COMPILATION FOR MORE PRACTICAL SECURE MULTI-PARTY COMPUTATION

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
von

Niklas Büscher, M.Sc.
geboren in Münster

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Referenten:	Prof. Dr. Stefan Katzenbeisser
	Prof. Dr. Florian Kerschbaum
Tag der Einreichung:	21.11.2018
Tag der Prüfung:	11.01.2019

D 17
Darmstadt, 2018

ABSTRACT

Within the last decade, smartphone applications and online services became universal resources and an integral part of nowadays life. Unfortunately, the ongoing digitization trend is also a huge risk for the individual's privacy because users of these interconnected devices and services become more and more transparent and reveal sensitive data to an untrusted and possibly unknown service provider. Yet, since the 1980's it is known that any computation between two or more parties can be evaluated securely such that the parties do not learn more about the inputs of the other parties than they can derive from the output of the computation. For a long time considered to be a purely theoretical concept, in the last fifteen years, this technique known as Secure Multi-Party Computation (MPC), transitioned into a powerful cryptographic tool to build privacy-enhancing technology. As such MPC could prevent mass surveillance of online services while maintaining the majority of their business use cases. Furthermore, MPC could be an enabler for novel business-to-business use cases, where mutually distrusting parties corporate by sharing data without losing control over it. Albeit its potential, the practicality of MPC is hindered by the difficulty to implement applications on top of the underlying cryptographic protocols. This is because their manual construction requires expertise in cryptography and hardware design. The latter is required as functionalities in MPC are commonly expressed by Boolean and Arithmetic circuits, whose creation is a complex, error-prone, and time-consuming task.

To make MPC accessible to non-domain experts, in this thesis we design, implement, and evaluate multiple compilation techniques that translate the high-level language ANSI C into circuit representations optimized for different classes of MPC protocols. Split in two parts, we focus on Boolean circuit based protocols in the first part of this thesis. We begin with an introduction into compilation and optimization of circuits with minimal size, which is required for constant round MPC protocols over Boolean circuits, such as Yao's Garbled Circuits protocol. For this purpose, we identify and evaluate classic logic minimization techniques for their application in compilation for MPC. Then, we present compiler assisted parallelization approaches for Yao's protocol that distribute the computational workload onto multiple processors, which can allow a faster or possibly more energy efficient protocol evaluation. By extending the protocol, we further show that parallelization leads to speed-ups even in single-core settings. As not only size minimization is of relevance for MPC, we also propose a compilation chain for the creation of depth-minimized

Boolean circuits, optimized for their use in multi-round protocols, such as the GMW protocol. For this purpose, we propose and implement new hand-optimized building blocks as well as code and circuit minimization techniques. In most cases the presented compilers create applications from high-level source code that outperform previous (hand-optimized) work.

In the second part, we introduce compilers for two advanced hybrid MPC protocols. First, we study the creation of MPC applications using multiple (standalone) MPC protocols at once. By combining protocols with different paradigms, e.g., Boolean and Arithmetic circuits based protocols, faster applications can be created. For the compilation of these hybrid applications we design and present novel code decomposition and optimization techniques. Moreover, we introduce solutions to the protocol selection problem to efficiently combine multiple protocols. Thus, we are able to present the first compiler that achieves full automatization from source code to hybrid MPC. Second, we investigate compilation for the combination of Oblivious RAM with MPC, also known as RAM based secure computation (RAM-SC). RAM-SC is required in data intensive applications, where circuit based protocols show limited scalability. A multitude of ORAMs based on different design principles with different trade-offs has been proposed. We explore all these design principles and corresponding deployment costs in different scenarios, before introducing a compiler that identifies an optimal selection of ORAM schemes for a given input source code. As such, we present the first fully automatized compile chain for RAM-SC programs.

In summary, we contribute in making MPC practical by improving both, efficiency and automatized application generation.

ZUSAMMENFASSUNG

Eine der größten technischen Entwicklungen in den letzten 15 Jahren ist der Fortschritt des Smartphones, die ständige Verfügbarkeit von Informationen und Konnektivität, sowie die damit verbundene Nutzung von mächtigen Onlinediensten. Leider birgt dieser Digitalisierungstrend auch große Risiken für die Privatsphäre der Nutzer, denn diese werden immer transparenter für den teilweise unbekannten Dienstanbieter. Es ist jedoch bereits seit Mitte der 1980er-Jahre bekannt, dass jegliche Art von Berechnung oder Interaktion zwischen zwei oder mehreren Parteien auch privatsphärefreundlich durchgeführt werden kann, so dass die teilnehmenden Parteien nicht mehr über die Eingaben der anderen Parteien lernen können, als das was sie sich von der gemeinsame Ausgabe ableiten können. Lange war diese Idee der sicheren Mehrparteienberechnung (MPC) nur ein theoretisches Konzept, jedoch hat es sich in den letzten 15 Jahren in ein

sehr praktisches kryptographisches Werkzeug entwickelt, um privatsphäreschützende Anwendungen zu realisieren. Als solches könnte MPC helfen die Massenüberwachung der Onlinedienste zu unterbinden, ohne deren Geschäftsmodelle komplett außer Kraft zu setzen. MPC ermöglicht auch neuartige Geschäftsmodelle, so dass beispielsweise zwei sich nicht vertrauende Unternehmen kooperieren und Daten teilen, ohne dabei die Kontrolle über diese Daten an den Geschäftspartner zu verlieren. Trotz des immensen Potentials von MPC findet es kaum Anwendung in der Praxis. Dies liegt vor allem an der Komplexität und dem Aufwand, um Anwendungen für existierende MPC Protokolle zu entwickeln. Die Erstellung dieser Anwendungen benötigt nämlich umfangreiche Kenntnisse der Kryptographie und des Logikdesigns (Schaltungsbau). Letzteres wird benötigt, da Funktionen in MPC üblicherweise in Form von logischen oder arithmetischen Schaltungen dargestellt werden, deren händische Erzeugung sehr fehleranfällig und zeitaufwendig ist.

Um MPC auch Nicht-Experten zugängliche zu machen, haben wir im Rahmen dieser Arbeit verschiedene Übersetzungstechniken (Compiler) entwickelt, implementiert und evaluiert, die es ermöglichen standard ANSI C Programmcode automatisiert in Schaltungen für verschiedene Klassen von MPC Protokollen zu übersetzen. Diese Arbeit besteht aus zwei Teilen, wobei sich der erste Teil mit der Erzeugung von logischen Schaltkreisen beschäftigt. Wir beginnen mit einer Betrachtung der Konstruktion von Schaltungen mit minimaler Größe. Diese werden üblicherweise in MPC Protokollen mit konstanter Rundenanzahl eingesetzt, wie beispielsweise Yao's Garbled Circuits Protokoll. Zu diesem Zweck analysieren und adaptieren wir klassische Logikminimierungstechniken für MPC. Anschließend zeigen wir, wie Yao's Protokoll mit Hilfe eines Compilers effektiv parallelisiert werden kann, um die Protokollausführung auf Mehrprozessorsystemen zu beschleunigen. Weiterhin präsentieren wir eine Protokollerweiterung, die selbst auf einem einzigen Rechenkern die Berechnungszeit durch Parallelisierung reduzieren kann. Neben der Minimierung der Schaltungsgröße, ist die Minimierung der Schaltungstiefe von besondere Relevanz für MPC Protokolle mit variabler Rundenanzahl, wie beispielsweise dem GMW Protokoll. Für diese Art von Protokollen präsentieren wir einen Übersetzungsansatz, bestehend aus neuen handminimierten Schaltungsbausteinen und Schaltungsminimierungstechniken. Die im Rahmen dieser Arbeit entstandenen Compiler, die die vorgestellten Techniken implementieren, erzeugen Schaltungen aus einer Hochsprache, die meist schneller evaluiert werden können als die zuvor vorgestellten und oft händisch optimierten Schaltungen.

Im zweiten Teil dieser Arbeit beschäftigen wir uns mit der Schaltungserzeugung für hybride MPC Protokolle. Hier zeigen wir zum einen die automatisierte Konstruktion von Anwendungen, die ver-

schiedene MPC Protokolle in einer Anwendung kombinieren. Durch die Kombination von Protokollen mit verschiedenen Designprinzipien, beispielsweise logische und arithmetische Schaltungen, können Anwendungen weiter beschleunigt werden. Zur Erzeugung dieser hybriden Anwendungen präsentieren wir neue Codezerlegungs- und Optimierungstechniken. Weiterhin stellen wir Lösungen vor, welche die effizienteste Kombination von Protokollen für eine gegebene Anwendungszerlegung identifizieren kann. Mit diesem Beitrag können wir den ersten Compiler präsentieren, der eine vollständige Automatisierung von Programmcode zu hybriden Anwendungen erzielt. Als zweite hybride Protokollklasse betrachten wir die Compilierung von RAM basierten Protokollen. Diese so genannten RAM-SC Protokolle kombinieren klassische MPC Protokolle mit Oblivious RAM (ORAM) Techniken und werden für datenintensive Anwendungen benötigt. Da existierende ORAMs verschiedenste Designprinzipien verwenden, präsentieren wir eine umfangreiche ORAM Kostenanalyse mit der die Laufzeit von RAM-SC in beliebigen Einsatzszenarien abgeschätzt werden kann. Diese Abschätzung ermöglicht es uns, den ersten voll automatisierten Compiler für RAM-SC vorzustellen, der sämtliche Speicherzugriffe in einem gegebenen Programmcode erfasst und die effizientesten ORAMs ermitteln kann.

Zusammenfassend trägt diese Arbeit dazu bei, MPC praxistauglicher und zugänglicher für Nicht-Experten zu machen, indem die Entwicklungsprozesse automatisiert und die Effizienz von MPC Anwendungen gesteigert wird.

ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

Niklas Büscher

AKADEMISCHER WERDEGANG

11/2013 - 10/2018	**Technische Universität Darmstadt** Wissenschaftlicher Mitarbeiter Promotionsstudium
04/2011 - 10/2013	**Technische Universität Darmstadt** Studium der Informatik Abschluss: Master of Science
08/2011 - 05/2012	**University of Boulder, Colorado, USA** Studium der Informatik
04/2008 - 03/2011	**Technische Universität Darmstadt** Studium der Informatik Abschluss: Bachelor of Science

PUBLICATIONS

This thesis is based on the following publications:

1. Niklas Büscher and Stefan Katzenbeisser. *"Faster Secure Computation through Automatic Parallelization"*. In: Proceedings of the 24th USENIX Security Symposium, USENIX Security 2015, Washington, D.C., USA.

2. Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. *"Compiling Low Depth Circuits for Practical Secure Computation"*. In: Proceedings of the 21st European Symposium on Research in Computer Security, ESORICS 2016, Heraklion, Greece.

3. Niklas Büscher, David Kretzmer, Arnav Jindal, and Stefan Katzenbeisser. *"Scalable Secure Computation from ANSI-C"*. In: Proceedings of the 8th IEEE International Workshop on Information Forensics and Security, WIFS 2016, Abu Dhabi, United Arab Emirates.

4. Niklas Büscher, Martin Franz, Andreas Holzer, Helmut Veith, and Stefan Katzenbeisser. *"On Compiling Boolean Circuits Optimized for Secure Multi-Party Computation"*. In: Formal Methods in System Design 51(2).

5. Niklas Büscher and Stefan Katzenbeisser, *"Compilation for Secure Multi-Party Computation"*, Springer Briefs in Computer Science 2017, ISBN 978-3-319-67521-3.

6. Niklas Büscher, Alina Weber, and Stefan Katzenbeisser, *"Towards Practical RAM-based Secure Computation"*. In: Proceedings of the 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain.

7. Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. *"HyCC: Compilation of Hybrid Protocols for Practical Secure Computation."* In: Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, Canada.

Further peer-reviewed publications published during my doctoral studies:

8. Johannes A. Buchmann, Niklas Büscher, Florian Göpfert, Stefan Katzenbeisser, Juliane Krämer, Daniele Micciancio, Sander

Siim, Christine van Vredendaal, Michael Walter. *"Creating Cryptographic Challenges Using Multi-Party Computation: The LWE Challenge"*. In: Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography, AsiaPKC@AsiaCCS 2016, Xi'an, China.

9. Niklas Büscher, Stefan Schiffner, Mathias Fischer. *"Consumer Privacy on Distributed Energy Markets"*. In: Proceedings of the Privacy Technologies and Policy - 4th Annual Privacy Forum, APF 2016, Frankfurt, Germany.

10. Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, Stefan Katzenbeisser. *"SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks"*. In: Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2017, Boston, USA.

11. Niklas Büscher, Spyros Boukoros, Stefan Bauregger, Stefan Katzenbeisser. *"Two Is Not Enough: Privacy Assessment of Aggregation Schemes in Smart Metering"*. In: Proceedings of the 17th Privacy Enhancing Technologies Symposium, PETS 2017, Minneapolis, USA.

12. Florian Kohnhäuser, Niklas Büscher, Stefan Katzenbeisser. *"SALAD: Secure and Lightweight Attestation of Highly Dynamic and Disruptive Networks"*. In: Proceedings of the 13th ACM Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea.

ACKNOWLEDGMENTS

Being at the final step of the doctorate allows me to thank everyone who supported me along the way. Foremost, I am very grateful to my advisor Stefan Katzenbeisser, for his invaluable guidance in and beyond research, the enjoyable working environment, and especially for the many mountaineering trips to Kleinwalsertal. Furthermore, I thank Florian Kerschbaum, for agreeing to be the second reviewer and even more for going to any length in trying to keep me in academia. My thank also goes to Christian Reuter, Thomas Schneider, and Neeraj Suri for joining the defense committee.

I owe my first contacts to research to Marc Fischlin and Michael Goesele, who both were independently willing to push undergraduate projects, jointly tackled with Benjamin, into papers and sponsoring our first conference visit. During my masters I was very warmly welcomed into the TK security group led by Stefan and Mathias, who gave me a first introduction into privacy research.

Then, during my doctoral studies, I had the honor to work with excellent colleagues in the SecEng team. Without revealing any details, e.g., with whom I shared the most coffee breaks, I have to say that it was my pleasure to work with André, Christian, Dominik, Erik, Florian, Heike, Marius, Markus, Nikolaos I., Nikolaos II., Nikolay, Philipp, Sebastian B., Sebastian G., Spyros, Tolga, and Ursula. Similarly, I would like to thank the ENCRYPTO group, namely, Ágnes, Amos, Christian, Daniel, Michael, Oleksandr, and Thomas for accepting me as an alien MPC researcher, sharing many conference visits, lunches, and coffees, which frequently led to fruitful discussions.

Likewise, I am very thankful for my co-authors and students, most notably Alina for her passion and endurance in performing circuitry, Andreas for introducing me to CBMC-GC, which provides a foundation for the tools developed as part of this thesis, and David for his commitment in compiler development as well as his exceptional expertise in C++ and information flow.

Many thanks go to my family for their continuous encouragement, teaching me open-mindedness, and the importance of education. But most of all, I am grateful for my wife Katharina, especially for her never-ending support and patience throughout the past years.

CONTENTS

ACRONYMS

AIG AND-Invert Graph

ABB Arithmetic Black Box

ASAP As-soon-as-possible

AST Abstract Syntax Tree

BMC Bounded Model Checking

CFG Control Flow Graph

CGP Coarse-Grained Parallelization

CNF Conjunctive Normal Form

CNN Convolutional Neural Network

CSA Carry-Save Adder

CSN Carry-Save Network

DAG Directed Acyclic Graph

DNF Disjunctive Normal Form

DPF Distributed Point Function

DSL Domain Specific Language

FA Full Adder

FGP Fine-Grained Parallelization

FSS Function Secret Sharing

GRR Garbled Row Reduction

HA Half Adder

IPP Inter-Party Parallelization

LSB Least Significant Bit

MPC Secure Multi-Party Computation

ORAM Oblivious RAM

PET Privacy-Enhancing Technology

PIR Private Information Retrieval

PPA Parallel Prefix Adder

PRF Pseudo Random Function

RCA Ripple Carry Adder

RTT Round Trip Time

SE Symbolic Execution

SIMD Single-Instruction-Multiple-Data

SSA Single Static Assignment

TPC Secure Two-Party Computation

"Die [...] Gestaltung von Datenverarbeitungssystemen sind an dem Ziel auszurichten, so wenig personenbezogene Daten wie möglich zu erheben, zu verarbeiten oder zu nutzen [...], ***soweit dies nach dem Verwendungzweck möglich ist und keinen im Verhältnis zu dem angestrebten Schutzzweck unverhältnismäßigen Aufwand erfordert.***"

— Bundesdatenschutzgesetz (BDSG)
§3a Datenvermeidung und Datensparsamkeit

1 INTRODUCTION

The continuous grow of data gathering and processing, which is fired by cheap sensors (e.g., in smart phones and wearables), cheap storage costs, and efficient machine learning algorithms enables many useful applications and powerful online services. However, this form of heavy data processing, often referred to as 'Big Data', is also a huge risk for the individual's privacy because users of these services become more and more transparent and reveal possibly sensitive data to an untrusted and possibly unknown service provider. Even when not assuming any malicious interest of the data collectors, the gathered data is often centrally stored, and thus prone to many possible breaches, e.g., hackers exploiting vulnerabilities, maliciously acting employees, requests by state agencies, or insufficient data disposal management.

