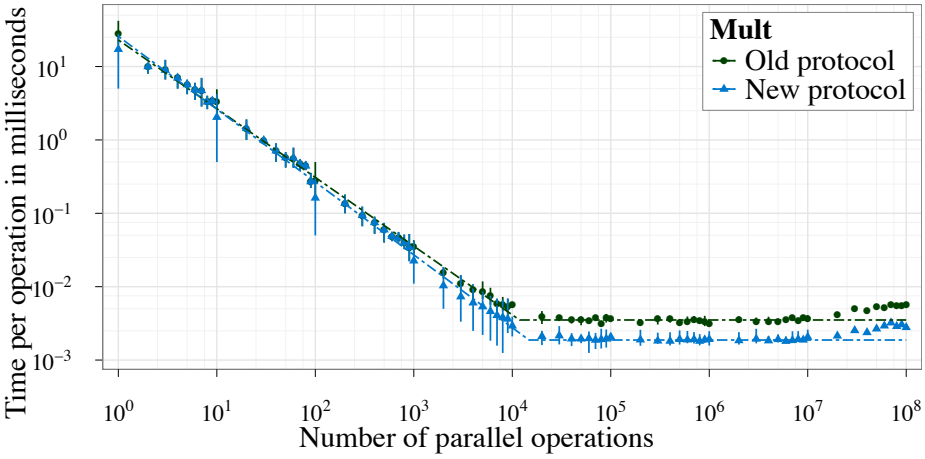


Figure 4.3: Price of a single multiplication depending on the size of the input vector [27].

Protocol	Lowest efficient input size	Speedup factor
Multiplication	15000	14000
Cast $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$	24000	19000
Equality	27000	20000
Bit shift right	12000	7700
Bit extraction	2600	2200
Division with public divisor	3500	2800
Division with private divisor	800	730

Table 4.3: SHAREMIND performance improvements from parallelization [27].

ning on SHAREMIND should be to process many values at once. It is safe to parallelize as much as possible, because SHAREMIND can automatically execute the protocol with the most efficient calibrated batch size. On the other hand, making a non-parallel algorithm use more parallel operations is a known hard problem.

Making an algorithm use parallel operations increases its memory footprint and also requires additional computational resources. Also, not all algorithms can be easily made parallel as some of them have strong sequential data dependencies. However, it pays off to try, as the possible speedup of more than 10 000 times should be a sufficient motivator for designing one's algorithms around massive parallelization. Also, practical tests have shown, that the overhead of building the input vectors for parallel processing is significantly smaller than the achieved reduction in running time.

In our analysis present in [27], we show that all the secure computation proto-

cols in SHAREMIND that use communication behave similarly. The main change is in the position of the saturation point among the different input sizes. For more complex protocols, the saturation point is lower, as the larger communication requirement fills the network capacity more quickly.

The benchmarks also illustrate another important fact—SHAREMIND is robust enough to handle very large input vectors. Figures 4.2 and 4.3 show multiplication operations processing up to 100 million input pairs at a time. In some data mining prototypes, SHAREMIND has successfully processed input vectors ranging to half a billion values. This is possible thanks to the batching mechanism described in Section 4.3.1.

4.3.4 The importance of network bandwidth and latency

Up to now, we have described the performance of SHAREMIND in near-ideal settings where the network links are very fast. In real-life deployments, we cannot always assume that this is possible. Therefore, we need to understand the effects of lower bandwidth and higher latency on secure computation performance.

We carried out performance experiments in various network settings and analyzed the results as joint work with Reimo Rebane [106]. We configured the experimental SHAREMIND cluster with special software that allowed us to emulate network links with lower bandwidth and higher latency. We performed experiments with a variety of network settings and built a linear regression model that linked the performance of secure operations with the network settings.

The analysis of the model indicated that an increase in network latency does not decrease the efficiency of secure operations on vectors. This is likely because SHAREMIND protocols use a single message for many parallel operations and the transfer of larger messages takes some time in any case. Furthermore, the secure computation performance is strongly dependent on the communication complexity of the protocol. As bandwidth decreased, so did the performance. Similar patterns were noticed for all protocols.

Another goal of the work in [106] was to experiment with SHAREMIND on the public cloud. This was driven by the practical consideration that it must be feasible to deploy SHAREMIND on the cloud as this will help with the adoption of the new technology. Each SHAREMIND installation requires three servers hosted by three independent hosts and controlled by separate entities. However, not all organizations have the capability of hosting a SHAREMIND server. Cloud service providers can rent the infrastructure required for deploying SHAREMIND to organizations with no such capability.

Rebane experimented with SHAREMIND using several cloud service providers. We chose two different settings for the experiments—the worst case scenario, where SHAREMIND servers are deployed as distantly from each other as possible,

and a more realistic scenario where the computing nodes are in the same part of the world—Europe. To determine, how far these settings are from the ideal lab setting, Rebane also compared the performance on the cloud to the performance in the lab.

In the European deployment, SHAREMIND was installed on servers provided by three different cloud providers located in the United Kingdom and Ireland. In the global deployment, one server was set up in the United States, one in the United Kingdom and one in Japan.

	Deployment		
Operation	**Lab**	**European cloud**	**Global cloud**
Multiplication	690000 ops	48300 ops	27850 ops
Share conversion	1360000 ops	120000 ops	58500 ops
Equality	254000 ops	21500 ops	12800 ops
Bit shift right	95000 ops	3000 ops	2600 ops
Bit extraction	28000 ops	1000 ops	600 ops

Table 4.4: SHAREMIND performance on the cloud in secure operations per second [106].

Table 4.4 gives a comparison of performance for five operations in the three settings. The data shows a slowdown factor of 10–30 times in the performance of secure operations. Even with decreased performance, SHAREMIND remains practical as it can still perform tens of thousands of secure operations each second.

	Deployment		
Operation	**Lab**	**European cloud**	**Global cloud**
Multiplication	27.79 Mbit/s	1.02 Mbit/s	0.57 Mbit/s
Share conversion	13.65 Mbit/s	1.17 Mbit/s	0.73 Mbit/s
Equality	18.84 Mbit/s	1.54 Mbit/s	0.42 Mbit/s
Bit shift right	39.96 Mbit/s	2.03 Mbit/s	0.84 Mbit/s
Bit extraction	24.85 Mbit/s	1.28 Mbit/s	0.78 Mbit/s

Table 4.5: Average network throughput comparison in the lab and on the cloud [106].

Table 4.5 shows the average bandwidth during the performance measurements. It clearly shows similar ratios between performances and bandwidths. However, the maximal measured bandwidth between individual SHAREMIND servers in the cloud (and also, in the lab) was significantly higher. For example, the lab deployment had direct gigabit network links between the servers, of which only up to 40 Mbit/s were reported as used during the experiments.

The results reported in [106] show that SHAREMIND was not using all the available bandwidth in the experiments. This suggests that there may be room for optimization in the SHAREMIND protocol execution pipeline. Further profiling is needed to determine, which computational step (secure transport, randomness generation or other) is limiting bandwidth usage.

Furthermore, the effect of communication complexity in the overall performance must be taken into account during the design of new protocols. We hypothesize that the best secure computation protocols are balanced—their rounds have a similar communication complexity so that the batching system works equally well on all the rounds of the protocol. Otherwise, if a protocol has rounds with significantly unbalanced communication, it is hard to choose a batch size that guarantees an even flow of data in all the rounds.

4.4 Optimization goals for future protocols

SHAREMIND protocol designers are faced with many choices as they create new protocols for the system. For example, should the number of rounds be the main optimization goal or should we limit communication instead? We propose general guidelines based on an analysis of a simplified model for SHAREMIND protocols.

Consider the protocol execution pipeline illustrated in Figure 4.1. Message transmission in this model has an inherent amount of latency. It is caused by the combination of local processing and network transfers. There are many factors that contribute to this latency, ranging from thread scheduling and other processes on the operating system to the physical parameters of the network connection. For these reasons, message transmission latency is stochastic and it cannot be avoided completely.

However, we can assume that the time of transferring a message over the network is mostly in the size of the data. Copying messages between threads, encrypting and authenticating them are all linear activities. The physical latency of the network is dependent on the quality of the network connection and not on the size of the data. The physical latency between two non-mobile nodes is typically constant over time, unless the network infrastructure between these nodes is overloaded or improved.

From this, we can derive a simplified statistical model for message delivery time. Let n be the message size in bits, b the bandwidth of the network in bits per millisecond and ℓ the roundtrip time on the network in milliseconds. We can express the transmission time t as

$$t = \frac{n}{b + \varepsilon_b} + \ell + \varepsilon_\ell \ , \tag{4.1}$$

where ε_b and ε_ℓ are random error terms for bandwidth and error, respectively.

Most secure computation protocols in SHAREMIND exchange messages in several rounds. In principle, one can create a model of the protocol running time by extending Equation (4.1) with details about the messages exchanged in each round. The resulting model of the protocol execution time can be used for determining how the number of rounds or the communication complexity affects the total running time. However, this approach is not practical without special tools to assist the developer in the analysis.

We can still derive useful rules from the simplified model (4.1). If the time cost of network latency is larger than the cost of transmitting the data bits so that

$$\frac{n}{b+\varepsilon_b} \ll \ell + \varepsilon_\ell \ ,$$

then the protocol running time grows linearly in the number of rounds. In this case, we should prefer protocols with a lower number of rounds. This situation is more probable, if we are running the protocol with just a few inputs, as otherwise the message size grows and the impact from the bandwidth becomes more significant. Therefore, if we know that a protocol will mostly be used with small inputs, we should aim for reducing the number of rounds in the protocol.

If the amount of transmitted data is sufficiently large then latency becomes less important as the time taken to transmit bits grows to be greater than the accumulated latency. Parallel operations on vectors can easily make the data size grow and reduce the importance of round complexity in protocol design.

It follows that developers who optimize secure computation protocols for SHAREMIND or a similar system must find a balance between bandwidth and latency. If the protocol must work on small inputs, one should aim for fewer rounds. If the protocol is intended to be run with multiple inputs, the designer should take steps to reduce the communication complexity of messages.

With SHAREMIND, one also has to understand that as the messages grow, SHAREMIND starts to automatically split messages to keep the input sizes of protocols near the saturation point. However, each piece of the message requires its own round trip time and this adds to the number of rounds. The perfect balance between the number of rounds and communication complexity is not yet known, but first steps towards finding this balance have already been taken [106].

CHAPTER 5

PROGRAMMING SECURE COMPUTATIONS

5.1 Motivation and design goals

In Chapter 3 we showed how to perform secure computations on SHAREMIND and presented general protocols for collecting data and processing them using secure computation. In Section 3.5, we presented a protocol for processing queries received from result parties. In practice, we need a way for specifying the algorithms that control the secure computation protocols in these queries. Our solution is to use domain-specific programming languages to specify the secure operations that SHAREMIND must perform to complete the query.

We set several goals for the whole programming experience.

1. The programming language must support operations on both public and private data.
2. The programming language must clearly separate public and private data and control when private data becomes public.
3. The programming language must be independent of any particular secure computation paradigm.
4. The programming language must provide tools that simplify the implementation of algorithms that process large databases.
5. It must be possible to run the programs written in the language on the SHAREMIND computing parties.

The choice of these particular goals were driven by several considerations.

First, the decision to include public operations in the programming model comes directly from the efficiency goal of SHAREMIND. It is not feasible to hide the whole state space of the secure computation. Each branching statement increases the number of parallel states we need to secure and maintain, because otherwise the security can be compromised using side channel attacks like timings. The situation is worse for loops, because each loop condition is basically a branching decisions and hiding the size of the loop either becomes very expensive or leaks bits about the loop condition. Also, most circuit evaluators unroll loops, thus needing to know the size (or, at least the maximum size) of the loop.

Instead, we focus on hiding the values in private data and supporting programming patterns that hide the control flow and defeat side-channel attacks. By restricting flow control to decisions based on public values we gain a lot of efficiency. For example, loop conditions can be evaluated using only public values, condition statements can be made only on public values. The naïve way for handling branching in secure circuits is to evaluate both branches and obliviously choose the results. It is easy to see how this can quickly grow the computation complexity of the program.

We can use significantly less secure computation resources by allowing the programmer to combine both public and private decisions in a program. For example, loop conditions can be evaluated publicly during runtime without requiring secure operations.

The second goal of SHAREMIND programming language design partially follows from the first. In order to prevent programming errors where private values are used instead of public ones, the programming language must keep a strict separation between the types. In our design we decided to allow the implicit conversion of public values to private values but allow only explicit conversions of private values to public values. The type system of the programming language takes care of the separation and enforces all possible assignments in the compiler.

Although the SHAREMIND design fulfills the goals we set for the secure computation engine, we acknowledge that cryptographic research moves on and more protocols can emerge, providing a comparable degree of efficiency and ease of use. Therefore, to keep the language independent from the underlying cryptographic protocols, we refrained from including protocol-specific constructs like parties in the programming language. The cryptographic parts of SHAREMIND are hidden in the type system and compiler. This allows us to adapt the language to other secure computation paradigms such as homomorphic encryption.

Fourth, we believe that secure aggregation and data mining will be a significant use case for SHAREMIND and, therefore, we intend to provide tools for the creation of such applications. This includes adding vector and matrix types to the language and supporting pointwise operations on these types. Furthermore, we

chose to design an imperative language to simplify the porting of data processing algorithms in the literature.

Finally, it is our goal to provide a runtime for the language. To achieve this without becoming protocol-specific, we created a low-level assembly language that is specific to the SHAREMIND system and supports all the operations supported by SHAREMIND. We then proceeded to design a high-level language called SECREC (pronounced as *secrecy*) that meets all the goals we have set and created a compiler to translate SECREC programs into SHAREMIND assembly language. As a final link, we added an assembly interpreter to the SHAREMIND machine, allowing SECREC language programs to be executed so that the separation of private and public data is enforced by the SHAREMIND runtime.

5.2 The SHAREMIND secure virtual machine and assembly language

The SHAREMIND assembly language is an interpreted language. The interpreter is implemented within SHAREMIND where it interfaces directly with two virtual machines—the privacy-preserving virtual machine that runs secure multiparty computation protocols and the public virtual machine that performs public operations.

The abstract machine of SHAREMIND 2 is a hybrid of a stack machine and a register machine. A private stack is used for passing operations to private operators. This allows the programmer to easily set up input vectors for larger SIMD operations. For intermediate results, however, there are public and private registers (including vector registers). The runtime state of the virtual machine is formed by the contents of the stack, the registers and also the private database.

As the main data type of the SHAREMIND 2 implementation is the 32-bit unsigned integer, it is also the type of the private stack. Public and private registers can contain scalar or vector values of the same type. Public registers can also store string data to help with the processing database metadata and handle logging.

To run a secure operation, the programmer pushes the inputs on the stack from either the database or the registers. The programmer can then invoke the secure computation operation that will take its inputs from the private stack and, after completion, will put the results back on the stack. Private operations can take parameter vectors of any size from the stack.

To speed up the implementation process, public expressions have been directly integrated into the assembly language. Public arithmetic works directly on registers.

SHAREMIND also has operations for loading data from the database and saving them to the database. For efficiency, the operations can push database columns

directly on the stack for immediate processing. This allows the programmer to optimize the code and skip the copying of large vectors through registers.

Even though the assembly interpreter keeps a strict separation between the public data and the private data, it does not have many built-in restrictions on making private data public. There are some operations (e.g., moving data from a private register to a public register, popping values from the private stack to a public register) that trigger the collection of shares and the reconstruction of secrets. As the assembly language represents the "hardware" layer in SHAREMIND, it must provide such operations. We give a more detailed description of the programming model of the SHAREMIND virtual machine in [24].

In the majority of cases, secure functionality is developed in the high-level SECREC programming language and therefore, it is useful to implement the relevant restrictions in that language. The translation from SECREC to assembly language must preserve the same security guarantees. It is possible to show formally, that the translation from a high-level language to a low-level language preserves the claims about the information flow. These claims include the movement of data between public and private types.

However, even with such claims, the compiler may contain errors that cause the generation of incorrect assembly code. One solution is to create a compiler that generates a proof of the data flow claims and embeds this proof in the assembly code. The interpreter of such proof-carrying assembly code will then be able to check the information flow claims during execution. Even then, it is important to control the quality of the compiler through testing and analysis of the resulting code, as developer errors may still create security risks. At the time of writing this thesis SHAREMIND does not use proof-carrying assembly code and adding support for this remains a future goal.

The assembly language interpreter of SHAREMIND 2 has been built and integrated into SHAREMIND as joint work with Roman Jagomägis [70]. The interpreter is fully functional and is powering the SECREC programming language.

5.3 SECREC—a high-level imperative language for implementing secure functionality

5.3.1 Secure data types

SHAREMIND assembly provided the hardware abstraction for secure computation protocols. We used this abstraction to design and implement the SECREC (*secrecy*) programming language that hides the details of the secure computation protocols. SECREC is a C-like language that separates public and private data on the type system level. Variables that are typed as private are processed using

```
// Declarations for public values.
public bool publicValue;
public uint32[10] publicVector;
public uint32[3][3] publicMatrix;
public string tableName;

// Private scalar value, vector and matrix declarations.
private uint32 secretValue;
private bool[5] secretVector;
private uint32[10][10] secretMatrix;
```

Figure 5.1: Variable declarations in SECREC

secure computation whereas public values are stored and processed as usual.

Each type in SECREC consists of a *data type* and a *security type*. As the SHAREMIND 2 platform works with 32-bit unsigned integers, the `uint32` data type in SECREC is a 32-bit unsigned value. There is also a `bool` type that is emulated on 32-bit shares. The `string` and `void` data types are only available for public types. The latter is used only for methods with no return value. Figure 5.1 shows how to declare variables in SECREC.

5.3.2 Secure operations and parallelism

As parallel operations are efficient on SHAREMIND, SECREC supports pointwise operations on vectors and matrices. There are also operations for aggregating whole vectors of values (e.g., summing them). Table 5.1 shows the available binary and unary operators for processing private data in SECREC. All the listed operators also work on scalar values and public variables. Additionally, SECREC can expand a scalar value to the same shape as a vector or matrix to simplify such operations for the programmer. See Figure 5.2 for examples.

Other functions are available through the standard library of the language. The reference is given in the documentation of the software development kit [115].

In Chapter 4, we showed that the use of parallelization can significantly simplify and optimize code. As processing a single value may take the same amount of time as processing a thousand values, SECREC programs should be designed so that they use pointwise operations and built-in aggregations as much as possible.

Figures 5.3 and 5.4 show two functions that perform a similar task. Both functions count the number of occurrences of a private value in a private vector (an interesting subtask in, e.g., histogram computation). The `countFast` function performs fewer operations, but these have a higher degree of parallelism.

Operands	Operation
private uint32[] a private uint32[] b private uint32[] c	c = a + b;
	c = a - b;
	c = a * b;
	c = a / b;
	c = a % b;
	c = -a;
private uint32[] a private uint32[] b private bool[] c	c = a < b; c = a > b;
	c = a <= b; c = a >= b;
	c = a == b; c = a != b;
private bool[] a private bool[] b private bool[] c	c = a || b;
	c = a && b;
	c = !a;

Table 5.1: Secure binary and unary operations in the SECREC language.

```
// Declare data.
private uint32 threshold;
private uint32[10] data;
private bool[10] result;
// Expand threshold into a vector.
private uint32[10] thresholdVector = threshold;
// Evaluate greater-than pointwise to get a boolean
// vector where a ``true'' means that the value at this
// position was greater or equal in the first parameter.
result = (data >= thresholdVector);
// Now count the number of true values.
private uint32 count = vecSum (result);
```

Figure 5.2: Pointwise and aggregation operations in SECREC

In Section 4.3.3, we showed that a single operation on a thousand values can take nearly the same amount of time than a single operation on one value. Therefore, the algorithm in Figure 5.4 has a significantly smaller running time on SHAREMIND than the algorithm in Figure 5.3.

In these examples, all intermediate values are declared to show the security types of each value and convince the reader that the secure inputs are not made public in the algorithm.

```
private uint32 count (private uint32[] data,
                      private uint32 value) {
  // Get the size of the data.
  public uint32 size; size = vecLength (data);
  // Loop over values and perform private comparisons.
  public uint32 i = 0;
  for (i = 0; i < size; i = i + 1) {
    // Perform secure comparison.
    private bool match; match = (data[i] == value);
    // Cast to integer (true = 1, false = 0) and add.
    public uint32 matchInt = boolToInt (match);
    matchcounter += match;
  }
  // Return result as a private value.
  return matchcounter;
}
```

Figure 5.3: A non-parallel counting function in SECREC

```
private uint32 countFast (private uint32[] data,
                          private uint32 value) {
  // Get the size of the data.
  public uint32 size; size = vecLength (data);
  // Expand the input to a vector.
  public uint32[size] valueVector = value;
  // Perform parallel comparison.
  private bool[size] matches;
  matches = (data == valueVector);
  // Cast and sum up the results.
  private uint32 matchcounter;
  matchcounter = vecSum (matches);
  // Return result as a private value.
  return matchcounter;
}
```

Figure 5.4: A parallelized counting function in SECREC

5.3.3 Making private data public

The only way to make a private value public in SECREC is to pass it to the `declassify` operator. No other operator or function can take a value typed

`private` as an input and output the same value to a value with a `public` type. Also, no function of the standard library uses the declassification operator internally. This restriction makes all share reconstructions explicit and simplifies the security analysis.

If the SHAREMIND assembly program in its execution reaches a call to the `declassify` operator, the computing nodes will send their shares of the declassified value to all other computing nodes and receive shares sent by others. This way, each computing node can reconstruct the value in a public register, replicated over the computing nodes. No computing node can send others a declassification call, each node must engage in the declassification synchronously for it to succeed.

When the execution of a secure algorithm is completed, the result can be published to the result party who requested the computation. This can be done using the `publish` function. The `main` function in SECREC acts as an entry point that processes the input parameters from the result party and sends the output. For example, consider the code in Figure 5.5.

```
// Count occurrences of a value in a database column
public void main (public string db, public string table,
            public string column, private uint32 value) {
  // Load the database by name.
  dbLoad (db);
  // Get the data from the database column.
  private uint32[0] data = dbGetColumn(column, table);
  // Call the more efficient counting function.
  private uint32 result = countFast (data, value);
  // Declassify and publish the result.
  public uint32 publicresult = declassify (result);
  publish ("countresult", publicresult);
}
```

Figure 5.5: Declassification and result publishing in a SECREC `main` function.

This code also demonstrates the use of database functions in the SECREC standard library. The result party publicly provides the database, table and column names and also gives a private query parameter. Data from the specified column is loaded and passed to a counting function together with the private parameter. The private result is returned and then published to the result party. The `main` function does not have a return type, as the `publish` function provides greater flexibility and makes returning several values easier.

The compiler for the SECREC language was designed and implemented as

joint work with Roman Jagomägis [71]. The compiler processes source code files with the `.sc` extension and outputs SHAREMIND assembly files with the `.sa` extension. The resulting assembly can be executed on the SHAREMIND machine.

5.4 Developing secure SECREC programs

SECREC algorithms must be constructed in such a way that the amount of declassified data is kept to a minimum. In the example discussed in Section 5.3.2, the following information is public:

- the location of the data in the SHAREMIND database,
- the amount of values loaded for processing, and
- the number of times the private parameter occurred in the private database column.

The following information remains private:

- the values in the database,
- the inputs and results of comparisons and
- information about which vector elements are added to the total sum of occurrences.

SHAREMIND and SECREC provide cryptographic privacy, but special care has to be taken to achieve other types of privacy. Secure multiparty computation guarantees that nothing except for the intended outputs of the algorithm is leaked. However, if the outputs are computed from the inputs, they always leak something about them. For example, if the number of occurrences is equal to the number of elements, then we know the contents of the whole database column. This is an example of an attack against output-level privacy (see Section 3.2 for details).

SHAREMIND achieves record-level privacy and source-level privacy in the storage phase, thanks to the use of secret sharing. Both kinds of privacy are harder to maintain during computations, as it will require that we do not change the program flow according to declassified variables among other things. However, declassifying intermediate values can significantly improve our performance and, therefore, a balance is needed. The issues affect source-level privacy.

The only known provable method for defeating attacks against output-level privacy is to design algorithms so they satisfy the property of differential privacy. However, this comes with the risk of lowered accuracy. Therefore, for practical applications, we use use clever algorithmic techniques for reducing the amount of

information that leaks from the output. This means accepting certain risks, but as long as these are acknowledged and quantified, the privacy leaks are controlled.

In practice, the algorithm developer is responsible for deciding when to declassify values. Until we build tools that can automatically point out privacy leaks in SECREC programs, the developers themselves have to consider the privacy implications of declassification. The privacy proof of an algorithm can be trivial, if only the final results are declassified. However, if intermediate results are used, the developer must consider the implications of such disclosures to computing parties.

We can analyze and quantify the privacy leaks of SECREC algorithm through an analysis of declassifications. Figure 5.6 illustrates the flow of private information in a SECREC program. A SECREC program can have two kinds of secure inputs in addition to the public parameters and hardcoded constants—the private parameters to the `main` function and data in the private database. These inputs are processed by the computing parties using secure multiparty computation. The program may declassify values that become visible to the computing parties. Some public values can be published to the result parties.

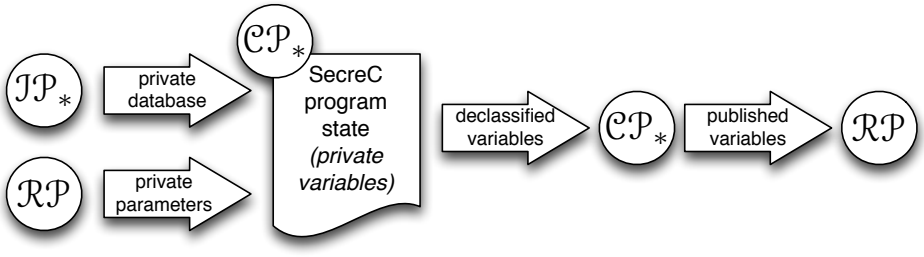


Figure 5.6: The flow of private data in SECREC programs.

Ideally, only the final results of the computation are published and the shares are sent straight to the result party so that the other computing parties do not learn them. However, as discussed earlier, we want to allow a program to declassify intermediate values that might affect the flow of the program. Therefore, we need to consider the information that these values disclose.

One way of proving that declassifications of intermediate data are not a privacy leak is to use a technique similar to the simulatability proofs that we use to prove the security of SHAREMIND protocols. To prove that declassifying the intermediate values in a SECREC program does not leak additional data, we show that the results of all these declassifications can be directly inferred from the final output.

If the final output is published and the miners can learn it, it is not a risk to publish some pieces leading to it earlier in the program to improve efficiency. This

technique has been used in proving the security of privacy-preserving frequent itemset mining algorithms in [22]. The model is applicable if the computing parties learn the output of the algorithm through declassification or collusion with the result party. However, if the final output is not available to the computing parties, such intermediate declassifications may leak too much information.

The other, more mechanical method is to consider the flow of individual private values through the execution of the program. Whenever a value is declassified, we look at how it has been processed up to that point. We prohibit the declassification of values that come directly from either of the two private input channels (parameters or the database) without any processing. We consider a declassification "safe", if the variable being declassified contains an aggregation of several private or public values. We stress that this kind of analysis provides a heuristic for detecting leaks, but does not give absolute guarantees.

The first software implementation of such an analysis was developed as joint work with Jaak Ristioja. The resulting static analysis framework for the SECREC programming language is described in [107]. We formalized the semantics of the SECREC language and created a prototype analysis tool that can detect several kinds of privacy leaks in SECREC programs. However, there are also several cases, where leaks are not detected and improving the analysis methods is a goal for future work.

5.5 Additional developer tools

5.5.1 The developer version of the SHAREMIND server

In production deployments, SHAREMIND is deployed on three separate servers connected over the network to satisfy the independence requirement of the security model. However, during the development of the system, the setup of three servers can be an unnecessary burden. We resolved this issue by creating a developer version of the SHAREMIND server, called the DEVMINER. Figure 5.7 shows the DEVMINER application immediately after starting up.

The DEVMINER is an application that runs three SHAREMIND computing parties on a single machine. The communication between these nodes is not performed over the network, but through in-memory channels. The DEVMINER still uses the network to service the requests of input parties and result parties. Figure 5.8 illustrates the difference between the production and developer deployments of SHAREMIND.

The DEVMINER application is built from the same components that are used in the standard SHAREMIND server application. The main differences are in some hardcoded configuration values, and a different communication model. Additionally, DEVMINER accepts developer commands such as instructions to debug a

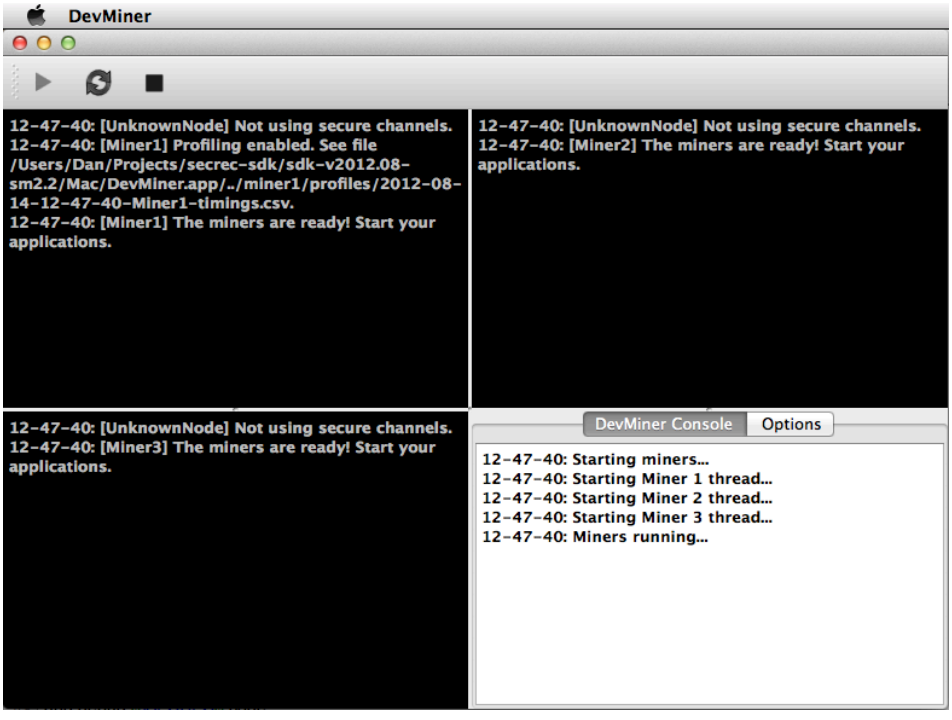


Figure 5.7: The DEVMINER application after startup.

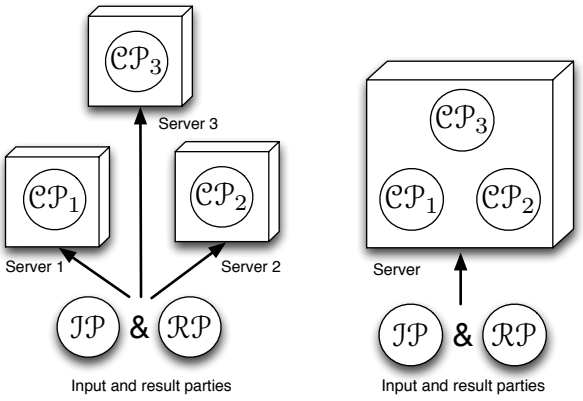
SHAREMIND assembly program by single-stepping or retrieving the contents of private values. Production deployments of SHAREMIND do not service such requests. DEVMINER is available as a part of the SHAREMIND SDK [115].

We stress that the DEVMINER system does not give the same security guarantees as a normal SHAREMIND installation, because all three computing party processes in DEVMINER are under the control of a single entity. However, a solution such as DEVMINER could be made secure if all three computing party processes are run in virtual machines that are separated from each other by a secure hypervisor process. Furthermore, SHAREMIND could be deployed in three separate servers in the same physical location, provided that physical access to the servers is restricted to three different owners.

5.5.2 The SECRECIDE integrated development environment

Modern software is often developed using integrated development environments (IDEs). These environments assist the developer by providing tools and documentation for project management, compilation and debugging.

SHAREMIND provides a fundamentally different programming paradigm when



(a) SHAREMIND in a production environment.

(b) SHAREMIND in a developer environment.

Figure 5.8: Different deployment options of SHAREMIND.

compared to systems running on standard computing hardware. The main difference is in the use of secret sharing for storing the data. This makes the development and debugging of SHAREMIND applications more complex, as inspecting the data at a single party yields only a single share and gives no useful information to the developer. These inconveniences inspired the creation of a tool that helps develop and debug SECREC code. The first version of the SECRECIDE tool was developed as joint work with Reimo Rebane and the general design is documented in [105].

SECRECIDE stands for the SECREC Integrated Development Environment. It supports the developer in several ways.

- SECRECIDE assists the developer in editing SECREC and SHAREMIND assembly source code by providing syntax highlighting and indenting.
- SECRECIDE simplifies compiling SECREC code into SHAREMIND assembly.
- SECRECIDE can connect to SHAREMIND miner servers to upload compiled code and execute it.
- SECRECIDE can be used for debugging SHAREMIND assembly on a running DEVMINER. This includes setting breakpoints, single-stepping and continuing code execution, inspecting public and private values in the private stack and both public and private registers.
- SECRECIDE contains reference documentation for both the SHAREMIND assembly and SECREC languages.

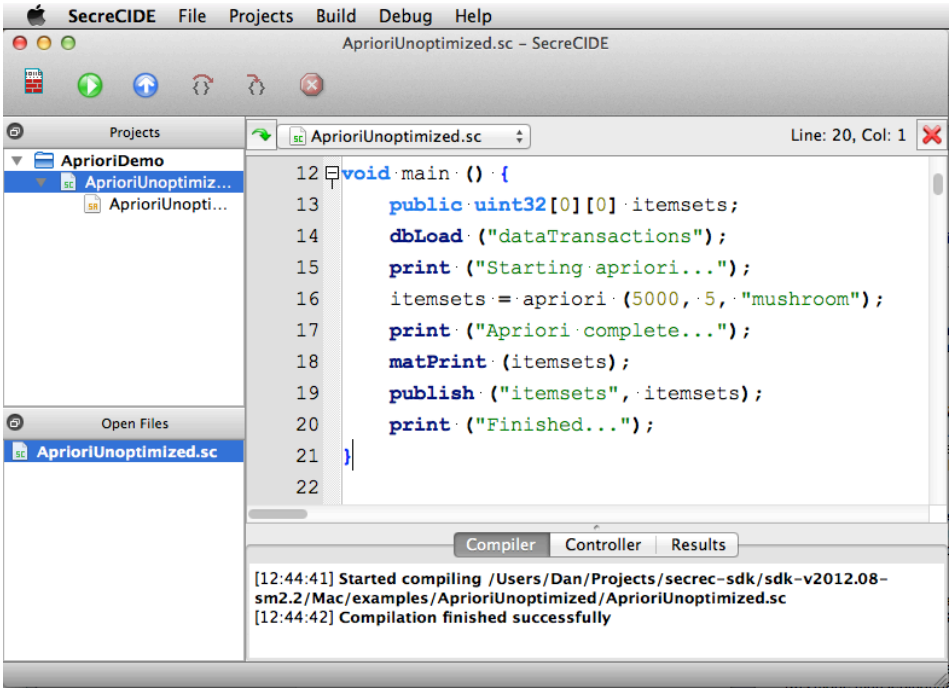


Figure 5.9: The SECRECIDE application with an open code file.