Currently, all these risks are mostly ignored by the service providers as well as the users. The former have (often legitimate) commercial interests in their business use cases and the latter are either unaware of the actual value of their data, lack alternatives, or have no possibilities to opt-out if they want to participate in nowadays life. Consequently, we observe a sincere conflict between business interest and the right on privacy when processing personal data. Yet, there is a solution to this dilemma: Namely, since the 1980's it is (theoretically) known that any computation over sensitive data from multiple parties can be performed securely, such that the participating parties do not learn more about the inputs of the other parties from the computation than they can already derive from the output [Yao82; Yao86]. This form of computation is known as Secure Computation or Secure Multi-Party Computation (MPC) and can be explained by an example application:

Consider two parties Alice and Bob that want to schedule a joint meeting. Yet, Alice does not want to reveal her own schedule, i.e., her availability, to Bob, as it might contain sensitive information. For the same reasons, Bob prefers to maintain his schedule secret. As a solution they can involve a third person, e.g., Charlie, whom they both

trust. Alice and Bob can send their private schedule, i.e., availability slots, to Charlie, who identifies a matching time slot and reveals this slot to Alice and Bob. Unfortunately, a third party that is trusted by both parties barely exists in practice. MPC solves this problem by simulating[1] such a trusted third party using cryptographic protocols. These protocols are run between the two parties, guarantee correctness of the computation and privacy of the users' data. Consequently, MPC is a powerful Privacy-Enhancing Technology (PET), as the data at the service providers can be processed under encryption, while almost all functionality required for business use cases can be maintained.

Naturally, the questions arises: If such powerful cryptographic techniques are available, why are they barely used in practice? The answer is twofold. First, MPC protocols have noticeable deployment costs in computation and communication. Second, creating, i.e., developing efficient applications for MPC by hand, as it had often be done previously, is a tedious and error-prone task. In this thesis, we address both challenges by *presenting a multitude of compilation techniques and implement these in compilers that automatically synthesize optimized MPC applications for different deployment scenarios from high-level code.* By automatizing the creation of MPC applications we significantly lower the entry barrier to MPC.

Our work is of practical relevance because of two main reasons: First, applied cryptography and IT infrastructures in general are getting more and more complex. To handle this growing complexity, simpler developer interfaces and further automatization are desirable to create secure applications. This situation has been identified by industry and academia alike. For example, replacements for standard cryptography libraries have been proposed that provide clearer interfaces to developers, e.g., [BLS12], and also dedicated tools have been developed that synthesize secure cryptographic implementations, e.g., [Krü+17]. Thus, automated compilation for MPC contributes in this direction, as it reduces the complexity when designing new PETs.

Second, the right to privacy is known for ages in most civilizations and therefore most countries have issued data protection laws that regulate the (commercial) use of data with different scopes and strengths. For example, an excerpt of the German privacy law, which has recently been adapted to the European General Data Protection Regulation (GDPR), is printed at the beginning of this chapter. Loosely translated, this excerpt says that all data processing systems should be developed with the goal to use, store, and process as little personal data as possible, *if this is reasonable given the technical efforts*. The proof-of-concept compilers developed as part of this thesis illustrate that the technical efforts to implement privacy-preserving computation are reasonable and consequently the use of MPC in highly sensi-

1 Not meant in the cryptography way.

tive areas, e.g., genome processing or the handling of health records, should be considered.

1.1 RESEARCH GOAL AND CONTRIBUTION

The goal of this thesis is the design, development, and evaluation of compilation techniques for efficiency improvements of MPC, when compiling from a high-level programming language. In this section, we specify this goal, outline our contributions, and illustrate the connection to related work.

SECURE MULTI-PARTY COMPUTATION. In MPC, two main research directions are distinguished. First, MPC protocols dedicated for specific applications have been developed, e.g., for recommender systems [KP08] or privacy-preserving face recognition [Erk+09]. Second, generic protocols have been proposed that allow to perform any computation between two or more parties securely[2]. Dedicated protocols require cryptographers to prove their correctness and security, yet in principal have the possibility to outperform the generic techniques. Generic protocols only need to be proven secure once and then offer significantly more versatility because any application can realized on top of these protocols without further security proofs, and thus without expert knowledge in cryptography. Due to these reasons, many generic protocols using different cryptographic primitives have been proposed, e.g., [Ara+17; BLW08; Dam+12; GMW87; Yao82], and also many theoretical and practical optimizations, e.g., [Bel+13; KS08; Pin+09; ZRE15] made these protocols ready for practice. The focus of this thesis is the compilation for generic MPC protocols.[3]

APPLICATION DESCRIPTIONS AND COMPILATION. At the heart of (generic) MPC is the ability to efficiently represent the functionality to be computed. Most existing protocols require either a formulation in terms of a Boolean or an Arithmetic circuit. Circuits are acyclic graphs, where values flow along the edges, i.e., 'wires', and are processed at nodes, representing either Boolean functions (in case of Boolean circuits), e.g., logical AND, or basic Arithmetic operations (in case of Arithmetic circuits). Hand-coding these circuits, i.e., efficiently formulating the function to be computed in terms of basic Boolean or Arithmetic operations, is practically infeasible even for moderately complex functions, as the manual construction of efficient

2 The term *secure* is defined more precisely in Section 2.2.

3 The tools developed as part of this thesis are creating applications for MPC with only two parties, commonly referred to as Secure Two-Party Computation (TPC). Nevertheless, we remark that the ideas presented in this work transfer for protocols with any number of parties. Because of this and for the sake of generality we will use the term MPC in the remainder of this thesis.

circuits for MPC is a complex and time-consuming task that requires expert knowledge in hardware design. Therefore, the first practical framework (protocol implementation) [Mal+04] for Yao's Garbled Circuits protocol [Yao86], which is one of the most studied MPC protocols, also provided a rudimentary compiler for a domain specific language. In following works, e.g., [Hol+12; Hua+11b], compilation has been identified as an independent task, separated from MPC protocol implementations. Since then, a multitude of compilers has been proposed. We give a detailed overview and classification of these in Section 2.4.

RESEARCH GOAL In this thesis, we study compilation approaches that translate a high-level language (ANSI C) into optimized circuits suiting the requirements of different classes of MPC protocols with a major focus on the optimization for Boolean circuit based protocols. As such, we aim at compiling circuit descriptions that are comparable or better than previous handmade constructions, given a suitable high-level description. Consequently, we aim at increasing both, usability and efficiency of MPC.

CONTRIBUTIONS. A major part of this thesis deals with the compilation and optimization for Boolean circuit based MPC protocols. Optimizing the created circuits is necessary, as MPC is still multiple orders of magnitude slower than generic computation. Therefore, we first study hand-optimized building blocks and a fixed-point optimization algorithm that adapts circuit synthesis techniques, known from logic or chip design [Gaj+12; She93], for their use in compilation for MPC. We implement and evaluate these techniques in the ANSI C compiler CBMC-GC by Holzer et al. [Hol+12] and illustrate their capabilities to compile efficient circuits that are capable of outperforming commercial hardware synthesis tools. As such, we present a compile-chain that creates small circuits from a standard programming language. These circuits are useful for all constant round MPC protocols that operate on Boolean circuits, e.g., Yao's Garbled Circuits protocol.

Although creating highly optimized applications, we observed that MPC in general, and Yao's protocol in particular, still require substantial computational resources. Therefore, we study parallelization techniques for Yao's protocol. We propose two parallelization approaches and introduce a new parallel protocol variant that allows to profit from parallelization even in single core settings. Furthermore, we present the first complete and automatized compile chain that creates and partitions circuits from standard ANSI C code that can be evaluated efficiently in parallel MPC frameworks.

While working on improving the application efficiency in Yao's protocol, many further MPC protocols also advanced and became of prac-

tical relevance. Especially multi-round protocols based on Boolean circuits, such as GMW [GMW86] and TinyOT [Nie+12] improved significantly due to the advancements in Oblivious Transfers [Ash+13]. Also newer and faster protocols, e.g., [Ara+17], following the same multi-round paradigm have been proposed. We address these developments and present an optimizing compiler from ANSI C that creates depth-minimized Boolean circuits for MPC. For this purpose, we present new depth-minimized building blocks and adapt high- and low-level optimization techniques, which allow to automatically generate circuits that outperform previously handmade constructions.

Recently also hybrid protocols, i.e., MPC protocols that combine different techniques, e.g., Boolean with Arithmetic circuits or Boolean circuits with Oblivious RAM (ORAM) [GO96], gained noticeable traction [DSZ15; Gor+12; Liu+14]. Therefore, we further extend the previously developed compilation chain to compile hybrid protocols. To achieve this goal and to also cope with the development of increasing circuit sizes, we present the first scalable compiler that allows to compile, optimize, and partition hybrid MPC protocols consisting of the combination of both Boolean and Arithmetic circuits.

Finally, we study the compilation and optimization for ORAM assisted MPC protocols, also known as RAM-SC [Liu+14], which are necessary for data intensive applications. For this purpose, we present a cost analyses of relevant ORAM schemes, optimize these, and introduce the first optimizing compilation chain for RAM-SC. In summary, we make a wide range of MPC protocols accessible for non-domain experts.

1.2 THESIS OUTLINE

The remainder of this thesis is structured in three parts.
In Part I – Preliminaries we discuss basic techniques and notations. In Chapter 2 we recap the basics of Boolean circuit design, give an overview of MPC protocols relevant for this work, and describe related compilers for MPC. Moreover, we describe example applications that are used as MPC benchmarks throughout this thesis. Then, in Chapter 3 we describe a compiler, named CBMC-GC [Hol+12], which translates ANSI C into Boolean circuits. This compiler is adapted and extended in the following chapters to compile and optimize circuits for the different protocols.

In Part II – Compilation and Optimization for Boolean Circuit based MPC Protocols we present compilation techniques that focus on standalone MPC protocols that use Boolean circuits as application descriptions. In Chapter 4 we present our first contribution by studying multiple size-minimizing optimization techniques for their adaption in circuit synthesis for MPC. The created circuits lead to ef-

ficiency improvements for all constant-round MPC protocols over Boolean circuits. Next, in Chapter 5, we study the parallelization of Yao's Garbled Circuits and present a compilation chain that creates circuits, which can be efficiently evaluated in parallel. In Chapter 6 we present a compilation and optimization approach that compiles depth-minimized Boolean circuits. These are required for multi-round MPC protocols that are deployed practical network environments.

In Part III – Compilation and Optimization for Hybrid MPC Protocols we extend our work towards more advanced MPC protocols. In Chapter 7 we study the compilation of hybrid MPC applications that consist of Boolean and Arithmetic circuits, which allows to mix constant and multi-round MPC protocols efficiently. Furthermore, we present a compilation chain for RAM-SC that suits the requirements of data intensive applications in Chapter 8.

Finally, in Chapter 9 we conclude our work and give an outlook on future research directions in compilation for MPC.

Part I

PRELIMINARIES

2

BASIC TECHNIQUES

Summary: In this chapter, we describe basic techniques required to follow the ideas presented in this thesis. First, in Section 2.1 we describe the circuit computation model, and recap the properties of Boolean circuits, which are predominately used in this thesis. Then, in Section 2.2, we describe three of the most studied secure computation protocols, namely Yao's Garbled Circuits protocol and the Goldreich-Micali-Wigderson (GMW) protocol, which are prototypic for constant-round and multi-round MPC protocols over Boolean circuits, as well as a multi-round additive secret sharing based protocol over Arithmetic circuits. Furthermore, we introduce the concept of Oblivious RAM (ORAM) in Section 2.3, and show how ORAM can be combined with secure computation protocols to build more efficient applications. Afterwards, in Section 2.4 we present a comparison of related compilers for MPC. Finally, we discuss multiple applications that are used for benchmarking purposes in Section 2.5.

2.1 DIGITAL LOGIC AND BOOLEAN CIRCUITS

Boolean circuits are a common representation of functions in computer science and a mathematical model for digital logic circuits. A Boolean circuit is a Directed Acyclic Graph (DAG) with l inputs, m outputs, and s gates. Technically, the graph consists of $|V| = l + m + s$ nodes and $|E|$ edges, where each node can either be of type input, output or gate. Due to the relation of digital circuit synthesis, the edges are called wires. We note that a Boolean circuit, as defined above, is also referred to as a combinatorial circuit. Combinatorial circuits are different to sequential circuits, which can be cyclic and carry a state. As all MPC protocols described in this thesis evaluate combinatorial circuits, their computational model is the combinatorial circuit model.

BOOLEAN GATES. Each gate in a Boolean circuit has one or multiple input wires and one output wire, which can be input to many subsequent gates. Each gate in a circuit represents a Boolean function f_g, which maps k input bits to one output bit, i.e., $g(w_1, w_2, \ldots, w_k) : \{0,1\}^k \rightarrow \{0,1\}$. In this thesis, we will only consider gates with at most two input wires, namely *unary* and *binary* gates. The relevant unary gate is the NOT gate ($\neg$), while typical binary gates are AND ($\wedge$), OR ($\vee$), and XOR ($\oplus$). Moreover, we distinguish between linear[1] (e.g., XOR) and non-linear (e.g., AND, OR) gates, as they have different

1 The Boolean function represented by linear gates can be expressed by a linear combination of its inputs over $\mathbb{Z}_2$, e.g., $\mathrm{XOR}(X, Y) = X + Y \mod 2$.

evaluation costs in secure computation [KS08]. In some works on MPC, non-linear gates are also referred to as non-XOR gates.

NOTATION. For a function f, we refer to its circuit representation as C_f. We use s to denote the total number of gates in a circuit, also referred to as size, i.e., $s = size(C_f)$. Moreover, we use s^{nX} to count the number of non-linear (non-XOR) gates per circuit. We denote the depth of a circuit by d and use d^{nX} to denote its non-linear depth, which is the maximum depth of all gates connected to an output node, defined recursively for a gate g as:

$$d^{nX}(g) = \begin{cases} 0 & \text{if } g \text{ is an input node,} \\ d^{nX}(w_1) & \text{if } g \text{ is an unary gate with input } w_1, \\ max(d^{nX}(w_1), d^{nX}(w_2)) & \text{if } g \text{ with inputs } w_1, w_2 \text{ is linear,} \\ max(d^{nX}(w_1), d^{nX}(w_2)) + 1 & \text{if } g \text{ with inputs } w_1, w_2 \text{ is non-linear.} \end{cases}$$

Furthermore, we denote bit strings in lower-case letters, e.g., x. We refer to individual bits, which can be part of a bit string, by capital letters X and denote their negation with $\overline{X}$. We refer to a single bit at position i within a bit string as X_i. The Least-Significant Bit (LSB) is denoted with X_0. When writing Boolean equations, we denote AND gates with $\cdot$, OR gates with $+$ and XOR gates with $\oplus$. Moreover, when useful, we abbreviate the AND gate $A \cdot B$ with AB.

INTEGER REPRESENTATION. Integers are represented in binary form, as typical in computer science and logic synthesis. Hence, the decimal value of a bit string x is $\sum_{i=0}^{n-1} 2^i \cdot X_i$. This common representation has the advantage that arithmetic operations for unsigned binary numbers such as addition, subtraction, and multiplication can be reused for signed numbers. In the two's complement, negative numbers are represented by flipping all bits and adding one: $-x = \overline{x} + 1$. In the following chapters, we assume a two's complement representation when referring to negative numbers.

FIXED AND FLOATING POINT REPRESENTATION. To represent real numbers, we use two representations. The fixed point representation has a fixed position of the radix point (the decimal point). Hence, a binary bit string is split into an integer part and a fractional part. The decimal value of a bit string in fixed point representation is computed as $\sum_{i=0}^{n-1} 2^{i-r} \cdot X_i$, where r is the position (counted from the LSB) of the radix point. The IEEE-754 floating point representation [Soc85] is the standard representation of floating point numbers. It divides a bit string in three components, namely sign s,

significant m and exponent e. Its real value is determined by computing $(-1)^s \cdot m \cdot 2^e$. In the IEEE-754 standard, the bit-width of each component, as well as their range is determined. Moreover, various error handling behavior is specified, e.g., overflow or division by zero [Gol91].

2.2 SECURE MULTI-PARTY COMPUTATION

Secure Multi-Party Computation (MPC), also referred to as *secure computation*, has been proposed in the 1980s as a more theoretical construct [Yao82]; it only became practical in the last fifteen years. MPC protocols are cryptographic protocols run between two or more parties that allow to perform a joint computation of a functionality $f(x_1, x_2, \dots)$ over the private inputs $x_1, x_2, \dots$ of the participating parties $P_1, P_2, \dots$ with guaranteed *correctness* and *privacy*. Privacy in MPC is understood as the guarantee that participating parties do not learn more about the other party's inputs than they could already derive from the observed output of the joint computation. Thus, informally speaking, MPC allows collaboration among mutually distrusting parties by realizing a virtual trusted party that receives the input of all parties, computes the functionality, and returns the output to the parties.