Figure 5.9 shows the SECRECIDE tool with an opened code file that has just been compiled.

5.6 A comparison of SECREC to other secure computation programming languages

We conclude the description of SECREC with a quick comparison to other languages designed for implementing secure multiparty computation protocols or applications. Table 5.2 gives a comparison of secure programming languages with a secure multiparty computation backend.

The table contains several aspects of the languages. First, we list the intermediate representation and runtime of each language. Then we describe the secure computation model represented by the language by giving the number of supported parties and describing, if the language requires the programmer to write code that is specific to a certain party. We show, which languages contain explicit syntax for exchanging messages between parties.

The overview continues with the secure and public computation features of the language and the runtime. We list the data types and operations and pay special

attention to how private values are made public. The table then describes the data structures and flow control mechanisms that the language provides. Finally, we list the papers from which the information in the table was gathered.

Based on the comparison, we can evaluate the suitability of each language for a particular task. We consider two main use cases—the development of new secure computation protocols and the development of secure data analysis applications. For protocol development, we require a language that can specify the functionality for each party and provides message exchange facilities. The best candidates for protocol development are L1 and TASTYL, as they have the listed capabilities and they also support several secure data representations such as garbled circuits and homomorphic encryption. While SMCL also has party-specific code, it does not support different secure data representations.

The best languages for data processing applications are SMCL and SECREC. The main difference between the two languages is the application development paradigm. Whereas SMCL has special objects representing different clients and their inputs and outputs, SECREC code is independent of the deployment of the underlying cryptographic protocols. Furthermore, the SECREC standard library contains functions for vector and matrix manipulation and secure database access that simplify the development of applications. Both languages have been used for developing real-world applications [100, 30].

The SFDL language is also suitable for applications and it is excellent for specifying purely secure functionalities. This is well illustrated by the fact that the SFDL language is used as an input language by several secure computation implementations, e.g., Fairplay, FairplayMP and TASTY. Its main downside is the inconvenience of working with public data. SFDL programs do not contain public variables and this makes the development of real-life applications significantly more complex.

We have discussed the strengths of each language and conclude that they share many similarities, but their intended purpose makes them different. For example, SECREC is probably the easiest language to develop algorithms in, as it does not require the developer to think about parties or networking. However, this makes SECREC less optimal for applications where the roles of parties are strictly fixed.

Language	**SFDL** (1 and 2)	**SMCL**	**TASTYL**	**L1**	**SecreC**
Compiles to	SHDL	Java DSL	—	Java DSL	assembly
Runtime	Fairplay, FairplayMP, TASTY	SMCR	TASTY	Java VM	Sharemind 2
Number of parties	2 or more	3 or more	2	2 or more	Not fixed
Code style	Party-specific	Party-specific	Party-specific	Party-specific	Universal
Networking	Implicit	Explicit	Explicit	Explicit	Implicit
Secure integers	Yes	Yes	Yes	Yes	Yes
Secure $+, \times$	Yes	Yes	Yes	Yes	Yes
Secure $==, <, >$	Yes	Yes	Yes	Yes	Yes
Secure $/, \mathrm{mod}$	No	No	No	Yes	Yes
Secure booleans	Yes	Yes	Yes	Yes	Yes
Secure logic	Yes	Yes	Yes	Yes	Yes
Declassification	Implicit	Explicit	Implicit	Implicit	Explicit
Public types and operations	None	Boolean, integer	Boolean, integer	Boolean, integer, string	Boolean, integer, string
Arrays, matrices	Only arrays	Only arrays	Both	Only arrays	Both
SIMD operations	No	No	Yes	Yes	Yes
Oblivious arrays	Yes	No	No	No	Special function
Private branching	Limited	Limited	None	None	None
Loops	Constant size	Public size	Constant size	Public size	Public size
References	[95, 15]	[101, 100]	[66]	[78, 111]	[71, 107], this thesis

Table 5.2: A selection of secure computation programming languages with secure multiparty computation runtimes.

CHAPTER 6

SHAREMIND IN PRACTICE

6.1 The process of developing a SHAREMIND application

6.1.1 Designing the application

We will now propose a process for developing secure applications based on the SHAREMIND system. This guidance has been developed through the study of several implemented and deployed prototypes. The process will be detailed further when SHAREMIND technology is developed into an industry-accepted toolkit.

The development of a secure computation system with SHAREMIND starts with determining the problem and its stakeholders. The important questions are:

1. Who has the data?
2. Who wants to process the data?
3. Are there any other organizations who benefit from processing the data?
4. Are there any other organizations whose goal it is to protect the data?

We must consider these answers as we determine the input parties, computing parties and result parties. The people that can provide the necessary data will act as input parties. The organizations who want to process data become the result parties. The three computing parties are chosen from among all the identified stakeholders in accordance to the following requirements.

First, the organizations must have an interest in preserving the privacy of the data providers. Second, these organizations must be as independent from each other as possible. Finally, each organization must be capable of hosting a SHAREMIND server or, alternatively, must arrange the hosting, e.g., on a cloud.

These requirements are in place to prevent the unauthorized exchange of secret shared data as that would compromise the privacy guarantees. For details, refer to the the passive adversary assumptions as discussed in Section 3.3.3.

The next step is to design the secure application as a collection of data models and algorithms for processing the data. As the result party knows what kind of data are needed to perform the analyses, it proposes the initial data model. The designed data model must include information on which attributes are stored as public values and what will be stored in secret shared form. The resulting data model will later be used to set up the SHAREMIND database.

While the result party is the initiator behind application design, the process should be witnessed by both the computing parties and the result parties. This way, the computing parties will get an understanding of the kind of data they will jointly be processing. Furthermore, it may be beneficial to involve a representative of the input parties as one of the computing parties. This way, the input parties have a stronger involvement in the whole process.

After agreeing on the data model, the result party presents the list of analyses to be performed on the data. Each analysis in the application should be described so that it is clear, what data are used and what will be declassified as a result of the computation. This transparent communication allows all involved parties to understand the extent of information use.

6.1.2 Implementing the application

Once the application description has been agreed on, the three main components of a SHAREMIND application are implemented.

First, the data entry application implements the behavior of the input parties. The data entry application is implemented using the controller library specific to the platform that will be used for data entry. For example, if the data are entered using a desktop application or imported from an existing database or file, a desktop version of the controller library should be used. A web-based data application can be implemented with a specific library that allows secret sharing to be performed in the web browser.

The developer will design a suitable user interface for collecting the secret data and implement it using the SHAREMIND controller library. The controller library will take care of secret sharing and uploading the collected data to the SHAREMIND database.

The data analysis application represents the behavior of the computing parties. It consists of secure data processing algorithms implemented in the SECREC language. The developer can use the SECRECIDE environment and the DEVMINER during development to simplify debugging and testing. All the analysis algorithms must be implemented so that the declassifications are in accordance with the ap-

plication description. Should it happen that an algorithm cannot be implemented without additional declassifications (or the implementation would be inefficient), the application description must be updated. Moreover, all parties must be informed of the new declassifications.

The data analysis application implements the behavior of the result parties. This application will also be implemented using the controller library. To send queries to SHAREMIND, the application will give the controller library the name of the algorithm to run, together with all the required parameters. The controller library will relay the information to SHAREMIND servers that will, in turn, start the algorithm with the given parameters.

Once the computation is complete, all the published values will be sent to the controller library application from where the query originated. The library will return the results to the analysis application. The application may then use any suitable presentation method to show the data. Alternatively, the analysis application can also store the results locally for later lookup.

6.1.3 Deploying the application

When the necessary components of the SHAREMIND application have been created, they can be deployed for use. The data entry applications must be delivered to the data provider in a trustworthy manner.

For desktop applications, this means ensuring the correctness and authenticity of the code by techniques such as code signing. For web applications it is also important to inform the users of the correct URL for the data entry application. Note that these precautions are not unique to secret sharing technology, but they provide additional guarantees. There is always the alternative of setting up trusted data entry terminals, but then we would have to convince the users that there are no additional tracking systems such as keyboard loggers on

The SHAREMIND server software must be deployed by each host individually to ensure that no party has access to more than one SHAREMIND server. The servers will then be configured with each others addresses and encryption keys. This information must be exchanged in the most direct manner possible to ensure that the configuration is correct and the keys for setting up secure channels between the servers are not compromised.

When SHAREMIND has been successfully set up, it is time to deploy the SECREC code. The code must be delivered to each SHAREMIND server host who can also exchange hashes of the code to make sure that they have the same version of it. For better security, the code could be digitally signed and each SHAREMIND server host could validate the signature.

It makes sense to deliver SECREC code instead of compiled code as it is easier to read and validate. As an optional step, the host can perform a final verification

to check that the code conforms to the agreed application description. The code is then compiled to SHAREMIND assembly and deployed at the server. This step effectively confirms the server's readiness to execute the given analysis algorithms.

We note here a possible research direction for improving the security of deploying SECREC code for SHAREMIND. It is possible that the compiler makes mistakes in the translation and generates assembly code that does not preserve the secure data flow of the SECREC program. We could enhance the SECREC language and compiler to employ *proof-carrying code* techniques so that the security properties of a SECREC program could easily be validated on the lower-level assembly code.

Finally, the setup of the data analysis application is similar to the setup of the data entry application. Whereas the data entry application was delivered to the input parties, the data analysis application is set up for the result parties who can use it for sending queries to the running SHAREMIND application.

6.2 Privacy-preserving application prototypes

6.2.1 Online surveys

We demonstrate a prototype application for SHAREMIND that helps a user perform surveys that ask for numeric values or selections from pre-determined options. These multiple choice questions are usually connected to a numeric scale or an encoded classifier. For now, we do not consider questionnaires with text fields.

It is possible to conduct the survey in the form of an interview, where a specialist asks the question from an individual and enters the results into a special data collection application. However, it is often cheaper for the survey organizer to make the form available on the internet and request that members of the target group fill it out themselves.

Deploying SHAREMIND data entry applications as web pages raises several technical challenges. For best privacy, we must ensure that the data leaves the input party in secret shared form and reaches the computing parties without intermediaries, secret sharing must be performed in the web browser. Hence, the web application forms secure connections to several servers and has access to a good source of randomness.

These issues were investigated as joint work with Riivo Talviste [122]. We implemented a web-based secret shared data collection mechanism that performs secret sharing in the web browser using a simple SHAREMIND controller library written in JavaScript. As most standard client-side web technologies do not provide developer access to a secure randomness source, we also described a general solution for obtaining secure randomness in a web browser that has no built-in access to good entropy.

We also describe graphical user interface elements that help the user distinguish secret-shared data collection from standard web forms. The proposed solution displays visual identifiers of the computing parties together with the web form to show that these are the parties that are responsible for preserving privacy.

A demonstration of the secure survey technology was developed by Estonian companies Cybernetica, Quretec and Software Technology and Applications Competence Centre (STACC). This application consisted of a data entry form that sent data to SHAREMIND servers and a report generator that regularly compiled a web page that represented the collected data as histograms.

Figure 6.1 illustrates this design. The figure shows an example deployment where all three computing nodes are deployed by the same host. As this is not the case in practical applications, an example of a similar application with different hosts will be presented in Section 6.3.

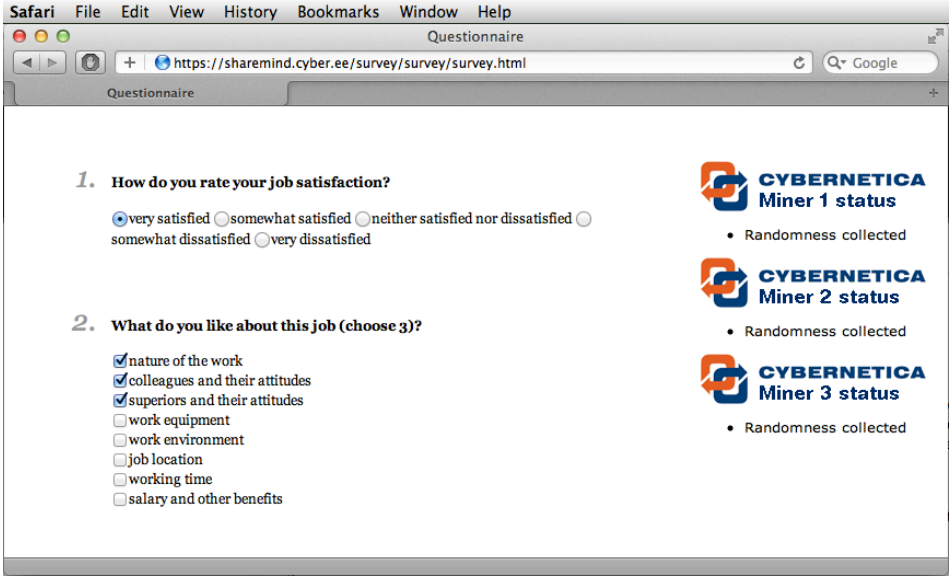


Figure 6.1: An example of a secure survey questionnaire using SHAREMIND.

To generate the report, a result party application requests the computing parties to execute a SHAREMIND assembly program that computes several histograms. The published data from these histograms is passed to a report generation tool that presents them in the form of bar charts, as shown in Figure 6.2. The demonstration has been published on the web [116].

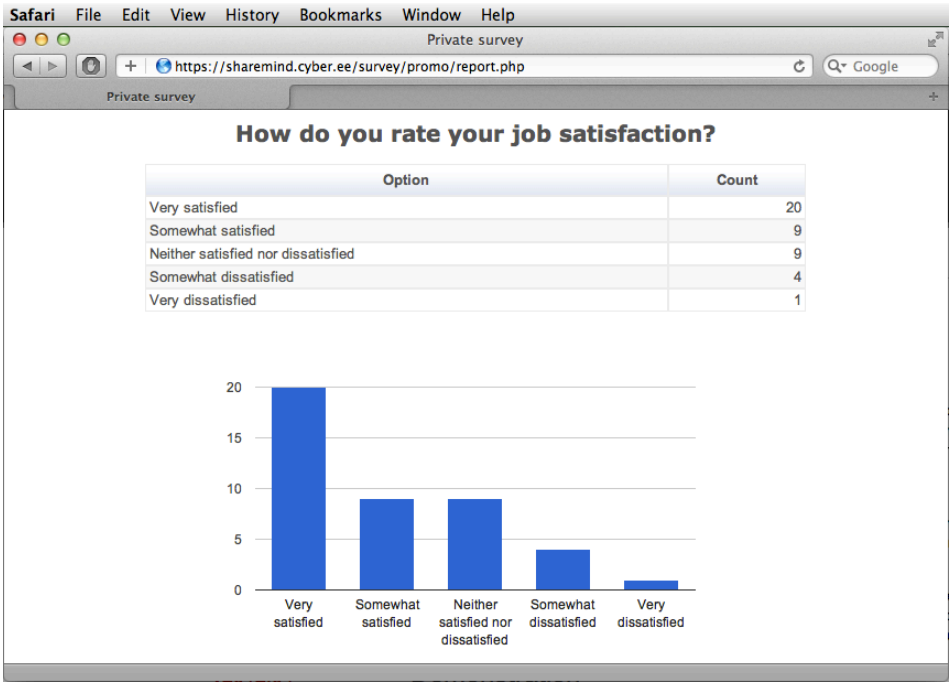


Figure 6.2: An example of a secure survey report.

6.2.2 Frequent itemset mining

Data mining is a technique for analyzing large databases in order to extract new information. We consider it to be a primary application of the SHAREMIND system and, therefore, we have also adapted data mining algorithms for SHAREMIND. Even though SHAREMIND and SECREC allow standard data mining algorithms to be naïvely converted into a secure form, we can often get better performance and security by redesigning algorithms specifically for SHAREMIND.

There are two main reasons for why optimizing algorithms for SHAREMIND brings significant benefits. First, the hybrid programming model of SECREC requires that private data are identified and separated. Furthermore, only the minimal amount of information should be declassified. For some algorithms, no intermediate declassifications are required, whereas others may need them to be more efficient.

Second, the properties of SHAREMIND motivate us to use parallelization as much as possible. While some data mining algorithms (e.g., ones based on breadth-first search) are easy to parallelize, others require more work. Our research group has found, that parallelization and minimal declassification are easy to achieve through the use of oblivious choice primitives [87].

The first non-trivial data mining problem we solved was that of frequent itemset mining [22, 71]. Frequent itemset mining is an underlying technique for collaborative filtering and market basket analysis. Privacy-preserving collaborative filtering can be used for a variety of tasks, including product suggestions in e-commerce and item suggestions in museums [28].

We chose to adapt and implement two well-known frequent itemset mining algorithms—Apriori [4, 96] and Eclat [130]. Apriori uses a breadth-first searching approach and Eclat uses depth-first search. The two algorithms were implemented in SECREC and also in the form of SHAREMIND protocols. The latter was done to measure the overhead of SHAREMIND assembly interpretation. The details of the implementation and the security analysis is given in [22].

Preliminary results showed that the highly parallelized secure Apriori implementation is significantly faster than the secure Eclat implementation. However, Apriori also used significantly more memory. Based on these results, we implemented a version of Apriori with controlled levels of parallelization and a version of Eclat with some levels of parallelization.

As expected, the *hybrid* Apriori used less memory and it also performed only slightly slower than the fully parallel version. Similarly, we observed that the *hybrid* Eclat algorithm performed better without requiring significantly more memory. A comparison of the performance of the algorithms is given in Figure 6.3. The experiments were run on the mushroom database from UC Irvine Machine Learning repository [59]. Note that in the experiments we only set limits on the threshold and not the contents of the itemset so that all the itemsets meeting the threshold criteria were found.

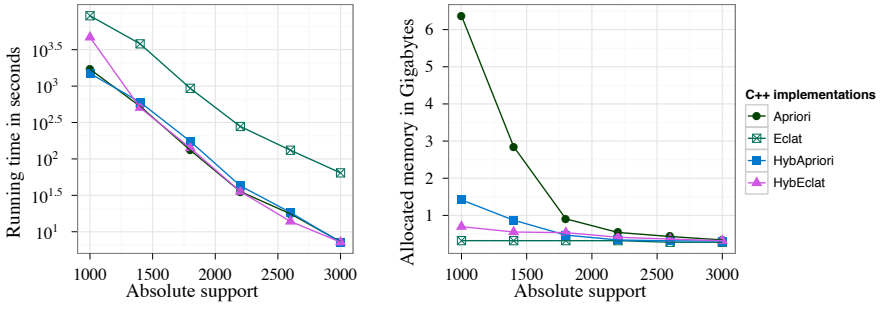


Figure 6.3: Comparison of secure frequent itemset mining performance [22].

To measure the overhead of SHAREMIND assembly execution we compared the running time of the SECREC implementation to the C++ protocol implementation. We also decided to compare SHAREMIND to other secure computing systems. Based on published benchmarks, we found that SEPIA [112] is the

best match for SHAREMIND in terms of secure computation performance. We implemented the fastest Apriori algorithm also on SEPIA and benchmarked it on the same hardware and network configuration that was used for measuring SHAREMIND. The results are given in Figure 6.4.

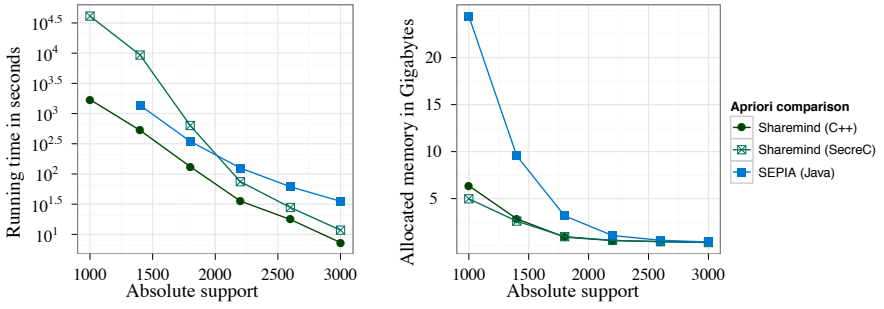


Figure 6.4: Comparison of the performance of different Apriori implementations [22].

According to the results the C++ protocol implementation is the fastest among the compared versions. The SECREC version is faster than the SEPIA implementation for some vector sizes, but becomes slower for larger vectors. The memory usage of both SHAREMIND implementations were similar, with SEPIA using significantly larger amounts of memory.

Profiling suggests that the slowdown of the SECREC implementation of the Apriori algorithm is caused by the inefficient handling of large vectors in the SHAREMIND assembly interpreter. The stack-based design of SHAREMIND operations is very easy to model and generate code for. However, copying large data vectors to and from the stack has a significant performance cost. Also, we found the virtual machine of SHAREMIND is not optimized for public operations and an algorithm with many public operations has a significant interpretation overhead.

Based on these result, we plan to improve the interpreter to reduce the overhead. We intend to resolve the data copying issue by moving towards a design where data vectors are passed around using handles in order to reduce copying them. Instead of pushing input data on the stack for secure processing, we will push handles to these vectors so that the protocol can work on the memory areas directly. We expect that this design change will further reduce the runtime memory requirements of the SHAREMIND system.

Furthermore, we will optimize the virtual machine that interprets SHAREMIND assembly to reduce the overhead of executing a single public operation. Both of these changes will be significant in ensuring that secure algorithms implemented in the SECREC programming language will not be significantly slower than their C++ counterparts.

6.2.3 Privacy-preserving k-means clustering

Our secure frequent itemset mining algorithm used secure operations for addition, multiplication and greater-than comparison. However, several algorithms require the division operation for normalization and similar tasks. SHAREMIND is currently one of the few systems with an implemented secure division operation. We decided to test this operation in practice and implemented the k-means clustering algorithm [91].

Clustering analysis is used to group similar objects or observations according to an attribute space. The k-means algorithm is considered geometrical or centroid-based clustering, as it determines groups by defining a representative objects for each group and then finding the closest items around that object.

We implemented k-means clustering on the SHAREMIND system as an example of applying the new protocols proposed in [27]. Our implementation of secure k-means clustering hides the coordinates of each data point, but does not hide the size of the cluster or the set of points in a cluster. In each iteration, we declassify the index of the closest centroid for each data point and if this index has changed, we move the data point to the new cluster. In cases where we also need to hide, which shares belong to which clusters, we can add random shuffling and oblivious lookups, but that may slow down the algorithm several times.

We tested the implementation on the example databases from the UC Irvine Machine Learning repository [59]. The experiments were conducted on the same experimental setting that was used for performance benchmarks and frequent itemset mining measurements. Table 6.1 gives an overview of the running times of the algorithms and the number of secure computation operations that were needed to complete the analysis. In each case, execution continued until the clusters converged.

During development, we found that for some databases, the accuracy of clusters was not good enough because integer division was inaccurate in computing the distances. We resolved the issue by emulating fixed point number using integer arithmetics. We multiplying the coordinates of the data points by 10000 to make them large enough so that the integer division results become accurate enough. This allowed us to get the same degree of accuracy that was achieved with a non-private implementation of the k-means algorithm.

We can conclude from these results that even though the secure clustering operations require hundreds of millions of secure operations, the running times remain in the practical range and take under an hour.

Database	k	Time	Iter.	Multiply	Compare	Divide
iris 150×4	3	1 s	4	9600	5400	44
synthetic 600×60	3	3 s	5	$7.2 \cdot 10^5$	$2.7 \cdot 10^4$	900
	5	6 s	8	$1.7 \cdot 10^6$	$1.2 \cdot 10^5$	2400
	8	8 s	7	$2.3 \cdot 10^6$	$2.7 \cdot 10^5$	3360
plants 34781×70	3	4 min 58 s	12	$1.2 \cdot 10^8$	$3.8 \cdot 10^6$	2520
	5	22 min 42 s	28	$4.1 \cdot 10^8$	$2.4 \cdot 10^7$	9800
	10	36 min 35 s	17	$4.6 \cdot 10^8$	$5.9 \cdot 10^7$	11900

Table 6.1: k-means clustering performance and operation counts on SHAREMIND.

6.3 The ITL financial benchmarking application

SHAREMIND has been used in a real-world setting. The Estonian Association of Information Technology and Telecommunications (ITL) is a trade organization of Estonian companies who are active in the field of information and communication technology. In 2010, we developed and deployed a secure economic benchmarking application that ITL uses to measure the health of the information and communication technology sector in the country [30, 123].

According to the problem statement, members of the ITL enter a selection of economic metrics twice a year. This information is analyzed to produce trends and aggregated metrics that can be visualized and presented to the member companies.

We learned of the problem statement of ITL in the summer of 2010. We drafted a solution proposal that followed the three-part application blueprint presented in Section 6.1.2. The proposal was accepted and we started to construct the application. According to the design, data would be entered using a collection form in the members area of the ITL web page. The same web page would also host the analytics and reporting application. A SHAREMIND installation was planned to host the collected data.

The data collection form was implemented using an improved version of the secure survey technology described in Section 6.2.1. The details of the construction is described in [123]. The role of the input parties is fulfilled by all eligible members of ITL.

Three organizations were chosen among the ITL members to hold the roles of computing parties and host the SHAREMIND nodes. In the context of SHAREMIND these nodes act as computing parties. Since all three companies were capable of hosting a SHAREMIND server, no outside hosting was needed.

In the solution, the result party is the ITL board who sends queries to the SHAREMIND installation and then presents the results to the member companies.

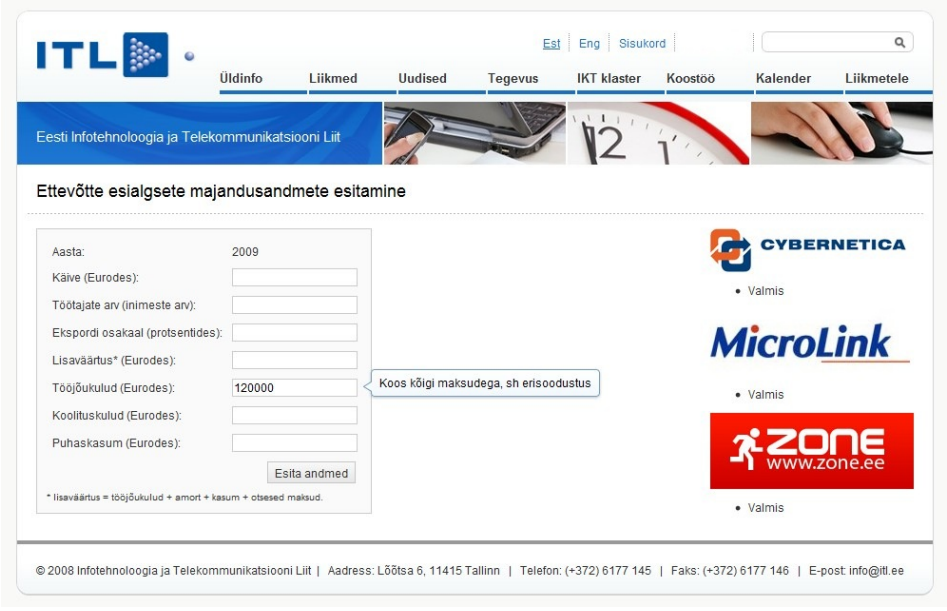


Figure 6.5: A screenshot of the ITL data entry form.

The application was deployed in late 2010 and the first data collection and analysis took place in early 2011. For the second data collection period in the third quarter of 2011, we added a small survey to the system to ask for user feedback and attitude towards the system. The results and analysis are presented in the paper [30]. Figure 6.5 shows a screenshot of the deployment.

The ITL application is unique for several reasons. First, it is the first real-world application where secure multiparty computations are performed over the public internet. Second, it is the most complex reported application in practice, as it uses a large set of different primitives and algorithms (see Table 6.2 for details). Third, it is built using a general purpose secure multiparty computation system, instead of a specific protocol. This shows that general purpose secure computation systems can be usable in practice.

Analysis operation	**Required secure computation capabilities**
Sorting financial indicator columns	The oblivious vector sorting network uses • secure multiplication, • secure addition and • secure comparison.
Privacy-preserving database filtering	Oblivious choice requires • casting secure booleans to secure integers and • secure multiplication.
Calculating the added value per employee	Oblivious ratio calculation requires a secure division operation.
Computing time series of financial indicators	The oblivious matrix sorting network is an extension of the oblivious vector sorting network and requires the same operations.

Table 6.2: The privacy-preserving algorithms used in the ITL application [30].

CONCLUSION

Secure processing of confidential data is a problem with many solutions. Some companies prefer organizational measures such as non-disclosure agreements and penalties, others apply standard data protection mechanisms. Unfortunately, ignorance is still a common approach to confidentiality, as some organizations disregard the risks associated with inadequate data protection.

The goal of this thesis is to improve the state of the art of practical secure computation by introducing a practical secure multiparty computation framework called SHAREMIND. SHAREMIND uses cryptographic techniques such as secret sharing and secure multiparty computation to ensure that confidential data is processed as securely as possible. In this thesis, we present the necessary tools for building secure computation protocols, using them in applications and deploying such applications in a real-world setting.

The main result of the thesis is a highly efficient and universally composable protocol suite for secure computation on integers. We show how SHAREMIND can securely perform basic arithmetic operations such as addition, multiplication, comparison and division. The basic operations can be composed sequentially to form programs and in parallel to achieve operations on vectors. Vector operations are important in SHAREMIND because they significantly reduce the amortized cost of secure computation. SHAREMIND encourages the developer to make use of parallelization by simplifying operations on vectors and matrices.

The efficient processing of vector data makes SHAREMIND a good platform for privacy-preserving data mining. We have implemented several data analysis application prototypes on SHAREMIND that can compute simple aggregations like sums and histograms. We have also developed secure algorithms for more complex tasks like frequent itemset mining and clustering.

All the secure computation capabilities of SHAREMIND are usable by non-cryptographers thanks to the specially designed programming language SECREC. SECREC allows data analysis algorithms to be implemented with a clear separation of public and private data. All operations on data that have been marked private are performed using secure computation. SECREC has been designed to be familiar to software developers and works well with the specially tailored in-

tegrated development environment called SECRECIDE. SHAREMIND also provides a special controller library for creating user interfaces for its applications. The controller library provides a simple interface to the secure computation capabilities of SHAREMIND, further reducing the complexity of applying the new technology in practice.

We believe that our work has brought secure computation technology closer to real-world applicability. The performance of SHAREMIND is good enough for secure processing of databases with millions of rows. Furthermore, our experiments show that SHAREMIND is robust enough to be deployed in real-world data centers internationally. Even more importantly, SHAREMIND has been successfully used in a real application for privacy-preserving financial benchmarking. This application is built on a SHAREMIND installation shared by three separate companies which makes it the first ever secure multiparty computation application to run on the public internet.

Data processing applications that use SHAREMIND are significantly more secure than ones that do not use secure multiparty computation. The risk of insider attacks is greatly reduced as individual computing parties cannot deduce information from the shares of the data available to them. The risk of data leaks through negligence or malicious behavior is reduced for the same reason. We believe that policy enforcement technologies like SHAREMIND will play a significant role in securing information systems and protecting confidential data.