In MPC different adverserial models are distinguished. Most relevant is the *semi-honest* (passive) model, where the adversary tries to learn as much from the protocol execution as possible, yet does not deviate from the protocol itself. This is opposed to the *malicious model*, where the adversary is allowed to actively violate the protocol. Examples of further adversarial models are the covert model by Aumann and Lindell [AL07] or the dual execution approach that leaks at most one bit by Huang et al. [HKE12]. Examples for further security properties of MPC protocols that are currently mostly of theoretic interest are adaptive security (i.e., the adversary is allowed to choose its input x while running the protocol), fairness (i.e., either all parties receive the output of the computation or no party learns a single bit), guaranteed output delivery, and robustness [CDN15].

As we focus on compilation in this thesis, we only give a high-level description of three MPC protocols, where each is a representative for a different class (in their application description and cost model) of MPC protocols. For simplicity and explanatory reasons, we focus on two-party protocols secure in the *semi-honest* adversary model, as these are already sufficient to follow the ideas presented in this thesis. Typically, MPC protocols in the semi-honest model are building blocks for protocols secure against malicious adversaries and are also used for many privacy-preserving applications on their own. We describe the two most prominent MPC protocols over Boolean circuits, namely *Yao's Garbled Circuits* and the *GMW* protocol. Moreover, in

Chapter 7, we study the compilation of hybrid protocols that consist of multiple protocols, which involve Yao's protocol, GMW, as well as an *additive secret sharing* based protocol. Therefore, we also give an introduction in MPC based on additive secret sharing.

We begin with an introduction into Oblivious Transfer, which is a building block used for many MPC protocols and also used by the protocols described in the remainder of this section.

2.2.1 *Oblivious Transfer*

Oblivious Transfer (OT) [Rab81; Kil88] is one of the most important building blocks for MPC protocols. An Oblivious Transfer protocol is a protocol in which a sender transfers one of multiple messages to a receiver, but it remains oblivious to the sender which message has been selected and retrieved by the receiver. At the same time, the receiver is only able to learn the content of the selected message, yet not anything about the other messages offered by the sender.

In this thesis, we mostly use 1-out-of-2 OTs, where the sender inputs two l-bit strings m_0, m_1 and the receiver inputs a bit $c \in \{0, 1\}$. At the end of the protocol, the receiver obliviously receives m_c such that neither the sender learns the choice c, nor the receiver learns anything about the other message m_{1-c}.

In a classic result, Impagliazzo and Rudich [IR89] showed that OT cannot be constructed from one-way functions in a black-box manner. Moreover, as non-black-box constructions have also not been presented, it was assumed that OT can only be constructed from asymmetric cryptography and thus was assumed to be very costly. However, in 2003 Ishai et al. [Ish+03] presented the idea of *OT Extension*, which significantly reduces the computational costs of OTs for most interesting applications of MPC. In OT Extension, a constant number κ, i.e., the security parameter (e.g., $\kappa = 128\,\text{bit}$), of OTs are realized using a traditional OT protocol based asymmetric encryption and referred to as Base OTs. These Base OTs establish a symmetric key of length κ that can subsequently be used to execute a larger number of OTs with comparably cheap symmetric cryptography. Since Base OTs only need to be established once between two parties, similar to a key exchange protocol, their costs becomes negligible through amortization in many applications. Finally, we remark that variants of OT have been proposed that are beneficial for MPC, such as *correlated* or *random OT* [Ash+13]. As a consequence, recent implementations of OT Extension are capable of performing multiple millions of OTs per second on a single core [Ash+13; KOS15].

2.2.2 *Yao's Garbled Circuits Protocol*

Yao's garbled circuits protocol, proposed in the 1980s [Yao86], is the most studied secure two-party computation protocol, secure in the semi-honest model. The protocol is run between two parties P_0, P_1 and operates on functionality descriptions in form of Boolean circuits over binary Boolean gates. To securely evaluate a functionality $f(x, y)$ over their private inputs x and y, both parties agree on a circuit $C_f(x, y)$, which can be seen as the machine code for the protocol.

During protocol execution, one party becomes the *circuit generator* (the garbling party), the other the *circuit evaluator*. The generator initializes the protocol by assigning each wire w_i in the circuit two random labels w_i^0 and w_i^1 of length κ (the security parameter), representing the respective Boolean values 0 and 1. For each gate the generator computes a *garbled truth table*. Each table consists of four encrypted entries of the output wire labels w_o^γ. These are encrypted according to the gate's Boolean functionality $\gamma = g(\alpha, \beta)$ using the input wire labels w_l^α and w_r^β as keys. Thus, an entry $gtt_o^{\alpha\beta}$ in the table is encrypted as

$$gtt_o^{\alpha\beta} = E_{w_l^\alpha}(E_{w_r^\beta}(w_o^{g(\alpha,\beta)})).$$

After their creation, the garbled tables are randomly permuted and sent to the evaluator, who, so far, is unable to decrypt a single row of any garbled table due to the random choice of wire labels. To initiate the circuit evaluation, the generator sends her input bits x in form of her input wire labels to the evaluator. Moreover, the wire labels corresponding to the evaluator's input y are transferred via an OT protocol, with the generator being the OT sender, who inputs the two wire labels, and evaluator being the OT receiver, who inputs its input bits of the computation. After the OT step, the evaluator is in possession of the garbled circuit and one input label per input wire. With this information the evaluator is able to iteratively evaluate the circuit by decrypting a single entry of each garbled table, starting from input wires and ending at the output wires. Due to the use of a double-encryption scheme, the evaluator is unable to decrypt more than one entry in each garbled table. Once all gates are evaluated, all output wire labels are known to the evaluator. In the last step of the protocol, the generator sends an output description table to the evaluator, containing a mapping between output label and actual bit value. The decrypted output is then shared with the generator. Note that this procedure can be adapted to provide different outputs to the two parties. Alternatively, the generator can encrypt the cleartext output bits in the garbled tables of all gates connected to output wires, which allows to reveal the output of the computation to the evaluator without further communication.

Selective security of Yao's Garbled Circuits in the semi-honest model has been proven by Lindell and Pinkas [LP09] and recently also adap-

tive security has been proven by Jafargholi and Wichs [JW16]. Yao's protocol has also been extended to be secure in the malicious model in multiple works, e.g., [Hua+14; HKE13; LP15; Lin16].

IMPLEMENTATIONS AND OPTIMIZATIONS. Yao's original protocol has seen multiple optimizations in the recent past. Most important are *point-and-permute* [BMR90; Yao86], which allows an efficient permutation of the garbled table such that only one entry in the table needs to be decrypted by the evaluator, *garbled-row-reduction* [NPS99], which reduces the number of ciphertexts that are needed to be transferred per gate, *free-XOR* [KS08], which allows to evaluate linear gates (XOR/XNOR) essentially for 'free' without any encryption or communication costs, and finally the communication optimal *half-gate* scheme [ZRE15], which requires only two ciphertexts per non-linear gate while being compatible with free-XOR. Important for practical implementations is the idea of *pipelining* [Hua+11b], which allows a parallel circuit generation and evaluation and which is necessary for the evaluation of larger circuits and a faster online execution of Yao's protocol, as well as the idea of *fixed-key garbling* [Bel+13], which allows to achieve garbling speeds of more than 10 million gates per second on a single CPU core using the AES instructions of modern CPUs.

COST MODEL. Considering all optimizations mentioned above, then for a given Boolean function $f(x, y)$ and its circuit representation $C_f(x, y)$, the evaluation costs of a circuit in Yao's protocol are dominated by the *number of the input bits*, and by the *number of non-linear gates in the circuit*. In an amortized setting, the input gates are transferred via OT Extension. Current implementations achieve a speed of more than 10 million OTs per second and require a bandwidth of two ciphertexts per OT. The garbling and evaluation costs of linear gates are negligible for most applications, as communication is the current practical bottleneck of Yao's protocol. To garble a non-linear gate, two entries from the garbled table of length κ each have to be transferred. Assuming a standard key length of $\kappa = 128$ bit, then a garbling throughput of 10 million non-linear gates per second produces $\approx 2.8\,\mathrm{Gbit}$ of data per second. Considering that the communication requirement of two ciphertexts per gate is a proven lower bound for known garbling schemes [ZRE15], minimizing the number of non-linear gates in a circuit is an important optimization goal when compiling applications into circuits for Yao's protocol. We remark that this optimization goal holds for almost all MPC protocols over Boolean circuits, as these have a mechanism similar to free-XOR, which allow the evaluation of linear gates without communication.

2.2.3 *Goldreich-Micali-Wigderson (GMW) Protocol*

The *Goldreich-Micali-Wigderson (GMW)* protocol [GMW87] also originates from the 1980s and allows two parties to securely compute a functionality described in form of a Boolean circuit. In contrast to Yao's protocol, which uses the idea of garbled tables, GMW is built on top of Boolean secret sharing. Each input and intermediate wire value is shared among the two parties using an XOR based sharing scheme over single bits, i.e., for every value $V \in \{0,1\}$ each party $P \in \{0,1\}$ holds a share V^P, indistinguishable from random, with $V = V^0 \oplus V^1$. Values can be revealed at the end of the computation by exchanging and XORing the shares.

Given shared values, XOR gates can be evaluated locally without any communication between the parties by XORing the respective wire shares, e.g., both parties compute $W^P = U^P \oplus V^P$ over their shares of U and V to compute shares of $W = U \oplus V$. An AND gate $Z = X \cdot Y$ requires the parties to run an interactive protocol. One approach is to perform an 1-out-of-4 OT protocol [SZ13], where the chooser inputs its share X^0 and Y^0, and the sender prepares the output accordingly. Thus, the sender samples a random Z^1 and computes its input, i.e., different Z^0's, into the OT protocol according to

$$Z^0 = Z^1 \oplus ((X^0 \oplus X^1) \cdot (Y^0 \oplus Y^1)),$$

such that $Z^0 \oplus Z^1 = 1$ iff $X^0 \oplus X^1 = 1$ and $Y^0 \oplus Y^1 = 1$.

The same functionality can be realized in a pre-processing phase, which is independent of the functionality to be computed and which enables a very fast online phase that only requires the computation of a few bit-wise operations. For this purpose Boolean *multiplication triples* [Bea92] are used. A multiplication triple consists of three bits A, B, and C, where $C = A \cdot B$ holds, that are shared between the parties. Given such a triple, the two parties compute an AND gate $Z = X \cdot Y$, by opening (reconstructing) two bits $E = A \oplus X$ and $F = B \oplus Y$ and computing their shares of Z as

$$Z^P = P \cdot E \cdot F \oplus F \cdot A^P \oplus E \cdot B^P \oplus C^P.$$

The multiplication triples can be generated in a preprocessing phase using the 1-out-of-4 OT protocol described above.

IMPLEMENTATIONS AND OPTIMIZATIONS. A first implementation was given by Choi et al. [Cho+12] that was subsequently improved by Schneider and Zohner [SZ13]. Furthermore, Asharov et al. [Ash+13] showed that the triples can be precomputed using only two random OTs with a total communication of 2κ bits, where κ denotes the key length in bits. A protocol secure against malicious adversaries, named TinyOT, was proposed by Nielsen et al. [Nie+12].

COST MODEL. Since the multiplication triples, which are the most expensive part of the computation, can be precomputed, only four bits (two per party) for every AND gate need to be communicated in the online phase. Moreover, we observe that the computation effort of AND and XOR gates is negligible in practice when compared to the communication costs. Thus, when considering a typical bandwidth ($\leqslant$ 1 Gbit) constrained deployment scenario, GMW outperforms Yao's Garbled Circuits in gate throughput significantly, as only a fraction bits per gate have to be transmitted. GMW is a multi-round protocol, where the number of rounds is depending on the non-linear circuit depth. All AND gates on the same layer in the circuit can be computed in parallel and in the same communication round. Input sharing is significantly cheaper than in Yao's Garbled Circuits, as only one bit per input wire has to be transmitted. We note that the computation and communication efforts in the preprocessing phase are comparable to the garbling cost of a gate in Yao's Garbled Circuits [DSZ15; Des+17].

In summary, the performance of the GMW protocol for a given circuit C_f is dominated by the *total number of non-linear gates* s^{nX} as well as *the non-linear depth* d^{nX} of the circuit (number of layers of AND gates), whereas the number of inputs and outputs marginally influences the performance of the GMW protocol. This cost model is prototypic for multi-round MPC protocols over Boolean circuits.

2.2.4 *MPC Based on Additive Secret Sharing*

In the later part in this thesis we make use of an MPC protocol based on additive linear secret sharing that evaluates Arithmetic circuits consisting of only addition and multiplication gates. Many other MPC protocols with different secret sharing schemes have been proposed, and we refer the reader to the book of Cramer et al. [CDN15] for a detailed introduction into the topic of MPC and secret sharing.

In this thesis, we make use of the two-party MPC protocol described in [Ata+04; DSZ15], where an l-bit value is shared additively in the ring $\mathbb{Z}_{2^l}$. Thus, for every value $v \in \mathbb{Z}_{2^l}$ in the protocol, each party $P \in \{0, 1\}$ holds a share v^P with $v = v^0 + v^1$ (this and all following computations are performed modulo 2^l). To enter a new value x into the protocol, i.e., to share a value, party P samples a random $x^P \in \mathbb{Z}_{2^l}$, computes $x^{1-P} = x - x^P$, and sends x^{1-P} to the other party. To output a shared value x, the parties send their own share x^P to the other party and locally compute $x = x^0 + x^1$. Additions can be performed locally by adding the respective shares. Thus, to compute $z = x + y$ over the shares of x and y, both parties compute $z^P = x^P + y^P$. Multiplications are performed with an interactive protocol, which can be precomputed using multiplication triples [Bea92; Gil99]. To compute $z = x \cdot y$ over shared values x and y, a multiplica-

tion triple $c = a \cdot b$ is used, such that both parties compute and then reveal $e^P = x^P - a^P$ and $f^P = y^P - c^P$. Given e^P and f^P, both parties compute their share z^P of z as follows:

$$z^P = P \cdot e \cdot f + f \cdot a^P + e \cdot b^P + c^P.$$

Multiplication triples can be generated in the preprocessing phase, independently of the computed functionality with the help of homomorphic encryption [Ata+04] and additive blinding. Alternatively, multiplication triples can be generated by performing a bit-wise OT based protocol [Gil99].

IMPLEMENTATIONS. An implementation of the protocol described above is given by Demmler et al. [DSZ15]. Alternative implementations of MPC protocols based on additive linear secret sharing are Sharemind [BLW08], which provides the richest set of functionalities, VIFF [Dam+09], and the SPDZ [Dam+12] implementation [KSS13] as well as FRESCO [Dam+16], which both provide security against malicious adversaries.

COST MODEL. The communication and computation costs in additive secret sharing based MPC is dominated by the *number of multiplication gates* and the *multiplicative depth* of the circuit. Similar to the Boolean circuit based protocols, linear operations, i.e., additions, are for free in the number of cryptographic operations and in communication. Multiplications have a small online cost, as only the respective shares have to be transmitted and only cheap local operations are performed. Moreover, similar to GMW, every multiplicative layer in the circuit increases the number of communication rounds. The precomputation phase is the most expensive part of the protocol execution. For the generation of the multiplication triples, either multiple (packed) encryptions in a homomorphic encryption systems or $O(l)$ OTs of length l have to be performed.

2.2.5 *Hybrid Protocols*

The efficiency of applications generated by compilers for all aforementioned protocols is bound by the efficiency to represent elementary operations, also referred to as building blocks, in their respective cost model. Because of this, the protocols can have substantial difference in their runtime for the same application. For example, expressing multiplications over n bit integers in Yao's protocol or GMW requires Boolean circuits of size $O(n^2)$. Due to this reason, multiple applications based on hybrid protocols, i.e., a mix of multiple standalone MPC protocols, have been proposed to achieve better efficiency by exploiting multiple function representations, e.g., [BG11; Hua+11a; Nik+13a; SK11]. An application independent combination

of protocols has been studied in [BLR13; Hen+10; SKM11; KSS14]. Most of these works combine Yao's Garbled Circuits with homomorphic encryption, i.e., an asymmetric encryption scheme, which allows to perform operations on the ciphertexts that resemble arithmetic operations on the respective cleartexts, e.g., addition.

In this thesis, we evaluate hybrid protocols consisting of all three aforementioned protocols, which has been described by Demmler et al. [DSZ15]. To combine multiple sequentially aligned MPC protocols into a hybrid protocol, an intermediate state has to be converted securely between the different protocols. Such an intermediate state consists of multiple values that are secretly shared between between the computing parties. In Yao's protocol, the wire label computed by the evaluator and the mapping of wire label to cleartext value form a sharing of one bit. In the two other protocols, the sharing is more straight-forward, namely either an XOR (Boolean) sharing is used or an additive (Arithmetic) sharing. The authors proposed conversion protocols that securely transform one sharing into the other. For example, a Yao sharing can be converted into a Boolean sharing at practically no cost by using the point-and-permute bits (cf. Section 2.2.2).

Unfortunately, other conversions require communication and computation. For example, the conversion from an n bit Arithmetic sharing into n Yao sharings or n Boolean sharings can be realized by evaluating a Boolean addition circuit. Thus, both parties have to enter the bit decomposition of their arithmetic shares as input into either GMW or Yao's protocol and then evaluate an addition circuit. A more detailed explanation of these conversion protocols and their costs is given in [DSZ15].

2.3 OBLIVIOUS RAM AND RAM-SC

In Chapter 8, we will present a compilation approach for RAM based MPC, which is often referred to as RAM-SC. In this section we give an overview of relevant ORAMs used in RAM-SC, before describing RAM-SC itself.

2.3.1 *Oblivious RAM (ORAM)*

Oblivious RAM (ORAM), first introduced by Goldreich and Ostrovsky in the context of software protection and simulation of RAM programs [GO96], is a cryptographic primitive that allows to obfuscate the access pattern to an outsourced storage to achieve *memory trace obliviousness*. Therefore, each logical access on some virtual address space is translated into a sequence of physical accesses on the memory, which appears to be random to observers, resulting in the security guarantee that two sequences of virtual accesses of the same length produce indistinguishable physical access patterns.

ORAMs are commonly modeled as protocols between an ORAM client, who is the data owner, and an untrusted ORAM server, who provides the physical storage. Typically, an ORAM construction is comprised of two distinct algorithms, the initialization and the access algorithm. Thereby, the initialization algorithm introduces a new oblivious structure for a given number of elements, while the access algorithm performs an access to one of the elements in the virtual memory space using a sequence of accesses on the physical memory, hiding the accessed virtual index and also the type of operation, i.e, read or write. An ORAM has a capacity m, which describes the number of data elements it can store. Moreover, most ORAMs require to store metadata for each data element, which in combination with the element itself is referred to as *block*.

The design goals of standalone ORAM constructions are manifold, e.g., minimizing client side storage, communication or computation costs. Therefore, many optimized ORAMs have been proposed, e.g., [GO96; KLO12; LO13; Ste+13]. For their combination with MPC (described next), a different cost model applies, because the ORAM client has to be evaluated as a circuit. In this thesis, we study the most efficient known ORAMs for MPC, namely Circuit-ORAM (C-ORAM) [WCS15], optimized Square-Root ORAM (SQ-ORAM) [Zah+16], FLORAM [Ds17a], and the FLORAM variant CPRG (FCPRG) [Ds17a].

CIRCUIT ORAM (C-ORAM). C-ORAM by Wang et al. [WCS15] is an MPC-optimized derivative of Path ORAM [Ste+13], which is the most practical known standalone ORAM to date. C-ORAM is a tree-based ORAM that stores m data elements in a binary tree structure of at most $\log_2(m)$ stages and an additional root level called *stash*, which can also be imagined as a cache. Each node in the tree is a smaller ORAM itself, called *bucket ORAM*, that stores multiple blocks. Each data element is randomly mapped to one of the leaf nodes, maintaining the invariant that an element with leaf identifier l is contained in one of the buckets on the path from the root node to leaf l or in the stash. To read or write an element in tree-based ORAMs, the according path from root to the leaf is read, a new leaf identifier for the read element is chosen, and the accessed path is moved to the stash. Moreover, after each access an *eviction* procedure is initiated, which writes blocks from stash into the tree while moving blocks as close as possible to their designated leaves.

C-ORAM requires a recursive position map in RAM-SC, which associates the virtual index of each element with the position in the tree. Henceforth, a tree-based ORAM has several construction parameters, including the size of the bucket ORAMs B, the number of recursive steps r, and a packing factor c describing the number of mappings contained in one block of the recursive ORAMs.

SQUARE-ROOT ORAM (SQ-ORAM). SQ-ORAM was one of the two ORAMs introduced in the seminal paper by Goldreich and Ostrovsky [GO96] and later optimized for MPC by Zahur et al. [Zah+16]. SQ-ORAM uses a fundamentally different strategy than Path ORAM. Its core idea is to randomly permute the memory and to periodically refresh this permutation. For m elements, the so-called *permuted memory* has size $m + \sqrt{m}$, Furthermore, a shelter/stash for $\sqrt{m}$ elements is used. The simulation of a RAM program takes place in so called epochs of $\sqrt{m}$ steps, consisting of three phases: In a first step the memory is obliviously permuted using a permutation π that assigns each element a position in the permuted memory by using random tags assigned to each element. Afterwards, $\sqrt{m}$ virtual accesses can take place, during which the updated values are written to the shelter. To access an element at index v, first the entire shelter is scanned. If the element cannot be found, the permuted memory is accessed to retrieve the element at position $\pi(v)$, otherwise, the element has previously been visited and can thus be found in the shelter, and a dummy access to the permuted memory is performed. As last step of an epoch, the permuted memory is updated according the shelter.

Zahur et al. [Zah+16], removed the use of Pseudo Random Functions (PRFs) (needed for the permutation), dummy elements, expensive oblivious sorting algorithms, and identified public metadata. Furthermore, they introduced the usage of recursive maps to compute the mapping between virtual and physical addresses.

FLORAM. FLORAM, recently introduced by Doerner and shelat [Ds17a], differs from the other ORAM constructions as it is built from Function Secret Sharing (FSS), introduced by Boyle et al. [BGI15]. FLORAM is a *distributed* ORAM [Ds17b; LO13], where the data is stored in a secret shared manner (XOR) between two servers. Elements are accessed using Private Information Retrieval (PIR) techniques, i.e., a short query is evaluated on all elements on the server to extracted the desired element. A point function $f_{\alpha,\beta}(x)$ evaluates to β if $x = \alpha$ and 0 otherwise. Informally, FSS allows to share a Distributed Point Function (DPF) in such a way, that the parties can evaluate the point function on arbitrary input, yet neither learn α nor β. This feature allows the client to send the servers specially crafted queries $q^0(i)$ or $q^1(i)$ using FSS to retrieve or to write an element at index i.

FLORAM distinguishes a read-only memory (OROM) and write-only memory (OWOM). Data in both is stored using an XOR sharing, yet in OROM, each share is additionally masked using a PRF with a key known only to the storing party. After a number of write accesses, the OWOM memory is converted into the OROM. This process is referred to as *refresh*. To read an index i, the client shares a DPF that evaluates to 1 on input i and to 0 otherwise. The servers evaluate

the DPF on all indices, multiply the result with each associated element share and return their aggregated result to the client. Writing is performed using a similar approach. A conversion between OWOM and OROM is too expensive to be performed after every write access, therefore a stash is used that functions as a cache between refreshes. The stash is scanned when performing read accesses to identify updated elements.

FLORAM CPRG (FCPRG). FCPRG [Ds17a] is an extension to FLORAM. The most expensive computation on the client side of FLORAM is the creation of the FSS scheme, which requires to compute $2 \cdot \log_2(m)$ PRFs for every access. In MPC the evaluation of the PRFs can render the scheme expensive and therefore, with FCPRG the authors propose to compute the PRFs locally, i.e., outside of MPC, with the trade-off that $O(\log_2(m))$ interactions between the computing parties are required.

2.3.2 *RAM based MPC (RAM-SC)*

MPC protocols evaluate functionalities represented as circuits. Circuits allow to express arbitrary computations, yet random memory accesses have to be expressed as a chain of multiplexers of the complete memory, referred to as linear scan (LS). This limits MPC for applications that rely on dynamic memory accesses. Therefore, Gordon et al. [Gor+12] proposed to combine MPC protocols with ORAM to enable dynamic memory accesses with sublinear overhead. The authors describe a RAM machine, where the circuit computes an oblivious machine that evaluates instructions and memory accesses. A complete RAM machine is often not necessary, and thus the so-called *RAM-SC* model was later refined by Liu et al. [Liu+14] for practical efficiency. Its major concepts are described in the following paragraphs.

First, the parties performing the MPC protocol also act as distributed ORAM server, and the ORAM client is implemented as circuit evaluated by the MPC protocol itself. Thus, both roles are shared between the computing parties. Intuitively, privacy is preserved because the MPC protocol acts as a virtual third party emulating the ORAM client. Second, a program is evaluated by interweaving the MPC protocol with ORAM accesses. Consequently, a RAM-SC program consists of many small protocols that either perform a computation or an ORAM access. This behavior is exemplary illustrated in Figure 1.

The construction of RAM-SC, as described, is very generic because it allows to combine different MPC protocols and ORAMs. When assuming composable ORAMs, we observe that in one RAM-SC program multiple ORAMs of possibly different type can be used, e.g., one ORAM for each array in the input program. Moreover, as in stan-

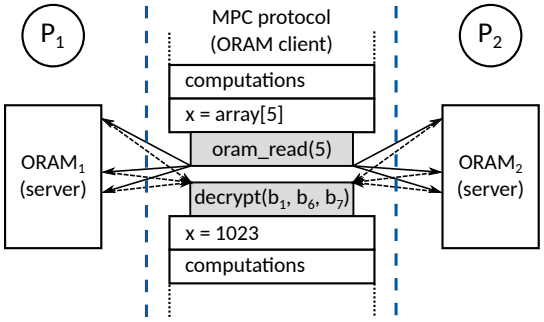


Figure 1: Exemplary and simplified illustration of RAM-SC. A program flow is illustrated that is computed within an MPC protocol, run between two parties P_1 and P_2. At some point, a value is read from an array with virtual index 5. Therefore, a circuit representing the ORAM client functionality is executed that translates the virtual index into multiple physical addresses. These addresses are revealed to both parties, who enter the blocks as input to the MPC protocol. Not shown here is, that often also an intermediate state has to be transferred between two MPC protocol.

dalone ORAMs, the blocks stored on the ORAM server have to be encrypted. This can be realized by performing an encryption and decryption within a circuit, which requires (even highly optimized) a substantial amount of gates, e.g., 5000 non-linear gates to encrypt a single block of 128 bits AES [BP12], using a secret sharing scheme, e.g., XOR sharing [Ds17a], or by soldering the existing garbled labels based on the publicly revealed index [Zah+16]. In the XOR sharing approach a physical block is read by entering the shares as input to the MPC protocol, which are then recombined within the protocol. Similarly, to write to one or multiple blocks, the MPC protocol outputs one share for each block to every party. When using the soldering approach, the circuit garbler re-uses the existing wire labels but remaps them according the accessed indices reveled to both parties, similar to a multiplexer (array) access with public index. We also remark that in RAM-SC, the ORAM access type, i.e., read or write, can be revealed to both parties, as the algorithm description is seen as public knowledge. This access type is also referred to as *semi-private access* [Ds17a].

SECURITY. RAM-SC provides the same privacy and correctness properties against semi-honest adversaries as traditional MPC protocols [Gor+12]. RAM-SC with security against malicious adversaries has been studied in [Afs+15].

COMPLEXITY. The computation and communication complexity of a RAM-SC protocol depends on the circuit complexity of the computation, the circuit complexity of the ORAM client, the number of protocol rounds, as well as additional ORAM protocol costs that are

performed outside of the MPC protocol. For ORAMs with less than $O(m)$ computations or less than $O(m)$ bandwidth RAM-SC is (asymptotically) more efficient than any circuit based MPC protocol.

2.4 COMPILERS FOR MPC

Due to the complexity of circuit design, multiple compilers for MPC have been proposed. In this section we give an overview and classification of these, which partially form the related work of this thesis.

COMPILER CLASSIFICATION. In general, compilers for MPC share many similarities with high-level synthesis tools [Gaj+12], known from digital circuit design. Nevertheless there are substantial differences between these tools and compilers for MPC, which will be studied in detail in the following chapters. Therefore, we restrict our comparison and classification to compilers that have explicitly been proposed for their use in MPC.

Compilers for MPC can be categorized based on whether they compile a (minimalistic) Domain Specific Language (DSL) or a widely used *common programming language*. Moreover, compilers can be *independent* or *integrated* into an MPC framework. Integrated compilers produce an intermediate representation, which is *interpreted* (instantiated by a circuit) only during the execution of an MPC protocol. These interpreted circuit descriptions commonly allow a more compact circuit representation. Independent compilers create circuits independent from the executing framework, and thus have the advantage that produced circuits can be optimized to the full extent during compile time and are more versatile in their use in MPC frameworks.

Especially for Arithmetic circuits it is often hard to make a strict separation. When compiling these circuits, compilers commonly target the so called Arithmetic Black Box (ABB) model [DN03], which is an abstraction between compiler and protocol implementation. The ABB model provides representations for values and operations on these, e.g., addition and multiplication, without providing details about their protocol implementation to the compiler. In principle, an ABB is Turing complete with only addition and multiplication. Yet, practical efficiency is only achieved if further functionalities, such as comparisons, are also provided. We consider an Arithmetic circuit compiler to be independent if it compiles towards an ABB model only consisting of elementary functions, involving comparisons and possibly floating point operations, yet not high-level functionalities such as sorting or statistical tests, as for example provided by Sharemind [BLW08].

Some integrated compilers support the compilation of *mixed-mode secure computation*. Mixed-mode computation allows to write code that distinguishes between oblivious (private) and public computa-

tion. Thus, the public computation is performed locally in the clear, interweaved by private computations insides an MPC protocol. This leads to an even tighter coupling between compiler and execution framework, but allows to express a mixed-mode program in a single language. Moreover, some integrated compilers support the compilation for *hybrid* secure computation protocols or RAM-SC. Next, we give an overview of Boolean circuit and Arithmetic circuit compilers.

BOOLEAN CIRCUIT COMPILERS. We begin by studying compilers targeting Boolean circuit based MPC protocols that use a DSL as input language. The Fairplay framework by Malkhi et al. [Mal+04] started research on practical MPC. Fairplay compiles a domain specific hardware description language called SFDL into a gate list for use in Yao's protocol. Following Fairplay, Henecka et el. [Hen+10] presented the TASTY compiler, which compiles its own DSL, called TASTYL, into an interpreted hybrid protocol. The PAL compiler by Mood et el. [MLB12] aims at low-memory devices as the compilation target. PAL also compiles Fairplay's hardware description language. The KSS compiler by Kreuter et al. [KSS12] is the first compiler that shows scalability up to a billion gates and uses gate level optimization methods, such as constant propagation and dead-gate elimination. KSS compiles a DSL into a flat circuit format. TinyGarble by Songhori et al. [Son+15] uses (commercial) hardware synthesis tools to compile circuits from hardware description languages such as Verilog or VHDL. On the one hand, this approach allows the use of a broad range of existing functionality in hardware synthesis, but also shows the least degree of abstraction by requiring the developer to be experienced in hardware design. We remark that high-level synthesis from C is possible, yet, as the authors note, this leads to significantly less efficient circuits. Recently, Mood et al. [Moo+16] presented the Frigate compiler, which aims at very scalable and extensively tested compilation of another DSL. Frigate and TinyGarble produce compact circuit descriptions that compress sequential circuit structures.

Examples for mixed-mode and hybrid compilers involving Boolean circuits from a DSL are L1, Wysteria, ObliVM, and Obliv-C. The L1 compiler by Schröpfer et al. [SKM11] compiles a DSL into a hybrid protocol involving homomorphic encryption. Wysteria by Rastori et al. [RHH14] creates Boolean circuits applied to GMW from a mixed-mode DSL that is supported by a strong formal calculus. ObliVM by Liu et al. [Liu+15b] extends SCVM [Liu+14], which both compile a DSL that support the combination of oblivious data structures with MPC. This approach allows the efficient development of oblivious algorithms. Yet, both compilers provide only very limited gate and source code optimization methods. The Obliv-C compiler by Zahur and Evans [ZE15] also supports oblivious data structures, but follows a different, yet elegant source-to-source translation approach by

Compiler	Language	Target Protocol	Interpreted	Mixed-mode	Maturity [Moo+16]	Code Level Optimization	Gate Level Optimization
CBMC-GC'12	ANSI C	Yao	no	no	yes	global	dead gate elimination, constant propagation, theorem rewriting, SAT sweeping
KSS'12	DSL	Yao	no	no	no	no	constant propagation, dead gate elimination
PCF'13	ANSI C	Yao	yes	yes	no	no	-
Wysteria'14	DSL	GMW	yes	yes	n/a	limited	local constant propagation, dead gate elimination
OblivC'15	DSL ANSI C	Yao	yes	yes	no	local	local and limited [Moo+16] constant propagation, dead gate elimination
ObliVM'15	DSL	RAM-SC	yes	yes	no	no	-
TinyGarble'15	Verilog VHDL	Yao & GMW	no	no	yes	n/a	dead gate elimination, constant propagation, rewriting, SAT sweeping
Frigate'16	DSL	Yao	no	no	yes	local	local constant propagation, dead gate elimination
CircGen'17	ANSI C	Yao	no	no	verified	global	dead gate elimination, constant propagation

Table 1: Comparison of recent Boolean circuit compilers for MPC. Compared is the source *language*, whether they compile to complete circuit or an intermediate representation that is *interpreted* during runtime, whether they support *mixed-mode* MPC, their *maturity* (compilation correctness) based on the study by Mood et al. [Moo+16], and the applied optimization techniques on the *source-code level* (constant propagation and constant folding) and on the *gate level* (logic minimization).

compiling a modified variant of C into efficient mixed-mode applications. Very recently, EzPC [Cha+17] has been presented that compiles mixed-mode hybrid protocols involving Boolean and Arithmetic circuits.