Bibliography

[1] Adam, N.R., Worthmann, J.C.: Security-control methods for statistical databases: a comparative study. ACM Computing Surveys 21(4), 515–556 (1989)

[2] Afyouni, S.: Database Security and Auditing: Protecting Data Integrity and Accessibility. Course Technology Press, Boston, MA, United States (2005)

[3] Agrawal, D., Aggarwal, C.C.: On the design and quantification of privacy preserving data mining algorithms. In: Buneman, P. (ed.) Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. PODS'01. ACM (2001)

[4] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) Proceedings of 20th International Conference on Very Large Data Bases. VLDB'01. pp. 487–499. Morgan Kaufmann (1994)

[5] Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: Chen, W., Naughton, J.F., Bernstein, P.A. (eds.) Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. pp. 439–450. ACM (2000)

[6] Ahmed, A.S., Bogdanov, D.: A Model for Automatically Evaluating Trust in X.509 Certificates. Tech. Rep. T-4-11, Cybernetica AS, Tartu, `http://research.cyber.ee/`. (2010)

[7] Aiello, W., Ishai, Y., Reingold, O.: Priced oblivious transfer: How to sell digital goods. In: Pfitzmann, B. (ed.) Proceedings of the 20^th International Conference on the Theory and Application of Cryptographic Techniques, EUROCRYPT '01. Lecture Notes in Computer Science, vol. 2045, pp. 119–135. Springer (2001)

[8] Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient

protocols for realistic adversaries. Journal of Cryptology 23(2), 281–343 (2010)

[9] Aumasson, J.P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. In: Nyberg, K. (ed.) 15th International Workshop on Fast Software Encryption. FSE'08. Lecture Notes in Computer Science, vol. 5086, pp. 470–488. Springer (2008)

[10] Barbaro, M., Jr., T.Z.: A face is exposed for AOL searcher no. 4417749. The New York Times (August 9th, 2006)

[11] Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) Proceedings of the 11th Annual International Cryptology Conference. CRYPTO '91. Lecture Notes in Computer Science, vol. 576, pp. 420–432. Springer (1991)

[12] Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: Ortiz, H. (ed.) Proceedings of the 22nd Annual ACM Symposium on Theory of Computing. STOC'90. pp. 503–513. ACM (1990)

[13] Bellare, M.: New proofs for NMAC and HMAC: security without collision-resistance. In: Dwork, C. (ed.) Proceedings of the 26th Annual International Cryptology Conference. CRYPTO'06. Lecture Notes in Computer Science, vol. 4117, pp. 602–619. Springer (2006)

[14] Bellare, M., Hoang, V.T., Rogaway, P.: Garbling schemes. Cryptology ePrint Archive, Report 2012/265 (2012), `http://eprint.iacr.org/`

[15] Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM Conference on Computer and Communications Security. CCS'08. pp. 257–266. ACM (2008)

[16] Ben-Or, M., Canetti, R., Goldreich, O.: Asynchronous secure computation. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing. STOC'93. pp. 52–61. ACM (1993)

[17] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In: Simon, J. (ed.) Proceedings of the 20th Annual ACM Symposium on Theory of Computing. STOC'88. pp. 1–10 (1988)

[18] Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) Proceedings of the 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT '11. Lecture Notes in Computer Science, vol. 6632, pp. 169–188. Springer (2011)

[19] Bernstein, D.: ChaCha, a variant of Salsa20. `http://cr.yp.to/chacha.html`. Last accessed August 14^{th}, 2012. (2008)

[20] Bogdanov, D.: How to securely perform computations on secret-shared data. Master's thesis, University of Tartu (2007)

[21] Bogdanov, D., Crispino, M.V., Čyras, V., Lapin, K., Panebarco, M., Zuliani, F.: Virtual World Platform VirtualLife: P2P, Security, Rule of Law and Learning Support. In: Proceedings of 2009 NEM Summit "Towards Future Media Internet". Distributed as an eBook. NEM Initiative (2009)

[22] Bogdanov, D., Jagomägis, R., Laur, S.: A universal toolkit for cryptographically secure privacy-preserving data mining. In: Chau, M., Wang, G.A., Yue, W.T., Chen, H. (eds.) Proceedings of the Pacific Asia Workshop on Intelligence and Security Informatics, PAISI '12. Lecture Notes in Computer Science, vol. 7299, pp. 112–126. Springer (2012)

[23] Bogdanov, D., Kamm, L.: Constructing privacy-preserving information systems using secure multiparty computation. Tech. Rep. T-4-13, Cybernetica AS, Tartu, `http://research.cyber.ee/`. (2011)

[24] Bogdanov, D., Laur, S.: The design of a privacy-preserving distributed virtual machine. In: Kaklamanis, C. (ed.) Collection of AEOLUS theoretical findings. Deliverable 1.0.6, pp. 269–280. Published online at `http://aeolus.ceid.upatras.gr/deliverables` (2010)

[25] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) Proceedings of the 13th European Symposium on Research in Computer Security, ESORICS '08. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008)

[26] Bogdanov, D., Livenson, I.: VirtualLife: Secure Identity Management in Peer-to-Peer Systems. In: Daras, P., Ibarra, O.M., Akan, O., Bellavista, P., Cao, J., Dressler, F., Ferrari, D., Gerla, M., Kobayashi, H., Palazzo, S., Sahni, S., Shen, X.S., Stan, M., Xiaohua, J., Zomaya, A., Coulson, G. (eds.) Proceedings of the 1st International ICST Conference on User

Centric Media, UCM '10. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 40, pp. 181–188. Springer (2010)

[27] Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. International Journal of Information Security 11(6), 403–418 (2012)

[28] Bogdanov, D., Sassoon, R.: Privacy-preserving collaborative filtering with sharemind. Tech. Rep. T-4-2, Cybernetica AS, Tartu, `http://research.cyber.ee/`. (2008)

[29] Bogdanov, D., Talviste, R.: A Comparison of Software Pseudorandom Number Generators. In: Cap, C. (ed.) Proceedings of Third Baltic Conference on Advanced Topics in Telecommunication - BaSoTi 2009. pp. 61–71. Universität Rostock, Wissenschaftsverbund IuK (2009)

[30] Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis - (short paper). In: Keromytis, A.D. (ed.) Proceedings of the 16th International Conference on Financial Cryptography and Data Security, FC '12. Lecture Notes in Computer Science, vol. 7397, pp. 57–64. Springer (2012)

[31] Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure Multiparty Computation Goes Live. In: Dingledine, R., Golle, P. (eds.) 13th International Conference of Financial Cryptography and Data Security. FC'09. Lecture Notes in Computer Science, vol. 5628, pp. 325–343. Springer (2009)

[32] Bogetoft, P., Damgård, I., Jakobsen, T.P., Nielsen, K., Pagter, J., Toft, T.: A practical implementation of secure auctions based on multiparty integer computation. In: Crescenzo, G.D., Rubin, A.D. (eds.) 10th International Conference on Financial Cryptography and Data Security. FC'06. Lecture Notes in Computer Science, vol. 4107, pp. 142–147. Springer (2006)

[33] Boneh, D., Kushilevitz, E., Ostrovsky, R., III, W.E.S.: Public Key Encryption That Allows PIR Queries. In: Menezes, A. (ed.) 27^{th} Annual International Cryptology Conference, CRYPTO'07. Lecture Notes in Computer Science, vol. 4622, pp. 50–67. Springer (2007)

[34] Boost C++ libraries. `http://www.boost.org/`. Last accessed June 1^{st}, 2012.

[35] Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.A.: SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In: 19th USENIX Security Symposium. USENIX'10. pp. 223–240. USENIX Association (2010)

[36] Canetti, R.: Studies in Secure Multiparty Computation and Applications. Ph.D. thesis, Weizmann Institute of Science (1995)

[37] Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067 (2000), http://eprint.iacr.org/. Last revision from 2005.

[38] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings of the 42nd Annual Symposium on Foundations of Computer Science. FOCS'01. pp. 136–145. IEEE Computer Society (2001)

[39] Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: Simon, J. (ed.) Proceedings of the 20th Annual ACM Symposium on Theory of Computing. STOC'88. pp. 11–19 (1988)

[40] Chaum, D., Damgård, I., van de Graaf, J.: Multiparty computations ensuring privacy of each party's input and correctness of the result. In: Pomerance, C. (ed.) Proceedings of the 7th Annual International Cryptology Conference. CRYPTO '87. Lecture Notes in Computer Science, vol. 293, pp. 87–119. Springer (1987)

[41] Chor, B., Goldwasser, S., Micali, S., Awerbuch, B.: Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In: 26th Annual Symposium on Foundations of Computer Science. FOCS'85. pp. 383–395. IEEE Computer Society (1985)

[42] Conway, R., Strip, D.: Selective partial access to a database. In: Proceedings of the 1976 Annual Conference of the ACM, ACM '76. pp. 85–89. ACM (1976)

[43] libcsv—a small, simple and fast CSV library. http://sourceforge.net/projects/libcsv/. Last accessed June 1^{st}, 2012.

[44] Damgård, I., Geisler, M., Krøigaard, M.: Homomorphic encryption and secure comparison. International Journal of Applied Cryptography 1(1), 22–31 (2008)

[45] Damgård, I., Geisler, M., Krøigaard, M.: A correction to "Efficient and secure comparison for on-line auctions". International Journal of Applied Cryptography 1(4), 323–324 (2009)

[46] Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Jarecki, S., Tsudik, G. (eds.) 12th International Conference on Practice and Theory in Public Key Cryptography, PKC'09. Lecture Notes in Computer Science, vol. 5443, pp. 160–179. Springer (2009)

[47] Damgård, I., Ishai, Y.: Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator. In: Shoup, V. (ed.) Proceedings of the 25th Annual International Cryptology Conference. CRYPTO'05. Lecture Notes in Computer Science, vol. 3621, pp. 378–394. Springer (2005)

[48] Damgård, I., Jurik, M.: A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In: Kim, K. (ed.) 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC'01. Lecture Notes in Computer Science, vol. 1992, pp. 119–136. Springer (2001)

[49] Damgård, I., Orlandi, C.: Multiparty Computation for Dishonest Majority: From Passive to Active Security at Low Cost. In: Rabin, T. (ed.) Proceedings of the 30th Annual Cryptology Conference. CRYPTO'10. Lecture Notes in Computer Science, vol. 6223, pp. 558–576. Springer (2010)

[50] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Proceedings of the 32nd Annual Cryptology Conference. CRYPTO'12. Lecture Notes in Computer Science, vol. 7417, pp. 643–662. Springer (2012)

[51] Dierks, T., Rescorla, E.: RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. Tech. rep., IETF (Aug 2008), http://tools.ietf.org/html/rfc5246

[52] Dobkin, D., Jones, A.K., Lipton, R.J.: Secure databases: protection against user influence. ACM Trans. Database Syst. 4, 97–106 (1979), http://doi.acm.org/10.1145/320064.320068

[53] Du, W., Atallah, M.J.: Protocols for Secure Remote Database Access with Approximate Matching. In: Ghosh, A.K. (ed.) E-Commerce Security and Privacy, Advances in Information Security, vol. 2, pp. 87–111. Springer (2001)

[54] Dwork, C.: Differential privacy: A survey of results. In: Agrawal, M., Du, D.Z., Duan, Z., Li, A. (eds.) Proceedings on the 5th International Conference of Theory and Applications of Models of Computation. TAMC'08. Lecture Notes in Computer Science, vol. 4978, pp. 1–19. Springer (2008)

[55] Ekdahl, P., Johansson, T.: A New Version of the Stream Cipher SNOW. In: Nyberg, K., Heys, H.M. (eds.) Proceedings of the 9th Annual International Workshop on Selected Areas in Cryptography. SAC'02. Lecture Notes in Computer Science, vol. 2595, pp. 47–61. Springer (2002)

[56] EU FP7 Project CACE: D4.6—MPC Virtual Machine Implementation. `http://www.cace-project.eu/` (2010)

[57] Fairplay. `http://www.cs.huji.ac.il/project/Fairplay/fairplay.html`. Last accessed June 1^{st}, 2012.

[58] FairplayMP. `http://www.cs.huji.ac.il/project/Fairplay/fairplayMP.html`. Last accessed June 1^{st}, 2012.

[59] Frank, A., Asuncion, A.: UCI Machine Learning Repository (2010), `http://archive.ics.uci.edu/ml`

[60] Fung, B.C.M., Wang, K., Chen, R., Yu, P.S.: Privacy-preserving data publishing: A survey of recent developments. ACM Computing Surveys 42, 14:1–14:53 (2010)

[61] Geisler, M.: Cryptographic Protocols: Theory and Implementation. Ph.D. thesis, Aarhus University (February 2010)

[62] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) Proceedings of the 41st Annual ACM Symposium on Theory of Computing. STOC'09. pp. 169–178. ACM (2009)

[63] Gentry, C., Halevi, S.: Implementing Gentry's Fully-Homomorphic Encryption Scheme. In: Paterson, K.G. (ed.) Proceedings of the 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques. EUROCRYPT'11. Lecture Notes in Computer Science, vol. 6632, pp. 129–148. Springer (2011)

[64] Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: Aho, A.V. (ed.) Proceedings of the 19th Annual ACM Symposium on Theory of Computing. STOC'87. pp. 218–229. ACM (1987)

[65] Goldreich, O., Ostrovsky, R.: Software Protection and Simulation on Oblivious RAMs. Journal of the ACM 43(3), 431–473 (1996)

[66] Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10. pp. 451–462. ACM (2010)

[67] Hirt, M., Maurer, U.M.: Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In: Burns, J.E., Attiya, H. (eds.) Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'97. pp. 25–34. ACM (1997)

[68] Hirt, M., Nielsen, J.B., Przydatek, B.: Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract). In: Cramer, R. (ed.) Proceedings of the 24^{th} Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT'05. Lecture Notes in Computer Science, vol. 3494, pp. 322–340. Springer (2005)

[69] Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: Proceedings of the 20th USENIX Security Symposium. USENIX'11. USENIX Association (2011)

[70] Jagomägis, R.: A programming language for creating privacy-preserving applications. Bachelor's thesis. University of Tartu (2008)

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

[72] Jarecki, S., Shmatikov, V.: Efficient Two-Party Secure Computation on Committed Inputs. In: Naor, M. (ed.) Proceedings of the 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques. EUROCRYPT'07. pp. 97–114 (2007)

[73] Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. Cryptology ePrint Archive, Report 2011/122 (2011), `http://eprint.iacr.org/`

[74] Kantarcioglu, M., Clifton, C.: Privacy-preserving distributed mining of association rules on horizontally partitioned data. IEEE Transactions on Knowledge and Data Engineering 16(9), 1026–1037 (2004)

[75] Kargupta, H., Datta, S., Wang, Q., Sivakumar, K.: Random-data perturbation techniques and privacy-preserving data mining. Knowledge and Information Systems 7(4), 387–414 (2005)

[76] Katz, J., Ostrovsky, R.: Round-optimal secure two-party computation. In: Franklin, M.K. (ed.) Proceedings of the 24^{th} Annual International Cryptology Conference. CRYPTO'04. Lecture Notes in Computer Science, vol. 3152, pp. 335–354. Springer (2004)

[77] Kenthapadi, K., Mishra, N., Nissim, K.: Simulatable auditing. In: Li, C. (ed.) Proceedings of the 24^{th} ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. PODS'05. pp. 118–127. ACM (2005)

[78] Kerschbaum, F.: Automatically optimizing secure computation. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS'11. pp. 703–714 (2011)

[79] Kleinberg, J.M., Papadimitriou, C.H., Raghavan, P.: Auditing boolean attributes. Journal of Computer and System Sciences 66(1), 244–253 (2003)

[80] Kolesnikov, V., Sadeghi, A.R., Schneider, T.: From Dust to Dawn: Practically Efficient Two-Party Secure Function Evaluation Protocols and their Modular Design. Cryptology ePrint Archive, Report 2010/079 (2010), `http://eprint.iacr.org/2010/079`

[81] Kolesnikov, V., Schneider, T.: Improved Garbled Circuit: Free XOR Gates and Applications. In: Proceedings of the 35th International Colloquium on Automata, Languages and Programming, ICALP'08. pp. 486–498 (2008)

[82] Kreuter, B., Shelat, A., Shen, C.H.: Towards billion-gate secure computation with malicious adversaries. IACR Cryptology ePrint Archive 2012, 179 (2012)

[83] Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (May 1998)

[84] Laur, S.: Cryptographic protocol design. Ph.D. thesis, Helsinki University of Technology (2008)

[85] Laur, S., Lipmaa, H.: A new protocol for conditional disclosure of secrets and its applications. In: Katz, J., Yung, M. (eds.) Proceedings of the 5^{th} International Conference on Applied Cryptography and Network Security,

ACNS '07. Lecture Notes in Computer Science, vol. 4521, pp. 207–225. Springer (2007)

[86] Laur, S., Lipmaa, H.: On the feasibility of consistent computations. In: Nguyen, P.Q., Pointcheval, D. (eds.) 13th International Conference on Practice and Theory in Public Key Cryptography. PKC'10. Lecture Notes in Computer Science, vol. 6056, pp. 88–106. Springer (2010)

[87] Laur, S., Willemson, J., Zhang, B.: Round-Efficient Oblivious Database Manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) Proceedings of the 14th International Conference on Information Security. ISC'11. Lecture Notes in Computer Science, vol. 7001, pp. 262–277. Springer (2011)

[88] Li, N., Li, T., Venkatasubramanian, S.: Closeness: A New Privacy Measure for Data Publishing. IEEE Transactions on Knowledge and Data Engineering 22(7), 943–956 (July 2010)

[89] Lindell, Y., Pinkas, B.: Privacy preserving data mining. Journal of Cryptology 15(3), 177–206 (2002)

[90] Lindell, Y., Pinkas, B.: An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In: Naor, M. (ed.) Proceedings of the 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques. EUROCRYPT'07. pp. 52–78 (2007)

[91] Lloyd, S.: Least squares quantization in PCM. IEEE Transactions on Information Theory 28(2), 129 – 137 (March 1982)

[92] Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M.: *L*-diversity: Privacy beyond *k*-anonymity. IEEE Transactions on Knowledge Discovery from Data 1(1) (2007)

[93] Malin, B., Sweeney, L.: How (not) to protect genomic data privacy in a distributed network: using trail re-identification to evaluate and design anonymity protection systems. Journal of Biomedical Informatics 37(3), 179–192 (2004)

[94] Malka, L.: VMCrypt: modular software architecture for scalable secure computation. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS'11. pp. 715–724 (2011)

[95] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - Secure Two-Party Computation System. In: Proceedings of the 13th USENIX Security Symposium. USENIX'04. pp. 287–302. USENIX (2004)

[96] Mannila, H., Toivonen, H., Verkamo, A.I.: Efficient Algorithms for Discovering Association Rules. In: Proceedings of the KDD Workshop '94. pp. 181–192 (1994)

[97] Mohassel, P., Franklin, M.K.: Efficiency tradeoffs for malicious two-party computation. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography, PKC '06. Lecture Notes in Computer Science, vol. 3958, pp. 458–473. Springer (2006)

[98] Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: Proceedings of the 1st ACM conference on Electronic Commerce, EC'99. pp. 129–139 (1999)

[99] Narayanan, A., Shmatikov, V.: Robust de-anonymization of large sparse datasets. In: Proceedings of the IEEE Symposium on Security and Privacy, S&P '08. pp. 111–125. IEEE Computer Society (2008)

[100] Nielsen, J.D.: Languages for Secure Multiparty Computation and Towards Strongly Typed Macros. Ph.D. thesis, Aarhus University (2009)

[101] Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: Hicks, M.W. (ed.) Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security. PLAS'07. pp. 21–30. ACM (2007)

[102] Nielsen, J.B., Orlandi, C.: Lego for two-party secure computation. In: Reingold, O. (ed.) TCC. Lecture Notes in Computer Science, vol. 5444, pp. 368–386. Springer (2009)

[103] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Proceedings of the 17th International Conference on the Theory and Application of Cryptographic Techniques, EUROCRYPT'99. Lecture Notes in Computer Science, vol. 1592, pp. 223–238. Springer (1999)

[104] RakNet—Multiplayer game network engine. `http://www.jenkinssoftware.com`. Last accessed June 1^{st}, 2012.

[105] Rebane, R.: An integrated development environment for the SecreC programming language. Bachelor's thesis. University of Tartu (2010)

[106] Rebane, R.: A Feasibility Analysis of Secure Multiparty Computation Deployments. Master's thesis, Institute of Computer Science, University of Tartu (2012)

[107] Ristioja, J.: An analysis framework for an imperative privacy-preserving programming language. Master's thesis, Institute of Computer Science, University of Tartu (2010)

[108] Rogaway, P.: The round complexity of secure protocols. Ph.D. thesis, MIT (1991)

[109] Rogaway, P., Bellare, M.: Robust computational secret sharing and a unified account of classical secret-sharing goals. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS '07. pp. 172–184. ACM (2007)

[110] Samarati, P.: Protecting respondents' identities in microdata release. IEEE Transactions on Knowledge and Data Engineering 13(6), 1010–1027 (Nov 2001)

[111] Schröpfer, A., Kerschbaum, F., Mueller, G.: L1 - An Intermediate Language for Mixed-Protocol Secure Computation. In: Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference. COMPSAC'11. pp. 298–307. IEEE Computer Society (2011)

[112] SEPIA—Security through Private Information Aggregation. `http://sepia.ee.ethz.ch/`. Last accessed June 1^{st}, 2012.

[113] Shamir, A.: How to share a secret. Communications of the ACM 22, 612–613 (November 1979)

[114] The SHAREMIND secure computation system. `http://sharemind.cyber.ee`. Last accessed June 1^{st}, 2012.

[115] The SHAREMIND Software Development Kit. Available from `http://sharemind.cyber.ee`. Last accessed January 1^{st}, 2013.

[116] A secure survey prototype based on SHAREMIND. `https://sharemind.cyber.ee/survey/promo/index.html`. Last accessed January 1^{st}, 2013.

[117] The Skein Hash Function Family. `http://www.skein-hash.info`. Last accessed August 3^{rd}, 2012.

[118] Smart, N., Vercauteren, F.: Fully Homomorphic SIMD Operations. Cryptology ePrint Archive, Report 2011/133 (2011), http://eprint.iacr.org/

[119] SQLite—a serverless relational database system. http://www.sqlite.org/. Last accessed June 1^{st}, 2012.

[120] Subramaniam, H., Wright, R.N., Yang, Z.: Experimental Analysis of Privacy-Preserving Statistics Computation. In: Jonker, W., Petkovic, M. (eds.) Proceedings on the VLDB 2004 Workshop on Secure Data Management. SDM'04. Lecture Notes in Computer Science, vol. 3178, pp. 55–66. Springer (2004)

[121] Sweeney, L.: k-anonymity: a model for protecting privacy. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10(5), 557–570 (2002)

[122] Talviste, R.: Web-based data entry in privacy-preserving applications. Bachelor's thesis. University of Tartu (2009)

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

[124] TASTY—Tool for Automating Secure Two-partY computations. http://tastyproject.net. Last accessed June 1^{st}, 2012.

[125] The Tokyo Cabinet database manager. http://fallabs.com/tokyocabinet/. Last accessed June 1^{st}, 2012.

[126] Vaidya, J., Clifton, C.: Privacy preserving association rule mining in vertically partitioned data. In: Proceedings of the 8^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD'02. pp. 639–644. ACM (2002)

[127] VIFF—the Virtual Ideal Functionality Framework. http://viff.dk. Last accessed June 1^{st}, 2012.

[128] Warner, S.: Randomized response: A survey technique for eliminating evasive answer bias. Journal of the American Statistical Association 60(309), 63–69 (1965)

[129] Yao, A.C.C.: Protocols for Secure Computations (Extended Abstract). In: 23rd Annual Symposium on Foundations of Computer Science. FOCS'82. pp. 160–164. IEEE (1982)

[130] Zaki, M.J.: Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering 12(3), 372–390 (2000)

ACKNOWLEDGMENTS

Parts of my doctoral studies have been supported by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science (EXCS), the Software Technology and Applications Competence Centre (STACC) and by the European Social Fund through the Estonian Doctoral School in Information and Communication Technology (IKTDK) and the Doctoral Studies and Internationalisation Programme (DoRa). I would also like to acknowledge the support from EU FP6-IST project No. 15964 "AEOLUS".

My research efforts have also received support from the Estonian Information Technology Foundation through the Tiger University programme and the Ustus Agur scholarship and from the the Estonian Science Foundation through grant No. 8124.

Throughout my studies and my professional career I have had the luck of being mentored by teachers, professors and leaders. I have been inspired and given opportunities that have helped me become a better specialist and a better researcher. I remain thankful for the trust that I have been granted.

I have had the luck to work together with a wonderful team of engineers and researchers who have helped with the design and construction of protocols, tools and applications for the SHAREMIND system. I have enjoyed the collaboration and I would like to thank the whole SHAREMIND team, my colleagues in Cybernetica and my co-authors for helping me build my vision.

This vision and my work on SHAREMIND were sparked from discussions with my supervisor, Sven Laur. His practical view on computer science has been an inspiration to my engineering spirit and I respect and appreciate his guidance throughout my studies. He has taught me many beautiful things about mathematics and computer science.

I give my deepest gratitude to all my family, whose interest in my work and continued support has been the best motivator there can be. I am truly happy to share this journey with my wife, Liina, who gives me an excellent reason for doing everything I do.

KOKKUVÕTE
(SUMMARY IN ESTONIAN)

SHAREMIND: PROGRAMMEERITAV TURVALINE ARVUTUSSÜSTEEM PRAKTILISTE RAKENDUSTEGA

Andmeid isiku tervise, uskumuste ja toimetuleku kohta peetakse tundlikuks ning seda infot tuleb hoolikalt kaitsta. Selliseid andmeid töötlevad asutused peavad kasutama meetmeid, mis välistavad andmete lekke kolmandatele osapooltele. Samas on nii avaliku kui erasektori asutused huvitatud andmete jagamisest, sest see annab neile ligipääsu suuremale infohulgale, mis aitab omakorda teha paremaid otsuseid.

Turvaline ühisarvutus on krüptograafiline meetod, mille abil mitu osapoolt saavad andmeid turvaliselt jagada ja töödelda. Kuigi esimesi protokolle esitleti juba eelmise sajandi 80ndatel aastatel, jõuti esimeste praktiliste lahendusteni alles käesoleva sajandi alguses. Sellest ajast alates on loodud järjest keerukamaid lahendusi ning turvalised ühisarvutused on muutumas teooriast tehnoloogiaks.

Käesolev doktoritöö esitleb autori poolt välja töötatud turvalist arvutussüsteemi nimega SHAREMIND. SHAREMIND on täielik lahendus turvalise ühisarvutuse kasutamiseks andmeid töötlevates rakendustes. Autor on kavandanud ja realiseerinud virtuaalmasina, mis suudab krüptograafiliste protokollide abil töödelda andmeid nii, et need ei ole arusaadavad isegi mitte neid töötlevale arvutusseadmele. Andmetöötlusrakenduste loomiseks on autor kavandanud ka virtuaalmasinaga ühilduva konfidentsiaalsust tagava andmebaasi ning liidesed kasutajaliideste ja programmeerimisvahendite loomiseks.

Töö kolm väidet on järgmised. Esiteks, me väidame, et turvalise ühisarvutusega on võimalik luua universaalseid komponente, mida kasutades saab omakorda ehitada keerukaid andmetöötlussüsteeme ilma iga kord uusi protokolle loomata.

Teiseks, me väidame, et realiseerituna on turvaline ühisarvutus piisavalt efektiivne miljonite andmekirjete töötlemiseks mõistlikus ajas. Viimaks, me väidame, et töövahendite loomisega on turvaline ühisarvutus võimalik kättesaadavaks teha ka neile, kellel puuduvad krüptograafilised teadmised.

Töö koosneb kuuest peatükist ja kolmest artiklist. Esimene peatükk juhatab töö sisse, kirjeldades turvalise andmetöötluse vajalikkust ja tutvustades töö ülesehitust. Teine peatükk kirjeldab erinevaid krüptograafilisi meetodeid andmete turvalisuse tagamiseks arvutusprotsessis ning võrdleb nende efektiivsust.

Kolmas peatükk kirjeldab SHAREMINDi protokollide turvamudelit, analüüsides reaalses maailmas juurutatud turvalise ühisarvutuse süsteemi varitsevaid ohte. Seejärel esitatakse protokollide kogumik, mille abil sellises mudelis on võimalik andmeid turvaliselt koguda ja töödelda. Peatükk tutvustab ka uudset, ühissalastusel põhinevat andmebaasisüsteemi ja kirjeldab päringute tegemist sellises andmebaasis. Lisaks näidatakse, kuidas sellises andmebaasis salvestatud andmeid saab töödelda turvalise ühisarvutusega.

Peatükki täiendavad kaks tööle lisatud artiklit. Esimene neist, "Sharemind: A Framework for Fast Privacy-Preserving Computations" ("Sharemind: raamistik andmete kiireks töötlemiseks privaatsust säilitaval moel"), kirjeldab turvalise ühisarvutuse protokolle täisarvuliste andmete töötlemiseks [25]. Artiklis alustatakse lihtsamatest operatsioonidest nagu liitmine ja korrutamine ning kirjeldatakse seejärel protokolle täisarvu bittide leidmiseks, suurem-kui võrdlemiseks ja võrdsuse kontrolliks. Artikkel sisaldab ka eksperimentide käigus mõõdetud jõudlusnäitajaid

Hilisemas artiklis, "High-performance secure multi-party computation for data mining applications" ("Suure jõudlusega turvaline ühisarvutus rakendustega andmekaeves"), esitatakse efektiivsemad protokollid täisarvude korrutamiseks ja võrdlemiseks [27]. Lisaks kirjeldatakse uusi turvalise ühisarvutuse protokolle bitikaupa nihete tegemiseks, jagamise ja jäägi leidmiseks. Protokollide efektiivsust näitab jõudlusvõrdlus eelmiste protokollidega. Lisaks kirjeldab artikkel eksperimente, kus uue jagamistehte kasulikkuse näitamiseks mõõdetakse SHAREMINDi jõudlust andmete klasterdamisel.

SHAREMINDi jõudlusest annab parema ülevaate neljas peatükk. Peatükis kirjeldatakse, kuidas erinevad aspektid nagu protsessori ja arvutivõrgu kiirus jõudlust mõjutavad. Jõudlustulemuste põhjal järeldatakse, et SHAREMIND on oluliselt efektiivsem, kui selle rakendused teevad mitu sarnast tehet paralleelselt. Jõudluses saadav võit on piisavalt suur, et selle saavutamiseks algoritme ja rakendusi kohandada.

Viies peatükk kirjeldab programmeerimiskeeli ja töövahendeid, millega saab SHAREMINDi jaoks rakenduste loomist lihtsustada. SHAREMINDi rakendusi saab programmeerida kahe programmeerimiskeele abil. SHAREMINDi virtuaalmasin

oskab interpreteerida madala taseme assemblerkeelt. Kuid kuna assemblerkeeles programmeerimine ei ole piisavalt mugav, siis oleme loonud ka mugavama imperatiivse programmeerimiskeele SECREC, mille abil saab kasutada SHAREMINDi kõiki võimalusi ning mis ei nõua krüptograafilisi teadmisi.

Kuuendas peatükis näidatakse SHAREMINDi praktilist rakendatavust erinevate prototüüprakenduste näitel. Üheks olulisemaks näiteks on SHAREMINDi esimene rakendus reaalses maailmas: Eestis juurutatud finantsandmete turvalise kogumise ja analüüsi süsteem. See süsteem on autori teada maailma esimene turvalise ühisarvutuse rakendus, milles infovahetus toimub avaliku interneti abil. Selles peatükis annab autor ka juhiseid, kuidas kavandada, realiseerida ja juurutada rakendust, mis kasutab turvalist ühisarvutust.

Peatüki juurde kuulub ka doktoritöö kolmas artikkel: “A Universal Toolkit for Cryptographically Secure Privacy-Preserving Data Mining” (“Universaalne töövahend krüptograafilise turvalisusega privaatsust säilitavaks andmekaeveks”), mis kirjeldab, kuidas SHAREMINDi abil lahendada tihtiesinevate alamhulkade otsingu ülesannet [22]. Artiklis kirjeldatakse SHAREMINDi jaoks kohandatud versioone populaarsetest tihtiesinevate alamhulkade leidmise algoritmidest Apriori [4, 96] ja Eclat [130]. Artikkel kirjeldab algoritme, nende privaatsusgarantiisid ja esitab eksperimenditulemused, mis näitavad piisavat jõudlust praktiliseks kasutuseks.

ORIGINAL PUBLICATIONS

Publication	Pages
Dan Bogdanov, Sven Laur, Jan Willemson ”Sharemind: A Framework for Fast Privacy-Preserving Computations”	133 – 149
Dan Bogdanov, Margus Niitsoo, Tomas Toft, Jan Willemson ”High-performance secure multi-party computation for data mining applications”	149 – 166
Dan Bogdanov, Roman Jagomägis, Sven Laur ”A Universal Toolkit for Cryptographically Secure Privacy-Preserving Data Mining”	166 – 182

I

Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) Proceedings of the 13th European Symposium on Research in Computer Security, ESORICS '08. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008).

Sharemind: A Framework for Fast Privacy-Preserving Computations

Dan Bogdanov[1,2,⋆], Sven Laur[1], and Jan Willemson[1,2,⋆]

[1] University of Tartu, Liivi 2, 50409 Tartu, Estonia
{db,swen,jan}@ut.ee
[2] Cybernetica AS, Akadeemia tee 21, 12618 Tallinn, Estonia

Abstract. Gathering and processing sensitive data is a difficult task. In fact, there is no common recipe for building the necessary information systems. In this paper, we present a provably secure and efficient general-purpose computation system to address this problem. Our solution—SHAREMIND—is a virtual machine for privacy-preserving data processing that relies on share computing techniques. This is a standard way for securely evaluating functions in a multi-party computation environment. The novelty of our solution is in the choice of the secret sharing scheme and the design of the protocol suite. We have made many practical decisions to make large-scale share computing feasible in practice. The protocols of SHAREMIND are information-theoretically secure in the honest-but-curious model with three computing participants. Although the honest-but-curious model does not tolerate malicious participants, it still provides significantly increased privacy preservation when compared to standard centralised databases.

1 Introduction

Large-scale adoption of online information systems has made both the use and abuse of personal data easier than before. This has caused an increased awareness about privacy issues among individuals. In many countries, databases containing personal, medical or financial information about individuals are classified as sensitive and the corresponding laws specify who can collect and process sensitive information about a person.

On the other hand, the use of sensitive information plays an essential role in medical, financial and social studies. Thus, one needs a methodology for conducting statistical surveys without compromising the privacy of individuals. Privacy-preserving data mining techniques try to address such problems. So far the focus has been on randomised response techniques [1,2,13]. In a nutshell, recipients of the statistical survey apply a fixed randomisation method on their responses. As a result, each individual reply is erroneous, whereas the global statistical properties of the data are preserved. Unfortunately, such transformations can preserve privacy only on average and randomisation reduces the precision of the outcomes. Also, we cannot give security guarantees for individual records. In fact, the corresponding guarantees are rather weak and the use of extra information might significantly reduce the level of privacy.

Another alternative is to consider this problem as a multi-party computation task, where the data donors want to securely aggregate data without revealing their private

⋆ This research has been supported by Estonian Science Foundation grant number 7081.

S. Jajodia, and J. Lopez (Eds.): ESORICS 2008, LNCS 5283, pp. 192–206, 2008.

inputs. However, the corresponding cryptographic solutions quickly become practically intractable when the number of participants grows beyond few hundreds. Moreover, data donors are often unwilling to stay online during the entire computation and their computers can be easily taken over by adversarial forces.

As a way out, we propose a hierarchical solution, where all computations are done by dedicated *miner* parties who are less susceptible to external corruption. Consequently, we can assume that only a few miner parties can be corrupted during the computation. Thus, we can use secret sharing and share computing techniques for privacy-preserving data aggregation. In particular, data donors can safely submit their inputs by sending the corresponding shares to the miners. As a result, the miners can securely evaluate any aggregate statistic without further interaction with the data donors.

Our Contribution. The presented theoretical solution does not form the core of this paper. Share computing techniques have been known for decades and thus all important results are well established by now, see [3,7] for further references. Hence, we focused mainly on practical aspects and developed the SHAREMIND framework for privacy-preserving computations. The SHAREMIND framework is designed to be an efficient and easily programmable platform for developing and testing various privacy-preserving algorithms. It consists of the computation runtime environment and a programming library for creating private data processing applications. As a result, one can develop secure multi-party protocols without the explicit knowledge of all implementation details. On the other hand, it is also possible to test and add your own protocols to the library, since the source code of SHAREMIND is freely available [17].

We have made some non-standard choices to assure maximal efficiency. First, the SHAREMIND framework uses additive secret sharing scheme over the ring $\mathbb{Z}_{2^{32}}$. Besides the direct computational gains, such a choice also simplifies many share computing protocols. When a secret sharing protocol is defined over a finite field $\mathbb{Z}_p$, then any overflow in computations causes modular reductions that corrupt the end result. In the SHAREMIND framework, all modular reductions occur modulo 2^{32} and thus results always coincide with the standard 32-bit integer arithmetic. On the other hand, standard share computing techniques are not applicable for the ring $\mathbb{Z}_{2^{32}}$. In particular, we were forced to roll out our own multiplication protocol, see Sect. 4.

Second, the current implementation of SHAREMIND supports the computationally most efficient setting, where only one of three miner nodes can be semi-honestly corrupted. As discussed in Sect. 3, the corresponding assumption can be enforced with a reasonable practical effort. Also, it is possible to extend the framework for other settings. For example, one can implement generic methodology given in [9].