The CBMC-GC compiler [Hol+12] is the first compiler that creates circuits for MPC from a common programming language (ANSI C). CBMC-GC follows the independent compilation approach and produces a single circuit description. Moreover, CBMC-GC applies source code optimization and provides a powerful symbolic execution. This compiler is extended throughout this thesis. The PCF compiler by Kreuter et al. [Kre+13] also compiles C, using the portable LCC compiler as a frontend. PCF compiles an intermediate bytecode representation given in LCC into a interpreted circuit format. PCF shows greater scalability than CBMC-GC, yet only supports comparably limited optimization methods that are only applied locally for every function. A (partially) formally verified compiler, named CircGen, has been proposed by Almeida et al. [Alm+17] that extends the Compcert compiler [Ler06] for ANSI C by a new backend that compiles Boolean circuits.

A detailed overview of Boolean circuits compilers for MPC is given in Table 1.

ARITHMETIC CIRCUIT COMPILERS. The Viff compiler [Dam+09] jointly with the proposition of a DSL for MPC [NS07] are two of the first works in the direction of compilation for Arithmetic circuit based MPC. The most prominent compiler is the Sharemind compiler by Bogdanov et al. [BLW08], which provides a production ready mixed-mode compiler suite (and protocol framework) for a DSL called SecreC. The PICCO compiler by Zhang et al. [ZSB13] implements a mixed-mode programming environment for C that supports parallelization. The Armadillo compiler by Carpov et al. [CDS15] and the ALCHEMY compiler by Crockett et al. [CPS18] target the creation of Arithmetic circuits for homomorphic encryption. Recently, also an integrated compiler for SPDZ has been proposed [Spd].

2.5 BENCHMARKING APPLICATIONS FOR MPC COMPILERS

MPC has many applications, e.g., privacy-preserving biometric authentication [Erk+09], private set intersection [FNP04], or secure auctions [Bog+09]. For research purposes, some (parts of) these applications have become popular for benchmarking purposes. The most prominent example is the AES block cipher, which is the de facto standard benchmark for MPC protocols. In compiler research on MPC, multiple different functionalities are commonly compiled into circuits, which are then compared regarding their properties, i.e., size, depth and fraction of non-linear gates.

In this section, we give an overview of these commonly benchmarked functionalities, ranging from very small snippets, e.g., integer addition, to larger functionalities, e.g., IEEE-754 compliant floating point operations or biometric authentication. In later chapters, we use these functionalities to evaluate the techniques presented in this book. Here, we discuss the functionalities together with their complexity and motivate why these functionalities are relevant in the context of MPC and the development of compilers.

Common benchmark functionalities are:

- *Integer arithmetic.* Due to their (heavy) use in almost every computational problem, arithmetic building blocks are of high importance in high-level circuit synthesis and are often benchmarked individually. Most common building blocks are addition, subtraction, multiplication, and division. When studying these building blocks, one should distinguish the results for different input and output bit widths. Namely, arithmetic operations can be differentiated in overflow-free operations, e.g., allocating $2n$ bits for the result of a $n \times n$ bit multiplication, and operations with overflow, e.g., when only allocating n bits for the same multiplication. In later chapters we explicitly state which input and output bit-widths are studied. To evaluate the scalability of compilers, multiple sequentially (in-)dependent arithmetic operations can be compiled as a single functionality.

- *Matrix multiplication.* Algebraic operations, such as matrix or vector multiplications, are building blocks for many privacy-preserving applications, e.g., for feature extractors in biometric matching or the privacy friendly evaluation of neural networks, and have therefore repeatedly been used to benchmark compilers for MPC [Dem+15; Hol+12; Kre+13]. From an optimizing compilers perspective, matrix multiplication is a comparably simple task, as it only involves the instantiation of arithmetic building blocks. However, it is very well suited to show the scalability of compilers or the capability to fuse multiple arithmetic operations into single statements, as required for depth minimization (see Chapter 6). Matrix multiplication can be parametrized according the matrix dimensions, the used number representation (integer, fixed or floating point) and its bit-width.

- *Modular exponentiation.* Modular exponentiation, i.e., $x^y \bmod p$, is relevant in secure computation, as it is a building block for various cryptographic schemes. For example, blind signatures can be realized with RSA using modular exponentiation. Blind signatures allow a signing process, where neither the message is revealed to the signing party, nor the signing key is revealed to the party holding the message. Therefore, modular exponen-

tiation has been used multiple times to study the performance of MPC [Bel+13; DSZ15; KSS12].

- *Distances.* Various distances need to be computed in many privacy preserving protocols and are therefore relevant for MPC, e.g., in biometric matching applications or location privacy applications. The *Hamming* distance is a measure between two bit strings. Measured is the number of pairwise differences in every bit position. Due to its application in biometrics, the Hamming distance has often been used for benchmarking MPC compilers, e.g., [Hol+12; KSS12; Moo+16]. The *Manhattan* distance

  $$\text{dist}_{\text{MH}} = |x_1 - x_2| + |y_1 - y_2|$$

  between two points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the distance along a two dimensional space, i.e., along the Manhattan grid, when only allowing horizontal or vertical moves. The *Euclidean* distance is defined as

  $$\text{dist}_{\text{ED}} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

  Due to the complexity of the square root function, it is common in MPC to benchmark the squared Euclidean distance separately, as its computation is usually sufficient when comparing multiple distances. All distances can be parameterized according to the used input bit-widths.

- *Biometric matching.* In biometric matching a party matches one biometric sample (a vector of features) against the other's party database of biometric templates. Example scenarios are face-recognition or fingerprint-matching [Erk+09]. One of the main concepts is the computation of a distance (see above) between a sample and all database entries. Once all distances are computed, the minimal distance determines the best match. The biometric matching application is very interesting for benchmarking, as it involves many parallel arithmetic operations, as well as a large number of sequential comparisons. The biometric matching application can be parameterized by the database size, the sample dimension (number of features per sample), the bit-width of each feature, and the used distance.

- *Database analytics.* Performing data analytics on sensitive data has numerous applications and therefore many privacy-preserving protocols have been studied, e.g., [Bog+15; DHC04]. Using generic MPC techniques is of special interest in analytics, as it allows to perform arbitrary computations, e.g., hypothesis testing, or allows to add data perturbation techniques, e.g., differential privacy, before releasing the result with minimal effort. This task can be parametrized according the database size(s) and the applied analyses.

- *Fixed and floating point arithmetics.* Fixed and floating point number representations allow the computation on real numbers using integer arithmetic. Therefore, these representations are necessary for all applications where numerical precision is required, e.g., in privacy preserving statistics. The floating point representation is the more versatile representation, as it allows to represent a larger range of values, whereas fixed point arithmetic can be realized with significant less costs and is therefore of interest in MPC. Floating point operations are also very suited to evaluate the gate level optimization methods of compilers since they require many bit operations. When implementing floating point arithmetic we follow the IEEE-754 standard.

- *Gaussian elimination.* Solving linear equations is required in many applications with Gaussian elimination being the most studied solving algorithm. Due to its wide range of use, it is also of relevance in privacy-preserving applications. In this thesis we evaluate our compiler using a textbook Gauss solver with partial pivoting. The task can be parameterized by the number of equations.

- *Location-aware scheduling.* Privacy-preserving availability scheduling is another application that can be realized with MPC [Bil+11]. The functionality matches the availability of two parties over a number of time slots, without revealing the individual schedule to the other party. Location-aware scheduling also considers the location and maximum travel distance of the two parties for a given time slot. Therefore, the functionality outputs a matching time slot where both parties are available and in close proximity to each other (if existent). The functionality can be parameterized by the number of time slots used per party, the representation of locations, and the distance measure, e.g., Manhattan or Euclidean distance.

- *Machine learning.* Machine learning (ML) with its distinction in super- and unsupervised learning techniques has many applications and is a very active field of research. Therefore, protecting the privacy of training data or ML inputs is also an active research area.

  Supervised machine learning – Neural networks. One of the most powerful ML techniques are Convolutional Neural Networks (CNNs). Because of this, many dedicated protocols for private data classification using CNNs have been proposed, e.g., [Gil+16; Liu+17; Ria+18]. Computationally, CNNs consists of multiple matrix multiplications concatenated with convolutions and non-linear activation functions.

Unsupervised machine learning – k-means. Clustering is another ML task, frequently used to identify centroids in unstructured data. One of the most well known clustering algorithms is k-means, and multiple works proposed dedicated privacy-preserving k-means protocols, e.g., [JW05; VC03]. The k-means algorithm can be parametrized according to the number of data points, the number of clusters, and the number of iterations to perform the algorithm.

- *Median computation.* The secure computation of the median is required in privacy preserving statistics. It is an interesting task for compilation, as it can be implemented using a sorting algorithm, because a sorted array allows a direct access to the median element. The compiled circuit depends on the used sorting algorithm. In this book, we evaluate Bubble and Merge sort. Moreover, the task can be parameterized according to the number of elements and their bit-width.

IMPLEMENTATION DIFFERENCES. We remark that for a fair comparison between multiple compilers that compile the same functionality, it has to be made sure that the same algorithm as well as the same abstraction level of the source code is used. To illustrate this thought we give three example implementations for computing the *Hamming* distance, which produce circuits of largely varying sizes (cf., Section 4.5). The distance can be computed by XOR-ing the input bit strings and then counting the number of bits. An exemplary implementation is given in Listing 1 that computes the distance between two bit-strings of length 160 bit, which are split over five unsigned integers. In Line 7 the number of ones in a string of 32 bit is computed. This task is also known as population count. In the following paragraphs we describe three different implementations.

```
1  #define N 5
2  void hamming() {
3    unsigned INPUT_A_x[N];
4    unsigned INPUT_B_y[N];
5    unsigned res = 0;
6    for(int i = 0; i < N; i++) {
7      res += count_naive32(INPUT_A_x[i]^INPUT_B_y[i]);
8    }
9    unsigned OUTPUT_res = res;
10 }
```

Listing 1: Hamming distance computation between two bit strings.

The first implementation is given in Listing 2. In this naïve approach, each bit is extracted using bit shifts and the logical AND

operator before being aggregated.

```
unsigned char count_naive32(unsigned y) {
  unsigned char m = 0;

  for(unsigned i = 0; i < 32; i++) {
    m += (y & (1 << i)) >> i;
  }

  return m;
}
```

Listing 2: Counting bits using a naïve bit-by-bit comparison approach.

A variant of this implementation is given in Listing 3. Here, the bit string of length 32 is first split into chunks of 8 bit (`unsigned char`). The ones set in each chunk are then counted as described above.

```
unsigned char count_naive8(unsigned char c) {
  unsigned char m = 0;
  for(int i = 0; i < 8; i++) {
    m += (c & (1 << i)) >> i;
  }
  return m;
}

unsigned char count_tree32(unsigned y) {
  unsigned char m0 = y & 0xFF;
  unsigned char m1 = (y & 0xFF00) >> 8;
  unsigned char m2 = (y & 0xFF0000) >> 16;
  unsigned char m3 = (y & 0xFF000000) >> 24;

  return count_naive8(m0) + count_naive8(m1) + \
    count_naive8(m2) + count_naive8(m3);
}
```

Listing 3: Counting bits over unsigned chars in a tree based manner.

Finally, in Listing 4 a variant optimized for a CPU with 32 bit registers and slow multiplication is given. This implementation uses only 14 CPU instructions.

```
unsigned count_reg32(unsigned y) {
  unsigned x = y - ((y >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = (x + (x >> 4)) & 0x0f0f0f0f;
  x += x >>  8;
  x += x >> 16;
  return x;
}
```

Listing 4: Counting bits, optimized for a 32 bit register machine.

Fundamental implementation differences will inevitably lead to different compilation results. Therefore, all benchmarked functionalities in this book are implemented using the same algorithms, data structures and data types when using them for the comparison with other compilers even when using different input languages.

3 FROM ANSI C CODE TO BOOLEAN CIRCUITS

Summary: The practicality of Secure Multi-party Computation (MPC) is hindered by the difficulty to implement applications on top of the underlying cryptographic protocols. This is because the manual construction of efficient applications, which need to be represented as Boolean or arithmetic circuits, is a complex, error-prone, and time-consuming task. For the practical use of MPC, and thus the development of further privacy-enhancing technologies, compilers supporting common programming languages are desirable to provide developers an accessible interface to MPC.

In this chapter, we describe the translation approach that is followed by the Boolean circuit compiler CBMC-GC [Hol+12], which is based on the model checker CBMC [CKL04], and which compiles ANSI C into circuits ready for their use in MPC protocols. Throughout this thesis, we will extend the here presented tool-chain to create optimized circuits for MPC.

Remarks: This chapter is based in parts on Chapter 3 of our book – *"Compilation for Secure Multi-Party Computation"*, Niklas Büscher and Stefan Katzenbeisser, Springer Briefs in Computer Science 2017, ISBN 978-3-319-67521-3.

3.1 MOTIVATION AND OVERVIEW

When visualizing an MPC protocol as a hardware architecture that can execute programs (functionalities), then the program descriptions are Boolean circuits (e.g., for Yao's Garbled Circuits [Yao86]) or Arithmetic circuits (e.g., for the SPDZ protocol [Dam+12]). Consequently, a compiler for MPC protocols compiles a functionality written in a high-level language into an (optimized) circuit representation. The core focus of this thesis is the creation of optimized Boolean circuits that also allow developers without a professional background in computer security and hardware design to create efficient applications for MPC. All techniques presented in this thesis extend an existing compiler by Holzer et al. [Hol+12] that is named *CBMC-GC* and described in detail in this chapter.

Technically, CBMC-GC is based on the software architecture of the bounded model checker CBMC by Clarke et al. [CKL04], which verifies assertions in ANSI C source code. The basic idea of Bounded Model Checking (BMC) [Bie+99] is to search for a counterexample in all possible program executions whose length is bounded by some integer k. The BMC problem can efficiently be reduced to a propositional satisfiability problem (SAT), and can therefore be solved by SAT methods. CBMC transforms an input C program, i.e., a function f, including assertions that encode properties to be verified, into a

Boolean formula B_f which is then (dis-)proved by a SAT solver. The formula B_f is constructed in such a way that the Boolean variables reflect the manipulation of all memory bits by the program and the assertions in the program. Moreover, CBMC is *bit-precise*, i.e., the formula B_f encodes the actual memory footprint of the analyzed program under ANSI C semantics for a specified hardware platform. Thus, CBMC is essentially a compiler that translates C source code into Boolean formulas.

The code must meet some requirements, detailed in Section 3.2.1, so that this transformation is possible in an efficient manner. In particular, the program must terminate in a finite number of steps. Because of this, CBMC expects a number k as input which bounds the size of program traces, yet CBMC also determines if this bound is sufficient. The compiler CBMC-GC inherits these constraints from CBMC. However, for MPC this property is actually a mandatory requirement rather than a limitation. This is because combinatorial circuits have a fixed size and thus deterministic evaluation time in any MPC protocol. This is a logical consequence from the requirement that the runtime of an MPC protocol should not leak any information about the inputs of either party.

We remark that the construction of a compiler out of a well tested bit-precise model checker is beneficial when aiming at a compilation chain from C to circuits for MPC that is sound and that covers almost all language features. Therefore, CBMC-GC is an reasonable choice to implement all optimization techniques presented in the later chapters. Next, we discuss the (dis-)advantages of using ANSI C as input language.

ANSI C AS INPUT PROGRAMMING LANGUAGE. In contrast to many other compilers for MPC (see Section 2.4), CBMC-GC does not use a new domain-specific programming language or even a hardware description language. Instead CBMC-GC uses common ANSI C as input language. ANSI C allows to write low-level as well as high-level (when compared to a hardware description language) code, and thus makes it a reasonable choice to perform high-level synthesis for MPC. Moreover, ANSI C has advantages as existing code can be reused and application developers do not need to learn a new language to explore MPC. For example, in the later chapters we study the use of floating point numbers in MPC, whose support can be added by compiling one of the many existing software floating point implementations.

A drawback of compiling a common programming language is that the data types are optimized for typical RAM based computing hardware. Thus, only fixed bit-widths of $8, 16, 32, 64$, or 128 bit are used. However, Boolean circuit based MPC protocols support arbitrary bit-widths, and hence the optimal bit-width for every variable can be

used for better efficiency. We observe in the next chapter that the actual required bit-width can often be deduced by the compiler and thus, is adapted to the actual required bit-width during optimization, which simplifies programming. Therefore, we are convinced that the advantages of a common programming language prevail this disadvantage.

CHAPTER OUTLINE. First, we describe the compilation chain of CBMC-GC in Section 3.2. Then we study the computational costs of ANSI C operations in an MPC circuit in Section 3.3.

3.2 CBMC-GC'S COMPILATION CHAIN

CBMC-GC's compilation pipeline, which is based in most parts on CBMC's original compilation pipeline, is illustrated in Figure 2. A given input source code passes through multiple steps, before being converted into a circuit:

1. *Parsing and type checking.* First, CBMC-GC parses the given source code into a parse tree and checks syntactical correctness of the source code.
2. *GOTO conversion.* Then, the code is translated into a GOTO program, which is the intermediate representation of code in CBMC-GC.
3. *Loop unrolling.* Next, the program is made loop free by unrolling all loops and recursions.
4. *Single-Static Assignment (SSA)* The loop-free program is rewritten in SSA form, where every variable is only assigned once.
5. *Expression simplification.* Using symbolic execution techniques, constants are propagated and expressions are simplified in the SSA form.
6. *Circuit instantiation.* Finally, given the simplified SSA form, a Boolean circuit is instantiated.