To make the presentation more fluent, we describe the SHAREMIND framework step by step through Sect. 2–5. Performance results are presented and analysed in Sect. 6. In particular, we compare our results with other implementations of privacy-preserving computations [6,16,18]. Finally, we conclude our presentation with some improvement plans for future, see Sect. 7.

Some of the details of this work have been omitted because of space limitations. The full version of this article that covers all these details can be found on the homepage of SHAREMIND project [17] and in the IACR ePrint Archive [5].

2 Cryptographic Preliminaries

Theoretical Attack Model. In this article, we state and prove all security guarantees in the *information-theoretical setting*, where each pair of participants is connected with a private communication channel that provides asynchronous communication. In other words, a potential adversary can only delay or reorder messages without reading them. We also assume that the communication links are authentic, i.e., the adversary cannot send messages on behalf of non-corrupted participants. The adversary can corrupt participants during the execution of a protocol. In the case of *semi-honest* corruption, the adversary can only monitor the internal state of a corrupted participant, whereas the adversary has full control over *maliciously* corrupted participants. We consider only *threshold adversaries* that can adaptively corrupt up to t participants. Such an attack model is well established, see [4,14] for further details.

Secondly, we consider only self-synchronising protocols, where the communication can be divided into distinct rounds. A protocol is *self-synchronising* if the adversary cannot force (semi-)honest participants to start a new communication round until all other participants have completed the previous round. As a result, this setting becomes equivalent to the standard synchronised network model with a rushing adversary.

Secure Multi-party Computation. Assume that participants $\mathcal{P}_1, \ldots, \mathcal{P}_n$ want to compute outputs $y_i = f_i(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are corresponding private inputs. Then the security of a protocol π that implements the described functionality is defined by comparing the protocol with the ideal implementation π°, where all participants submit their inputs $x_1, \ldots, x_n$ securely to the trusted third party $\mathcal{T}$ that computes the necessary outputs $y_i = f_i(x_1, \ldots, x_n)$ and sends $y_1, \ldots, y_n$ securely back to the respective participants. A malicious participant $\mathcal{P}_i$ can halt the ideal protocol π° by submitting $x_i = \perp$. Then the trusted third party $\mathcal{T}$ sends $\perp$ as an output for all participants. Now a protocol π is secure if for any plausible attack $\mathcal{A}$ against the protocol π there exists a plausible attack $\mathcal{A}^\circ$ against the protocol π° that causes comparable damage.

For brevity, let us consider only the stand-alone setting, where only a single protocol instance is executed and all honest participants carry out no side computations. Let $\phi_i = (\sigma_i, x_i)$ denote the entire input state of $\mathcal{P}_i$ and let $\psi_i = (\phi_i, y_i)$ denote the entire output state. Similarly, let ϕ_a and ψ_a denote the inputs and outputs of the adversary and $\boldsymbol{\phi} = (\phi_1, \ldots, \phi_n, \phi_\mathrm{a})$, $\boldsymbol{\psi} = (\psi_1, \ldots, \psi_n, \psi_\mathrm{a})$ the corresponding input and output vectors. Then a protocol π is *perfectly secure* if for any plausible τ_re-time real world adversary $\mathcal{A}$ there exists a plausible τ_id-time ideal world adversary $\mathcal{A}^\circ$ such that for any input distribution $\boldsymbol{\phi} \leftarrow \mathfrak{D}$ the corresponding output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ in the real and ideal world coincide and the running times τ_re and τ_id are comparable.

In the asymptotic setting, the running times are *comparable* if τ_id is polynomial in τ_re. For fixed time bound τ_re, one must decide an acceptable time bound τ_id by him- or herself. All security proofs in this article are suitable for both security models, since they assure that $\tau_\mathrm{id} \leq c \cdot \tau_\mathrm{re}$ where c is a relatively small constant.

In our setting, a real world attack $\mathcal{A}$ is plausible if it corrupts up to t participants. The corresponding ideal world attack $\mathcal{A}^\circ$ is plausible if it corrupts the same set of participants as the real world attack. Further details and standard results can be found in the manuscripts [3,7,8,11].

Universal Composability. Complex protocols are often designed by combining several low level protocols. Unfortunately, stand-alone security is not enough to prove the security of the compound protocol and we must use more stringent security definitions. More formally, let $\varrho\langle\cdot\rangle$ be a global context that uses the functionality of a protocol π. Then we can compare real and ideal world protocols $\varrho\langle\pi\rangle$ and $\varrho\langle\pi^\circ\rangle$.

Let $\boldsymbol{\phi}$, $\boldsymbol{\psi}$, $\boldsymbol{\psi}^\circ$ denote the input and output vectors of the compound protocols $\varrho\langle\pi\rangle$ and $\varrho\langle\pi^\circ\rangle$. Then a protocol π is *perfectly universally composable* if for any plausible τ_{re}-time attack $\mathcal{A}$ against $\varrho\langle\pi\rangle$ there exists a plausible τ_{id}-time attack $\mathcal{A}^\circ$ against $\varrho\langle\pi^\circ\rangle$ such that for any input distribution $\boldsymbol{\phi} \leftarrow \mathfrak{D}$ the output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ coincide and the running times τ_{re} and τ_{id} are comparable. We refer to the manuscript [8] for a more formal and precise treatment.

Secret Sharing Schemes. Secret sharing schemes are used to securely distribute private values to a group of participants. More precisely, let $\mathcal{M}$ be the set of possible secrets and let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be the sets of possible shares. Then shares for the participants are created with a randomised sharing algorithm $\mathsf{Deal} : \mathcal{M} \to \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$. Participants can use a recovery algorithm $\mathsf{Rec} : \mathcal{S}_1 \times \ldots \times \mathcal{S}_n \to \mathcal{M} \cup \{\perp\}$ to restore the secret form shares. For brevity, we use a shorthand $[\![s]\!]$ to denote the shares $[s_1, \ldots, s_n]$ generated by the sharing algorithm $\mathsf{Deal}(s)$.

Secret sharing schemes can have different security properties depending on the exact details of Deal and Rec algorithms. The SHAREMIND framework uses *additive sharing* over $\mathbb{Z}_{2^{32}}$, where a secret value s is split to shares $s_1, \ldots, s_n \in \mathbb{Z}_{2^{32}}$ such that

$$s_1 + s_2 + \cdots + s_n \equiv s \mod 2^{32}$$

and any $n-1$ element subset $\{s_{i_1}, \ldots, s_{i_{n-1}}\}$ is uniformly distributed. As a result, participants cannot learn anything about s unless all of them join their shares.

3 Privacy-Preserving Data Aggregation

As already emphasised in the introduction, organisations who collect and process data may abuse it or reveal the data to third parties. As a result, people are unwilling to reveal sensitive information without strong security guarantees. Although proper legislation and auditing reduces the corresponding risks, data donors must often unconditionally trust institutions that gather and process data. In the following, we show how to use cryptographic techniques to avoid such unconditional trust.

The SHAREMIND framework for privacy-preserving computations uses secret sharing to split confidential information between several nodes (*miners*). By sending the shares of the data to the miners, data donors effectively delegate all rights over the data to the consortium of miners. Let t be the prescribed corruption threshold such that no information can be learnt about the inputs if the number of collaborating corrupted parties is below t. We allow some miner nodes to be corrupted, but require that the total number of corrupted nodes is below the threshold t. The latter can be achieved with physical and organisational security measures such as dedicated server rooms and software auditing. This is achievable, since the framework needs only a few miner nodes. In practice, each miner node should be hosted by a separate respected organisation.

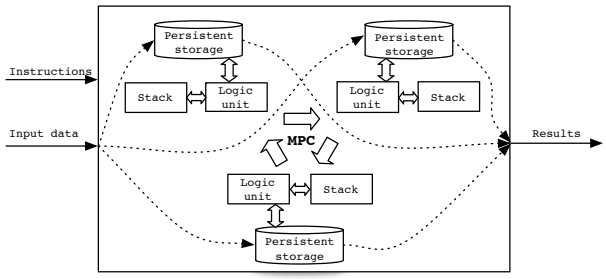


Fig. 1. In SHAREMIND, input data and instructions are sent to miner nodes that use multi-party computation to execute the algorithm. The result is returned when the computation is complete.

The high level description of the SHAREMIND framework is depicted in Fig. 1. Essentially, one can view SHAREMIND as a virtual processor that provides secure storage for shared inputs and performs privacy-preserving operations on them. Each miner node $\mathcal{P}_i$ has a local *database* for persistent storage and a local *stack* for storing intermediate results. All values in the database and stack are shared among all miners $\mathcal{P}_1, \ldots, \mathcal{P}_n$ by using an *additive secret sharing* over $\mathbb{Z}_{2^{32}}$. The framework provides efficient protocols for basic mathematical operations so that one could easily implement more complex tasks. In particular, one should be able to construct such protocols without any knowledge about underlying cryptographic techniques. For that reason, all implementations of basic operations in the SHAREMIND framework are perfectly universally composable.

The current version of SHAREMIND framework is based on three miner nodes and tolerates semi-honest corruption of a single node, i.e., no information is leaked unless two miner nodes collaborate. The latter is a compromise between efficiency and security. Although a larger number of miner nodes increases the level of tolerable corruption, it also makes assuring semi-honest behaviour much more difficult. Secondly, the communication complexity of multi-party computation protocols is roughly quadratic in the number of miners n and thus three is the optimal choice. Besides, it is difficult to find more than a handful of independent organisations that can provide adequate protection measures and are not motivated to collaborate with each other.

To achieve maximal efficiency, we also use non-orthodox secret sharing and share computing protocols. Recall that most classical secret sharing schemes work over finite fields. As a result, it is easy to implement secure addition and multiplication modulo prime p or in the Galois field $\mathbb{F}_{2^k}$. However, the integer arithmetic in modern computers is done modulo 2^{32}. Consequently, the most space- and time-efficient solution is to use additive secret sharing over $\mathbb{Z}_{2^{32}}$. There is no need to implement modular arithmetic and we do not have to compensate the effect of modular reductions. On the other hand, we have to use non-standard methods for share computing, since Shamir secret sharing scheme does not work over $\mathbb{Z}_{2^{32}}$. We discuss these issues further in Sect. 4.

Initially, the database is empty and data donors have to submit their inputs by sending the corresponding shares privately to miners who store them in the database. We describe this issue more thoroughly in Sect. 4.2. After the input data is collected, a data analyst can start privacy-preserving computations by sending instructions to the miners. Each instruction is a command that either invokes a share computing protocol or just

reorders shares. The latter allows a data analyst to specify complex algorithms without thinking about implementation details. More importantly, the corresponding complex protocol is guaranteed to preserve privacy, as long as the execution path in the program itself does not reveal private information. This restriction must be taken into account when choosing data analysis algorithms for implementation on SHAREMIND.

Each arithmetic instruction invokes a secure multi-party protocol that provides new shares. These shares are then stored on the stack. For instance, a unary stack instruction f takes the top shares $[\![u]\!]$ of the stack and pushes the resulting shares $[\![f(u)]\!]$ to the stack top. Analogously, a fixed binary stack instruction $\otimes$ takes two top most shares $[\![u]\!]$ and $[\![v]\!]$ and pushes $[\![u \otimes v]\!]$ to the stack. For efficiency reasons, we have also implemented vectorised operations to perform the same protocol in parallel. This significantly reduces the number of rounds required for applying similar operations on many inputs.

The current implementation of SHAREMIND framework provides privacy preserving addition, multiplication and greater-than-or-equal comparison of two shared values. It can also multiply a shared value with a constant and extract its bits as shares. Share conversion from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$ and bitwise addition are mostly used as components in other protocols, but they are also available to the programmer. We emphasise here that many algorithms for data mining and statistical analysis do not use other mathematical operations and thus this instruction set is sufficient for many applications. Moreover, note that bit extraction and arithmetic primitives together are sufficient to implement any Boolean circuit with a linear overhead and thus the SHAREMIND framework is also Turing complete. We acknowledge here that there are more efficient ways to evaluate Boolean circuits like Yao circuit evaluation (see [15]) and we plan to include protocols with similar properties in the future releases of SHAREMIND.

We analyse the security of all share manipulation protocols in the information-theoretical attack model that was specified in Sect. 2. How to build such a network form standard cryptographic primitives is detailed in Sect. 5. Also, note that the next section provides only a general description of all protocols, detailed technical description of all protocols can be found in the full version of this article [5].

4 Share Computing Protocols

All computational instructions in the SHAREMIND framework are either unary or binary operations over unsigned integers represented as elements of $\mathbb{Z}_{2^{32}}$ or their vectorised counterparts. Hence, all protocols have the following structure. Each miner $\mathcal{P}_i$ uses shares u_i and v_i as inputs to the protocol to obtain a new share w_i such that $[\![w]\!]$ is a valid sharing of $f(u)$ or $u \otimes v$. In the corresponding idealised implementation, all miners send their input shares to the trusted third party $\mathcal{T}$ who restores all inputs, computes the corresponding output w and sends back newly computed shares $[\![w]\!] \leftarrow \mathsf{Deal}(w)$. Hence, the output shares $[\![w]\!]$ are independent of input shares and thus no information is leaked about the input shares if we publish all output shares.

Although share computing protocols are often used as elementary steps in more complex protocols, they themselves can be composed from even smaller atomic operations. Many of these atomic sub-protocols produce output shares that are never published. Hence, it makes sense to introduce another security notion that is weaker than universal

1. Each party $\mathcal{P}_i$ sends a random mask $r_i \leftarrow \mathbb{Z}_{2^{32}}$ to the right neighbour $\mathcal{P}_{i+1}$.
2. Each party $\mathcal{P}_i$ uses the input share u_i to compute the output $w_i \leftarrow u_i + r_{i-1} - r_i$.

Fig. 2. Re-sharing protocol for three parties

composability. We say that a share computing protocol is *perfectly simulatable* if there exists an efficient universal non-rewinding simulator $\mathcal{S}$ that can simulate all protocol messages to any real world adversary $\mathcal{A}$ so that for all input shares the output distributions of $\mathcal{A}$ and $\mathcal{S}\langle\mathcal{A}\rangle$ coincide. Most importantly, perfect simulatability is closed under concurrent composition. The corresponding proof is straightforward.

Lemma 1. *If all sub-protocols of a protocol are perfectly simulatable, then the protocol is perfectly simulatable.*

Proof (Sketch). Since all simulators $\mathcal{S}_i$ of sub-protocols are non-rewinding, we can construct a single compound simulator $\mathcal{S}_*$ that runs simulators $\mathcal{S}_i$ in parallel to provide the missing messages to $\mathcal{A}$. As each simulator $\mathcal{S}_i$ is perfect, the final view of $\mathcal{A}$ is also perfectly simulated. □

However, perfect simulatability alone is not sufficient for universal composability. Namely, output shares of a perfectly simulatable protocol may depend on input shares. As a result, published shares may reveal more information about inputs than necessary. Therefore, we must often re-share the output shares at the end of each protocol.

The corresponding ideal functionality is modelled as follows. Initially, the miners send their shares $[\![u]\!]$ to the trusted third party $\mathcal{T}$ who recovers the input $u \leftarrow \mathsf{Rec}([\![u]\!])$ and sends new shares $[\![w]\!] \leftarrow \mathsf{Deal}(u)$ back to the miners. The simplest universally composable re-sharing protocol is given in Fig. 2. Indeed, we can construct a non-rewinding *interface* $\mathcal{I}_0$ between the ideal world and a real world adversary $\mathcal{A}$ such that for any input distribution the output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ coincide. The corresponding interface $\mathcal{I}_0$ forwards the input share u_i of a corrupted miner $\mathcal{P}_i$ to $\mathcal{T}$, provides randomness $r_i \leftarrow \mathbb{Z}_{2^{32}}$ to $\mathcal{P}_i$, and given w_i form $\mathcal{T}$ sends $r_{i-1} \leftarrow w_i - u_i + r_i$ to $\mathcal{P}_i$.

The next lemma shows that perfect simulatability together with re-sharing assures universal composability in the semi-honest model. In the malicious model, one needs additional correctness guarantees against malicious behaviour.

Lemma 2. *A perfectly simulatable share computing protocol that ends with perfectly secure re-sharing of output shares is perfectly universally composable.*

Proof. Let $\mathcal{S}$ be the perfect simulator for the share computing phase and $\mathcal{I}_0$ the interface for the re-sharing protocol. Then we can construct a new non-rewinding interface $\mathcal{I}$ for the whole protocol:

1. It first submits the inputs of the corrupted miners $\mathcal{P}_i$ to the trusted third party $\mathcal{T}$ and gets back the output shares w_i.
2. Next, it runs, possibly in parallel, the simulator $\mathcal{S}$ and the interface $\mathcal{I}_0$ with the output shares w_i to simulate the missing protocol messages.

Now the output distributions ψ and ψ° coincide, since the sub-routines $\mathcal{S}$ and $\mathcal{I}_0$ perfectly simulate protocol messages and $\mathcal{I}_0$ assures that the output shares of corrupted parties are indeed w_i. The latter assures that the adversarial output ψ°_{a} is correctly matched together with the outputs of honest parties. Since the interface $\mathcal{I}$ is non-rewinding, the claim holds even if the protocol is executed in a larger computational context $\varrho\langle\cdot\rangle$. □

4.1 Protocols for Atomic Operations

Due to the properties of additive sharing, we can implement share addition and multiplication by a public constant c with local operations only, as $[u_1 + v_1, \ldots, u_n + v_n]$ and $[cu_1, \ldots, cu_n]$ are valid shares of $u + v$ and cu. However, these operations are only perfectly simulatable, since the output shares depend on input shares.

A share multiplication protocol is another important atomic primitive. Unfortunately, we cannot use the standard solutions based on polynomial interpolation and re-sharing. Shamir secret sharing just fails in the ring $\mathbb{Z}_{2^{32}}$. Hence, we must roll out our own multiplication protocol. By the definition of the additive secret sharing scheme

$$uv = \sum_{i=1}^{n} u_i v_i + \sum_{j \neq i}^{n} u_i v_i \mod 2^{32} \tag{1}$$

and thus we need sub-protocols for computing shares of $u_i v_j$. For clarity and brevity, we consider only a sub-protocol, where $\mathcal{P}_1$ has an input x_1, $\mathcal{P}_2$ has an input x_2 and the miner $\mathcal{P}_3$ helps the others to obtain shares of $x_1 x_2$. Du and Atallah were the first to publish the corresponding protocol [12] although similar reduction techniques have been used earlier. Fig. 3 depicts the corresponding protocol. Essentially, the correctness of the protocol relies on the observation

$$x_1 x_2 = -(x_1 + \alpha_1)(x_2 + \alpha_2) + x_1(x_2 + \alpha_2) + (x_1 + \alpha_1)x_2 + \alpha_1\alpha_2 \ .$$

The security follows from the fact that for uniformly and independently generated $\alpha_1, \alpha_2 \leftarrow \mathbb{Z}_{2^{32}}$ the sums $x_1 + \alpha_1$ and $x_2 + \alpha_2$ have also uniform distribution.

Lemma 3. *The Du-Atallah protocol depicted in Fig. 3 is perfectly simulatable.*

Proof. Let us fix inputs x_1 and x_2. Then $\mathcal{P}_1$ receives two independent uniformly distributed values and $\mathcal{P}_2$ receives two independent uniformly distributed values. $\mathcal{P}_3$ receives no values at all. Hence, it is straightforward to construct a simulator $\mathcal{S}$ that simulates the view of a semi-honest participant. □

1. $\mathcal{P}_3$ generates $\alpha_1, \alpha_2 \leftarrow \mathbb{Z}_{2^{32}}$ and sends α_1 to $\mathcal{P}_1$ and α_2 to $\mathcal{P}_2$.
2. $\mathcal{P}_1$ computes $x_1 + \alpha_1$ and sends the result to $\mathcal{P}_2$.
   $\mathcal{P}_2$ computes $x_2 + \alpha_2$ and sends the result to $\mathcal{P}_1$.
3. Parties compute shares of $x_1 x_2$:
   (a) $\mathcal{P}_1$ computes its share $w_1 = -(x_1 + \alpha_1)(x_2 + \alpha_2) + x_1(x_2 + \alpha_2)$.
   (b) $\mathcal{P}_2$ computes its share $w_2 = (x_1 + \alpha_1)x_2$.
   (c) $\mathcal{P}_3$ computes its share $w_3 = \alpha_1\alpha_2$.

Fig. 3. Du-Atallah multiplication protocol

Execute the following protocols concurrently:

1. Compute locally shares u_1v_1, u_2v_2 and u_3v_3.
2. Use six instances of the Du-Atallah protocol for computing shares of u_iv_j where $i \neq j$.
3. Re-share the final sum of all previous sub-output shares.

Fig. 4. High-level description of the share multiplication protocol

Fig. 4 depicts a share multiplication protocol that executes six instances of the Du-Atallah protocol in parallel to compute the right side of the equation (1). Since the protocols are executed concurrently, the resulting protocol has only three rounds.

Theorem 1. *The multiplication protocol is perfectly universally composable.*

Proof. Lemma 1 assures that the whole protocol is perfectly simulatable, as local computations and instances of Du-Atallah protocol are perfectly simulatable. Since the output shares are re-shared, Lemma 2 provides universal composability. □

4.2 Protocol for Input Gathering

Many protocols can be directly built on the atomic operations described in the previous sub-section. As the first example, we discuss methods for input validation. Recall that initially the database of shared inputs is empty in the SHAREMIND framework and the data donors have to fill it. There are two aspects to note. First, the data donors might be malicious and try to construct fraudulent inputs to influence data aggregation procedures. For instance, some participants of polls might be interested in artificially increasing the support of their favourite candidate. Secondly, the data donors want to submit their data as fast as possible without extra work. In particular, they are unwilling to prove that their inputs are in the valid range.

There are two principal ways to address these issues. First, the miners can use multiparty computation protocols to detect and eliminate fraudulent entries. This is computationally expensive, since the evaluation of correctness predicates is a costly operation. Hence, it is often more advantageous to use such an input gathering procedure that guarantees validity by design. For instance, many data tables consist of binary inputs (yes-no answers). Then we can gather inputs as shares over $\mathbb{Z}_2$ to avoid fraudulent inputs and later use share conversion to get the corresponding shares over $\mathbb{Z}_{2^{32}}$.

Let $[u_1, u_2, u_3]$ be a valid additive sharing over $\mathbb{Z}_2$. Then we can express the shared value u through the following equation over integers:

$$f(u_1, u_2, u_3) := u_1 + u_2 + u_3 - 2u_1u_2 - 2u_1u_3 - 2u_2u_3 + 4u_1u_2u_3 = u \ .$$

Consequently, if we treat u_1, u_2, u_3 as inputs and compute the shares of $f(u_1, u_2, u_3)$ over $\mathbb{Z}_{2^{32}}$, then we obtain the desired sharing of u. More precisely, we can use the Du-Atallah protocol to compute the shares $[\![u_1u_2]\!]$, $[\![u_1u_3]\!]$, $[\![u_2u_3]\!]$ over $\mathbb{Z}_{2^{32}}$. To get the shares $[\![u_1u_2u_3]\!]$, we use the share multiplication protocol to multiply $[\![u_1u_2]\!]$ and the shares $[\![u_3]\!]$ created by $\mathcal{P}_3$. Finally, all parties use local addition and multiplication

1. Generate random bit shares $[\![r^{(31)}]\!], \ldots, [\![r^{(0)}]\!]$ over $\mathbb{Z}_{2^{32}}$.
2. Compute the corresponding shares $[\![r]\!] = 2^{31} \cdot [\![r^{(31)}]\!] + \cdots + 2^0 \cdot [\![r^{(0)}]\!]$.
3. Compute and publish the shares of the difference $[\![a]\!] = [\![u]\!] - [\![r]\!]$.
4. Mimic bitwise addition algorithm to compute bit shares $[\![u^{(31)}]\!], \ldots, [\![u^{(0)}]\!]$ from the known bit representation of a and the bit shares $[\![r^{(31)}]\!], \ldots, [\![r^{(0)}]\!]$.

Fig. 5. High-level description of the bit extraction protocol

routines to obtain the shares of $f(u_1, u_2, u_3)$ and then re-share them to guarantee the universal composability. The resulting protocol has only four rounds, since we can start the first round of all multiplication protocols simultaneously.

Theorem 2. *The share conversion protocol is perfectly universally composable.*

Proof. The proof follows again directly from Lemmata 1 and 2, since all sub-protocols are perfectly simulatable and the output shares are re-shared at the end. □

Note that input gathering can even be an off-line event, if we make use of public-key encryption. If everybody knows the public keys of the miners, they can encrypt the shares with the corresponding keys and then store the encryptions in a public database. Miners can later fetch and decrypt their individual shares to fill their input databases.

4.3 Protocols for Bit Extraction and Comparison

Various routines for bit manipulations form another set of important operations. In particular, note that for signed representation of $\mathbb{Z}_{2^{32}} = \{-2^{31}, \ldots, 0, \ldots, 2^{31} - 1\}$ the highest bit indicates the sign and thus the evaluation of greater-than-or-equal (GTE) predicate can be reduced to bit extraction operations. In the following, we mimic the generic scheme proposed by Damgård et al. [10] for implementing bit-level operations. As this construction is given in terms of atomic primitives, it can be used also for settings where there are more than three miners, see Fig. 5.

For the first step in the algorithm, miners can create random shares over $\mathbb{Z}_2$ and then convert them to the shares over $\mathbb{Z}_{2^{32}}$. The second step can be computed locally. The third step is secure, since the difference $a = u - r$ has uniform distribution over $\mathbb{Z}_{2^{32}}$ and thus one can always simulate the shares of a. For the final step, note that addition and multiplication protocols are sufficient to implement all logic gates when all inputs are guaranteed to be in the range $\{0, 1\}$. Hence, we can use the classical bitwise addition algorithm to compute $[\![u^{(31)}]\!], \ldots, [\![u^{(0)}]\!]$. However, the number of rounds in the corresponding protocol is linear in the number of bits, since we cannot compute carry bits locally. To minimise the number of rounds, we used standard look-ahead carry construction to perform the carry computations in parallel. The latter provides logarithmic round complexity. More precisely, the final bitwise addition protocol has 8 rounds and the corresponding bit extraction protocol has 12 rounds. Both protocols are also universally composable, since all sub-protocols are universally composable.

Theorem 3. *The bitwise addition protocol is perfectly universally composable. The bit extraction protocol is perfectly universally composable.*

As a simple extension, we describe how to implement greater-than-or-equal predicate if both arguments are guaranteed to be in $\mathbb{Z}_{2^{31}} \subseteq \mathbb{Z}_{2^{32}}$. This allows us to define

$$\textsc{Gte}(x, y) = \begin{cases} 1, & \text{if the highest bit of the difference } x - y \text{ is } 0, \\ 0, & \text{otherwise.} \end{cases}$$

It is straightforward to see that the definition is correct for unsigned and signed interpretation of the arguments as long as both arguments are in the range $\mathbb{Z}_{2^{31}}$. Since the range $\mathbb{Z}_{2^{31}}$ is sufficient for most practical computations, we have not implemented the extended protocol for the full range $\mathbb{Z}_{2^{32}} \times \mathbb{Z}_{2^{32}}$, yet.

Theorem 4. *The greater-than-or-equal protocol is perfectly universally composable.*

Proof. The protocol is universally composable, since the bit extraction protocol that is used to split $x - y$ into bit shares is universally composable. □

5 Practical Implementation

The main goal of the SHAREMIND project is to provide an easily programmable and flexible platform for developing and testing various privacy preserving algorithms based on share computing. The implementation of the SHAREMIND framework provides a library of the most important mathematical primitives described in the previous section. Since these protocols are universally composable, we can use them in any order, possibly in parallel, to implement more complex algorithms. To hide the execution path of the algorithm, we can replace if-then branches with oblivious selection clauses. For instance, we can represent **if** a **then** $x \leftarrow y$ **else** $x \leftarrow z$ as $x \leftarrow a \cdot y + (1 - a) \cdot z$.

The software implementation of SHAREMIND is written in the C++ programming language and is tested on Linux, Mac OS X and Windows XP. The "virtual processor" of SHAREMIND consists of the *miner* application which performs the duties of a secure multiparty computation party and the *controller* library for developing controller applications that work with the miners. Secure channels between the miners are implemented using standard symmetric encryption and authentication algorithms. As a result, we obtain only computational security guarantees in the real world. The latter is unavoidable if we want to achieve a cost-efficient and universal solution, as building dedicated secure channels is currently prohibitively expensive.

One of the biggest advances of the framework is its modularity. At the highest abstraction level, the framework behaves as a virtual processor with a fixed set of commands. However, the user can design and experiment with new cryptographic protocols. On this level, the framework hides all technical details, such as network setup and exact details of message delivery. Finally, the user can explicitly change networking details at the lowest level, although we have put a lot of effort into optimising network behaviour.

To facilitate fast testing and algorithm development, we implemented the most obvious execution strategy, where the controller application executes a program by asking the miners to sequentially execute operations described by the program. When a computational operation is requested from the miner, it is scheduled for execution. When the operation is ready to be executed, the miners run the secure multi-party computation

protocols necessary for completing the operation. Like in a standard stack machine, all operations read their input data from the stack and write output data to the stack upon completion. The shares of the final results are sent back to the controller.

Of course, such a simplistic approach neglects many practical security concerns. In particular, the controller has full control over the miners and thus we have a single point of failure. Therefore, real-world applications must be accompanied with auxiliary mechanisms to avoid such high level attacks. For instance, the miners must be configured with the identities of each other and all possible controllers to avoid unauthorised commands. This can be achieved by using public-key infrastructure. Similarly, the complete code should be analysed and signed by an appropriate authority to avoid unauthorised data manipulation. However, the time-complexity of these operations is constant and thus our execution strategy is still valid for performance testing.

6 Performance Results

We have measured the performance of the SHAREMIND framework on two computational tasks—scalar product and vectorised comparison. These tests are chosen to cover the most important primitives of SHAREMIND: addition, multiplication and comparison. More importantly, it also allowed us to compare SHAREMIND to other secure multi-party computation systems [6,16,18].

The input datasets were randomly generated and the corresponding shares were stored in local databases. For each vector size, we ran the computation many times and measured the results for each execution. To identify performance bottlenecks, we measured the local computation time, the time spent on sending data, and the time spent on waiting. The time was measured at the miners to minimise the impact of overhead from communication with the controller. The tests were performed on four computers in a computing cluster. Each machine had a dual-core Opteron 175 processor and 2 GB of RAM, and ran Scientific Linux CERN 4.5. The computers were connected by a local switched network allowing communication speeds up to 1 gigabit per second.

As one would expect, the initial profiling results showed that network roundtrip time has significant impact on the performance. Consequently, it is advantageous to execute many operations in parallel and thus the use of vectorised operations can lead to significant performance gains. The latter is a promising result, since many data mining algorithms are based on highly parallelisable matrix operations.

Nevertheless, we also observed that sometimes data vectors become too large and this starts to hinder the performance of the networking layer. To balance the effects of vectorisation, we implemented a load balancing system. We fixed a certain threshold vector size after which the miners start batch processing of large vectorised queries. In each sub-round, a miner processes a fragment of its inputs and sends the results to the other miners before continuing with the next fragment of input data.

Fig. 6 shows the impact of our optimisations on the waiting time caused by network delays. In particular, note that the impact of network delays is small during scalar product computation—the miners do not waste too many CPU cycles while waiting for inputs. Consequently, further optimisations can only lead to marginal improvements. The same is true for the multiplication protocol, since the performance characteristics

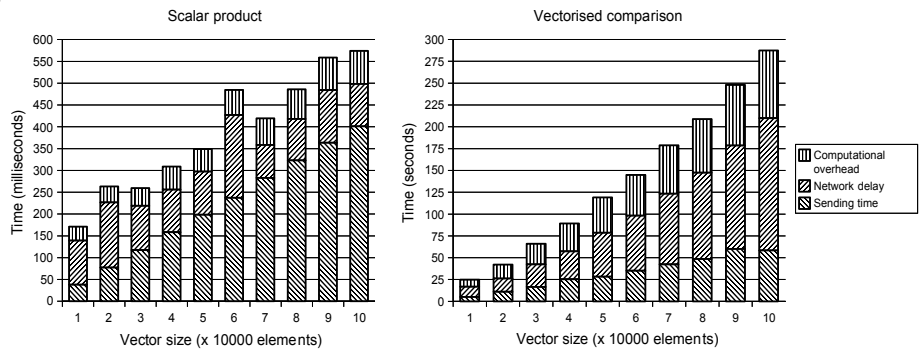


Fig. 6. Performance of the SHAREMIND framework. Left and right pane depict average running times for test vectors with $10,000$–$100,000$ elements in $10,000$-element increments.

of the scalar product operation practically coincide with the multiplication protocol: addition as a local operation is very fast. For the parallel comparison, the effect of network delays is more important and further scheduling optimisations may decrease the time wasted while waiting for messages. In both benchmarks, the time required to send and receive messages is significant and thus the efficiency of networking layer can significantly influence performance results.

Besides measuring the average running time, we also considered variability of timings. For the comparison protocol, the running times were rather stable. The average standard deviation was approximately 6% from the average running time. The scalar computation execution time was significantly more fluctuating, as the average standard deviation over all experiments was 24% of the mean. As most of the variation was in the network delay component of the timings, the fluctuations can be attributed to low-level tasks of the operating system. This is further confirmed by the fact that all scalar product timings are small, so even relatively small delays made an impact on our execution time. We remind here that the benchmark characterises near-ideal behaviour of the SHAREMIND framework, since no network congestion occurred during the experiments and the physical distance between the computers was small. In practice, the effect of network delays and the variability of running times can be considerably larger.

We also compared the performance of SHAREMIND with other known implementations of privacy-preserving computations. Our first candidate was the FAIRPLAY system [16], which is a general framework for secure function evaluation with two parties that is based on garbled circuit evaluation. According to the authors, a single comparison operation for 32-bit integers takes 1.25 seconds. A single SHAREMIND comparison takes, on average, 500 milliseconds. If we take into account the improvements in hardware we can say that the performance is similar when evaluating single comparisons. The authors of FAIRPLAY noticed that parallel execution gives a speedup factor of up 2.72 times in a local network. Experiments with SHAREMIND have shown that parallel execution can increase execution up to 27 times. Hence, SHAREMIND can perform

parallel comparison more efficiently. The experimental scalar product implementation in [18] also works with two parties. However, due to the use of more expensive cryptographic primitives, it is slower than SHAREMIND even with precomputation. For example, computing the scalar product of two 100000-element binary vectors takes a minimum of 5 seconds without considering the time of precomputation.