In this section, we first describe the differences between standard C code and code for CBMC-GC. Then, we give an overview of all the compilation steps mentioned before and explain them in detail. Moreover, we give code examples and point out the differences between compiling a program into a SAT formula (CBMC) and compiling into a logical circuit for MPC (CBMC-GC).

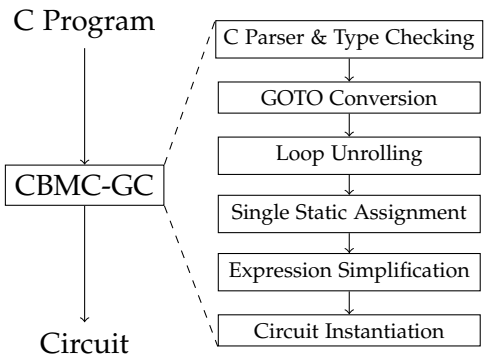


Figure 2: CBMC-GC's compilation chain from C source code into circuits without optimization.

3.2.1 *Input Language and Circuit Mapping*

When programming for CPU/RAM architectures, inputs and outputs of a program are commonly realized by standard libraries that themselves invoke system calls of the operating system. Contrasting, in MPC the only input and output interface available are the *input/output (I/O)* wires of the circuit. To realize the I/O mapping between C code and circuits, a special naming convention of I/O variables is used. The input variables have to be left uninitialized in the source code and are only assigned a value during the evaluation of the circuit in an MPC framework. Hence, instead of adding additional standard libraries, CBMC-GC requires the developer to name input and output variables accordingly.

To illustrate this naming convention, we give an example source code of the millionaires' problem (cf. [Yao82]) in Listing 5. The function shown is a standard C function, where only the input and output variables are annotated as designated inputs of party P_0 or P_1 (Line 2 and Line 3) or as output (Line 4). Hence, variables that are inputs of party P_0 or P_1 have to be named with a preceding `INPUT_A` or `INPUT_B`. Similar, output variable names have to start with `OUTPUT`. Aside from this naming convention, arbitrary C computations are allowed to produce the desired result, in this case a simple comparison (Line 6).

```
1 void millionaires() {
2   int INPUT_A_income;         // Input Party A
3   int INPUT_B_income;         // Input Party B
4   int OUTPUT_result = 0;      // Output
5
6   if (INPUT_A_income > INPUT_B_income) {
7     OUTPUT_result = 1;
8   }
9 }
```

Listing 5: CBMC-GC code example for Yao's Millionaires' problem [Yao82].

For simplicity, CBMC-GC only distinguishes between two parties and uses a shared output, which is the simplest case of secure two-party computation. We follow this approach throughout this thesis. However, we remark that this is not preventing the compilation of code for more than two parties or code with outputs that are designated for specific parties only. This is because, during compilation only input and output variables are distinguished from other variables, but not the association with any party. Hence, to compile code for more parties, an application developer can use her own naming scheme that extends the one introduced by CBMC-GC. CBMC-GC outputs a mapping between every I/O variable and their associated wires in the circuit; this information can be used in any MPC protocol implementation to correctly map wires back to the designated parties.

3.2.2 *C Parser and Type Checking*

CBMC-GC parses the given source code using standard compilation techniques, e.g., using an off-the-shelf C preprocessor (e.g., `gcc -E`) that implements the macro language of C. The preprocessed code is then parsed using a common lexer and parser setup, namely GNU `flex` and `bison` to parse the code into an Abstract Syntax Tree (AST) representation. During parsing, a symbol table is created, which tracks all occurring symbols and their bit-level type information. Type checking is already performed during parsing using the symbol table. If any inconsistencies are detected, CBMC-GC will abort the compilation. The whole parsing process resembles a typical compiler *frontend*, as it is also required when compiling for CPU/RAM architectures. More background on this part of the compile chain can be found in [Muc97].

3.2.3 *GOTO Conversion*

In the second phase of the compilation chain, the parsed AST is translated in a *GOTO* program, which is the intermediate representation of code in CBMC-GC. In this representation all operations responsible for diverging control flow, such as `for`, `while`, `if`, or `switch` statements are replaced by equivalent *guarded* goto statements. These statements can be seen as if-then-else statements with conditional jumps, similarly to conditional branches in assembly language. Using a GOTO representation allows for a uniform treatment of all non-sequential control flow, i.e., no distinction between recursion and loops has to be made, which is beneficial for the next compilation step to unroll all loops. We also remark that in this phase CBMC-GC is capable of handling complex language features, such as function point-

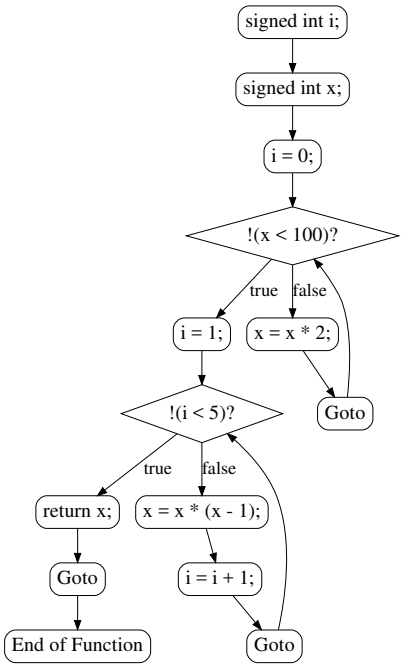


Figure 3: Shown is the CFG of an exemplary program using the GOTO representation as generated by CBMC(-GC) for the example code in Listing 6.

ers, which are resolved using static analysis to identify and branch all candidate functions.

GOTO CONVERSION EXAMPLE. To illustrate the conversion of source code into a GOTO program, we study the example code given in Listing 6. In this code, two types of loops are used, which will have the same representation in the GOTO program.

```
1  int main() {
2    int i, x;
3    i = 0;
4    while(x < 100) {
5      x *= 2;
6    }
7    for(i = 1; i < 5; i++) {
8      x = x * (x - 1);
9    }
10   return x;
11 }
```

Listing 6: Exemplary code snippet with one `for` and one `while` loop.

The resulting Control Flow Graph (CFG) after the conversion into a GOTO program is shown in Figure 3. We observe that both loops have been replaced by a conditional branch ending in a GOTO statement. Recursive functions are translated in the same manner.

3.2.4 *Loop Unrolling*

Boolean formulas and circuits are purely combinatorial, i.e., they only consist of Boolean operators (gates). Cyclic structures, which are still present in the GOTO program, have to be removed during compilation, because these cannot be represented in combinatorial form. To make the program acyclic, CBMC-GC performs symbolic execution to (greedily) *unwind* all loops and recursion in the program. Unwinding is done until either a termination of the cycle is detected during symbolic execution, e.g., the range of a `for` loop has been exceeded, or a user specified bound `k` has been reached. This bound can be set locally for every function, or globally for the complete program.

Technically, unwinding works by replacing all loops (cyclic GOTOs) by a sequence of `k` nested `if` statements. In CBMC the sequence is followed by a special assertion (called *unwinding assertion*), which can be used to detect insufficient `k`. For CBMC-GC, the assertion is omitted because it cannot be checked during circuit evaluation and therefore, the programmer has to ensure a correct upper bound for all loops. Automatically deriving an upper bound is impossible in many cases, as this relates to the undecidable halting problem. Thus, instead of restricting the input language, which would be the alternative solution, CBMC-GC's approach allows the programmer to use any cyclic code structure, as long as an upper bound for every cycle can be provided.

LOOP UNWINDING EXAMPLE. We explain the process of loop unwinding using a generic while loop expression (which is easier to read, yet actually replaced by a GOTO program at this compilation step):

```
while(condition) {
  body;
}
```

In the first unwinding iteration $i = 1$, the first iteration of the loop body is unrolled, which leads to the following code:

```
if(condition) {
  body;
  while(condition) {
    body;
  }
}
```

In the second unwinding iteration $i = 2$, the second loop iteration is unrolled:

```
if(condition) {
  body;
  if(condition) {
    body;
    while(condition) {
      body;
    }
  }
}
```

In the last iteration $i = k$, the loop is completely unrolled:

```
if(condition) {
          body;
          if(condition) {
                    body;
                    [...]
                    if(condition) {
                              body;
                    }
          }
}
```

To provide a concrete example, the loop given below

```
int x[3];
int i = 0;
while(i < 3){
  x[i] = x[i] * x[i];
  i = i + 1;
}
```

is unrolled to the following code:

```
int x[3];
int i = 0;
if(i < 3) {
  x[i] = x[i] * x[i];
  i = i + 1;
  if(i < 3) {
    x[i] = x[i] * x[i];
    i = i + 1;
    if(i < 3) {
      x[i] = x[i] * x[i];
      i = i + 1;
    }
  }
}
```

We remark that recursive functions can be handled in the same way as loops are unrolled by expanding the function body k times, as recursion and loops use the same representation in a GOTO program. We also observe that for loops with at most k steps, unwinding preserves the semantics of the program.

3.2.5 *Conversion into Single Static Assignment Form*

Once the program is acyclic, CBMC-GC turns it into SSA form. This means that each variable x in the program is replaced by fresh variables $x_1, x_2, \ldots$, where each of them is assigned a value only once. For instance, the code sequence

```
x = x + 1;
x = x * 2;
```

is replaced by

```
x2 = x1 + 1;
x3 = x2 * 2;
```

Conditionals occurring in a program are translated into *guarded assignments*. The core idea is that instead of branching at every conditional, both branches are evaluated on distinct program variables. After both branches have been evaluated, the effect on the variables, which are possibly modified by either of the two branches, is applied depending on the branch condition, also referred to as the guard. For example, the body of the following function, which computes the absolute of a value,

```
int abs(int x) {
  if (x < 0)
    x = -x;
  return x;
}
```

is transformed into guarded SSA form as follows:

```
int abs(int x1) {
  _Bool guard1 = (x1 < 0);
  int x2 = -x1;
  int x3 = guard1 ? x2 : x1
  return x3;
}
```

The SSA representation has the important advantage that (guarded) assignments of program variables can be used and manipulated as mathematical equations. In contrast, the straightforward interpretation of any assignment as equation, leads to unsolvable equations, e.g., `x=x+1;`. The indices of the variables in SSA form essentially correspond to different intermediate states in the computation.

3.2.6 *Expression Simplification*

Expression simplification, e.g., constant folding, and constant propagation allow to perform computation on constants and to remove unnecessary computations already during compile time. Hence, removed operations do not need to be translated into circuits.

For example, the expressions

```
x = a + 0;
y = a * 0 + b * 1;
z = -3 + a + 6 / 2;
```

can be simplified to

```
x = a;
y = b;
z = a;
```

by reordering the expressions and evaluating constant parts or by template based matching, such as rewriting `0 * X` by `0`, where `X` can be any side-effect free arithmetic expression. In Section 3.3, we will show that array accesses in MPC are comparably inefficient. Therefore, expression simplification is especially effective and important for the computation of array indicies. Expression simplification can also be performed before loop unrolling, yet, in most cases more simplifications are possible after unrolling, as some variables will become constant, e.g., copies of a loop index variable.

3.2.7 *Circuit Instantiation*

In the final step, CBMC-GC translates the simplified code given in SSA form, which can be seen as a sequence of arithmetic equations, into Boolean formulas (circuits). To this end, CBMC-GC first replaces all variables by bit vectors. For instance, an integer variable is represented as a bit vector consisting of 32 literals. For more complex variables such as arrays and pointers, CBMC-GC replaces all dereference operators by a function that maps all pointers to their dereferenced expression. More details can be found in [CKY03].

Correspondingly, operations over variables (e.g., arithmetic computations) are naturally translated into Boolean functions over these bit vectors. Internally, CBMC-GC realizes these Boolean functions as circuits of Boolean gates whose construction principles are inspired by methods from hardware design. For example, the guarded assignment `(INPUT_A_X == INPUT_B_Y)? t = 0 : t = 1;` is translated into a circuit consisting of two Boolean functions, namely an equivalence check and a multiplexer. We refer to these elementary Boolean functions as *building blocks*. The resulting circuit in this example is illustrated in Figure 4. An implementation of the equivalence building block is illustrated in Figure 5 for a bit-width of two bits. In the next chapter, we present optimized building blocks for all elementary operations. We also remark that for some operations, e.g., bit shifts or multiplications, building blocks with constant and variable inputs are distinguished during circuit instantiation. This is because operations with inputs known to be constant during compile time can be represented by a smaller circuit than those with variable input.

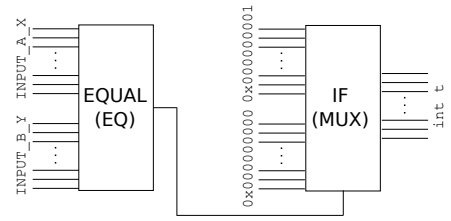


Figure 4: Circuit (left: input, right: output) consisting of two Boolean functions, as created by CBMC-GC from the guarded assignment `(INPUT_A_X==INPUT_B_Y)? t = 0 : t = 1;`.

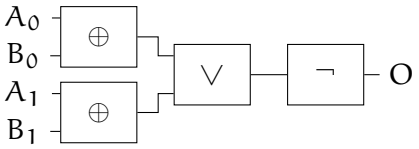


Figure 5: The Boolean equivalence function for two input bit strings of length two, $a = A_0A_1$ and $b = B_0B_1$, implemented as a circuit consisting of two XOR, one OR and one NOT gate.

In the default setting, CBMC translates the resulting circuit into a Boolean formula in Conjunctive Normal Form (CNF) form, ready for its use in a SAT solver. In CBMC-GC the translation into CNF is omitted. Moreover, the circuit generation of CBMC has to be adapted for CBMC-GC: Since CBMC aims to produce efficiently solvable instances for a SAT solver, it is allowed to instantiate circuits which are equisatisfiable with the expected circuits, but not logically equivalent, e.g., integer division is transformed into a multiplication. This is a useful trick to decrease the runtime of SAT solvers for all operations that result in a large Boolean formulas. In these cases CBMC introduces circuits with free input variables, and adds constraints which require them to coincide with other variables. All operations that instantiated equisatisfiable circuits in CBMC have been adapted to instantiate purely logical circuits for CBMC-GC.

3.3 COMPLEXITY OF OPERATIONS IN MPC

When programming efficient applications for Boolean circuit based MPC it is important to consider that the resulting program will be evaluated in the circuit computation model, cf. Section 2.2.2. Thus, some operations that are efficient in the CPU/RAM model can be very inefficient in the circuit model of MPC. In this section, we give a high-level overview on the performance of various operations of ANSI C in the circuit model. A detailed discussion on how the different operations are implemented as a circuit and how these are optimized for MPC is given in the next chapter.

Table 2 shows a summary of the circuit complexities, i.e., the number of non-linear gates, which describe the computation costs in MPC, of various operations in ANSI C. Moreover, we illustrate the actual circuit sizes for a standard integer with a bit-width of $n = 32\,\mathrm{bit}$ and give an estimate on the number of possible operations per second, when garbling each operation in Yao's protocol secure against semi-honest adversaries. The estimates are given using the fastest known garbling scheme [Bel+13], which garbles around 10 million (M) gates per second on single core of a commodity laptop CPU. We remark that already 4 M non-linear gates per second are sufficient to saturate a network link with a capacity of 1 Gbit [ZRE15].

Bit level operations in ANSI C, such as logical ANDs, OR, XOR are very efficient on the circuit level, as they directly translate into the according gate types with a total circuit size that is linear in the data type's bit-width (AND, OR) or even zero (XOR). Shifts (« or ») with variable offset are more costly by a logarithmic factor. All logical operations are for free, when being used with a constant.

Some arithmetic operations, i.e., addition and subtraction, are likewise efficient with a circuit size linear in the bit-width. For example, it is possible to perform more than 300,000 additions per second on a single core in Yao's protocol. Multiplications and divisions are significantly more costly with a circuit size that is quadratic in the bit-width[1]. Hence, for a typical integer, multiplications are by a factor of 32 slower than additions. Nevertheless, in Yao's protocol more than 10,000 multiplications can be performed per second.

The assignment or copying of variables requires no gates in MPC, as assignments are represented by wires. The same holds for array access (read and write) with an index known at compile time. However, a main performance bottleneck in MPC is the access to an array with a variable index. Each access (read or write) of an array element requires a circuit that has the size of the array itself. Assuming an array with m elements of bit-width n, a circuit for read or write access has size $O(mn)$. Consequently, for an exemplary, moderately sized array with 1024 integers, only 300 read or write access can be performed per second.

In summary, integer arithmetic and logical operations are very fast, while multiplications and divisions are the slowest operations, yet still can be performed at a speed of 10,000 of operations per second. However, dynamic array access can become an obstacle for any practical application that requires arrays of noticeable size. Therefore, in the next chapter we also study techniques that aim at detecting constant array accesses to avoid the use of compilation of dynamic accesses whenever possible.

1 For larger bit-widths a complexity of multiplication of $O(n^{1.6})$ can be achieved using Karatsuba multiplication [KS08].