The SCET system used in [6] is similar to SHAREMIND as it is also based on share computing. Although SCET supports more than three computational parties, our comparison is based on results with three parties. The authors have presented performance results for multiplication and comparison operations as fitted linear approximations. The approximated time for computing products of x inputs is $3x + 41$ milliseconds and the time for evaluating comparisons is $674x + 90$ milliseconds (including precomputation). The performance of SHAREMIND can not be easily linearly approximated, because for input sizes up to 5000 elements parallel execution increases performance significantly more than for inputs with more than 5000 elements. However, based on the presented approximations and our own results we claim that SHAREMIND achieves better performance with larger input vectors in both scalar product and vectorised comparison. A SHAREMIND multiplication takes, on the average, from 0.006 to 57 milliseconds, depending on the size of the vector. More precisely, multiplication takes less than 3 milliseconds for every input vector with more than 50 elements. The timings for comparison range from 3 milliseconds to about half a second which is significantly less than 674 milliseconds per operation.

7 Conclusion and Future Work

In this paper, we have proposed a novel approach for developing privacy-preserving applications. The SHAREMIND framework relies on secure multi-party computation, but it also introduces several new ideas for improving the efficiency of both the applications and their development process. The main theoretical contribution of the framework is a suite of computation protocols working over elements in the ring of 32-bit integers instead of standard finite fields.

We have also implemented a fully functional prototype of SHAREMIND and showed that it offers enhanced performance when compared to other similar frameworks. Besides that, SHAREMIND also has an easy to use application development interface allowing the programmer to concentrate on the implementation of data mining algorithms without worrying about the details of cryptographic protocols.

However, the current implementation has several restrictions. Most notably it can use only three computing parties and can deal with just one semi-honest adversary. Hence the main direction for future research is relaxing these restrictions by developing computational primitives for more than three parties. We will also need to study the possibilities for providing security guarantees against active adversaries. Another aspect needing further improvement is the application programmer's interface. A compiler from a higher-level language to our current assembly-like instruction set is definitely needed. Implementing and benchmarking a broad range of existing data-mining algorithms will remain the subject for further development as well.

References

1. Agrawal, D., Aggarwal, C.C.: On the design and quantification of privacy preserving data mining algorithms. In: Proc. of PODS 2001, pp. 247–255 (2001)
2. Agrawal, R., Srikant, R.: Privacy-preserving data mining. SIGMOD Rec. 29(2), 439–450 (2000)
3. Beaver, D.: Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. Journal of Cryptology 4(2), 75–122 (1991)
4. Ben-Or, M., Kelmer, B., Rabin, T.: Asynchronous secure computations with optimal resilience (extended abstract). In: Proc. of PODC 1994, pp. 183–192 (1994)
5. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. Cryptology ePrint Archive, Report 2008/289 (2008)
6. Bogetoft, P., Damgård, I., Jakobsen, T., Nielsen, K., Pagter, J., Toft, T.: A practical implementation of secure auctions based on multiparty integer computation. In: Di Crescenzo, G., Rubin, A. (eds.) FC 2006. LNCS, vol. 4107, pp. 142–147. Springer, Heidelberg (2006)
7. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology 13(1), 143–202 (2000)
8. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proc. of FOCS 2001, pp. 136–145 (2001)
9. Cramer, R., Fehr, S., Ishai, Y., Kushilevitz, E.: Efficient multi-party computation over rings. In: Proc. of EUROCRYPT 2003. LNCS, vol. 4107, pp. 596–613 (2003)
10. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (2006)
11. Dodis, Y., Micali, S.: Parallel reducibility for information-theoretically secure computation. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 74–92. Springer, Heidelberg (2000)
12. Du, W., Atallah, M.J.: Protocols for secure remote database access with approximate matching. In: ACMCCS 2000, Athens, Greece, November 1-4 (2000)
13. Evfimievski, A.V., Srikant, R., Agrawal, R., Gehrke, J.: Privacy preserving mining of association rules. In: Proc. of KDD 2002, pp. 217–228 (2002)
14. Hirt, M., Maurer, U.M.: Player simulation and general adversary structures in perfect multiparty computation. Journal of Cryptology 13(1), 31–60 (2000)
15. Lindell, Y., Pinkas, B.: A proof of Yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175 (2004)
16. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - secure two-party computation system. In: Proc. of USENIX Security Symposium, pp. 287–302 (2004)
17. The SHAREMIND project web page (2007), `http://sharemind.cs.ut.ee`
18. Yang, Z., Wright, R.N., Subramaniam, H.: Experimental analysis of a privacy-preserving scalar product protocol. Comput. Syst. Sci. Eng. 21(1) (2006)

II

Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. International Journal of Information Security 11(6), 403–418 (2012)

Int. J. Inf. Secur.
DOI 10.1007/s10207-012-0177-2

REGULAR CONTRIBUTION

High-performance secure multi-party computation for data mining applications

Dan Bogdanov · Margus Niitsoo · Tomas Toft · Jan Willemson

Abstract Secure multi-party computation (MPC) is a technique well suited for privacy-preserving data mining. Even with the recent progress in two-party computation techniques such as fully homomorphic encryption, general MPC remains relevant as it has shown promising performance metrics in real-world benchmarks. SHAREMIND is a secure multi-party computation framework designed with real-life efficiency in mind. It has been applied in several practical scenarios, and from these experiments, new requirements have been identified. Firstly, large datasets require more efficient protocols for standard operations such as multiplication and comparison. Secondly, the confidential processing of financial data requires the use of more complex primitives, including a secure division operation. This paper describes new protocols in the SHAREMIND model for secure multiplication, share conversion, equality, bit shift, bit extraction, and division. All the protocols are implemented and benchmarked, showing that the current approach provides remarkable speed improvements over the previous work. This is verified using real-world benchmarks for both operations and algorithms.

D. Bogdanov ( )· J. Willemson
Cybernetica, Ülikooli 2, 51003 Tartu, Estonia
e-mail: dan@cyber.ee

D. Bogdanov · M. Niitsoo
Institute of Computer Science, University of Tartu,
J. Liivi 2, 50409 Tartu, Estonia
e-mail: margus.niitsoo@ut.ee

T. Toft
Department of Computer Science, Aarhus University,
IT-Parken, Åbogade 34, 8200 Aarhus N, Denmark
e-mail: ttoft@cs.au.dk

J. Willemson
STACC, Ülikooli 2, 51003 Tartu, Estonia
e-mail: janwil@cyber.ee

Keywords Secure computation · Performance · Applications

1 Introduction

The aim of secure multi-party computation (MPC) is to enable a number of networked parties to carry out distributed computing tasks on private information. During the computations, no one party (and, more generally, no certain subsets of the parties) should be able to learn any information about any other party's input other than what can be inferred from the output.

The theory behind MPC is fairly well developed, but practical solutions based on it are lagging considerably behind. In the last few years, several MPC frameworks based on various techniques have been proposed and implemented, for example FairPlayMP [1], VIFF [12], SEPIA [6], SecureSCM [18], VMcrypt [15], TASTY [14] and SHAREMIND [2].

Existing frameworks for MPC can broadly be split into two-party and multi-party frameworks. The former requires computational security; hence, solutions are based on homomorphic, public-key encryption (HE), or garbled circuits (GC). TASTY combines the two, utilizing whichever is best for the immediate task at hand. Both solutions have drawbacks. HE is computationally expensive, while the GC approach requires that a circuit computing the function is generated. For large-scale problems—such as data mining—even simply generating and storing the garbled circuit may be challenging. VMcrypt avoids this issue by generating and evaluating the circuit on the fly.

Multi-party frameworks are generally more efficient, as the computational primitives for information theoretic solutions are simpler than those of two-party computation. For example, addition and multiplication in a ring are more

Published online: 09 September 2012

efficient than public key or even symmetric key cryptography. Of the multi-party frameworks, FairPlayMP's circuit-based approach suffers from the issues with large-scale applications as discussed above. The passively secure VIFF and SEPIA both use Shamir's secret sharing scheme to implement MPC; both are more general than SHAREMIND, as they allow an arbitrary number of parties. while SHAREMIND is limited to three.

The first published application of MPC was the Danisco sugar beet auction held in 2008 [5]. Since then, the practical feasibility of MPC has also been shown for network anomaly detection [6] and joint analysis of financial data [4]. However, neither of the presented applications required MPC protocols to process large databases. Nevertheless, it is clear that many important areas of human endeavor, such as biomedical research and business data aggregation, do require such a capacity, often working with databases with thousands or even millions of records.

This research was mainly motivated by the need to build an MPC application suitable for the statistical analysis of financial data from competing companies, who are interested in finding the common trends in the entire sector. Several trend indicators require more complicated computational primitives, for example, private division to find out different ratios (like turnaround to personnel size). This is impossible on many of the current systems (such as those described in [5,6]) that use only relatively simple primitive operations like multiplication and comparison of two numbers.

The primary target of this paper is to develop a set of MPC protocols with high real-life performance on large databases. We will present new primitives for secure multiplication, share conversion, equality, bit shift, bit extraction, and division (both by public and private divisor).

The presented protocols have been implemented as improvements to the SHAREMIND framework [2,3]. In addition to the theoretical round and communication complexities, we also present benchmark results illustrating the achieved performance improvements.

2 Preliminaries

As stated above, SHAREMIND [2,3] is a secure multi-party computation system operating on additively secret-shared values. Although general m-party protocols can be devised for such a setting (see [2]), this paper concentrates on the case with three computing parties identified as $\mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{P}_3$. Designing and implementing protocols with a specific scenario in mind allows significant efficiency gains over general protocols; such optimizations can make the difference as to whether a problem can be solved or not.

The SHAREMIND framework uses additive secret sharing over the finite ring $\mathbb{Z}_{2^n}$. Secret-shared value $u \in \mathbb{Z}_{2^n}$ is represented as a triple $[\![u]\!] = (u_1, u_2, u_3)$, with the element u_i held by party $\mathcal{P}_i$ $(i = 1, 2, 3)$ and $u_1 + u_2 + u_3 \equiv u \bmod 2^n$. In the current implementation, $n = 32$ and this value is used for the performance benchmarks. However, it is stressed that all the protocols presented here work for any choice of n.

Let $\oplus$, $\vee$, and $\wedge$ denote the bitwise XOR, OR, and AND operations, respectively. Many of the protocols in this paper make use of these operations over the shared bits. In these cases, it is convenient to think of a value $u \in \mathbb{Z}_{2^n}$ as a bit vector $\overline{u} \in (\mathbb{Z}_2)^n$. We stress that both u and $\overline{u}$ represent the same value and the difference is only to denote whether it is used bitwise or as an integer, so $\overline{(\cdot)}$ is best thought of as a typing indicator. Hence, in general, we have $\overline{u} = \overline{u_1 + u_2 + u_3} \neq \overline{u_1} \oplus \overline{u_2} \oplus \overline{u_3}$.

Bit-level protocols also make use of vectors that are shared bitwise. We thus introduce special notation $\overline{[\![u]\!]} = (\overline{u}_1, \overline{u}_2, \overline{u}_3)$ to refer to such a bitwise sharing that $\overline{u} = \overline{u}_1 \oplus \overline{u}_2 \oplus \overline{u}_3$. This is a fairly natural extension of the notation.

To allow elementwise access to the bit vectors, we will let $\overline{u}^{(j)}$ stand for the jth bit of $\overline{u}$. We will also use notation $\overline{[\![u]\!]}^{(j)}$ to denote the share tuple $(\overline{u}_1^{(j)}, \overline{u}_2^{(j)}, \overline{u}_3^{(j)})$, so that $\overline{u}^{(j)} = \overline{u}_1^{(j)} \oplus \overline{u}_2^{(j)} \oplus \overline{u}_3^{(j)}$.

3 Proving security of Sharemind protocols

The security proofs in this paper are presented in the universal composability framework of Canetti [7]. To be precise, we assume that we have three distinct computational entities $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ all of which have an ideally secure authenticated channel to the two others. Security is proven in the passive (honest-but-curious) model in which the adversary is allowed to corrupt at most one of the three parties before the execution of the protocol. The adversary is then handed both the inputs and all the incoming messages of the corrupted party ("curiosity"), but he has no control over its outputs, which are assumed to be chosen as specified in the protocols ("honesty"). This model roughly corresponds to the real-world situation where we assume the protocol implementations are fairly hard to tamper with, whereas their inputs and outputs could be eavesdropped on—which is a sensible assumption for most practical purposes.

In the following, we will use the following definition:

Definition 1 We say that a share computing protocol is *perfectly simulatable* if there exists an efficient universal non-rewinding simulator $\mathcal{S}$ that can simulate all protocol messages to any real-world adversary $\mathcal{A}$ so that for all input shares, the output distributions of $\mathcal{A}$ and $\mathcal{S}(\mathcal{A})$ coincide.

To prove that a protocol is universally composable, it suffices to show that it is perfectly simulatable and that the outputs are independent of the inputs (Lemma 2 of [2]).

In order to prove perfect simulatability, we consider the incoming views of all the computing parties and prove that

they are independent of the input shares of the other parties, hence proving existence of the simulator. We will use the sequences-of-games formalism in our proofs. Denote the distribution of the incoming view G as $(\!|G|\!)$ and let the original incoming view of the party $\mathcal{P}$ be G_0. Then, we are interested in finding a sequence $G_0, G_1, \ldots, G_n$ such that

$$(\!|G_0|\!) = (\!|G_1|\!) = \ldots = (\!|G_n|\!)$$

and that the view G_n does not contain any references to input shares of the other parties.

The main tool that allows us to construct such sequences is the following simple folk lemma.

Lemma 1 *Let the incoming view G contain incoming messages $a_1 \pm r, a_2, \ldots, a_k$ where a_1, r are elements of finite additive group $\mathcal{A}$ and where r is a uniformly random element of $\mathcal{A}$, independent from all a_i. Then,*

$$(\!|G|\!) = (\!|G[a_1 \pm r/r]|\!),$$

where $G[a_1 \pm r/r]$ denotes the game, where the occurrence of $a_1 \pm r$ has been replaced by r.

Proof If $r \in \mathcal{A}$ is uniformly distributed and independent from all a_i, then so is $r \pm a_1$ since $f_r(x) := r \pm x$ is a bijective mapping for $\mathcal{A}$. □

In the following security proofs, this lemma will be used for various groups, including $\mathbb{Z}_2$, $\mathbb{Z}_{2^n}$, and $(\mathbb{Z}_2)^n$.

The lemma is often used in combination with Lemma 1 of [2], which states that a concurrent composition of perfectly simulatable protocols is also perfectly simulatable, which allows us to use already defined perfectly simulatable protocols as subroutines without analyzing their internal messages.

All of the protocols described in the following are fully simulatable. To achieve full security in the universal composability framework, one extra step is needed to also guarantee the independence of the protocol outputs from its inputs. This resharing step was already described in [2] and is brought here for completeness as Algorithm 1. The input shared value $[\![u]\!]$ is reshared as $[\![w]\!]$ so that $u = w$, all shares w_i are uniformly distributed and u_i and w_j are independent for all i, j. This is accomplished by masking the input shares with values that are randomly generated, but shared by only two of the three parties. Sharing the random values between two parties requires one round of communication, but since the values being sent are independent of the inputs, it can generally be done in parallel with the last round of the protocol on whose outputs it is to be applied.

Theorem 1 *Algorithm* 1 *is correct.*

Proof It is easy to see that

$$\begin{aligned} w &= w_1 + w_2 + w_3 \\ &= u_1 + r_{12} - r_{31} + u_2 + r_{23} - r_{12} + u_3 + r_{31} - r_{23} \\ &= u_1 + u_2 + u_3 = u. \end{aligned}$$

Algorithm 1: Resharing protocol $[\![w]\!] \leftarrow \mathsf{Reshare}([\![u]\!])$.

Data: Shared value $[\![u]\!]$.
Result: Shared value $[\![w]\!]$ such that $w = u$, all shares w_i are uniformly distributed and u_i and w_j are independent for $i, j = 1, 2, 3$.

1. $\mathcal{P}_1$ generates random $r_{12} \leftarrow \mathbb{Z}_{2^n}$.
2. $\mathcal{P}_2$ generates random $r_{23} \leftarrow \mathbb{Z}_{2^n}$.
3. $\mathcal{P}_3$ generates random $r_{31} \leftarrow \mathbb{Z}_{2^n}$.
4. All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
5. $\mathcal{P}_1$ computes $w_1 \leftarrow u_1 + r_{12} - r_{31}$.
6. $\mathcal{P}_2$ computes $w_2 \leftarrow u_2 + r_{23} - r_{12}$.
7. $\mathcal{P}_3$ computes $w_3 \leftarrow u_3 + r_{31} - r_{23}$.
8. Return $[\![w]\!]$.

Algorithm 2: Protocol for multiplying two shared values $[\![w']\!] \leftarrow \mathsf{Mult}([\![u]\!], [\![v]\!])$.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shared value $[\![w']\!]$ such that $w' = uv$.

1. $[\![u']\!] \leftarrow \mathsf{Reshare}([\![u]\!])$
2. $[\![v']\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
3. $\mathcal{P}_1$ sends u'_1 and v'_1 to $\mathcal{P}_2$.
4. $\mathcal{P}_2$ sends u'_2 and v'_2 to $\mathcal{P}_3$.
5. $\mathcal{P}_3$ sends u'_3 and v'_3 to $\mathcal{P}_1$.
6. $\mathcal{P}_1$ computes $w_1 \leftarrow u'_1 v'_1 + u'_1 v'_3 + u'_3 v'_1$.
7. $\mathcal{P}_2$ computes $w_2 \leftarrow u'_2 v'_2 + u'_2 v'_1 + u'_1 v'_2$.
8. $\mathcal{P}_3$ computes $w_3 \leftarrow u'_3 v'_3 + u'_3 v'_2 + u'_2 v'_3$.
9. Return $[\![w']\!] \leftarrow \mathsf{Reshare}([\![w]\!])$.

Proofs of independence can be given exactly as in Lemma 1, since all the elements w_i are of the form $u_j + r - s$ for randomly generated elements r, s. □

We stress that for practical applications, this step needs to be added to the end of all the protocols that are available to the end user for use. However, for all the intermediate steps, perfect simulatability is enough, which is why we omit the resharing step from the descriptions of the protocols in this paper.

4 Multiplication protocol

Instead of using the Du-Atallah multiplication protocol as in [2] and [3], we propose a new protocol which is based on the following observation. If we have two values u and v shared additively as $u = u_1 + u_2 + u_3$ and $v = v_1 + v_2 + v_3$, their product is $uv = \sum_{i=1}^{3} \sum_{j=1}^{3} u_i v_j$. The addends of the form $u_i v_i$ can be computed locally by party $\mathcal{P}_i$. In order to find an addend of the form $u_i v_j$ ($i \neq j$), the share u_i can be sent from $\mathcal{P}_i$ to $\mathcal{P}_j$ (or the share v_j from $\mathcal{P}_j$ to $\mathcal{P}_i$). Knowing the shares u_i and u_j, party $\mathcal{P}_j$ is still unable to get any information concerning u, but in order to obtain universal composability, all the shares u_i and v_j still need to be reshared.

The new multiplication protocol is presented in Algorithm 2.

Theorem 2 *Algorithm* 2 *is correct and secure against a passive attacker.*

Proof For correctness, we note that

$$\begin{aligned} w &= w_1 + w_2 + w_3 = u'_1 v'_1 + u'_1 v'_3 + u'_3 v'_1 \\ &\quad + u'_2 v'_2 + u'_2 v'_1 + u'_1 v'_2 + u'_3 v'_3 + u'_3 v'_2 + u'_2 v'_3 \\ &= (u'_1 + u'_2 + u'_3)(v'_1 + v'_2 + v'_3) \\ &= (u_1 + u_2 + u_3)(v_1 + v_2 + v_3) \\ &= uv. \end{aligned}$$

To prove security, we note that Algorithm 2 is symmetric for all the parties. Thus, it will be enough to consider just the incoming view of $\mathcal{P}_1$, which consists of just two values u'_3 and v'_3. Both incoming values are uniformly distributed and independent of the private inputs other than u_1, v_1 due to Lemma 1. Therefore, the protocol is secure since we can build a perfect simulator by generating uniformly distributed values. □

We will use the standard arithmetic shorthand and write $[\![w]\!] \leftarrow [\![u]\!] \cdot [\![v]\!]$ to mean $[\![w]\!] \leftarrow \mathsf{Mult}([\![u]\!], [\![v]\!])$. We note that this protocol works over any ring, so we can also use it for $\wedge$ operation for shares from $\mathbb{Z}_2$. In that case, we will also use a shorthand notation to write $[\![u]\!] \wedge [\![v]\!]$ instead of calling the multiplication protocol. We stress, however, that while $+$ and $\oplus$ require only local addition of the shares and are thus essentially free, both $\cdot$ and $\wedge$ *require communication* and are hence considerably more costly.

Even though the description of the multiplication protocol involves two rounds of sending different values between the computing parties, the resharing round can be carried out as precomputation since it is independent of inputs $[\![u]\!]$ and $[\![v]\!]$. Hence, the real overhead of multiplication is just one round. It is actually possible to combine the two rounds of communication into a single round, but this provided for no noticeable increase in performance in implementation so we omit the details.

5 Bit-level protocols

Most of the high-level protocols presented in Sects. 6 and 7 depend on low-level bit operations. Since the SHAREMIND virtual machine operates on values shared additively over $\mathbb{Z}_{2^n}$, accessing the bits of the shared value is a non-trivial problem.

The first step for all the protocols in the current section is to consider the shares $u_1, u_2, u_3 \in \mathbb{Z}_{2^n}$ as the elements $\overline{u_1}, \overline{u_2}, \overline{u_3}$ of the ring $(\mathbb{Z}_2)^n$ and carry out all the operations bitwise. Recall that the value $\overline{[\![u]\!]}$ represented by the shares $\overline{u}_1, \overline{u}_2, \overline{u}_3$ is, generally speaking, not equal to u when converted back to $\mathbb{Z}_{2^n}$, since it does not take into account the carry bits that occur during addition.

Algorithm 3: $\overline{[\![p']\!]} \leftarrow \mathsf{PrefixOR}(\overline{[\![p]\!]})$.

Data: Bitwise shared vector $\overline{[\![p]\!]}$.
Result: The vector $\overline{[\![p']\!]}$ which has the form $00\ldots011\ldots1$, where the initial part $00\ldots01$ coincides with the vector originally represented by $\overline{[\![p]\!]}$.

```
l ← |[[p]]|.
if l = 1 then
    Return [[p']] ← [[p]].
else
    [[p']]^(l−1...⌊l/2⌋) ← PrefixOR([[p]]^(l−1...⌊l/2⌋)).
    [[p']]^(⌊l/2⌋−1...0) ← PrefixOR([[p]]^(⌊l/2⌋−1...0)).
    for i ← 0 to ⌊l/2⌋ − 1 do .
        [[p']]^(i) ← [[p']]^(i) ∨ [[p']]^(⌊l/2⌋).
    Return [[p']].
end
```

The bit-level protocols used in SHAREMIND utilize the basic principles of digital circuit design [16] and build on the general bit extraction framework proposed by Damgård et al. [8].

One elementary protocol used is $[\![b]\!] \leftarrow \mathsf{BitConj}(\overline{[\![u]\!]})$ for finding the conjunction of all the bits of a bitwise shared vector $\overline{[\![u]\!]}$ and representing the result as a shared bit $[\![b]\!]$. This protocol can be implemented using a natural recursive split-in-half approach and achieves round complexity logarithmic in the input vector length.

Another protocol we need is $\overline{[\![s]\!]} \leftarrow \mathsf{CarryBits}(\overline{v}, \overline{[\![r]\!]})$, where the value $\overline{v} \in \mathbb{Z}_{2^n}$ is known by $\mathcal{P}_2$ and $\mathcal{P}_3$, the value r is shared bitwise over $(\mathbb{Z}_2)^n$ between $\mathcal{P}_2, \mathcal{P}_3$ (so $\overline{r}_1 = 0$), and the output is a shared vector $\overline{[\![s]\!]}$ representing the carries occurring when the addition $v + r$ is performed. The protocol can be implemented exactly as in [3] using carry look-ahead technique that also works in a logarithmic number of rounds.

Whenever a bit-level protocol needs to $\wedge$ together two bit values, the (perfectly simulatable) protocol $\mathsf{Mult}(\cdot, \cdot)$ can be called. The security proofs for protocols $\mathsf{CarryBits}(\cdot, \cdot)$ and $\mathsf{BitConj}(\cdot)$ are standard applications of universal composability Lemmas 1 and 2 of [2] and we will skip them here.

As an example of a more involved bit-level protocol, we will present and analyze the protocol for finding the most significant nonzero bit position of a bitwise shared value $\overline{[\![u]\!]}$.

We will proceed in two steps. First, we set all the bits after the first 1 to be 1 as well by a recursive prefix-OR procedure presented as Algorithm 3. In the procedure, let $|\overline{[\![p]\!]}|$ denote the number of bits that the vector represented by $\overline{[\![p]\!]}$ contains. Also, let $\overline{[\![p]\!]}^{(i\ldots j)}$ denote the shared subvector containing the bits represented by $\overline{[\![p]\!]}^{(i)}, \ldots, \overline{[\![p]\!]}^{(j)}$ (note that we will write more significant bits to the left, so in this notation $i \geq j$).

Algorithm 4: Protocol for the most significant nonzero bit position $\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![u]\!]})$.

Data: Bitwise shared value $\overline{[\![u]\!]}$.
Result: Shared vector $\overline{[\![s]\!]}$ such that $\overline{[\![s]\!]}^{(j)}$ represents 1, where j is the most significant position, where $\overline{[\![u]\!]}^{(j)}$ represents 1, and 0 otherwise. If all the bits of $\overline{[\![u]\!]}$ represent 0, all the shared bits of $\overline{[\![s]\!]}$ represent 0 as well.

1 $\overline{[\![u']\!]} \leftarrow \mathsf{PrefixOR}(\overline{[\![u]\!]})$.
2 **for** $i \leftarrow 0$ **to** $n-2$ **do**
3 $\quad \overline{[\![s]\!]}^{(i)} \leftarrow \overline{[\![u']\!]}^{(i)} \oplus \overline{[\![u']\!]}^{(i+1)}$.
4 $\overline{[\![s]\!]}^{(n-1)} \leftarrow \overline{[\![u']\!]}^{(n-1)}$.
5 Return $\overline{[\![s]\!]}$.

Note that the $\log_2 n$ recursive calls of Algorithm 3 require one round of multiplication each (to compute $\vee$ of the shared bits $[\![b_1]\!]$ and $[\![b_2]\!]$ as $[\![b_1]\!] \oplus [\![b_2]\!] \oplus ([\![b_1]\!] \wedge [\![b_2]\!])$), hence the overall round complexity of Algorithm 3 is $\log_2 n$.

To compute the most significant bit, it now suffices to zero all the bits that are to the right of the first 1. This can be done by a simple loop given on the lines 2 to 3 of the main protocol described as Algorithm 4. As this computation is local and requires no communication, the complexity is the same as for Algorithm 3.

Theorem 3 *Algorithm* 4 *is correct and secure against one passive attacker.*

Proof Correctness of the protocol follows directly from the discussion given above.

Security of the protocol is trivial as well, since we are only composing perfectly simulatable primitives. □

6 Improved high-level protocols

We now describe the new and improved high-level protocols for the framework: the operators for share conversion, bit extraction, and equality. We have also developed protocols for bit shift under a public shift, which are presented in Appendix 1.

6.1 Share conversion

Bit-level operations are typically used as building blocks within algorithms working with the full shares over $\mathbb{Z}_{2^n}$. Hence, the problem of converting the bits shared over $\mathbb{Z}_2$ to shares over $\mathbb{Z}_{2^n}$ arises. Converting individual shares locally does not solve the problem, since we will lose reduction modulo 2. Hence, a different approach is needed.

The routine presented as Algorithm 5 first splits the bit u as $u = m \oplus s$ so that $m = b \oplus u_1$ for a random bit b. The bit m can then be directly converted to $\mathbb{Z}_{2^n}$ by one party and

Algorithm 5: Protocol $[\![v]\!] \leftarrow \mathsf{ShareConv}([\![u]\!])$ for converting a share $[\![u]\!] \in \mathbb{Z}_2$ to $[\![v]\!] \in \mathbb{Z}_{2^n}$.

Data: Shared value $[\![u]\!]$ in bit shares.
Result: Shared value $[\![v]\!]$ such that $u = v$ and $[\![v]\!]$ is shared in $\mathbb{Z}_{2^n}$.

1 $\mathcal{P}_1$ generates random $b \leftarrow \mathbb{Z}_2$ and sets $m \leftarrow b \oplus u_1$.
2 $\mathcal{P}_1$ locally converts m to $\mathbb{Z}_{2^n}$, generates random $m_{12} \leftarrow \mathbb{Z}_{2^n}$ and computes $m_{13} = m - m_{12}$.
3 $\mathcal{P}_1$ generates random $b_{12} \leftarrow \mathbb{Z}_2$ and computes $b_{13} = b - b_{12} = b \oplus b_{12}$.
4 All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
5 $\mathcal{P}_2$ sets $s_{23} \leftarrow b_{12} \oplus u_2$.
6 $\mathcal{P}_3$ sets $s_{32} \leftarrow b_{13} \oplus u_3$.
7 All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
8 $\mathcal{P}_1$ sets $v_1 \leftarrow 0$
9 $\mathcal{P}_2$ and $\mathcal{P}_3$ set $s \leftarrow s_{23} \oplus s_{32}$
10 **if** $s = 1$ **then**
11 $\quad \mathcal{P}_2$ sets $v_2 \leftarrow (1 - m_{12})$.
12 $\quad \mathcal{P}_3$ sets $v_3 \leftarrow (-m_{13})$.
13 **else**
14 $\quad \mathcal{P}_2$ sets $v_2 \leftarrow m_{12}$.
15 $\quad \mathcal{P}_3$ sets $v_3 \leftarrow m_{13}$.
16 **end**
17 Return $[\![v]\!]$.

the value of s can be used to select whether the real value of u should be $1 - m$ or m.

Theorem 4 *Algorithm* 5 *is correct and secure against one passive attacker.*

Proof For correctness, we first note that

$$u = u_1 \oplus u_2 \oplus u_3 = b \oplus m \oplus b_{12} \oplus s_{23} \oplus b_{13} \oplus s_{32} = m \oplus s.$$

Hence, if $s = 1$, we have $v = v_1 + v_2 + v_3 = 1 - m_{12} - m_{13} = 1 - m$, which is equal to $m \oplus 1$ when embedded to $\mathbb{Z}_{2^n}$. If $s = 0$, we have $v = v_1 + v_2 + v_3 = m_{12} + m_{13} = m$, which is equal to $m \oplus 0$ when embedded to $\mathbb{Z}_{2^n}$.

To prove security, we will consider all the three computing parties and prove that their incoming views can be perfectly simulated. The view of party $\mathcal{P}_1$ contains no incoming messages, so the corresponding simulator is trivial. The incoming view of $\mathcal{P}_2$ can be perfectly simulated, since using Lemma 1 we see that its distribution

$$(\!|m_{12}, b_{12}, b \oplus b_{12} \oplus u_3|\!) = (\!|m_{12}, b_{12}, b|\!)$$

is independent of private inputs other than u_2.

Similarly, the incoming view of $\mathcal{P}_3$ can be perfectly simulated, since using Lemma 1 we see that its distribution

$$\begin{aligned}
&(\!|b \oplus u_1 - m_{12}, b \oplus b_{12}, b_{12} \oplus u_2|\!) \\
&\quad = (\!|m_{12}, b \oplus b_{12}, b_{12} \oplus u_2|\!) \\
&\quad = (\!|m_{12}, b, b_{12} \oplus u_2|\!) \\
&\quad = (\!|m_{12}, b, b_{12}|\!)
\end{aligned}$$

is independent of private inputs other than u_3. Furthermore, the values m_{12}, b_{12}, and b are uniformly distributed so we can build a perfect simulator for $\mathcal{P}_2$ and $\mathcal{P}_3$ and show security. □

Algorithm 6: Protocol $\overline{[\![w]\!]} \leftarrow \mathsf{BitExtr}([\![u]\!])$ for bit extraction.

Data: Additively shared value $[\![u]\!]$.
Result: Bitwise shared vector $\overline{[\![w]\!]}$ representing the bits of u.
1 $\mathcal{P}_1$ generates random $r, r_2 \leftarrow \mathbb{Z}_{2^n}, \overline{q}_2 \leftarrow (\mathbb{Z}_2)^n$, sets $\overline{q}_1 = \overline{0}$ and computes $r_3 \leftarrow u_1 - r - r_2, \overline{q}_3 \leftarrow \overline{r} \oplus \overline{q}_2$.
2 $\mathcal{P}_1$ sends $r_i, \overline{q}_i$ to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ $(i = 2, 3)$ computes the share $v_i \leftarrow u_i + r_i$ and sends it to $\mathcal{P}_{6/i}$.
4 $\mathcal{P}_2, \mathcal{P}_3$ compute $v = v_2 + v_3$.
5 $\overline{[\![s]\!]} \leftarrow \mathsf{CarryBits}(\overline{v}, \overline{[\![q]\!]})$.
6 Define $s_i^{(-1)} = 0, (i = 1, 2, 3)$.
7 **for** $j \leftarrow 0$ **to** $n - 1$ **do**
8 In $\mathcal{P}_1$: $\overline{w}_1^{(j)} \leftarrow \overline{s}_1^{(j-1)}$.
9 In $\mathcal{P}_2$: $\overline{w}_2^{(j)} \leftarrow \overline{v}^{(j)} \oplus \overline{q}_2^{(j)} \oplus \overline{s}_2^{(j-1)}$.
10 In $\mathcal{P}_3$: $\overline{w}_3^{(j)} \leftarrow \overline{q}_3^{(j)} \oplus \overline{s}_3^{(j-1)}$.
11 Return $\overline{[\![w]\!]}$.

6.2 Bit extraction

In order to perform bit-level computations, we first need to extract the bits, which is a non-trivial task for shared values. The basic working principle of Algorithm 6 is the same as of the bitwise addition protocol explained in [3]. The initial value u is represented as the sum $v + r$, where r is a random value with a known shared bitwise decomposition. We can then use the carry look-ahead algorithm to determine the carry bits that occur in the addition $v + r$ and use them to compute the bits of u.

Theorem 5 *Algorithm* 6 *is correct and secure against one passive attacker.*

Proof During the initial stage, u is represented as

$$\begin{aligned} u &= u_1 + u_2 + u_3 = (r + r_2 + r_3) + (v_2 - r_2) + (v_3 - r_3) \\ &= v_2 + v_3 + r = v + r, \end{aligned}$$

where r has a known shared bit decomposition $r = \overline{q}_2 \oplus \overline{q}_3$. Thus, in order to find the bits of u, we can use bitwise addition to compute the bits of $v + r$. To do that, one needs to compute the carry bits, and this is done by calling the Algorithm $\mathsf{CarryBits}(\cdot, \cdot)$ (laid out in [3]).

As a result, the bitwise shared vector $\overline{[\![s]\!]}$ will represent exactly the carry bits from the corresponding positions when computing $v + r$, hence it remains to add these bits to the shared bitwise $\oplus$ of v and r, which is done on lines 6 to 10. Indeed, we see that

$$\begin{aligned} \overline{w}^{(j)} &= \overline{w}_1^{(j)} \oplus \overline{w}_2^{(j)} \oplus \overline{w}_1^{(j)} \\ &= \overline{s}_1^{(j-1)} \oplus \overline{v}^{(j)} \oplus \overline{q}_2^{(j)} \oplus \overline{s}_2^{(j-1)} \oplus \overline{q}_3^{(j)} \oplus \overline{s}_3^{(j-1)} \\ &= \overline{v}^{(j)} \oplus \overline{r}^{(j)} \oplus \overline{s}^{(j-1)}. \end{aligned}$$

To prove security, we will consider all three computing parties and prove that their incoming views can be perfectly simulated. The incoming view of party $\mathcal{P}_1$ contains no other incoming messages than the ones determined by Algorithm $\mathsf{CarryBits}(\cdot, \cdot)$, which can be perfectly simulated. The incoming view of $\mathcal{P}_2$ looks mostly the same as that of $\mathcal{P}_1$, only the initial part differs. We use Lemma 1 again to see that

$$(\!|r_2, \overline{q}_2, u_3 + r_3, \ldots|\!) = (\!|r_2, \overline{q}_2, r_3, \ldots|\!),$$

which does not depend on any input values and can hence be perfectly simulated.

Similarly, for party $\mathcal{P}_3$, we have the initial part of the incoming view

$$\begin{aligned} &(\!|u_1 - r - r_2, \overline{r} \oplus \overline{q}_2, u_2 + r_2, \ldots|\!) \\ &\quad = (\!|u_1 - r - r_2, \overline{q}_2, u_2 + r_2, \ldots|\!) \\ &\quad = (\!|r, \overline{q}_2, u_2 + r_2, \ldots|\!) \\ &\quad = (\!|r, \overline{q}_2, r_2, \ldots|\!), \end{aligned}$$

which does not depend on any input values once again. □

We note that this protocol can also be used to improve the efficiency of comparison protocols, which usually use bit extraction as a subroutine [3].

Algorithm 7: Protocol $[\![w]\!] \leftarrow \mathsf{Equal}([\![u]\!], [\![v]\!])$ for evaluating the equality predicate.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shared value $[\![w]\!]$ such that $w = 1$ if $u = v$, and 0 otherwise.
1 $\mathcal{P}_1$ generates random $r_2 \leftarrow \mathbb{Z}_{2^n}$ and computes $r_3 \leftarrow (u_1 - v_1) - r_2$.
2 $\mathcal{P}_1$ sends r_i to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ computes $e_i = (u_i - v_i) + r_i$ $(i = 2, 3)$.
4 $\mathcal{P}_1$ sets $\overline{p}_1 \leftarrow 2^n - 1 = 111 \ldots 1$.
5 $\mathcal{P}_2$ sets $\overline{p}_2 \leftarrow e_2$.
6 $\mathcal{P}_3$ sets $\overline{p}_3 \leftarrow (0 - e_3)$.
7 Return $[\![w]\!] \leftarrow \mathsf{BitConj}(\overline{[\![p]\!]})$.

6.3 Equality testing

Equality testing can be accomplished fairly easily via bit extraction. However, since equality comparison is used quite often in practical applications, it makes sense to provide a separate and more efficient protocol specifically designed for that task. Algorithm 7 first shares the difference $u - v$ as $e_2 + e_3$ between $\mathcal{P}_2$ and $\mathcal{P}_3$. Then, it remains to determine whether $e_2 + e_3 = 0$, which can be done by comparing e_2 and $-e_3$ bitwise.

Theorem 6 *Algorithm* 7 *is correct and secure against one passive attacker.*

Proof For correctness, note first that

$$\begin{aligned} e_2 + e_3 &= (u_2 - v_2) + r_2 + (u_3 - v_3) + r_3 \\ &= (u_2 - v_2) + (u_3 - v_3) + (u_1 - v_1) \\ &= u - v, \end{aligned}$$

hence $u = v$ iff $e_2 = 0 - e_3$. Algorithm 7 compares u and v by comparing $\overline{p}_2 = e_2$ and $\overline{p}_3 = (0 - e_3)$ bitwise;

for that, we analyze the bitwise sum (XOR) of $\overline{p}_1 = 2^n - 1 = 111\ldots1$, $\overline{p}_2$ and $\overline{p}_3$. We see that $u = v$ iff all the bits represented $\overline{[\![p]\!]}$ are 1, which is exactly the case when the conjunction $[\![w]\!] = \wedge_{i=0}^{n-1}\overline{[\![p]\!]}^{(i)}$ is 1. This is exactly what is verified by calling Algorithm BitConj$(\cdot)$.

To prove security, we will consider all the three computing parties and prove that their incoming views can be perfectly simulated.

The incoming view of party $\mathcal{P}_1$ coincides with its view in Algorithm BitConj$(\cdot)$ and can hence be simulated. The incoming view of $\mathcal{P}_2$ is almost equivalent to that of $\mathcal{P}_1$ with the only exception of receiving one extra independently and uniformly chosen element r_2, which is trivial to simulate. The same holds for $\mathcal{P}_3$ who receives $r_3 = (u_1 - v_1) - r_2$, which can be replaced by r_2 by Lemma 1. □

7 Division protocols

In this paper, we introduce two division protocols—one where the divisor is a public constant and the other where the divisor is a shared value. The protocols make use of two subroutines ReshareToTwo$(\cdot)$ and Overflow$(\cdot)$ meant for resharing a value to two parties and for computing the overflow bit once, the values are shared in this way, respectively. As both protocols are fairly straightforward, the exact details for them are given in Appendix 1.

7.1 Division by a public value

The main idea for the division protocol comes from [13] and essentially consists of publicly finding the inverse $d' = 1/d$ of the divisor d and then multiplying the dividend a with the previously found inverse value d'. This trick reduces division to multiplication by a constant, which is often used on normal processors to speed up batch division of many numbers with a single value. In our case, however, it allows to do the division publicly and then only perform a secret multiplication, which is fairly efficient.

Since we have access to only integer arithmetic, it makes sense to denote the inverse value $d' \approx c2^{-k}$ where we choose k in such a way that c is an integer. It is shown in [13] that one can choose c and k in such a way that the final outcome w of the protocol is equal to $\lfloor \frac{u}{d} \rfloor$. All that is left to do is to compute the division by multiplying c and u and then shifting the result cu right k positions.

It is shown in [13] that if we are working with integers from $\mathbb{Z}_{2^n}$, it suffices to choose $k = n + 1$ and the problem can thus be reduced to just finding the highest n bits of the $2n + 1$ bit multiplication result.[1]

[1] In [13] the authors actually transform the problem so that it is enough to use just $2n$ bits. However, the transformation assumes that bit shifts are cheap, making it impractical in the current MPC setting.

Algorithm 8: Protocol $[\![w]\!] \leftarrow$ PubDiv$([\![u]\!], d)$ for division by a public value d.

Data: Shared value $[\![u]\!]$ and a public divisor d.
Result: Shares $[\![w]\!]$ of the value $w = \lfloor \frac{u}{d} \rfloor$.
1 $[\![u']\!] \leftarrow$ ReshareToTwo(u).
2 $\mathcal{P}_2, \mathcal{P}_3$ find $c \in \mathbb{Z}_{2^{n+1}}$ such that $c2^{-(n+1)} \approx \frac{1}{d}$ as in [13].
3 $\mathcal{P}_1$ sets $v_1^1, v_1^2 \leftarrow 0 \in \mathbb{Z}_{2^{2n+1}}$.
4 $\mathcal{P}_2$ sets $v_2^1 \leftarrow cu'_2$, $v_2^2 \leftarrow c(u'_2 - 2^n) \in \mathbb{Z}_{2^{2n+1}}$.
5 $\mathcal{P}_3$ sets $v_3^1, v_3^2 \leftarrow cu'_3 \in \mathbb{Z}_{2^{2n+1}}$.
6 $[\![\lambda]\!] \leftarrow$ Overflow$([\![u']\!])$.
7 $[\![\lambda^1]\!] \leftarrow$ Overflow$([\![v^1 \ll n]\!])$.
8 $[\![\lambda^2]\!] \leftarrow$ Overflow$([\![v^2 \ll n]\!])$.
9 Every $\mathcal{P}_i$ sets $w_i^1 \leftarrow v_i^1 \gg (n+1)$ and $w_i^2 \leftarrow v_i^2 \gg (n+1)$.
10 Return $[\![w]\!] \leftarrow (1 - [\![\lambda]\!])([\![w^1]\!] + [\![\lambda^1]\!]) + [\![\lambda]\!]([\![w^2]\!] + [\![\lambda^2]\!])$.

In our setting, multiplying two n bit values means that the higher bits are thrown away. To avoid that, both values would temporarily need to be converted to $2n + 1$-bit values, then multiplied and then have the result truncated back to n bit value.

Converting a secret n bit value $[\![u]\!]$ into a value of $m > n$ bits requires that we know whether the sum $u_1 + u_2 + u_3$ produces a carry into the higher order bits when viewed in $\mathbb{Z}_{2^m}$. We can use Algorithm 11 to obtain the carry bit. We can then carry out the multiplication and use Algorithm 11 again to perform the truncation.[2]

However, the crucial observation is that we can parallelize determining the carry bit and truncation—since there are just two different choices for the carry and we can just compute the results for both and only decide in the end which of the two to use. This approach leads to Algorithm 8.

Theorem 7 *Algorithm* 8 *is correct and secure against one passive attacker.*

Proof In order to compute $\frac{u}{d} = u \cdot \frac{1}{d}$, the algorithm first chooses c such that $\frac{1}{d} \approx c2^{-(n+1)}$ and then computes cu. After running Algorithm 10, we have either $u = u'_2 + u'_3$ or $u = u'_2 + u'_3 - 2^n$. Hence, the values

$$v^1 = v_1^1 + v_2^1 + v_3^1 = 0 + cu'_2 + cu'_3 = c(u'_2 + u'_3) \text{ and}$$
$$v^2 = v_1^2 + v_2^2 + v_3^2 = 0 + c(u'_2 - 2^n) + cu'_3$$
$$= c(u'_2 + u'_3 - 2^n)$$

are the two candidates for cu. Now, it remains to divide both of these values by 2^{n+1} and choose the correct one. The division is performed by right shift on line 9 and the correct value is chosen based on the value of the bit λ on line 10. Note that when performing the right shift, we may still need to add 1 to compensate for the carry we lose when truncating; this is achieved by running Algorithm 10 first to reshare the

[2] We do not need to use Algorithm 12 because we do not introduce new digits on the left like we would in the case of a normal bit shift.

values $[\![v^i]\!] \ll (n+1)$ and then Algorithm 11 in order to obtain known carries.

To prove security, we note that sending messages only occurs within subprotocols proven above to be perfectly simulatable, hence building the required simulator is trivial. □

7.2 Division by a shared value

In order to implement the protocol, we will use the Goldschmidt iteration method, which is an adaptation of Newton iteration designed especially for efficient implementation in digital computers. When dividing u by v, the algorithm keeps track of N and D so that $\frac{N}{D} = \frac{u}{v}$ but where $D \to 1$ as the number of iterations increases, which guarantees that $N \to \frac{u}{v}$ as well.

To be precise, the method works by starting with $N_0 = c_0 u$ and $D_0 = c_0 v$ where c_0 is a scaling constant designed to guarantee $0.5 \leq D_0 < 1$. In each step of the iteration, a scaling coefficient $F_i = 2 - D_{i-1}$ is computed after which the new values $D_i = F_i D_{i-1}$ and $N_i = F_i N_{i-1}$ can be calculated.

The Goldschmidt iteration method has many desirable properties. First, it was conceived with parallelism in mind, so that the two multiplications in each iteration can be done in parallel. Secondly, it has quadratic convergence, which means that if $0.5 \leq D_0 < 1$ then $1 - 2^{-2^i} \leq D_i < 1$ for all $i \geq 0$. Consequently, the relative error $\frac{u/v - N_i}{u/v} = 1 - D_i \in (0, 2^{-2^i}]$, implying that $\log_2 n$ iterations suffice for n bits of precision (see Appendix 2 for details). Thirdly, the convergence is monotonic, so that $N_0 < N_1 < \ldots < N_i < \ldots < \frac{u}{v}$. This becomes crucial when one is interested in division that always rounds in a fixed direction (i.e., either always upwards or downwards).

In practice, the method is most often used for floating-point division. However, analogously to the public division, it is pretty straightforward to convert everything to purely integer arithmetic by simply emulating fix-point arithmetic with corresponding integer operations followed by the appropriate bit shifts. The details of such an approach can be found in [17].

However, some interesting technical problems arise when attempting an efficient implementation of such an iteration method within SHAREMIND. The key difference between the standard model and our MPC setting is the cost of bit operations—they are virtually costless in the standard model, but extremely costly in SHAREMIND.

This causes the most problems for computing the initial scaling constant c_0, which is usually done by just finding the most significant bit position h_v of v and taking $c_0 = 2^{-h_v - 1}$. We will follow the same approach (combining Algorithms 6 and 4), but note that doing so is very costly—finding c_0 constitutes roughly one-third of the whole protocol in terms of both round and communication complexity. Nevertheless, there seems to be no obvious way around it as the iteration methods converge very slowly if an initial estimate of comparable quality is not used and the relevant literature does not seem to discuss any other methods of finding such an estimate.

A second problem arises when we note that emulating fractional multiplication requires an efficient shift right operator. However, this is again something that the current framework does not provide, as Algorithm 12 is rather costly. The same is true for modulus expansion, which is required to get the high bits of the multiplication result.

However, these problems can be solved fairly efficiently. Firstly, the ring is expanded to m bits just once before the iterations, and m is simply chosen large enough, so that no further expansions would be necessary. Since this can be done in parallel with finding c_0, doing so is essentially free in terms of rounds. Since exact truncation is also very expensive, we will replace it with an imprecise one where all the shares are truncated individually without worrying about the possible carry bit. This introduces additional imprecision into the computation, but that can be dealt with by slightly increasing the precision of the arithmetic from 2^n to $2^{n'}$ and adding an additional iteration. The details of the corresponding error analysis and the details of the choice of m and n' are presented in Appendix 2. Using these two tricks brings the cost of each iteration step down to just two parallel (large modulus) multiplications, making it relatively fast and efficient. Additional care needs to be taken to enforce strict downward rounding. As mentioned before, Goldschmidt iteration ensures monotonic convergence from below. This means that N will always be less than the real value $\frac{u}{v}$, which also means that generally we expect $\lfloor N \rfloor = \lfloor \frac{u}{v} \rfloor$ to hold. However, there are cases where we can get $\lfloor N \rfloor < \lfloor \frac{u}{v} \rfloor$. This can be easily fixed by adding a suitably small value Δ to N before truncation. The details of choosing Δ are presented in Appendix 2 along with the analysis of error terms introduced by imprecise truncation during the iteration steps.

To make convergence faster, we will alter the first iteration a bit by setting $F_1 = 2\sqrt{2} - 2D_0$, which is standard practice for Newton iteration, but somewhat less used in the case of Goldschmidt algorithm. Assuming $\sqrt{2} - 1 < D_0 < 1$, it is easy to see that $2\sqrt{2} - 2 < D_1 < 1$. Note that $\sqrt{2} - 1 \approx 0.41 < 0.5$, but $2\sqrt{2} - 2 \approx 0.83$, which gives us a better estimate compared to $1 - 2^{-2^1} = 0.75$ provided by the original first iteration of Goldschmidt method. This will guarantee sufficient extra precision to compensate for the additional errors introduced with truncation, so no extra iteration step is needed.

The protocol for division is formalized as Algorithm 9. In this protocol, we will be working with values from several

Algorithm 9: Protocol $[\![w]\!] \leftarrow \mathsf{Div}([\![u]\!], [\![v]\!])$ for division.

Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shares $[\![w]\!]$ such that $w = \lfloor \frac{u}{v} \rfloor$.

1 $\overline{[\![v']\!]} \leftarrow \mathsf{BitExtr}([\![v]\!])$.
2 $\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![v']\!]})$.
3 Compute $[\![c^i]\!] \leftarrow \mathsf{ShareConv}^{\mathsf{m}}(\overline{[\![s]\!]}^{(i)})$ $(i = 0, \ldots, n-1)$.
4 Set $[\![\widehat{c_0}]\!] \leftarrow \sum_{i=0}^{n-1} 2^{n'-i-1}[\![c^i]\!]$.
5 $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}([\![u]\!])$, $[\![\lambda^1]\!] \leftarrow \mathsf{Overflow}^{\mathsf{m}}([\![u']\!])$.
6 $[\![v']\!] \leftarrow \mathsf{ReshareToTwo}([\![v]\!])$, $[\![\lambda^2]\!] \leftarrow \mathsf{Overflow}^{\mathsf{m}}([\![v']\!])$.
7 $[\![u'']\!] \leftarrow [\![u']\!] - 2^n[\![\lambda^1]\!] \in \mathbb{Z}_{2^m}$.
8 $[\![v'']\!] \leftarrow [\![v']\!] - 2^n[\![\lambda^2]\!] \in \mathbb{Z}_{2^m}$.
9 Compute $[\![\widehat{N_0}]\!] \leftarrow [\![u'']\!] \cdot [\![\widehat{c_0}]\!]$ and $[\![\widehat{D_0}]\!] \leftarrow [\![v'']\!] \cdot [\![\widehat{c_0}]\!]$.
10 Set $[\![\widehat{F_1}]\!] \leftarrow \lfloor 2^{n'} \cdot 2\sqrt{2} \rfloor - 2[\![\widehat{D_0}]\!]$.
11 Compute $[\![\widehat{\widehat{N_1}}]\!] \leftarrow [\![\widehat{N_0}]\!] \cdot [\![\widehat{F_1}]\!]$ and $[\![\widehat{\widehat{D_1}}]\!] \leftarrow [\![\widehat{D_0}]\!] \cdot [\![\widehat{F_1}]\!]$.
12 **for** $k \leftarrow 1$ **to** $\log_2 n$ **do**
13 Each party $\mathcal{P}_i$ computes $(\widehat{N_k})_i \leftarrow ((\widehat{\widehat{N_k}})_i \gg n') + 1$ and $(\widehat{D_k})_i \leftarrow (\widehat{\widehat{D_k}})_i \gg n'$ $(i = 1, 2, 3)$.
14 Set $[\![\widehat{F_{k+1}}]\!] \leftarrow 2^{n'} \cdot 2 - [\![\widehat{D_k}]\!]$.
15 Compute $[\![\widehat{\widehat{N_{k+1}}}]\!] \leftarrow [\![\widehat{N_k}]\!] \cdot [\![\widehat{F_{k+1}}]\!]$ and $[\![\widehat{\widehat{D_{k+1}}}]\!] \leftarrow [\![\widehat{D_k}]\!] \cdot [\![\widehat{F_{k+1}}]\!]$.
16 Compute $[\![R]\!] \leftarrow [\![\widehat{\widehat{N_{\log_2 n+1}}}]\!] + \Delta$.
17 Return $[\![w]\!] \leftarrow \mathsf{ShiftR}^{n+n'}([\![R]\!], n') \bmod 2^n$.

different domains. The inputs u, v and the output w belong to $\mathbb{Z}_{2^n}$. At the first stage, the input values and the initial approximation c_0 will be converted to $\mathbb{Z}_{2^m}$ using the procedures $\mathsf{ShareConv}^{\mathsf{m}}(\cdot)$ and $\mathsf{Overflow}^{\mathsf{m}}(\cdot)$. They behave exactly as $\mathsf{ShareConv}(\cdot)$ and $\mathsf{Overflow}(\cdot)$ with their outputs considered to be shared over $\mathbb{Z}_{2^m}$.

The intermediate values are essentially fixed point numbers of the form $2^{-n'} \cdot \widehat{x}$ with $\widehat{x} \in \mathbb{Z}_{m'}$ for some m'. When such values are multiplied, we also get numbers of the form $2^{-2n'} \cdot \widehat{\widehat{x}} \in \mathbb{Z}_{m''}$. Since SHAREMIND can only handle integer values, these numbers will be represented by the values $\widehat{x}$ and $\widehat{\widehat{x}}$, respectively. In order to retain constant precision, the numbers $2^{-2n'} \cdot \widehat{\widehat{x}} \in \mathbb{Z}_{m''}$ need to be converted back to the form $2^{-n'} \cdot \widehat{x}$. This is done by right shifting all the shares of $\widehat{\widehat{x}}$ by n' positions and rounding, if necessary (see line 13). As a result, the modulus will also be decreased by n', that is, $m' = m'' - n'$. This truncation will introduce rounding errors; see Appendix 2 for details on how they are accounted for.

Theorem 8 *Algorithm* 9 *is correct and secure against one passive attacker.*

Proof Correctness directly follows from the discussion above and error computations presented in Appendix 2. Security of the protocol is trivial as well, since we are only using perfectly simulatable building blocks. □

Table 1 Complexities of protocols

Protocol	Rounds	Communication
Mult	1	$15n$
ShareConv	2	$5n + 4$
Equal	$\ell + 2$	$22n + 6$
ShiftR	$\ell + 3$	$12(\ell + 4)n + 16$
BitExtr	$\ell + 3$	$5n^2 + 12(\ell + 1)n$
PubDiv	$\ell + 4$	$(108 + 30\ell)n + 18$
Div	$4\ell + 9$	$2mn + 6m\ell + 39\ell n + 35\ell n' + 126n + 32n' + 24$

8 Performance analysis

8.1 Complexity of protocols

Communication and round complexities of the described protocols are presented in Table 1. Here, $\ell = \log_2 n$ and the details of selecting n' and m for the general division protocol are presented in Appendix 2.

8.2 Experimental setup

Since we had access to the source code of the SHAREMIND virtual machine, we extended it by implementing the described protocols as primitive operations. We conducted a series of experiments to verify that the new protocols are an improvement over the previous protocols presented in [2].

Since SHAREMIND is designed to be a data mining platform, its instruction set follows the SIMD (single instruction, multiple data) principle. This requires protocols to accept vectors of integers as inputs and provide vectors as outputs. Previous tests conducted on the platform showed that vector operations can be more efficient than single operations. Additionally, we wanted to verify the scalability of the implementation by testing large input vectors. Therefore, we benchmarked each operation with input vectors with sizes ranging from 1 up to 10^8. The input vectors consisted of random values.

The order of the experiments was randomized to reduce the impact of outside factors such as flow control and low-level processes of the operating system. For comparison, we also benchmarked the protocols from [2]. Not all vector sizes could be tested for the old protocols, as their inefficiency overloaded the networking layer and the time-outs caused SHAREMIND to cancel the protocol.

We performed the benchmarks on a high-performance computation cluster. The servers run the Debian Linux operating system, contain 12-core Intel® Xeon® processors, have 48 GB of memory and are connected with network

interface cards allowing for speeds up to 1 Gb/s. We used three of these servers to run the SHAREMIND virtual machine.

SHAREMIND can also run successfully, albeit with lower performance, on weaker hardware and with smaller communication channels. We note that on each machine, SHAREMIND used one core, leaving the other cores with no load. This is because the performance of SHAREMIND is communication-bounded, as opposed to circuit-based solutions. Further experiments must be conducted to measure the effect of weaker communication channels on the computation speed.

8.3 Benchmark results

After conducting the experiments, we fitted the protocol execution times using linear regression. Two distinct lines emerged. One of the regression lines was a fit for input vector size smaller than the point n_s and the other for input vector sizes larger than n_s. This point n_s was identified for each protocol. We call this point the *saturation point* of a protocol.

According to our tests, the saturation point depends on the communication complexity of the protocol. For smaller input vectors, the required network messages fit into the network channel without fragmentation. When the network traffic exceeds the available bandwidth, the flow control algorithms on the SHAREMIND network layer start to work. However, this reduces the efficiency of the protocol and further growth is characterized by a different linear function.

A direct result of this is that practical applications should try to run each protocol with vector sizes equal or larger to the saturation point. This will guarantee that network is used to its full capabilities, and private operations are run on their maximum efficiency.

We note that for input sizes larger than 10^7, the implementation started using large amounts of memory and this affected the running time. This can be seen best on Fig. 3 where a third line seems to emerge. However, since we propose that the implementation of algorithms run the algorithms in batches with the size of the the saturation point n_s, we do not consider the performance of huge vectors a major issue. It may also be reduced by further fine-tuning of the implementation.

Table 2 presents an overview of the benchmark results. For each benchmarked protocol, the table contains the time needed to process a single input, the estimated input size that causes saturation in the communication channel, and finally, the time needed to process a single value in input vectors larger than the saturation point (which is presented in microseconds as it is generally at least a thousand times smaller than the time needed in the case of a single value operation). The values are taken from the linear fits and are therefore estimates.

Table 2 Overview of experimental results

Protocol	Single op. (ms)	n_s	Saturated op. (μs)	Old (μs)
ShareConv	15.3	24000	0.8	18
Mult	25.9	15000	1.8	3.5
Equal	101	27000	5.0	2225
BitExtr	113	2600	51	1426
ShiftR	122	12000	15.7	–
PubDiv	124	3500	44	–
Div	390	800	534	–

For comparison, Table 2 also contains the saturated operation cost for the previous generation of protocols ("Old"). It is clear from the data that the speed of all complex protools has increased by several orders of magnitude. For a visual comparison of the new and old protocols, please refer to Appendix 3.

We conclude that the protocols presented in this paper are significantly more efficient than the ones in Bogdanov et al. [2]. The improvement is most visible for the bit extraction and comparison protocols, where the new protocols are more than a hundred times faster than the previous ones. We also note that our implementation actually achieves a speed of 1 MIPS (million instructions per second) for the private share conversion operations. This is a significant milestone for secure multi-party computation, as data miners typically work with large datasets.

Another goal of our experiments was to show that it is possible to run secure multi-party computation protocols with large input vectors containing up to 100 million values. This demonstrates the robustness of the implementation and therefore its suitability for practical applications.

8.4 Secure k-means clustering

k-means clustering is a cluster analysis algorithm for partitioning a set of points into k clusters according to their distances from each other. Cluster analysis helps to identify similar object groups and is used in a range of areas from business intelligence to computational biology. k-means clustering is a fitting benchmark for the protocols in this paper, as it requires multiplications, greater-than comparison, and division.

Our implementation of k-means uses secure computation to hide the values of the clustered data. We did not hide the size of the clusters or the assignment of points to the clusters. However, this is not a significant limitation, since the cluster sizes would typically be published anyway. While certain techniques could be used to hide the movement of points between clusters, they would significantly lower the performance of the computation. It is a generally accepted tradeoff between privacy and efficiency also taken by previous implementations [9,19]. We modified the algorithm to use

Table 3 Privacy-preserving k-means clustering performance

Dataset	k	Time	Iter.	Multiplications	Less-thans	Divisions
iris 150×4	3	1 s	4	12.6% (9600 ops)	42.9% (5400 ops)	38.5% (44 ops)
synthetic 600×60	3	3 s	5	44.4% ($7.2 \cdot 10^5$ ops)	29% ($2.7 \cdot 10^4$ ops)	21.7% (900 ops)
	5	6 s	8	41.2% ($1.7 \cdot 10^6$ ops)	33% ($1.2 \cdot 10^5$ ops)	16% (2,400 ops)
	8	8 s	7	42.2% ($2.3 \cdot 10^6$ ops)	44.2% ($2.7 \cdot 10^5$ ops)	10.7% (3,360 ops)
plants $34{,}781 \times 70$	3	4 min 58 s	12	75.8% ($1.2 \cdot 10^8$ ops)	21.1% ($3.8 \cdot 10^6$ ops)	0.6% (2,520 ops)
	5	22 min 42 s	28	41.2% ($4.1 \cdot 10^8$ ops)	33% ($2.4 \cdot 10^7$ ops)	0.4% (9,800 ops)
	10	36 min 35 s	17	51.3% ($4.6 \cdot 10^8$ ops)	47.6% ($5.9 \cdot 10^7$ ops)	0.2% (11,900 ops)

vector operations as much as possible to take advantage of the increased performance.

The points are distributed into initial clusters on a round-robin basis (the initial cluster number of point i is $i \bmod k$). Note that different initial cluster numbers can affect the number of iterations needed. We did not attempt to achieve more favorable initial clusters. The algorithm runs until stabilizing.

The benchmarking results presented in Table 3 are based on the *iris, synthetic* and *plants* datasets from the UCI Machine Learning Repository [11]. The databases were stored in secret-shared form after scaling fractional values to allow integer computation; the scaling did not affect the final clustering. Each row shows one experiment: the parameters, overall number of operations, and measured runtime. The latter is further broken down by secure operation. Note that the percentages do not add up to 100 % as a fraction of the time was also used for disk operations and secret sharing.

The synthetic control chart time series data set was also used by Doganay et al. in [9] to benchmark k-means clustering algorithms developed by themselves and by Vaidya and Clifton [19]. Their algorithms only work on vertically partitioned data, which is a considerably weaker security model compared to the one used by SHAREMIND. Despite that, the time they required to cluster the *synthetic* dataset was considerably larger compared to our implementation. Whereas SHAREMIND required 3–8 s for this task, the implementation of the algorithms introduced in [9] needed roughly 30 s, and the time required by the algorithm of [19] was even several orders of magnitude larger.

9 Conclusions

In this paper, we have presented numerous advancements for computational primitives used in the SHAREMIND multi-party computation engine. Compared to the original implementation presented in Bogdanov et al. [2], the performance of all the primitives (multiplication, share conversion, bit extraction, equality testing, comparison) has been increased. Additionally, new protocols for right shift by a public offset and division by both public and private value have been implemented and benchmarked. All the proposed protocols have been proven secure in the semi-honest model with one passive adversary, and a convenient game-based framework for improving the readability of the proofs has been presented.

The original motivation for developing SHAREMIND has come from the needs of mining large volumes of data. Our benchmarks show that with the current improvements, input vectors of up to 10^8 elements can be processed in reasonable time and that speeds up to 1 MIPS can be achieved for basic primitives on state-of-the-art hardware. Real performance of the primitives has improved up to 100 times. Benchmarks with the k-means clustering algorithm show that secure MPC is ready to handle real-world data mining tasks, as algorithm runs needing hundreds of millions of private operations can be executed in reasonable time.

Acknowledgments Authors Dan Bogdanov, Margus Niitsoo and Jan Willemson acknowledge support from the European Regional Development Fund through the Estonian Center of Excellence in Computer Science (EXCS). Authors Dan Bogdanov and Jan Willemson acknowledge support from the European Regional Development Fund through the Software Technology and Applications Competence Centre (STACC) and from the Estonian Science Foundation through grant No. 8124. Author Dan Bogdanov also acknowledges support from the European Social Fund through the Estonian Doctoral School in Information and Communication Technology (IKTDK) and the Doctoral Studies and Internationalisation Programme (DoRa). Author Tomas Toft is supported by Confidential Benchmarking, financed by The Danish Agency for Science, Technology and Innovation; and acknowledges support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within which part of this work was performed, as well as from the Center for Research in the Foundations of Electronic Market (supported by the Danish Strategic Research Council) within which part of this work was performed.

Appendix 1: Bit shift protocols under a public shift

The protocols in this section allow us to perform two more standard bit-level operations on shared values, namely left and right shifts ($\ll$ and $\gg$).[3]

[3] Note that a bit shift can be used for efficient comparison as the highest bit of x is just $x \gg 31$.

First, note that the left shift protocol is actually trivial, since left shift by p positions can be accomplished by multiplying the shared value by a public constant 2^p. This, in turn, can be done by locally multiplying all the shares by the same constant. Since no messages are exchanged, the protocol is trivially secure against a passive adversary.

Right shift, on the other hand, is more complicated because of the unknown overflow carry modulo 2^n. Thus, in order to build a right shift protocol, we first need a protocol to compute the overflow. This is considerably easier to do if the value in question is (temporarily) secret-shared between just two parties, because then the overflow is guaranteed to be either 0 or 1. We thus present two routines: Algorithm 10 for resharing a value to just two parties and Algorithm 11 for computing the overflow bit once the values are shared in this way.

Algorithm 10: Protocol $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}([\![u]\!])$ for resharing a value $[\![u]\!]$ between the parties $\mathcal{P}_2$ and $\mathcal{P}_3$.

Data: Shared value $[\![u]\!]$.
Result: Shared value $[\![u']\!]$ so that $u = u'$ and $u'_1 = 0$.
1 $\mathcal{P}_1$ generates random $r_2 \leftarrow \mathbb{Z}_{2^n}$ and computes $r_3 \leftarrow u_1 - r_2$.
2 $\mathcal{P}_1$ sets $u'_1 = 0$ and sends r_i to $\mathcal{P}_i$ $(i = 2, 3)$.
3 $\mathcal{P}_i$ computes $u'_i \leftarrow u_i + r_i$ $(i = 2, 3)$.
4 Return $[\![u']\!]$.

Theorem 9 *Algorithm* 10 *is correct and secure against one passive attacker.*

Proof Correctness of the Algorithm is straightforward:

$$u' = u'_1 + u'_2 + u'_3 = 0 + u_2 + r_2 + u_3 + r_3 = u_2 + r_2 + u_3 + u_1 - r_2 = u.$$

For security, note that $\mathcal{P}_1$ has no incoming messages, whereas the only incoming messages for $\mathcal{P}_2$ and $\mathcal{P}_3$ are r_2 and $u_1 - r_2$, respectively. These messages can be easily simulated with a random value. □

The correctness proof for Algorithm 11 is somewhat more complicated.

Theorem 10 *Algorithm* 11 *is correct and secure against one passive attacker.*

Proof To prove correctness, we need to compute the overflow bit λ. The overflow occurs exactly when $u'_2 + u'_3 \geq 2^n$, or equivalently $u'_2 \geq 2^n - u'_3$. Note that modulo 2^n the value $2^n - u'_3$ is represented just as $-u'_3$ (unless $u'_3 = 0$, which has to be treated separately). Thus,

$$\lambda = 1 \iff u'_2 \geq (-u'_3) \bmod 2^n \wedge u'_3 \neq 0.$$

In order to perform the comparison between u'_2 and $-u'_3$, we first run Algorithm 4 and obtain a bitwise shared vector

Algorithm 11: Protocol $[\![\lambda]\!] \leftarrow \mathsf{Overflow}([\![u']\!])$ for obtaining the overflow bit $[\![\lambda]\!]$ for $[\![u']\!]$ if share $u'_1 = 0$.