	Circuit complexity	Complexity for 32 bit integers	Ops / sec in Yao's protocol
Logical Operators			
AND (&)	$O(n)$	32	312,500
OR (|)	$O(n)$	32	312,500
XOR (^)	$O(n)$	0	$> 10\,M$
Shift («/») variable	$O(n \log(n)))$	160	62,500
one of the above with constant	0	0	$> 10\,M$
Arithmetic Operators			
Addition (+)	$O(n)$	31	322,580
Subtraction (-)	$O(n)$	32	312,500
Multiplication (*)	$O(n^2)$	993	10,070
Division (/)	$O(n^2)$	1,085	9,216
Modulo (%)	$O(n^2)$	1,085	9,216
Assignments			
var $\leftarrow$ var	0	0	$> 10\,M$
var $\leftarrow$ Array_m[const]	0	0	$> 10\,M$
Array_m[const] $\leftarrow$ var	0	0	$> 10\,M$
var $\leftarrow$ Array_m[var]	$O(mn)$	31744	305
Array_m[var] $\leftarrow$ var	$O(mn)$	34816	287

Table 2: Circuit complexity of operations in ANSI C depending on the bit-width n in the number of non-linear gates and the size of an array m. Shown is also the size of each operation for a typical integer bit-width of $n = 32\,bit$ and an exemplary array size of $m = 1024$, as well as the number of possible operations in current state-of-the-art semi-honest Yao's Garbled Circuits on a single CPU core (10 M non-linear gates per second [Bel+13]).

Part II

COMPILATION AND OPTIMIZATION FOR BOOLEAN CIRCUIT BASED MPC PROTOCOLS

4

COMPILING SIZE-OPTIMIZED CIRCUITS FOR CONSTANT-ROUND MPC PROTOCOLS

Summary: Even though MPC became 'practical' in recent years, it is still multiple orders of magnitude slower than generic computation. Therefore, when developing efficient applications on top of MPC protocols, it is of interest to optimize these to the full extent. In this chapter, we detail the need for optimization in application development for MPC, before deriving requirements for efficient circuit design for Boolean circuit based constant-round MPC protocols. Then we describe the optimization techniques that we applied in CBMC-GC to optimize circuits for this class of protocols. This approach consists of two parts, namely hand-optimized building blocks and a fixed point optimization algorithm employing multiple gate level optimization techniques, such as pattern rewriting or SAT sweeping. The effectiveness of our approach is demonstrated by an experimental evaluation of various benchmark functionalities. For example, we show that the optimization techniques of CBMC-GC lead to circuits that are 70% smaller for the computation of an Euclidean distance or between $20-50\%$ smaller for different implementations of the the Hamming distance when compared to the best compilers in related work.

Remarks: This chapter is based in parts on our article *"On Compiling Boolean Circuits Optimized for Secure Multi-party Computation"*, Niklas Büscher, Martin Franz, Andreas Holzer, Helmut Veith, Stefan Katzenbeisser, which appeared in Formal Methods in System Design, vol. 51, no. 2, 2017, and on Chapter 4 of our book – *"Compilation for Secure Multi-party Computation"*, Niklas Büscher and Stefan Katzenbeisser, Springer Briefs in Computer Science 2017, ISBN 978-3-319-67521-3.

4.1 MOTIVATION AND OVERVIEW

When evaluating a circuit in an MPC protocol each gate in the circuit is evaluated in software using different cryptographic instructions depending on the gate types (see Section 2.2.2). Therefore, assuming a fixed set of input and output wires, the performance of MPC applications depends both on the size of a circuit that represents the functionality to be computed and the gate types used therein [KS08].

We also observe that for almost all deployment scenarios of MPC protocols, the circuit is generated once, whereas the MPC protocol itself will be run multiple times. Especially when considering the high computational and communication costs of MPC protocols (compared to generic computation), it is worthwhile to optimize circuits during their creation. In this chapter, we identify and evaluate optimization techniques that minimize a Boolean circuit for an application in MPC protocols with constant rounds.

OPTIMIZATION CHALLENGE. The compilation of an efficient high-level description of a functionality for RAM based architectures does not necessarily translate into an efficient representation as a circuit. For example, in Section 4.5 we compare the circuit sizes when compiling a Merge and a Bubble sort algorithm. Merge sort is commonly superior in efficiency when computed on a RAM based architecture, yet Bubble sort compiles to a significantly smaller circuit. Due to this reasons dedicated algorithms have been developed in hardware synthesis (e.g., sorting networks), which are optimized for a circuit based computation. To the best of our knowledge, no generic (and practical) approach is known that is capable of transforming a functionality optimized for a RAM machine into a representation that is best suited for circuit synthesis. Therefore, in this chapter *we focus on minimizing circuits for a given algorithmic representation*.

ILLUSTRATING THE NEED FOR OPTIMIZATION. Optimization is important for the wide-spread use of MPC. For example, when writing source code for RAM based architectures, developers commonly rely on a few data types that are available, e.g., `unsigned int` or `long`. In contrast, on the circuit level, arbitrarily bit-widths can be used. However, it is a tedious programming task to specify precise bit-widths for every variable to achieve minimal circuits. Consequently, optimizing compilers should (beside other means) identify overly allocated bit-widths on the source code level and adjust them on the gate-level accordingly. Without advanced optimization techniques, the developer is required to be very familiar with circuit synthesis for MPC and compiler internals to write code that compiles into an efficient circuit and thus, efficient application.

To further illustrate the need for optimization in circuit compilation, we study an example code snippet given in Listing 7. The main part of the code begins in Line 7, where an input variable is declared that is only instantiated during protocol evaluation. Hence, its actual value is unknown during compile time. Next, a variable `t` is declared and initialized with a constant value of 43210. Then, a helper function, declared in Line 1, is called that checks whether the given argument is an odd number. Depending on the result of the helper function, `t` will be incremented by one in Line 10.

A naive translation of this code into a Boolean circuit leads to a circuit consisting of four building blocks, namely, a logical AND, an integer equality check, an integer addition as well as a conditional integer assignment. Figure 6 illustrates the circuit that is generated by such a direct translation. When using the best known building blocks (these are described in Section 4.3) and assuming a standard integer bit-width of 32 bit, 31 or 32 non-linear gates are required for

```
1  int is_odd(int val) {
2    return ((val & 1) == 1);
3  }
4
5  int main() {
6    [...]
7    int INPUT_A_x;
8    int t = 43210;
9    if(is_odd(INPUT_A_x) {
10     t = t + 1;
11   }
12   [...]
13 }
```

Listing 7: Code example to illustrate the need for optimization.

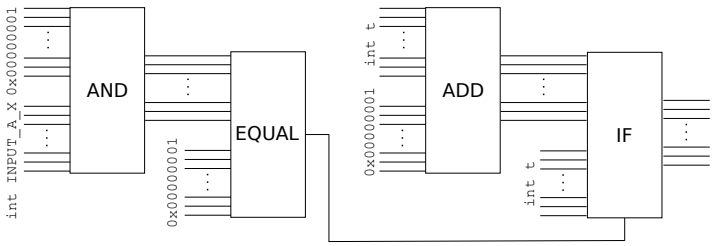


Figure 6: Circuit after naïve translation from source code in Listing 7 to three building blocks of bit-width 32 bit.

each building block. This results in a total circuit size of $s^{nX} = 126$ non-linear gates.

However, an optimizing compiler could detect that the comparison is only a single bit comparison, whose result is equal to the Least Significant Bit (LSB) of `INPUT_A_x`. Hence, no gate is required for the comparison. Moreover, as variable `t` is initialized by an even constant, the addition in Line 10 can be folded into an assignment of the LSB, which leads to a circuit that consists only of a single wire and that does not even contain a single gate, i.e., the LSB of `t` is set to the LSB of `INPUT_A_x`. In this example, the difference in circuit sizes between the naïve translation and an optimized compilation is significant.
Such a size reduction directly relates to an improvement in the protocol's runtime, as the (amortized) computational cost of evaluating an MPC protocol scales linear with the circuit size.

Therefore, in this chapter we present the optimization techniques that we implemented in CBMC-GC that allow the compilation of efficient circuits for MPC protocols with constant rounds, i.e., circuits with a minimal number of non-linear gates. With these techniques, we are able to offer a high level of abstraction from both MPC and circuits by compiling from standard ANSI C source code. For the creation of size-minimized circuits, we first provide an overview of optimized building blocks and then present a fixed point algorithm

that combines techniques from logic minimization, such as constant propagation, SAT sweeping, and pattern rewriting, to compile circuits that are significantly smaller than those generated by compilers from related work.

CHAPTER OUTLINE. We first given an overview on the circuit optimization process in CBMC-GC in Section 4.2. Then, in Section 4.3 we describe size-optimized building blocks for MPC, before describing multiple gate-level optimization techniques in Section 4.4. Finally, the effectiveness of these optimization techniques is studied in Section 4.5.

4.2 CIRCUIT MINIMIZATION FOR MPC

OPTIMIZATION GOAL. In Section 2.2.2, we introduced Yao's Garbled Circuits protocol, which is the most researched constant round MPC protocol. We use Yao's protocol to derive a cost model and circuit optimization goal, yet, we remark that all known practical MPC protocols over Boolean circuits with constant rounds have a very similar cost model and therefore profit from the ideas presented here.

The runtime of Yao's protocol depends on the input and output sizes as well as the circuit that is used to compute the functionality $f(x, y)$. Assuming a correct specification of inputs and outputs[1], the runtime of an application in Yao's protocol can only be improved by changing the circuit representation $C_f(x, y)$ into a more efficient representation $C'_f(x, y)$. The circuit runtime depends on the number of linear and non-linear gates. For security models relevant in practice[2], only non-linear gates require computation of encryptions and communication between the parties, whereas the linear gates are considered as being for 'free' (see Section 2.2.2), because they neither require communication nor cryptographic operations. Thus, the primary goal of circuit optimization for Yao's protocol, and most other known constant round MPC protocols, is to minimize the number of non-linear (AND) gates. Nevertheless, once the number of non-linear gates is minimized, as a secondary goal the total number of linear (XOR) gates could also be minimized, because in a practical deployment linear gates also generate (albeit very small in comparison to AND gates) computation costs, e.g., fetching and storing wire labels from the memory. Moreover, for the best known protocol secure in the standard model, XOR gates have a relevant cost, as they cannot be garbled for free [Gue+15]. Albeit less costs than garbling AND

1 Our optimization methods also detect and display unused input as well as constant output.

2 The best known techniques secure in the standard model also require encryptions for linear gates [Gue+15].

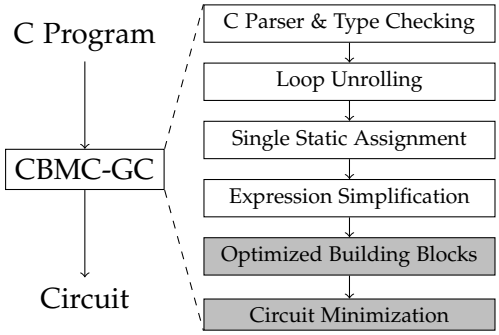


Figure 7: CBMC-GC's compilation pipeline with circuit optimization. Marked in gray are the parts described in this chapter that lead to size-minimized circuits for MPC.

gates, a minimal number of XOR gates is thus also relevant in this scenario.

With communication being the most relevant bottleneck for practical MPC, the techniques described in the next sections mainly aim at achieving the primary goal, yet an unnecessary increase in XOR gates is always avoided. Finally, we remark that circuit optimization is a time constrained problem, as a programmer in practices expects the compiler to finish its optimization after a reasonable time T.

MINIMIZING STRATEGY. Unfortunately, finding a minimal circuit for a given circuit description is known to be Σ_2^P complete [BU11]. Therefore, in CBMC-GC we follow a heuristic approach. First, during circuit instantiation (see Chapter 3), we use size-minimized building blocks that are described in Section 4.3. Second, the instantiated circuit is optimized on the gate level by using a best-effort fixed point optimization algorithm that is discussed in detail Section 4.4 and added as an additional step to the compilation chain of CBMC-GC. The full compilation chain of CBMC-GC, including optimizations, is shown in Figure 7.

4.3 BUILDING BLOCKS FOR BOOLEAN CIRCUIT BASED MPC

Optimized building blocks are an essential part of designing complex circuits. They facilitate efficient compilation, as they can be highly optimized once and subsequently instantiated at practically no cost during compilation. In the following paragraphs, we give a comprehensive overview over the currently best known building blocks with a minimal number of non-linear gates for the most common arithmetic and control flow operations.

ADDER. An n-bit *adder* takes two bit strings x and y of length n, representing two (signed) integers, as input and returns their sum as an output bit string S of length $n+1$. An adder is commonly con-

structed of smaller building blocks, namely Half Adders (HAs) and Full Adders (FAs). A HA is a combinatorial circuit that takes two bits A and B and computes their sum $S = A \oplus B$ and carry bit $C_{out} = A \cdot B$. A FA allows an additional carry-in bit C_{in} as input. The best known constructions [KS08] for computing the sum bit of a FA is by XOR-ing all inputs

$$S = A \oplus B \oplus C_{in},$$

while the carry-out bit can be computed by

$$C_{out} = (A \oplus C_{in})(B \oplus C_{in}) \oplus C_{in}.$$

Both HA and FA have a non-linear size $s^{nX} = 1$. The standard and best known n bit adder is the Ripple Carry Adder (RCA) that consists of a successive composition of n FAs. Thus, RCA has a circuit size $s^{nX}_{RCA}(n) = n$. We note that, according to the semantics of ANSI C, an addition is computed as $x + y \mod 2^n$ and no overflow bit is returned, which reduces the circuit size by one non-linear gate.

SUBTRACTOR. A subtractor for two n-bit strings can be implemented with one additional non-linear gate by using the two's complement representation $x - y = x + \overline{y} + 1$. Thus, the addition of negative numbers in the two's complement representation is equivalent to an addition of positive numbers.

COMPARATOR. An *equivalence (EQ)* comparator checks whether two input bit strings of length n are equivalent and outputs a single result bit. The comparator can be implemented naïvely by a successive OR composition over pairwise XOR gates that compare single bits. This results in a size of $s^{nX}_{EQ}(n) = n - 1$ gates [KS08]. A *greater-than (GT)* comparator that compares two integers can be implemented with help of a subtractor by observing that $x > y \Leftrightarrow x - y - 1 \geqslant 0$ and returning the carry out bit, which yields to a circuit size of $s^{nX}_{GQ}(n) = n$.

MULTIPLIER. In classic hardware synthesis, a multiplier (MUL) computes the product of two n bit strings x and y, which has a bit-width of $2n$ bit. However, in many programming languages, e.g., ANSI C, multiplication is defined as an $n \to n$ bit operation, where the product of two unsigned integers is computed as $x \cdot y \mod 2^n$. The standard approach for computing an $n \to 2n$ bit multiplication is often referred to as the "school method". Using a bitwise multiplication and shifted addition, the product is computed as $\sum_{i=0}^{n-1} 2^i (X_i y)$. This approach leads to a circuit requiring n^2 1-bit multiplications and $n - 1$ shifted n-bit additions, which in total results in a circuit size of $s^{nX}_{MUL}(n) = 2 \cdot n^2 - n$ gates [KSS09]. When compiling a $n \to n$ bit multiplication with the same method, only half of the one bit multiplications are relevant, leading to a circuit size of $s^{nX}_{MUL}(n) = n^2 - n$

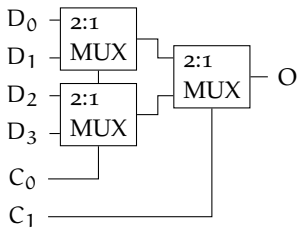


Figure 8: Circuit consisting of a multiplexer tree when compiling an exemplary array read of four bits D_0, D_1, D_2, D_3.

gates. The $n \to n$ multiplication of negative numbers in the two's complement representation can be realized with the same circuit. Alternatively, for a $n \to 2n$ bit multiplication the Karatsuba-Ofmann multiplication (KMUL) can be used, achieving an asymptotic complexity of $O(n^{\log_2(3)})$. Henecka et al. [Hen+10] presented the first adoption for MPC, which was subsequently improved by Demmler et al. [Dem+15] by 3% using commercial hardware synthesis tools. For an $n \to 2n$ bit multiplication their construction outperforms the school method for bit-widths $n \geqslant 19$ bit.

MULTIPLEXER. Control flow operations, e.g., branches and array read accesses, are expressed on the circuit level through multiplexers (MUX). A 2:1 n-bit MUX consists of two input bits strings d^0 and d^1 of length n and a control input bit C. The control input decides which of the two input bit strings is propagated to the output bit string o. For an array read access, a multiplexer with more inputs is required. A 2:1 MUX can be extended to a m:1 MUX that selects between m input strings $d^0, d^1, \ldots, d^m$ using $\lceil \log_2(m) \rceil$ control bits $C_0, C_1, \ldots, C_{\lceil \log(m) \rceil}$ by a tree based composition of 2:1 MUXs. For example, a read access to an array consisting of four bits D_0, D_1, D_2, D_3 and two index bits $i = C_1 C_0$ can be realized as illustrated in Figure 8.