Data: Shared value $[\![u']\!]$ where $u'_1 = 0$.
Result: Share $[\![\lambda]\!]$ so that $u' = u'_2 + u'_3 - \lambda 2^n$.
1 $\mathcal{P}_1$ sets $p_1 = 0$.
2 $\mathcal{P}_2$ sets $p_2 = u'_2$.
3 $\mathcal{P}_3$ sets $p_3 = -u'_3$.
4 $\overline{[\![s]\!]} \leftarrow \mathsf{MSNZB}(\overline{[\![p]\!]})$.
5 Share the value $-u'_3$ bitwise as a vector $\overline{[\![-u'_3]\!]}$.
6 $[\![\lambda^0]\!] \leftarrow 1 \oplus \bigoplus_{i=0}^{n-1} \overline{[\![s]\!]}^{(i)} \wedge \overline{[\![-u'_3]\!]}^{(i)}$.
7 $\mathcal{P}_3$ checks whether $u'_3 = 0$. If so, $\lambda^0_3 = 1 \oplus \lambda^0_3$.
8 Return $[\![\lambda]\!] \leftarrow \mathsf{ShareConv}([\![\lambda^0]\!])$.

$\overline{[\![s]\!]}$, which contains all zeroes if $u'_2 = -u'_3 \bmod 2^n$, or has just one bit in the highest position where they differ. Thus, the dot product $\bigoplus_{i=0}^{n-1} \overline{[\![s]\!]}^{(i)} \wedge \overline{[\![-u'_3]\!]}^{(i)} = 1$ iff $u'_2 < -u'_3 \bmod 2^n$ and hence $\lambda^0 = 1$ iff $u'_2 \geq -u'_3 \bmod 2^n$, as required. The only exception appears when $u'_3 = 0$, in which case, no overflow can occur, but λ^0 is set to 1. This mistake is easy to correct locally by $\mathcal{P}_3$ who has the original u'_3 and can flip his own share of λ^0 in case u'_3 happens to be 0.

The security of the protocol is still trivial as it is just a composition of perfectly simulatable protocols. □

We are now ready to present the right shift protocol. The main idea behind the public right shift protocol is to convert the input to a sum of two values (known to two of the parties) and then shift these down. This leaves us with two problems. First, discarding the low bits discards the carry bit for the least significant position that is retained. Second, the top carry bit of the addition would previously implicitly disappear as we consider addition modulo 2^n. Since the values have been shifted down, the carry bit will be present. The bulk of the work of the protocol consists of determining and correcting for these two carry bits.

The protocol itself is presented as Algorithm 12.

Algorithm 12: Protocol $[\![w]\!] \leftarrow \mathsf{ShiftR}([\![u]\!], p)$ for evaluating right shift.

Data: Shared value $[\![u]\!]$ and a public shift p.
Result: Shares $[\![w]\!]$ such that $w = u \gg p$.
1 $[\![u']\!] \leftarrow \mathsf{ReshareToTwo}([\![u]\!])$.
2 $[\![s]\!] \leftarrow [\![u' \ll n - p]\!]$ (locally).
3 $[\![\lambda_1]\!] \leftarrow \mathsf{Overflow}(u')$.
4 $[\![\lambda_2]\!] \leftarrow \mathsf{Overflow}(s)$.
5 $\mathcal{P}_i$ computes $v_i \leftarrow u'_i \gg p$.
6 Return $[\![w]\!] = [\![v]\!] - 2^{n-p}[\![\lambda_1]\!] + [\![\lambda_2]\!]$.

Theorem 11 *Algorithm* 12 *is correct and secure against one passive attacker.*

Proof Correctness of the algorithm follows from the discussion above. Since $u'_2 + u'_3 = u + \lambda_1 2^n$, we have

$$v = v_1 + v_2 + v_3 \bmod 2^n = (u'_2 \gg p) + (u'_3 \gg p) \bmod 2^n$$
$$= u \gg p + \lambda_1 2^{n-p} - \lambda_2 \bmod 2^n,$$

hence

$$u \gg p = v - \lambda_1 2^{n-p} + \lambda_2 \bmod 2^n.$$

For security note that we are only composing perfectly simulatable subroutines. □

This protocol can also be used for extracting the most significant bit for comparison purposes. As it is also slightly more efficient than the full bit extraction, we use it as the basis of the comparison in the current implementation for the comparison operator.

Appendix 2: Error calculation of Goldschmidt division

We will present an analysis of the effects of rounding errors. This is done by looking at the divergence from the "ideal" computation where no rounding takes place and for which the error terms can be fairly easily estimated. A similar analysis was performed in [10]. Their analysis was more detailed, but relied on using floating-point numbers, making it hard to apply it here directly.

Let N_i, D_i, F_i, c_0 denote the actual real numbers encountered during the run of Newton Goldschmidt iterations as described in Sect. 9. In SHAREMIND, we are using the approximations of values x by fixed point numbers $\widetilde{x} = 2^{-n'} \cdot \widehat{x}$ being represented by $\widehat{x} \in \mathbb{Z}_{m'}$ for some m'.

Recall that both the sequences (N_i) and (D_i) were converging from below to $\frac{u}{v}$ and 1, respectively. To preserve the convergence from below in the presence of errors, extra care needs to be taken with rounding errors to make sure they are also one-sided.

Let the differences between the real values N_i, D_i and their approximations be ΔN_i and ΔD_i selected so that $\widetilde{N_i} = N_i + \Delta N_i$ and $\widetilde{D_i} = D_i - \Delta D_i$. Note that on line 13 of Algorithm 9, the value of $\widehat{N_k}$ is always rounded up and the value of $\widehat{D_k}$ is always rounded down. This guarantees that we have $\Delta N_k, \Delta D_k \geq 0$ for all $k \geq 1$. When the shares of $\widehat{\widehat{N_k}}$ and $\widehat{\widehat{D_k}}$ are right shifted to convert the elements back to the precision $2^{-n'}$ (Algorithm 9, line 13), additional truncation errors are introduced. Since there are three computing parties and we shift by n' positions, the errors occurring at both upwards and downwards rounding are bounded by $\delta = 3 \cdot 2^{-n'}$. Thus, for $k \geq 1$, we obtain

$$\widetilde{D_{k+1}} > \widetilde{D_k} \cdot \widetilde{F_{k+1}} - \delta = \widetilde{D_k} \cdot (2 - \widetilde{D_k}) - \delta$$
$$= (D_k - \Delta D_k) \cdot (2 - D_k + \Delta D_k) - \delta$$
$$= D_{k+1} - 2\Delta D_k (1 - D_k) - (\Delta D_k)^2 - \delta$$

and

$$\widetilde{N_{k+1}} \leq \widetilde{N_k} \cdot \widetilde{F_{k+1}} + \delta = \widetilde{N_k} \cdot (2 - \widetilde{D_k}) + \delta$$
$$= (N_k + \Delta N_k) \cdot (2 - D_k + \Delta D_k) + \delta$$
$$= N_{k+1} + N_k \Delta D_k + \Delta N_k (2 - D_k + \Delta D_k) + \delta.$$

This implies

$$\Delta D_{k+1} = D_{k+1} - \widetilde{D_{k+1}} < 2\Delta D_k (1 - D_k) + (\Delta D_k)^2 + \delta$$
$$\leq 2\Delta D_k 2^{-2^k} + (\Delta D_k)^2 + \delta$$

and

$$\Delta N_{k+1} = \widetilde{N_{k+1}} - N_{k+1}$$
$$\leq \Delta N_k (2 - D_k + \Delta D_k) + N_k \Delta D_k + \delta$$
$$< \Delta N_k (1 + 2^{-2^k} + \Delta D_k) + \frac{u}{v} \Delta D_k + \delta.$$

Since the first rounding error is introduced only after multiplication by F_1, we have $\Delta D_1, \Delta N_1 \leq \delta$. Thus, we can iterate these recurrent inequalities to get bounds for $\Delta D_k, \Delta N_k$ in terms of $\frac{u}{v}$ and δ.

In order to guarantee that truncation of the result will lead to a proper value, we will have to ensure that the end result R satisfies $\lfloor \frac{u}{v} \rfloor \leq \lfloor R \rfloor < \lfloor \frac{u}{v} \rfloor + 1$. Let $1 - D_k < 2^{-p}$, in which case $N_k < \frac{u}{v}(1 - 2^{-p})$ since $\frac{N_k}{D_k} = \frac{u}{v}$. Recall that $\widetilde{c_0}$ was chosen so that $0.5 \leq v\widetilde{c_0} < 1$, hence $\widetilde{c_0} < \frac{1}{v} \leq 2\widetilde{c_0}$. Consequently, $\widetilde{N_k} \geq N_k < \frac{u}{v}(1 - 2^{-p}) > \frac{u}{v} - u\widetilde{c_0} 2^{-p+1}$.

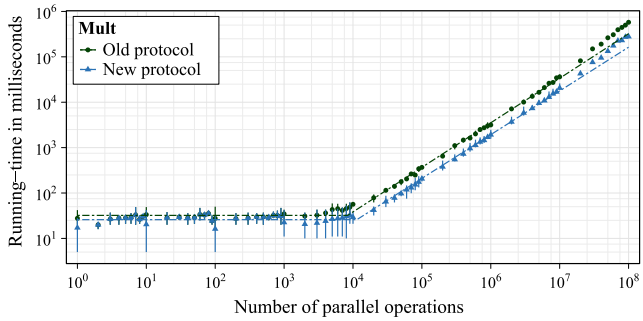


Fig. 1 Benchmark results for the multiplication conversion operation

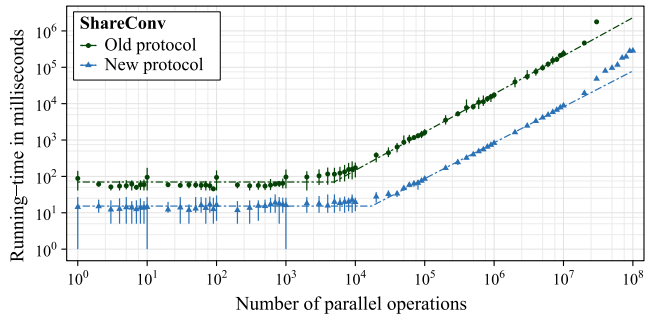


Fig. 2 Benchmark results for the share conversion operation

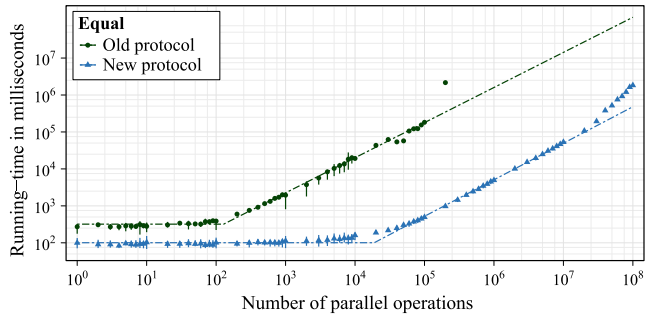


Fig. 3 Benchmark results for the equality comparison operation

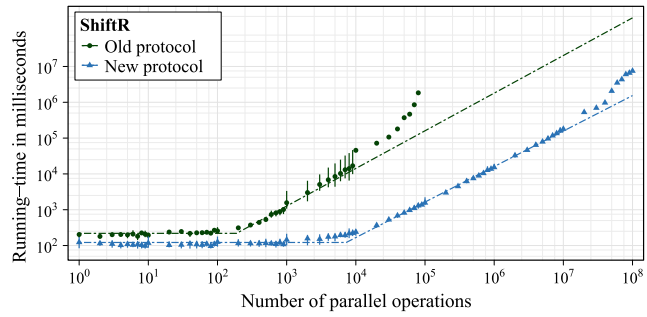


Fig. 4 Benchmark results for the greater-than comparison operation

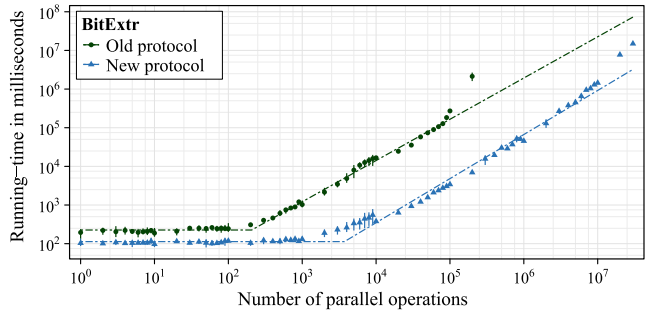


Fig. 5 Benchmark results for the bit extraction operation

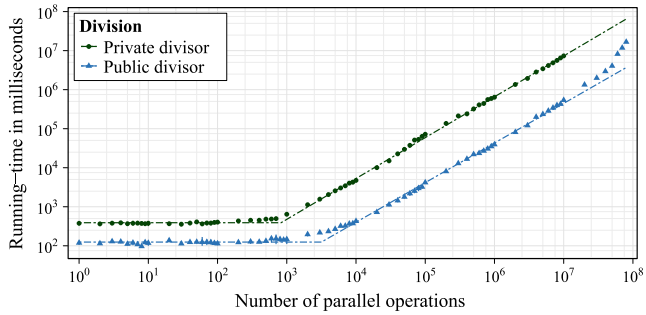


Fig. 6 Benchmark results for the division operations

Taking $R = \widetilde{N_k} + \Delta$ where $\Delta = u\widetilde{c_0}2^{-p+1}$ thus guarantees $R \geq \frac{u}{v}$ and $\lfloor R \rfloor \geq \lfloor \frac{u}{v} \rfloor$.

We are left to show that $R < \lfloor \frac{u}{v} \rfloor + 1$. Let $\Delta N_k < a + b\frac{u}{v}$ as obtained after iterating the above recurring inequalities k times. Then, $R = \widetilde{N_k} + u\widetilde{c_0}2^{-p+1} < N_k + a + 2u\widetilde{c_0}(2^{-p} + b)$. Since $N_k < \frac{u}{v} \leq (\lfloor \frac{u}{v} \rfloor + 1) - \frac{1}{v} < (\lfloor \frac{u}{v} \rfloor + 1) - \widetilde{c_0}$, it suffices to show that $a + 2u\widetilde{c_0}(2^{-p} + b) < \widetilde{c_0}$, or equivalently $\frac{a}{\widetilde{c_0}} + 2ub + u2^{-p+1} < 1$. Since $2^{-n} \leq \widetilde{c_0} < 1$ and $0 \leq u < 2^n$, this can be achieved by showing $2^n(a + 2b + 2^{-p+1}) < 1$.

For $n = 32$, the required inequality can be guaranteed by taking $k = 5$, $n' = 37$, in which case $p > 40.68$ (if the first iteration is done with $F_1 = 2\sqrt{2} - 2D_0$), $a, b < 0.2 \times 2^{-32}$. These choices imply $m = 32 + (5 + 1) \times 37 = 254$.

Appendix 3: Benchmark diagrams

Figures 1, 2, 3, 4, 5, and 6 compare the running times for the protocols in this paper with the protocols in [2]. The range between the minimal and maximal result is shown where multiple experiments were conducted. Missing data points indicate that the protocol was too inefficient to perform at that input size. The axes on the diagrams are drawn on a logarithmic scale. Since the right shift protocol is also used to implement greater-than comparisons, we compared it with the greater-than comparison protocol from [2]. This is an honest comparison, since the greater-than comparison can be implemented in computing the difference on two values and finding the highest bit using the right shift operation.

References

1. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS '08: Proceedings of the 15th ACM conference on Computer and Communications Security, pp. 257–266. ACM, New York, NY, USA (2008). http://doi.acm.org/10.1145/1455770.1455804
2. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS 2008: Proceedings of the 13th European Symposium on Research in Computer Security, Málaga, Spain, Oct 6–8, 2008, LNCS, vol. 5283, pp. 192–206. Springer (2008)
3. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. Cryptology ePrint Archive, Report 2008/289 (2008). http://eprint.iacr.org/
4. Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis. (short paper). In: Keromytis, A. (ed.) Proceedings of the 16th International Conference on Financial Cryptography and Data Security. FC'12. Lecture Notes in Computer Science, vol. 7397, pp. 57–64. Springer Berlin/Heidelberg (2012)
5. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: FC '09: Proceedings of the 13th International Conference on Financial Cryptography, pp. 325–343 (2009)
6. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-Preserving aggregation of multi-domain network events and statistics. In: Proceedings of the USENIX Security Symposium '10, pp. 223–239. Washington, DC, USA (2010)
7. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS '01: 42nd Annual Symposium on Foundations of Computer Science, pp. 136–145 (2001)
8. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Proceedings of The 3rd Theory of Cryptography Conference, TCC 2006, LNCS, vol. 3876. Springer (2006)
9. Doganay, M.C., Pedersen, T.B., Saygin, Y., Savaş, E., Levi, A.: Distributed privacy preserving k-means clustering with additive secret sharing. In: Proceedings of the 2008 International Workshop on Privacy and Anonymity in Information Society, PAIS '08, pp. 3–11 (2008)
10. Even, G., Seidel, P.M., Ferguson, W.E.: A parametric error analysis of Goldschmidt's division algorithm. J. Comput. Syst. Sci. **70**(1), 118–139 (2005)
11. Frank, A., Asuncion, A.: UCI Machine Learning Repository (2010). URL http://archive.ics.uci.edu/ml
12. Geisler, M.: Cryptographic Protocols: Theory and Implementation. Ph.D. thesis, Aarhus University (2010)
13. Granlund, T., Montgomery, P.L.: Division by invariant integers using multiplication. In: PLDI '94: Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, pp. 61–72 (1994)
14. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: CCS '10: Proceedings of the 17th ACM conference on Computer and Communications Security, pp. 451–462. ACM (2010)

15. Malka, L.: VMCrypt: modular software architecture for scalable secure computation. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS'11. pp. 715–724 (2011)
16. Parhami, B.: Computer Arithmetic: Algorithms and Hardware Designs. Oxford University Press, Oxford (2010)
17. Rodeheffer, T.: Software integer division. Microsoft Research Tech, Report MSR-TR-2008-141 (2008)
18. SecureSCM. Technical report D9.1: Secure Computation Models and Frameworks. http://www.securescm.org (2008)
19. Vaidya, J., Clifton, C.: Privacy-preserving *k*-means clustering over vertically partitioned data. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data mining, KDD '03, pp. 206–215 (2003)

III

Bogdanov, D., Jagomägis, R., Laur, S.: A universal toolkit for cryptographically secure privacy-preserving data mining. In: Chau, M., Wang, G.A., Yue, W.T., Chen, H. (eds.) Proceedings of the Pacific Asia Workshop on Intelligence and Security Informatics, PAISI '12. Lecture Notes in Computer Science, vol. 7299, pp. 112–126. Springer (2012).

A Universal Toolkit for Cryptographically Secure Privacy-Preserving Data Mining

Dan Bogdanov[1,2,⋆], Roman Jagomägis[1,2], and Sven Laur[2]

[1] AS Cybernetica, Akadeemia tee 21, 12618 Tallinn, Estonia
{dan,lighto}@cyber.ee
[2] University of Tartu, Institute of Computer Science, Liivi 2, 50409 Tartu, Estonia
swen@math.ut.ee

Abstract. The issue of potential data misuse rises whenever it is collected from several sources. In a common setting, a large database is either horizontally or vertically partitioned between multiple entities who want to find global trends from the data. Such tasks can be solved with secure multi-party computation (MPC) techniques. However, practitioners tend to consider such solutions inefficient. Furthermore, there are no established tools for applying secure multi-party computation in real-world applications. In this paper, we describe Sharemind—a toolkit, which allows data mining specialist with no cryptographic expertise to develop data mining algorithms with good security guarantees. We list the building blocks needed to deploy a privacy-preserving data mining application and explain the design decisions that make Sharemind applications efficient in practice. To validate the practical feasibility of our approach, we implemented and benchmarked four algorithms for frequent itemset mining.

1 Introduction

The ability to combine different data sources is crucial in data analysis. For instance, not all causal dependencies are discoverable in small-scale medical studies. Therefore, the data of individual studies is often merged to get more reliable results. In other occasions, a combination of different studies can lead to previously undiscovered relations.

However, combining databases gives rise to serious privacy concerns. In the most severe cases, the database owners cannot give their data to other parties for processing, as their database contains either personally identifiable information or company trade secrets. The latter makes it almost impossible to carry out the analysis even if all data owners would support it. Moreover, it can even be illegal to merge the data, as data protection laws restrict the collection and processing of personal and medical data.

In this paper, we show how to set up a privacy-preserving data mining service using the SHAREMIND secure computation framework. Differently from many approaches, our solution provides cryptographic security and is efficient enough to be usable in real-life applications. In common terms, nothing is leaked during the data aggregation except

⋆ This research has been supported by Estonian Science Foundation grant number 8124, the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and the Software Technology and Applications Competence Centre, STACC and by European Social Funds Doctoral Studies and Internationalisation Programme DoRa.

M. Chau et al. (Eds.): PAISI 2012, LNCS 7299, pp. 112–126, 2012.

the desired outputs. Of course, there are certain underlying assumptions that must be met. We discuss the features and applicability of our approach and compare it with other methods in Section 2. Section 3 presents our practical contribution—four privacy-preserving frequent itemset mining algorithms for use with SHAREMIND. Section 4 gives benchmark results from actual experiments performed on a SHAREMIND system.

2 Data Mining Using Secure Multi-Party Computation

A typical data mining study involves data donors, data collectors and data analysts. Data donors are the owners of the data. However, they typically do not perform data mining themselves. Instead, they send their data to data collectors who carry out the analysis or contract analysts to do that. This is a potential cause of privacy and trust issues, since data collectors and analysts can misuse data or forward it to other parties. Ideally, we want to guarantee that data collectors and analysts learn nothing about the input data so that they can *rightfully* refute all claims of abuse. Similarly, analysts should learn nothing beyond the desired end results. Additionally, data donors should not have to participate in the processing as the popular data entry platforms (web browsers, mobile devices) lack the power to do so.

Secure multi-party computation techniques can be used to achieve these goals, provided that there are at least two non-colluding data collectors. However, there is a big conceptual difference depending on the number of data collectors. Namely, non-trivial privacy preserving computation must rely on slow cryptographic primitives, whenever the data is split between two entities [8,14,15,21].

As a solution, we propose a setting where data donors use secret sharing to distribute the data between several data collectors (*miners*) as depicted in Figure 1. Secret sharing is a cryptographic technique used to distribute confidential data into shares so that the shares leak no information about the original value [19].

Secret sharing assures that the entries stored by individual miners are completely random bit strings. Hence, a data provider *does not have to trust* any of the miners. Instead, the donor must believe that *miners as a group obey certain rules*, i.e., that no two miners collude with each other during the computations. In practice, miners will be well-guarded computers that belong to independent companies or government agencies.

The properties of secure multi-party computation protocols are proven in certain models. For example, in the *honest-but-curious* security model, it is assumed that the miners follow the secure computation protocols. If no two miners among three collude, then no miner can access any single input value of the processed database. Also, the inputs and intermediate values do not leak during computation.

As a result, the individual miners can rightfully refute all abuse complaints by showing that they followed the restrictions posed on group members. The cryptographic protocols guarantee their inability to draw conclusions about the data beyond the desired outcomes. Also, we do not have to make any assumptions on the honesty of data donors. The donors do not participate in computations directly as they only send shares of their inputs to the miners. Hence, they can influence the outcome only by altering their inputs, which is unavoidable in any case.

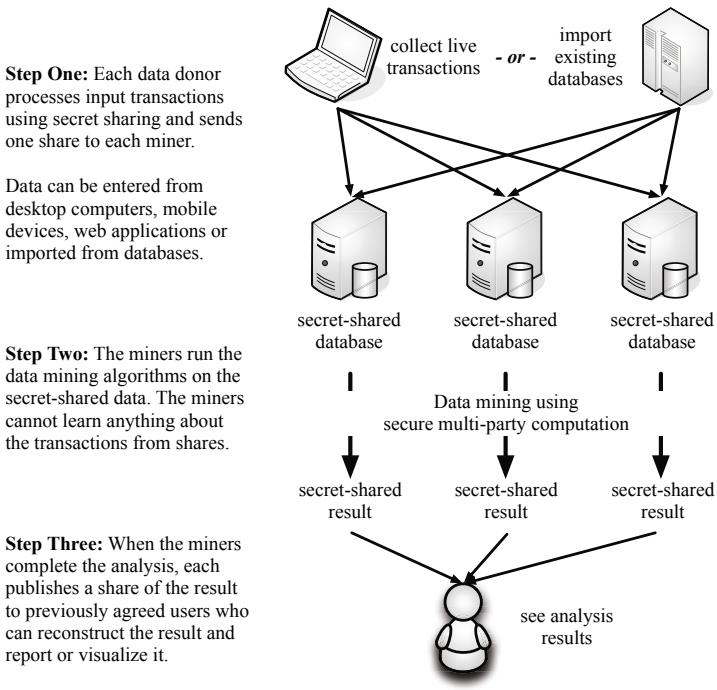


Fig. 1. Architecture of the secure data mining system

2.1 The Sharemind Toolkit

The SHAREMIND platform is a practical implementation of the model described above. Currently, the system supports three miner nodes and is proven secure in the honest-but-curious security model [3]. Having three miners is optimal w.r.t. efficiency, as the overall communication complexity grows quadratically with the number of miners.

For all practical purposes, one can treat SHAREMIND as a virtual machine with a general purpose arithmetic processor and access to a secret-shared secure relational database. These shared values are modified with share computing protocols that leak no information about inputs and outputs. We deliberately omit all details on the secure computation protocols, since all the SHAREMIND protocols together with formal security analysis have been published separately [3]. In brief, all protocols can be executed sequentially or in parallel without losing security and miners learn only which operations are performed on shared data and values that are explicitly reconstructed from shares. The occurrences of such reconstruction can be controlled in the algorithm.

Programs for SHAREMIND can be written in three different ways. There is a high-level language SECREC, a low-level assembly language and finally, it is possible to program in C++ using special libraries to invoke secure computation routines. Programs written in SECREC look like ordinary C programs with two important distinctions. First, variables have explicit confidentiality types: **public** and **private**. All private values are secret shared and conversion to the public type requires an explicit call of the **declassify**

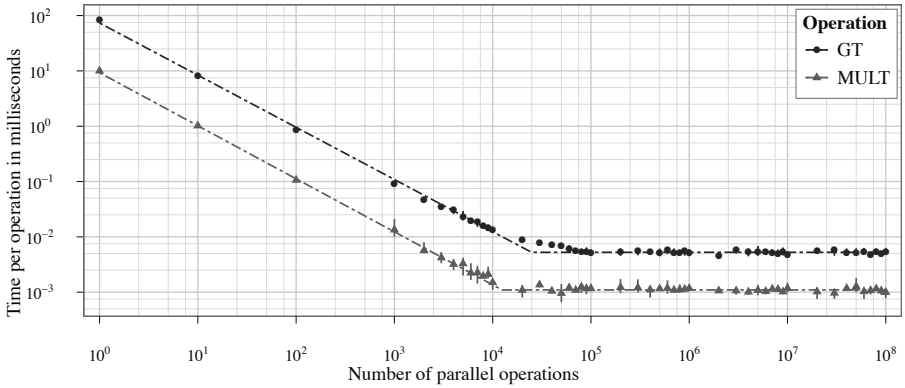


Fig. 2. The cost of secure multiplication (MULT) and secure comparison (GT) operations with different input vector sizes

operator. Conversion from public to private is automatic. The goal of these language elements is to explicitly bring out the locations in the algorithm where private values are made public. Ideally, **declassify** should be used as rarely as possibly and preferably only for final results or intermediate results with a low privacy risk. The second important feature of SECREC is its explicit support for vector and matrix data types. Since the parallel execution of several share computing protocols increases efficiency, the compiler automatically parallelizes vector and matrix operations. We do not give further details about the SECREC language as it is not in the scope of this paper. Further details can be found on the SHAREMIND web page [18].

The assembly language provides fine-grained control—a programmer can specify computations down to the level of register manipulations. The C++ interface allows to avoid code interpretation entirely and provides an implementation with maximum efficiency. As a drawback, one must know each implementation detail, whereas programming in SECREC requires no knowledge of the underlying structures.

The SHAREMIND toolkit also includes several developer tools, such as a designated SECREC development environment, tools for profiling and debugging and an easy-to-use virtual machine. Developer versions are available from the website [18].

2.2 Performance Tuning Tricks

Most share computing protocols involve online communication over the network. Due to network delays, all such operations are guaranteed to take several milliseconds and the performance of the system is likely to be communication-bounded. However, if we execute several instructions in parallel, the impact of network delays remains roughly the same and only the amount of time spent on transferring the data and performing computations increases marginally, see Figure 2. As a result, algorithms that are practically infeasible based on the costs of individual operations can be computable in nearly real-time if most operations are parallelized.

Performance profiles of this shape are not specific to SHAREMIND—any other platform, where individual operations require online communications, behaves similarly. An idealized performance profile is determined by the initial cost t_{i}, saturation point n_{s} and limiting cost t_{ℓ}. The cost of a single operation is t_{i}, but the amortized complexity of an operation drops as more values are processing parallel. This holds for up to n_{s} inputs, as beyond that, each new input value raises the total complexity by t_{ℓ}. If we use initial costs in the running-time analysis, we get a conservative upper bound on the running time. Similarly, the use of limiting costs gives us a lower bound. The saturation point is dependent on the bandwidth of the network channel. If the channel is full, then further vectorization does not improve performance.

2.3 Deployment Scenarios

SHAREMIND has two main deployment scenarios. For survey-type services, three respectable organizations should deploy SHAREMIND servers. Each of them should be motivated to both run the data mining task and preserve the privacy of the data owners. Data donors and analysts can then use a designated desktop, web or mobile applications to work with data entry and analysis applications running on the resulting platform. This deployment is most fitting, when the data owners are individuals or companies who provide information for a larger study.

When the data owners are organizations belonging to a consortium, they can adopt a more democratic approach to setting up the data mining system. They should choose three amongst themselves to deploy the SHAREMIND miner software. All organizations will now provide data into the system, including the hosts themselves. This way, their dedication to privacy preservation is even stronger, since if they try to break the privacy of the other parties, they can also compromise their own inputs.

2.4 Related Work

The theoretical model where dedicated miner nodes are used to collect and process inputs form data donors was first proposed by Damgård and Ishai [9]. SHAREMIND provides a right mix of cryptographic techniques and implementation techniques to get maximal efficiency. Indeed, the alternative multi-party computation frameworks VIFF [12] and SIMAP [4] are less optimized for large input sizes, see Table 1. The SEPIA framework [7] is comparable in the speed of multiplication, but it is slower in comparisons.

SHAREMIND has some unique features that are not found in other secure computation implementations. First, it has a database for securely storing large datasets prior to aggregation. Second, the high-level SECREC algorithm language hides the details of cryptographic protocols.Third, SHAREMIND has strong support for vector and matrix operations which are executed as efficient parallel operations. All these features greatly simplify the development of data mining algorithms.

Table 1. Published running-times of systems related to SHAREMIND

Framework	Multiplication	Less than or equal
Running time of a single operation t_{i}		
SHAREMIND	11.4 ms	101 ms
VIFF	0.63 ms	126 ms
SIMAP	42 ms	774 ms
Limiting cost for a single operation t_ℓ		
SHAREMIND	$1.1 \cdot 10^{-3}$ ms	$5.3 \cdot 10^{-3}$ ms
SEPIA	$6.9 \cdot 10^{-3}$ ms	1.6 ms
SIMAP	3 ms	674 ms

3 Frequent Itemset Mining

To show the practical applicability of the SHAREMIND framework, we implemented four privacy-preserving algorithms for frequent itemset mining. This problem is a good test case, as it is simple enough, but at the same time all solutions are moderately computation intensive.

Frequently co-occurring events or actions often reveal information about the underlying causal dependencies. Hence, frequent itemset mining is often used as one of the first steps in the analysis of transactional data. Market basket analysis is one of the most well-known application areas for these algorithms. Although privacy issues are important in this context, a shop can function only if all transactions are correctly recoded into the central database and in this case the loss of record-level privacy is inevitable. In this case, the use of privacy-preserving algorithms is justified only if we want to combine data from different sources. For instance, one can combine shopping behavior with demographic data without publishing either of the datasets.

We formalize frequent itemset mining as follows. Let $\mathcal{A} = (a_1, \ldots, a_m)$ be the complete list of attributes that can appear in transactions, e.g., the items sold in the shop. Then a transaction is a subset of $\mathcal{A}$ and the list of transactions $\mathcal{T}_1, \ldots, \mathcal{T}_n$ can be represented as an $n \times m$ zero-one matrix $\mathcal{D}$ where $\mathcal{D}[\,i, j\,] = 1$ iff $a_j \in \mathcal{T}_i$. The *support of an itemset* $\mathcal{X}$ is the number of transactions that contain all items of $\mathcal{X}$. The *cover of an itemset* is the set of transaction identifiers that contain the itemset $\mathcal{X}$.

Many frequent itemset mining algorithms [22,23] convert the database into a set of covers, as support counting becomes more efficient provided that all necessary covers fit into the main memory. Covers of itemsets can also be used in privacy-preserving algorithms; however, their representation should not leak information about the individual transactions. Consequently, we must represent covers with index vectors $\boldsymbol{x}$ such that $x_i = 1$ if $\mathcal{X} \in \mathcal{T}_i$ and $x_i = 0$ otherwise.

Let $\mathrm{cover}(\mathcal{X})$ denote the corresponding index vector and $\boldsymbol{x} \odot \boldsymbol{y}$ denote the pointwise multiplication of vectors. Also, let $\mathcal{D}[\,*, a\,]$ denote the column of $\mathcal{D}$ that corresponds to the attribute a. Then the pair of recursive equations

$$\mathrm{cover}(\{a\}) = \mathcal{D}[\,*, a\,] \ , \tag{1}$$

$$\mathrm{cover}(\mathcal{X} \cup \mathcal{Y}) = \mathrm{cover}(\mathcal{X}) \odot \mathrm{cover}(\mathcal{Y}) \ . \tag{2}$$

where $a \in \mathcal{A}$ can be any individual attribute and $\mathcal{X}$ and $\mathcal{Y}$ are arbitrary itemsets, is sufficient to compute the covers of all itemsets. The natural correspondence between support and cover of an itemset

$$\mathrm{supp}(\mathcal{X}) = |\mathrm{cover}(\mathcal{X})| \tag{3}$$

where $|\boldsymbol{x}| = x_1 + \cdots + x_n$ for any index vector $\boldsymbol{x}$ allows us to express supports in terms of addition and multiplication operations. This property significantly simplifies the design of privacy-preserving frequent itemset mining algorithms.

The aim of frequent itemset mining is to find all itemsets $\mathcal{X}$ such that their support is above a prescribed threshold t. As support is anti-monotone:

$$\mathcal{X} \subseteq \mathcal{Y} \quad \Rightarrow \quad \mathrm{supp}(\mathcal{X}) \geq \mathrm{supp}(\mathcal{Y}) \ , \tag{4}$$

all subsets of a frequent set must be frequent. This observation gives rise to two basic search strategies in the lattice of all itemsets. The APRIORI algorithm [1,16] uses breadth-first search. Given a list of frequent ℓ-element itemsets $\mathcal{F}_\ell$, the algorithm generates a list of $(\ell+1)$-element itemsets $\mathcal{C}_{\ell+1}$ such that all ℓ-element subsets of a candidate set $\mathcal{X} \in \mathcal{C}_{\ell+1}$ belong to $\mathcal{F}_\ell$. After that, supports are computed for all candidates and the list of frequent sets $\mathcal{F}_{\ell+1}$ is put together. The process continues until there are no valid candidates to test. The ECLAT algorithm [22] uses depth-first search. Given a frequent seed pattern $\mathcal{X}$ and the set of frequent items $\mathcal{F}_1$, the algorithm forms a new candidate set $\mathcal{C} = \{\mathcal{X} \cup \{a\} : a \in \mathcal{F}_1\}$ and recursively applies the same search procedure for all frequent candidates. As a result, the candidate set remains small enough to fit all covers into the main memory, which makes the search more efficient. There are more elaborate algorithms for frequent itemset mining, see the overviews [13,5] and references provided therein.

3.1 Designing Privacy-Preserving Algorithms

All privacy-preserving algorithms presented below use private values stored as shares and public values available for all miners. To emphasize this distinction, we surround private variables with double brackets and write public variables as usual. For example, $[\![x]\!]$ denotes that variable x is secret shared and $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$ means that the shares of $z = x \cdot y$ are securely computed from the shares of x and y. The same notational convention applies to vectors and matrices. As a result, all algorithms given below are quite similar to the actual SECREC programs used in our experiments.

There are certain aspects to consider while designing privacy-preserving algorithms. First, a large amount of sequential operations can considerably increase running times. Second, the miners inevitably learn whether a branching condition holds or not. Hence, only public values can be used for branching. If a private value must be used to make decisions, it has to be declassified beforehand. For the same reason, array indices must be public. While these limitations seem rather strong, they considerably simplify the formal security analysis. Moreover, these limitations can be satisfied by representing the entire algorithm as an arithmetical circuit.

3.2 A Privacy-Preserving Apriori Algorithm

We assume that the miners have a secret shared matrix $\mathcal{D}$. For a horizontally split database, the data donors have to secret share their rows. For a vertical split, data donors have to secret share their columns.

In all algorithms, we need to test whether a candidate set $\mathcal{X} = \{x_1, \ldots, x_k\}$ is frequent, i.e. we want to evaluate $[\![\mathrm{supp}(\mathcal{X})]\!] \geq t$. When the database $\mathcal{D}$ is secret shared, the support of an itemset be computed from the cover as

$$\mathrm{supp}(\mathcal{X}) = |\mathcal{D}[\,*, x_1\,] \odot \cdots \odot \mathcal{D}[\,*, x_k\,]| \quad .$$

A naive implementation of this formula produces k sequential multiplication operations, whereas a balanced evaluation strategy has only $\lceil \log_2 k \rceil$ sequential operations. For example, two sequential operations are needed to compute the cover of $\{x_1, \ldots, x_4\}$ if we concurrently execute the following multiplications:

$$(\mathcal{D}[\,*, x_1\,] \odot \mathcal{D}[\,*, x_2\,]) \odot (\mathcal{D}[\,*, x_3\,] \odot \mathcal{D}[\,*, x_4\,]) \quad .$$

The latter is important, as parallel operations are much faster. As a downside, the required amount of memory increases by k times.

For further gains, note that each candidate in the APRIORI algorithm is obtained by merging two frequent itemset found on the previous level. Hence, the number of multiplication operations can be reduced to just one large vector operation, provided that we store all covers of frequent itemsets. By performing frequency tests $[\![\mathrm{supp}(\mathcal{X})]\!] \geq t$ for all candidate sets in parallel, we increase the efficiency of the comparison operation.

These two optimizations form the core of Figure 3, where the elements of a vector $\boldsymbol{s}_i$ are supports of a candidate set $\mathcal{C}_i$ and the columns of the matrices M_i correspond to the cover vectors of frequent itemsets $\mathcal{F}_i$. Our version of GENCANDIDATES outputs a list of candidate sets $\mathcal{C}_{i+1}$ and two index sets $\mathcal{I}_1$ and $\mathcal{I}_2$ such that

$$\mathcal{C}_{i+1} = \{\mathcal{I}_1[\,j\,] \cup \mathcal{I}_2[\,j\,] : j = 1, \ldots, |\mathcal{C}_{i+1}|\} \quad .$$

Hence, the matrix assignment $[\![M_{i+1}]\!] \leftarrow [\![M_i]\!][\,*, \mathcal{I}_1\,] \odot [\![M_i]\!][\,*, \mathcal{I}_2\,]$ produces respective cover vectors for $\mathcal{C}_{i+1}$ and the algorithm is formally correct.

The main drawback of this algorithm is its high memory consumption. As cover vectors for all candidate sets have to be allocated, the algorithm quickly allocates large amounts of memory when there are many frequent itemsets.

3.3 A Privacy-Preserving Eclat Algorithm

Due to cached covers, the memory footprint of Figure 3 is several magnitudes higher than for the vanilla APRIORI algorithm. Moreover, we cannot use cover compaction methods, such as vertical tid-lists and diffsets [22,23], since the reduced cover size inevitably leaks information about the supports. In principle, one could drop cover caching, but this leads to significant performance penalties.

In these circumstances, the depth-first search strategy followed by the ECLAT algorithm can be more compelling. The core step in our privacy-preserving ECLAT algorithm is depicted in Figure 4. The function `EclStep` takes a frequent itemset $\mathcal{X}$ and

procedure APRIORI($[\![\mathcal{D}]\!]$, t, k)
  // Compute support for all cover vectors
  $[\![\boldsymbol{s}]\!] \leftarrow \texttt{ColSum}([\![\mathcal{D}]\!])$
  // Declassify index vector of frequent columns
  $\boldsymbol{f}_1 \leftarrow$ **declassify**$([\![\boldsymbol{s}]\!] \geq [\![t]\!])$
  // Gather frequent column data
  $\mathcal{F}_1 \leftarrow \{\mathcal{A}\texttt{[}\,i\,\texttt{]} : f_1\texttt{[}\,i\,\texttt{]} = 1\}$
  $[\![M_1]\!] \leftarrow [\![\mathcal{D}]\!]\texttt{[}\,*\,,\mathcal{F}_1\,\texttt{]}$
  // Validate candidate itemsets until size k
  for $i \in \{1, \ldots, k-1\}$ **do**
    // Generate candidates
    $(\mathcal{C}_{i+1}, \mathcal{I}_1, \mathcal{I}_2) \leftarrow$ GENCANDIDATES($\mathcal{F}_i$)
    // Compute covers for all candidate sets
    $[\![M_{i+1}]\!] \leftarrow [\![M_i]\!]\texttt{[}\,*\,,\mathcal{I}_1\,\texttt{]} \odot [\![M_i]\!]\texttt{[}\,*\,,\mathcal{I}_2\,\texttt{]}$
    // Compute support for all covers
    $[\![\boldsymbol{s}]\!] \leftarrow \texttt{ColSum}([\![M_{i+1}]\!])$
    $\boldsymbol{f}_{i+1} \leftarrow$ **declassify**$([\![\boldsymbol{s}]\!] \geq [\![t]\!])$
    // Remember frequent sets
    $\mathcal{F}_{i+1} \leftarrow \{\mathcal{C}_{i+1}\texttt{[}\,i\,\texttt{]} : \boldsymbol{f}_{i+1}\texttt{[}\,i\,\texttt{]} = 1\}$
    $[\![M_{i+1}]\!] \leftarrow [\![M_{i+1}]\!]\texttt{[}\,*\,,\mathcal{F}_{i+1}\,\texttt{]}$
  end for
  return $\mathcal{F}_1 \cup \mathcal{F}_2 \cup \ldots \cup \mathcal{F}_k$
end procedure

Fig. 3. Privacy-preserving APRIORI algorithm

tries to elongate by adding new items into it. For efficiency reasons, `EclStep` uses a list of potential extension itemsets $\mathcal{N}$ and the matrix $[\![M]\!]$ of corresponding cover vectors. We emphasize that each single matrix column corresponds to an itemset. For example, $[\![M]\!]\texttt{[}\,*\,,\mathcal{X}\,\texttt{]}$ is a column corresponding to the set $\mathcal{X}$ and $[\![M]\!]\texttt{[}\,*\,,\mathcal{N}\,\texttt{]}$ is a list of columns where each column corresponds to an itemset in $\mathcal{N}$.

The recursion in `EclStep` is aborted if new itemsets are larger than k. Otherwise, the shares of new cover vectors and supports are computed. Then, `EclStep` is applied for each newly found frequent itemset and results are merged. The full ECLAT algorithm consists of a call to the function `EclStep`($\emptyset$, $\mathcal{A}$, $\mathcal{D}$, k, t).

3.4 Hybrid Traversal over the Search Space

A second look on Algorithm 3 reveals that parallel generation of all candidate covers is a major contributor on memory consumption, since $|\mathcal{C}_{i+1}| \gg |\mathcal{F}_{i+1}|$. Thus, generating and testing candidates in smaller blocks will significantly decrease the memory footprint. We implemented this optimization in the HYB-APRIORI algorithm.

The efficiency of `EclStep` is determined by the number of extensions $\mathcal{N}$. If $\mathcal{N}$ is always large, there is no performance penalties. The HYB-ECLAT algorithm traverses the search space so that the number of candidates tested in each iteration is large enough, but the number of covers that must be cached is still small. The algorithm keeps a stack of frequent itemsets $\mathcal{F}_\circ$ that must be cached. In each iteration, it takes ℓ first

procedure ECLSTEP($\mathcal{X}, \mathcal{N}$, $[\![M]\!]$, k, t)
  // Combine frequent set with the candidate cover vectors
  $[\![M]\!] \leftarrow [\![M]\!][\,*, \mathcal{X}\,] \odot [\![M]\!][\,*, \mathcal{N}\,]$
  // Compute supports
  $[\![\boldsymbol{s}]\!] \leftarrow \texttt{ColSum}([\![M]\!])$
  $\boldsymbol{f} \leftarrow$ **declassify**$([\![\boldsymbol{s}]\!] \geq [\![t]\!])$
  // Construct new frequent item sets
  $\mathcal{F} \leftarrow \{\mathcal{X}\}, \mathcal{F}_* = \{\mathcal{X} \cup \mathcal{N}[\,i\,] : \boldsymbol{f}[\,i\,] = 1\}$
  // If we have reached the target set size, return
  if $|\mathcal{X}| + 1 \geq k$ **then**
    return $\mathcal{F}_*$
  end if
  // See how we could extend the current frequent sets
  for $\mathcal{Y} \in \mathcal{F}_*$ **do**
    $\mathcal{N}_* = \{\mathcal{Z} \in \mathcal{F}_* : \mathcal{Y} \preceq \mathcal{Z}\}$
    // Recursively extend the frequent itemset candidate
    $\mathcal{F} \leftarrow \mathcal{F} \cup \texttt{EclStep}(\mathcal{Y}, \mathcal{N}_*, [\![M]\!][\,*, \mathcal{N}_*\,], k, t)$
  end for
  return $\mathcal{F}$
end procedure

Fig. 4. The core step of privacy-preserving ECLAT

elements from $\mathcal{F}_\circ$ and finds the corresponding frequent extensions and adds them on top of $\mathcal{F}_\circ$. As a result, the algorithm does parallel breadth-first search and achieves a smaller memory footprint than HYB-APRIORI.

3.5 Security Analysis

Since the miners learn only the execution flow and the values of public and declassified variables, it is sufficient to show that these observations do not leak more information than originally intended. Formally, we need an efficient *simulator* that, given the desired outputs, reconstructs the execution path together with all declassified values. As all secure protocols used in SHAREMIND are universally composable [3], such a simulator will provide us with a formal security proof.

Theorem 1. *If the setup assumptions of the* SHAREMIND *platform are not violated, then all four algorithms reveal nothing beyond the list of frequent itemsets.*

Proof. Given a list $\mathcal{F}$ consisting of frequent itemsets with up to k elements, it is straightforward to determine the sets $\mathcal{F}_1, \ldots, \mathcal{F}_k$. From these sets, we can easily compute $\mathcal{C}_2, \ldots, \mathcal{C}_k$ and all other public variables used in the APRIORI algorithm. Note that the sets $\mathcal{F}_i$ and $\mathcal{C}_i$ completely determine the declassification results, since $\boldsymbol{f}_i[\,j\,] = 1$ if and only if $\mathcal{C}_i[\,j\,] \in \mathcal{F}_i$. Consequently, the entire execution flow can be reconstructed and it is straightforward to simulate the execution of the APRIORI algorithm. An analogous argumentation also holds for the ECLAT, HYB-APRIORI and HYB-ECLAT algorithms, since their execution path depends only the outputs of frequency tests.

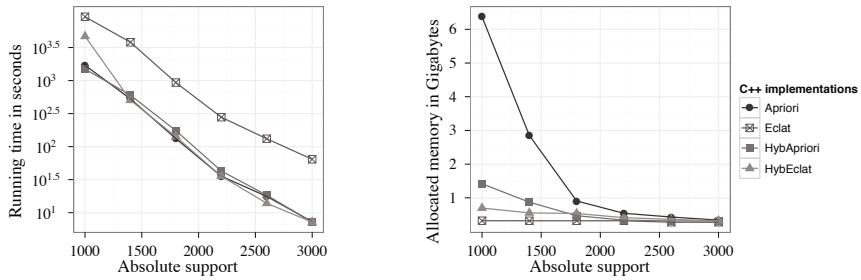


Fig. 5. Execution time and memory consumption on the MUSHROOM dataset

4 Experimental Results

We benchmarked the proposed algorithms on the SHAREMIND platform. We implemented APRIORI, ECLAT and HYB-APRIORI in SECREC and also using the low-level C++ interface. We implemented the HYB-ECLAT algorithm only in C++.

We conducted experiments on a dedicated SHAREMIND cluster, where three computers hosted the miners and a fourth computer acted as a client. A data import tool was used to import the datasets into the SHAREMIND database. Then, a client tool was used to submit frequent itemset mining queries. Execution times were measured at the miner nodes—the experiment started when the query was received and ended when the results were sent to the client. We used operating system calls to determine the amount of physical memory used by the miner application.

The conditions in the cluster are more ideal than the ones typically occurring in real life deployments. The nodes in the cluster are connected by fast point-to-point network connections. The machines contain 2.93 GHz six-core Intel Xeon CPUs and 48 GB of memory. We note that only a single CPU core was in use.

4.1 Analysis of Computational Experiments

We tested our algorithms on the MUSHROOM and CHESS datasets from the UC Irvine Machine Learning Repository [11]. MUSHROOM is a sparse (8124 transactions, 119 items, 19.3% density) and CHESS is a dense dataset (3196 transactions, 75 items, 49.3% density). In addition, we benchmarked the RETAIL dataset [6] (88163 transactions, 16470 items, 0.06% density) to see whether SHAREMIND can handle large datasets. We are not considering larger databases, as random sampling techniques can be used to approximate supports with high enough precision [20]. Typically, 50 000 transactions are enough for achieving a relative precision of 1%.

Figures 5 and 6 depict the behavior of all four algorithms on the MUSHROOM and CHESS datasets. As expected, the ECLAT algorithm has the smallest memory footprint but is also the slowest one. The running times of the other algorithms are comparable, whereas the memory consumption metrics justify hybrid traversal strategies.

We chose the HYB-APRIORI algorithm for conducting benchmarks on the RETAIL dataset due to its small memory footprint. Since RETAIL is large and sparse, we can use

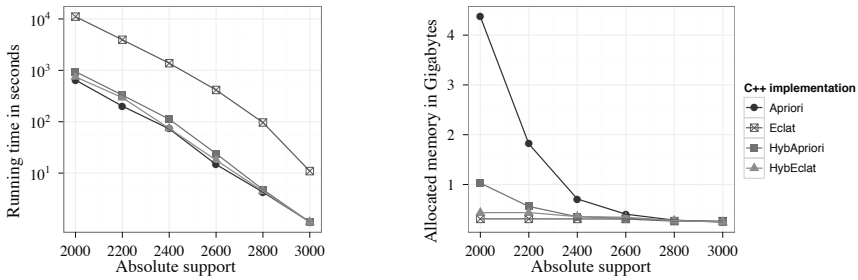


Fig. 6. Execution time and memory consumption on the CHESS dataset

smaller thresholds than for the other datasets. Table 2 gives the results of the benchmarks. The larger runtimes are expected, given the size of the database. However, the results illustrates the feasibility of private frequent itemset mining on large datasets.

The number of frequent itemsets found is over $50,000$ for the MUSHROOM and CHESS datasets. In most cases, such an amount of frequent sets reveals most of the useful associations. Discovery of more subtle relations requires search space pruning based on background information.

Table 2. Benchmark results for the RETAIL dataset

Support	**1000**	**800**	**600**	**400**
Running time	9 min	12 min	22 min	86 min
Memory usage	470 MB	476 MB	498 MB	615 MB

4.2 Related Work

To our knowledge, there are no directly comparable implementations of frequent itemset mining running on secure multi-party computation systems. Therefore we decided to implement a directly comparably algorithm on one of the competing platforms. Based on the performance timings in Table 1 we decided to use the SEPIA system, since it also performs well with vector inputs and is therefore a fair match for SHAREMIND.

The SEPIA framework provides the user with an application programming interface written in the Java programming language. We used this interface to implement the same APRIORI algorithm that was tested on SHAREMIND. Since SEPIA can also process a vector of values together, we also made use of this feature.

We compared the ease of developing the APRIORI implementation using these techniques. The results of the comparison are shown in Table 3. We did not measure the time required for implementing the algorithm, because our developers had an unequal experience with the platforms.

Table 3. Ease of development with the SHAREMIND and SEPIA platforms

	SHAREMIND with SECREC	SHAREMIND with C++ API	SEPIA with Java API
Secret sharing	Automatic	Automatic	Manual
Data storage	Built-in database	Built-in database	Manual
Algorithm development	SECREC programming language with data mining support	C++ API for operations on confidential data	Java API for secure computation protocols
Modifying an application	Change SECREC code and load it on the server	Change protocol implementation, recompile and restart server	Change protocol implementation, recompile and restart server

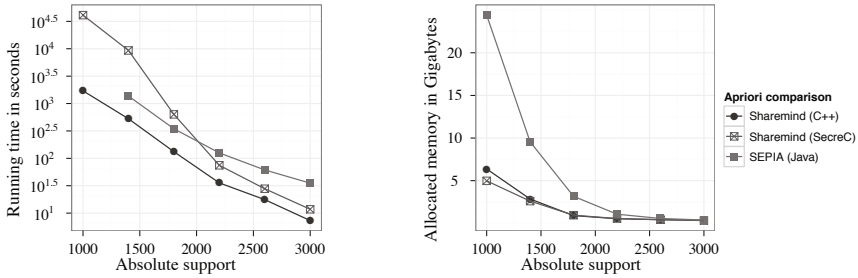


Fig. 7. Comparison of performance between the SHAREMIND and SEPIA frameworks

Performance was measured in exactly the same conditions. The results are given in Figure 7. The SHAREMIND C++ implementation is the fastest of the three with SEPIA being about two to five times slower across tested support sizes. The SECREC implementation starts up fast, but loses performance as more and more data is processed. Profiling showed that this is caused by inefficient vector handling in the virtual machine processing SECREC. This inefficiency will be resolved so that SECREC will have the same efficiency as the C++ API.

In the the memory consumption comparison, both SECREC and SHAREMIND/C++ are significantly more memory-efficient than SEPIA. This could be explained with a lower level of optimizations in large vector operations. There is no measurement for SEPIA with support 1000, because the protocol did not complete its work in several attempts. Measurements showed that network traffic between two SEPIA nodes ceased during processing a large vector multiplication operation.

We note that it does not make sense to compare our solution with randomized response techniques developed for frequent itemset mining [10,17], since they are shown to be rather imprecise for high privacy levels [2]. The algorithm by Kantarcioglu and Clifton [15] is the only algorithm that is probably more efficient, as it uses properties of horizontal database partition to do most computations locally. The solution can be also adopted for SHAREMIND with the same efficiency.

4.3 Feasibility in Real World Applications

In real-world applications, network latency can be significantly greater and bandwidth significantly lower than in our experiments. The initial costs of private operations are roughly proportional to the end-to-end latency of the communication channel and the limiting costs are roughly proportional to bandwidth.

In our experiments, the end-to-end latency on the application level is roughly 40–50 milliseconds. In real world settings, the physical network latency component is 40–100 milliseconds depending on network topology and geographical location. Hence, the end-to-end latency can increase up to five times. Similarly, it is quite plausible that the network bandwidth is up to 10 times smaller. For the most pessimistic settings, the amount of time spent on the private operations can thus grow 10 times, which still preserves practical feasibility.

While SHAREMIND guarantees the secrecy of data during operations, the developer may leak confidential information by declassifying too much values. Often, the security analysis is trivial—especially, when only the final result is declassified. In other cases, it is sufficient to show that all declassified values can be derived from the final result.

These issues can be solved by providing a developer's guide to help a programmer decide on the security of a declassification. Also, static program analysis may be a suitable tool for detecting trivial leaks in SECREC programs.

5 Conclusion

We have shown that privacy-preserving data mining using secure multiparty computation is practically feasible. Although the theoretical feasibility has been known since the late 1980s, SHAREMIND is one of the few implementations capable of processing large databases. We have validated our approach by implementing four privacy-preserving frequent itemset mining algorithms. The APRIORI and ECLAT provide high speeds and a low memory footprint, respectively. HYB-APRIORI and HYB-ECLAT provide more fine-tuned controls for memory usage and the degree of parallelization.

We have provided benchmarks for all the algorithms and also a performance comparison with another secure multiparty computation system. The optimizations for large scale databases allow SHAREMIND to outperform other systems. The ease of practical use can be as important as performance. Arguing about the privacy of a algorithm implemented with SHAREMIND requires no cryptographic proofs. Therefore, a data mining expert does not have to be a cryptography expert to use SHAREMIND and SECREC for creating privacy-preserving data mining applications.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. of VLDB 1994, pp. 487–499. Morgan Kaufmann (1994)
2. Agrawal, S., Haritsa, J.R., Prakash, B.A.: FRAPP: a framework for high-accuracy privacy-preserving mining. Knowledge Discovery and Data Mining 18(1), 101–139 (2009)

3. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A Framework for Fast Privacy-Preserving Computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008)
4. Bogetoft, P., Damgård, I., Jakobsen, T., Nielsen, K., Pagter, J., Toft, T.: A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In: Di Crescenzo, G., Rubin, A. (eds.) FC 2006. LNCS, vol. 4107, pp. 142–147. Springer, Heidelberg (2006)
5. Bramer, M.: Principles of Data Mining. Springer (2007)
6. Brijs, T., Swinnen, G., Vanhoof, K., Wets, G.: Using association rules for product assortment decisions: A case study. In: Proc. of KDD 1999, pp. 254–260. ACM (1999)
7. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: privacy-preserving aggregation of multi-domain network events and statistics. In: Proc. of USENIX Security 2010, p. 15. USENIX Association (2010)
8. Chor, B., Kushilevitz, E.: A zero-one law for boolean privacy. In: Proc. of STOC 1989, pp. 62–72. ACM Press (1989)
9. Damgård, I., Ishai, Y.: Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 378–394. Springer, Heidelberg (2005)
10. Evfimievski, A.V., Srikant, R., Agrawal, R., Gehrke, J.: Privacy preserving mining of association rules. In: Proc. of KDD 2002, pp. 217–228 (2002)
11. Frank, A., Asuncion, A.: UCI machine learning repository (2010)
12. Geisler, M.: Cryptographic Protocols: Theory and Implementation. PhD thesis, Aarhus University (2010)
13. Goethals, B.: Frequent set mining. In: The Data Mining and Knowledge Discovery Handbook, ch. 17, pp. 377–397. Springer (2005)
14. Goethals, B., Laur, S., Lipmaa, H., Mielikäinen, T.: On Private Scalar Product Computation for Privacy-Preserving Data Mining. In: Park, C., Chee, S. (eds.) ICISC 2004. LNCS, vol. 3506, pp. 104–120. Springer, Heidelberg (2005)
15. Kantarcioglu, M., Clifton, C.: Privacy-preserving distributed mining of association rules on horizontally partitioned data. IEEE Transactions on Knowledge and Data Engineering 16(9), 1026–1037 (2004)
16. Mannila, H., Toivonen, H., Verkamo, A.I.: Efficient algorithms for discovering association rules. In: KDD Workshop, pp. 181–192 (1994)
17. Rizvi, S., Haritsa, J.R.: Maintaining data privacy in association rule mining. In: Proc. of VLDB 2002, pp. 682–693 (2002)
18. The Sharemind framework, `http://sharemind.cyber.ee/`
19. Shamir, A.: How to share a secret. Communications of the ACM 22(11), 612–613 (1979)
20. Toivonen, H.: Sampling large databases for association rules. In: Proc. of VLDB 1996, pp. 134–145. Morgan Kaufmann (1996)
21. Yang, Z., Wright, R.N., Subramaniam, H.: Experimental analysis of a privacy-preserving scalar product protocol. Computer Systems: Science & Engineering 21(1) (2006)
22. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. 12(3), 372–390 (2000)
23. Zaki, M.J., Gouda, K.: Fast vertical mining using diffsets. In: Proc. of KDD 2003, pp. 326–335 (2003)

CURRICULUM VITAE

Personal data

Name	Dan Bogdanov
Birth	February 28th, 1983
Citizenship	Estonian
Marital Status	Married
Languages	Estonian, English, German, Russian, French
Contact	+372 52 75 525 dan@cyber.ee

Education

2007–	University of Tartu, Ph.D. candidate in Computer Science
2005–2007	University of Tartu, M.Sc. in Computer Science
2001–2005	University of Tartu, B.Sc. in Computer Science
1998–2001	Pärnu Sütevaka School of Humanities, secondary education
1993–1998	Pärnu Sütevaka School of Humanities, primary education
1989–1993	Pärnu 1st Secondary School, primary education

Employment

2007–	Cybernetica AS, researcher
2006–2007	OÜ Quretec, systems analyst
2005–2006	AS EGeen, systems analyst
2003–2005	OÜ Web Expert, software developer
2000–2001	OÜ Maripuu Meedia, software developer

ELULOOKIRJELDUS

Isikuandmed

Nimi	Dan Bogdanov
Sünniaeg ja -koht	28. veebruar 1983
Kodakondsus	eestlane
Perekonnaseis	abielus
Keelteoskus	eesti, inglise, saksa, vene, prantsuse
Kontaktandmed	+372 52 75 525 dan@cyber.ee

Haridustee

2007–	Tartu Ülikool, informaatika doktorant
2005–2007	Tartu Ülikool, MSc informaatikas
2001–2005	Tartu Ülikool, BSc informaatikas
1998–2001	Pärnu Sütevaka Humanitaargümnaasium, keskharidus
1993–1998	Pärnu Sütevaka Humanitaargümnaasium, põhiharidus
1989–1993	Pärnu 1. keskkool, algharidus

Teenistuskäik

2007–	Cybernetica AS, teadur
2006–2007	OÜ Quretec, analüütik
2005–2006	AS EGeen, analüütik
2003–2005	OÜ Web Expert, tarkvaraarendaja
2000–2001	OÜ Maripuu Meedia, tarkvaraarendaja

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

1. **Mati Heinloo.** The design of nonhomogeneous spherical vessels, cylindrical tubes and circular discs. Tartu, 1991, 23 p.
2. **Boris Komrakov.** Primitive actions and the Sophus Lie problem. Tartu, 1991, 14 p.
3. **Jaak Heinloo.** Phenomenological (continuum) theory of turbulence. Tartu, 1992, 47 p.
4. **Ants Tauts.** Infinite formulae in intuitionistic logic of higher order. Tartu, 1992, 15 p.
5. **Tarmo Soomere.** Kinetic theory of Rossby waves. Tartu, 1992, 32 p.
6. **Jüri Majak.** Optimization of plastic axisymmetric plates and shells in the case of Von Mises yield condition. Tartu, 1992, 32 p.
7. **Ants Aasma.** Matrix transformations of summability and absolute summability fields of matrix methods. Tartu, 1993, 32 p.
8. **Helle Hein.** Optimization of plastic axisymmetric plates and shells with piece-wise constant thickness. Tartu, 1993, 28 p.
9. **Toomas Kiho.** Study of optimality of iterated Lavrentiev method and its generalizations. Tartu, 1994, 23 p.
10. **Arne Kokk.** Joint spectral theory and extension of non-trivial multiplicative linear functionals. Tartu, 1995, 165 p.
11. **Toomas Lepikult.** Automated calculation of dynamically loaded rigid-plastic structures. Tartu, 1995, 93 p, (in Russian).
12. **Sander Hannus.** Parametrical optimization of the plastic cylindrical shells by taking into account geometrical and physical nonlinearities. Tartu, 1995, 74 p, (in Russian).
13. **Sergei Tupailo.** Hilbert's epsilon-symbol in predicative subsystems of analysis. Tartu, 1996, 134 p.
14. **Enno Saks.** Analysis and optimization of elastic-plastic shafts in torsion. Tartu, 1996, 96 p.
15. **Valdis Laan.** Pullbacks and flatness properties of acts. Tartu, 1999, 90 p.
16. **Märt Põldvere.** Subspaces of Banach spaces having Phelps' uniqueness property. Tartu, 1999, 74 p.
17. **Jelena Ausekle.** Compactness of operators in Lorentz and Orlicz sequence spaces. Tartu, 1999, 72 p.
18. **Krista Fischer.** Structural mean models for analyzing the effect of compliance in clinical trials. Tartu, 1999, 124 p.

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
20. **Jüri Lember.** Consistency of empirical k-centres. Tartu, 1999, 148 p.
21. **Ella Puman.** Optimization of plastic conical shells. Tartu, 2000, 102 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
25. **Maria Zeltser.** Investigation of double sequence spaces by soft and hard analitical methods. Tartu, 2001, 154 p.
26. **Ernst Tungel.** Optimization of plastic spherical shells. Tartu, 2001, 90 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 p.
28. **Rainis Haller.** *M(r,s)*-inequalities. Tartu, 2002, 78 p.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
30. **Eno Tõnisson.** Solving of expession manipulation exercises in computer algebra systems. Tartu, 2002, 92 p.
31. **Mart Abel.** Structure of Gelfand-Mazur algebras. Tartu, 2003. 94 p.
32. **Vladimir Kuchmei.** Affine completeness of some ockham algebras. Tartu, 2003. 100 p.
33. **Olga Dunajeva.** Asymptotic matrix methods in statistical inference problems. Tartu 2003. 78 p.
34. **Mare Tarang.** Stability of the spline collocation method for volterra integro-differential equations. Tartu 2004. 90 p.
35. **Tatjana Nahtman.** Permutation invariance and reparameterizations in linear models. Tartu 2004. 91 p.
36. **Märt Möls.** Linear mixed models with equivalent predictors. Tartu 2004. 70 p.
37. **Kristiina Hakk.** Approximation methods for weakly singular integral equations with discontinuous coefficients. Tartu 2004, 137 p.
38. **Meelis Käärik.** Fitting sets to probability distributions. Tartu 2005, 90 p.
39. **Inga Parts.** Piecewise polynomial collocation methods for solving weakly singular integro-differential equations. Tartu 2005, 140 p.
40. **Natalia Saealle.** Convergence and summability with speed of functional series. Tartu 2005, 91 p.
41. **Tanel Kaart.** The reliability of linear mixed models in genetic studies. Tartu 2006, 124 p.
42. **Kadre Torn.** Shear and bending response of inelastic structures to dynamic load. Tartu 2006, 142 p.

43. **Kristel Mikkor.** Uniform factorisation for compact subsets of Banach spaces of operators. Tartu 2006, 72 p.
44. **Darja Saveljeva.** Quadratic and cubic spline collocation for Volterra integral equations. Tartu 2006, 117 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
46. **Annely Mürk.** Optimization of inelastic plates with cracks. Tartu 2006. 137 p.
47. **Annemai Raidjõe.** Sequence spaces defined by modulus functions and superposition operators. Tartu 2006, 97 p.
48. **Olga Panova.** Real Gelfand-Mazur algebras. Tartu 2006, 82 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
50. **Margus Pihlak.** Approximation of multivariate distribution functions. Tartu 2007, 82 p.
51. **Ene Käärik.** Handling dropouts in repeated measurements using copulas. Tartu 2007, 99 p.
52. **Artur Sepp.** Affine models in mathematical finance: an analytical approach. Tartu 2007, 147 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
54. **Kaja Sõstra.** Restriction estimator for domains. Tartu 2007, 104 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
57. **Evely Leetma.** Solution of smoothing problems with obstacles. Tartu 2009, 81 p.
58. **Ants Kaasik.** Estimating ruin probabilities in the Cramér-Lundberg model with heavy-tailed claims. Tartu 2009, 139 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
60. **Indrek Zolk.** The commuting bounded approximation property of Banach spaces. Tartu 2010, 107 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
63. **Marek Kolk.** Piecewise Polynomial Collocation for Volterra Integral Equations with Singularities. Tartu 2010, 134 p.

64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
65. **Larissa Roots.** Free vibrations of stepped cylindrical shells containing cracks. Tartu 2010, 94 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
68. **Olga Liivapuu.** Graded q-differential algebras and algebraic models in noncommutative geometry. Tartu 2011, 112 p.
69. **Aleksei Lissitsin.** Convex approximation properties of Banach spaces. Tartu 2011, 107 p.
70. **Lauri Tart.** Morita equivalence of partially ordered semigroups. Tartu 2011, 101 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.
74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
75. **Nadežda Bazunova.** Differential calculus $d^3 = 0$ on binary and ternary associative algebras. Tartu 2011, 99 p.
76. **Natalja Lepik.** Estimation of domains under restrictions built upon generalized regression and synthetic estimators. Tartu 2011, 133 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
80. **Marje Johanson.** $M(r, s)$-ideals of compact operators. Tartu 2012, 103 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
82. **Vitali Retšnoi.** Vector fields and Lie group representations. Tartu 2012, 108 p.