Kolesnikov and Schneider [KS08] presented a construction of a 2:1 MUX that only requires one single non-linear gate for every pair of input bits by computing the output bit as $O = (D^0 \oplus D^1)C \oplus D^0$. This leads to a circuit size for an n-bit 2:1 MUX of $s^{nX}_{MUX}(n) = n$. The circuit size of a tree based m:1 MUX depends on the number of choices m, as well as the bit-width n, yielding $s^{nX}_{MUX\_tree}(m, n) = (m-1) \cdot s^{nX}_{MUX}(n)$.

DEMULTIPLEXER. Write accesses to an array require a building block, where only the element addressed by a given index is replaced. All other elements should be unchanged. Hence, this resembles very closely the inverse of a multiplexer, referred to as demultiplexer (DEMUX). A 1:m DEMUX has an input index i, an input bit string x, a number m of input bit strings $d_1, d_2, \ldots, d_m$ and outputs m bit

strings $d'_1, d'_2, \ldots, d'_m$, where an output d'_j is set to x if $j = i$ and to d_j if otherwise.

A construction for a 1:m DEMUX is given by Malkhi et al. [Mal+04]: Each output $d_{out,i}$ is controlled by a multiplexer, which assigns $d'_i \leftarrow x$, if the constant index j is equivalent to the index input bit string i. This construction is similar to a sequence of `if` clauses, i.e.,

```
if(i==0)
  d[0] = x;
else if(i==1)
  d[1] = x;
[...].
```

This yields a circuit size of

$$s^{nX}_{DMUX\_EQ}(m,n) = m \cdot \left(s^{nX}_{EQ}(n) + s^{nX}_{MUX}(n)\right) = m \cdot (\lceil \log_2(m) \rceil - 1 + n).$$

A 1:m DEMUX can be constructed more efficiently using a one-hot encoder that translates the input choice string c into an output bit string o of length m, where the c'th bit is set to one, and all other bits are set to zero. The output bit string can be connected to a multiplexer for every data input, as described above. A one-hot encoder can be constructed in a tree based manner using 1-bit encoders that have an input bit I, a choice input C and two output O_0 and O_1. The outputs are computed as $O_0 = (I \wedge C) \oplus I$ and $O_1 = (I \wedge C) \oplus I$. The root input is set to $I = 1$ and the m final outputs are connected to the multiplexers controlling the data inputs. In total $n - 1$ 1-bit encoders are required, which can be computed with only a single AND gate. This is because the intermediate result $I \wedge C$ has only be computed once to be used for both outputs. In total, this yields a circuit size of

$$\begin{aligned} s^{nX}_{DMUX\_tree}(m,n) &= \text{encoders} + \text{multiplexers} \\ &= n - 2 + mn \\ &\approx n \cdot (m+1). \end{aligned}$$

DIVISION. A divisor computes the quotient and remainder for a division of two binary integer numbers. The standard approach for integer division is known as long division and works similar to the school-method for multiplication. Namely, the divisor is iteratively shifted and subtracted from the remainder, which is initially set to the dividend. Only if the divisor fits into the remainder, which is efficiently decidable by overflow free subtraction, a bit in the quotient is set and the newly computed remainder is used. Thus, a divisor can be built with help of n subtractors and n multiplexers, each of bit-width n, leading to a circuit size of $s^{nX}_{SDIV}(n) = 2n^2$. The divisor can be improved by using restoring division [Rob58], which leads to a circuit size of $s^{nX}_{RDIV}(n) = n^2 + 2n + 1$.

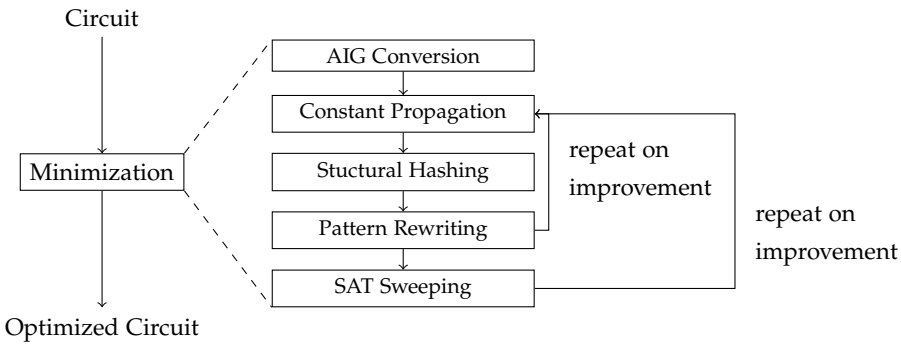


Figure 9: Illustration of CBMC-GC's minimization procedure.

4.4 GATE-LEVEL CIRCUIT MINIMIZATION

As identified in Section 4.1, with the compilation being a one-time task, it is very useful to invest computation time in circuit optimization during compilation. Moreover, as shown, a naïve translation of code into optimized building blocks does not directly lead to a minimal circuit. Therefore, after the instantiation of all building blocks, and thus construction of the complete circuit, a circuit minimization procedure is run that reduces the number of non-linear gates. The procedure is an heuristic approach that itself consists of multiple different algorithms. We begin with a discussion of the general procedure, before discussing the different components in detail.

MINIMIZATION PROCEDURE. An overview of the minimization routine is illustrated in Figure 9. It begins with the translation of the circuit into an intermediate AND-Invert Graph (AIG) representation, which is a circuit description consisting of only AND and inverter (NOT) gates that allows efficient gate-level optimizations [MCB06]. After the AIG translation, a fixed point minimization routine is initiated. The algorithm is run until no further improvements in the circuit size are observed or a user given time bound T has been reached. In both cases, the result of the latest iteration is returned.

In every iteration of the algorithm a complete and topological pass over all gates from from inputs to outputs is initiated. During this pass, constants (zero or one) are propagated (*constant propagation*), duplicate gate structures are eliminated (*structural hashing*) and small sub-circuits are matched and replaced by hand-optimized sub-circuits (*rewrite patterns*). If any improvement, i.e., reduction in the number of non-linear gates, is observed, a new pass is initiated. If no improvement is observed, a more expensive optimization routine is invoked that detects constant and duplicate gates using a SAT solver (*SAT sweeping*).

AIG, CONSTANT PROPAGATION AND STRUCTURAL HASHING. An AND-inverter graph is a representation of a logical functionality using only binary AND gates (nodes) with (inverted) inputs. AIGs have been identified as a very useful representation for circuit minimization, as they allow very efficient graph manipulations, such as addition or merging of nodes. CBMC-GC utilizes the ABC library [Ber] for AIG handling, which provides state-of-the-art circuit synthesis methods. As a first step in CBMC-GC, input and output wires are created for every input and output variable. Then, during the instantiation of building blocks, the AIG is constructed by substituting every gate type that is different from an AND gate by a Boolean equivalent AIG sub-graph. For example, an XOR gate ($A \oplus B$) can be replaced by the following AIG: $\overline{\overline{V_A} \cdot \overline{V_B}} \cdot \overline{V_A \cdot V_B}$, where V_A is the node representing A in the AIG.

Whenever a node is added to the AIG, two optimization techniques are directly applied. First, constant inputs are propagated. Hence, whenever an input to a new node is known as constant, the added node is replaced by an edge. For example, when adding a node V_{new} with inputs from some node V_j and node V_{zero}, which is the node for the constant input zero, then V_{new} will not be added to the AIG. Instead, all edges originating V_{new} will be remapped to originate from V_{zero} instead, because $V_{new} \cdot 0$ is equivalent to 0. Similarly, $V_{new} = V_j \cdot 1$ will be replaced by V_j. Second, structural hashing is applied. Structural hashing [Dar+81] is used to detect and remove duplicate sub-graphs, i.e., graphs that compute the same functionality over the same inputs. Duplicate sub-graphs are also detected when new nodes are added. Hence, when adding a node V_{new} to the AIG, it is checked that no other node exists that uses the same inputs. If such a node V_j is found, V_{new} will be replaced by V_j, as described above.

PATTERN REWRITING. Circuit (and AIG) rewriting is a greedy optimization algorithm used in logic synthesis [MCB06], which was first proposed for hardware verification [BB04]. A rewrite pattern consists of a (small) template circuit to be matched and a substitute circuit. Both circuits are functionally equivalent, yet can have a different structure or different gates. Pattern based rewriting has been shown to be a very effective optimization technique in logic synthesis, as it can be applied with very little computational cost [MCB06]. In CBMC-GC's compilation chain, rewrite patterns are of high importance due to multiple reasons. First, they are responsible for translating the AIG back into a Boolean circuit representation with a small number of non-linear gates. This is necessary, because all linear gates have been replaced by AND gates during the translation in the AIG representation. Second, pattern based rewriting allows for MPC specific optimizations by applying patterns that favor linear gates and reduce

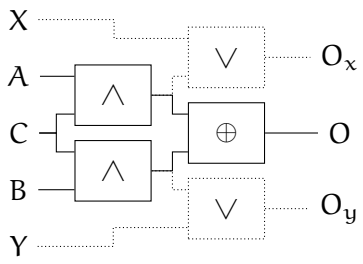


Figure 10: Counter example for circuit rewriting. The sub-circuit indicated by the solid lines is a candidate for rewriting. Yet, due to its intermediate outputs, marked with dotted lines, a rewriting would actually increase the overall circuit size.

the number of non-linear gates. Finally, in CBMC-GC each rewrite pass is also used for constant propagation and structural hashing, as described above. We remark that these techniques are responsible for reducing the bit-width declared on the source code level to the actual required bit-width, by identifying unused or constant gates.

For circuit rewriting all gates are first ordered in topological order by their circuit depth. Subsequently, by iterating over all gates, the patterns are matched against all gates, and thus all possible sub-circuits. Whenever a match is found, the sub-circuit becomes a candidate for a replacement. However, the sub-circuit will only be replaced if the substitution leads to an actual improvement in the circuit size. This is often not the case, because matched gates might provide inputs to other gates, which can render the substitution ineffective. In these cases the sub-circuit will not be replaced. An example is shown in Figure 10, where a sub-circuit matches a template: $(A \cdot B) \oplus (A \cdot C) \rightarrow (A \oplus C) \cdot A$. Yet the detected sub-circuit has intermediate outputs to gates outside the template, which would still need to be computed when rewriting the computation of O.

The outcome and performance of this greedy replacement approach depends not only on the patterns themselves, but also on the order of patterns applied. Therefore, in CBMC-GC, small patterns are matched first, e.g., single gate patterns, as they can be matched with little cost and offer guaranteed improvements, before matching more complex patterns that require to compare sub-circuits consisting of multiple gates and inputs. Table 3 lists some exemplary rewrite patterns that are used in CBMC-GC and that have been shown to be very effective in our evaluation. In total more than 80 patterns are used for rewriting.

SAT SWEEPING. SAT sweeping is a powerful minimization tool, widely used for equivalence checking of combinatorial circuits [Kue04; Mis+06]. The core idea of SAT sweeping is to prove that the output of a sub-circuit is either constant or equivalent to another sub-circuit (detection of duplicity). In both cases one of the two sub-circuit is un-

Search pattern	Substitute	Size reduction (s^{nX}/s)
Propagate Pattern		
$\overline{0}$	1	0 / 1
$0 \cdot A$ or $A \cdot 0$	0	1 / 1
$0 + A$ or $A + 0$	A	1 / 1
$0 \oplus A$ or $A \oplus 0$	A	0 / 1
$\overline{1}$	0	0 / 1
$1 \cdot A$ or $A \cdot 1$	A	1 / 1
$1 + A$ or $A + 1$	1	1 / 1
Trivial Patterns		
$A \cdot A$	A	1 / 1
$A \cdot \overline{A}$	0	1 / 2
$A + A$	A	1 / 1
$A + \overline{A}$	1	1 / 2
$A \oplus A$	0	0 / 1
$A \oplus \overline{A}$	1	0 / 2
$\overline{\overline{A}}$	A	0 / 2
AND/OR Patterns		
$(A \cdot B) \cdot (A \cdot C)$	$A \cdot B \cdot C$	1 / 1
$(A \cdot B) + (A \cdot C)$	$A \cdot (B + C)$	1 / 1
$(A + B) \cdot (A + C)$	$A + (B \cdot C)$	1 / 1
$(A + B) + \overline{A}$	1	2 / 3
$(A \cdot B) \cdot \overline{A}$	0	2 / 3
$(A + B) + (\overline{A} + C)$	1	3 / 4
$(A \cdot B) \cdot (\overline{A} \cdot C)$	0	3 / 4
XOR Patterns		
$\overline{A} \oplus \overline{B}$	$A \oplus B$	0 / 2
$(A + B) \oplus (A \cdot B)$	$A \oplus B$	2 / 2
$(A + B) \cdot \overline{(A \cdot B)}$	$A \oplus B$	3 / 3
$\overline{\overline{A} \cdot \overline{B}} \cdot \overline{A \cdot B}$	$A \oplus B$	3 / 6
$(A \cdot (B \oplus (A \cdot C)))$	$A \cdot (B \oplus C)$	1 / 1
$(A \cdot B) \oplus (A \cdot C)$	$(A \oplus C) \cdot A$	1 / 1
$\overline{(A \cdot C) \oplus (B \cdot C)}$	$\overline{(A \oplus B)} \cdot C$	1 / 1

Table 3: Exemplary rewrite patterns used in CBMC-GC. The patterns consist of a search pattern and a substitute circuit, which lead to different improvements, i.e., size reductions.

necessary and can be removed. As common, SAT sweeping is applied in CBMC-GC in a probabilistic manner. A naïve application, which checks every possible combination of sub-circuits, would result in infeasible computational costs. Thus, for efficient equivalence checking, the circuit is first evaluated (simulated) multiple times with different random inputs. The gates in every run are then grouped by their output, i.e., gates with the same output behavior over all runs form a candidate equivalence class. Moreover, gates that always output one or always output zero are presumingly constant. These hypotheses are then verified using the efficient tool of a SAT solver. For this purpose, sub-circuit are converted into CNF and passed to the solver. Due to its high computational cost in comparison with the circuit rewriting, SAT sweeping is only applied if other optimization methods cannot minimize the circuit any further.

4.5 EXPERIMENTAL EVALUATION

To evaluate the effectiveness of different optimization techniques, we study the circuit sizes when compiling example applications that emerged as standard benchmarks for MPC, described in detail in Section 2.5. We first present a comparison of circuits created by the different releases of CBMC-GC, which use an increasing number of optimization techniques described in this chapter. Then, we present a comparison of the circuits generated by CBMC-GC with circuits generated by other state of the art compilers for Boolean circuit based MPC.

4.5.1 *Evaluation of Circuit Minimization Techniques*

It is almost impossible to benchmark the described circuit minimization techniques in isolation. This is because of the many dependencies between the optimization techniques. For example, SAT sweeping is highly ineffective without efficient constant propagation. Moreover, often rewrite patterns can become ineffective without the application of other rewrite patterns. To provide a further example, when using less efficient building blocks, the circuit minimization phase will partly be able to compensate inefficient building blocks and start optimizing these. However, the computation time spent on optimizing the building blocks can consequently not be applied to the remaining parts of the circuit. For these reasons, we abstain from an individual evaluation of the described optimization methods but rather show that the combination of techniques described in this thesis have evolved over previous work, such that the circuit sizes of all benchmark applications have significantly been reduced when compared to earlier compiler versions.

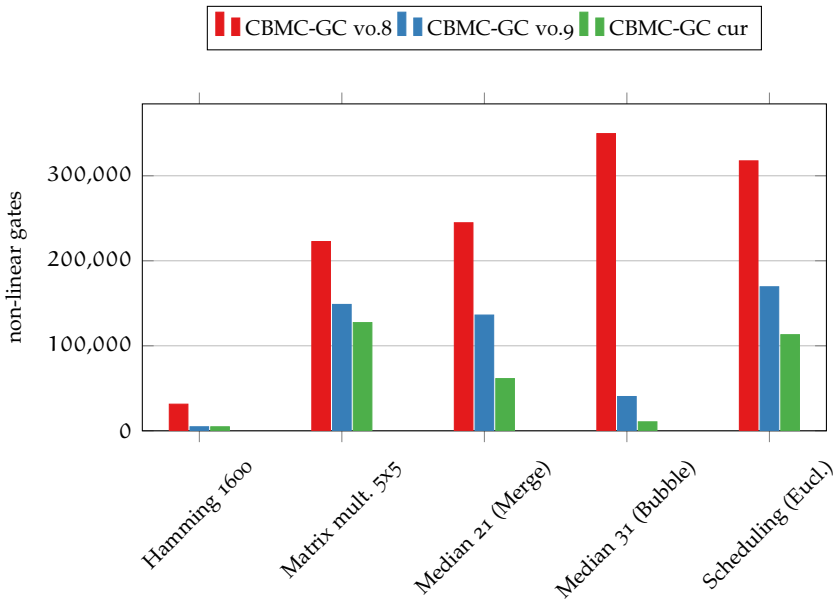


Figure 11: The circuit sizes in the number of non-linear gates produced by CBMC-GC v0.8 [Hol+12], v0.9 [Fra+14], and the current version for five example applications, cf. Table 4.

In Table 4, which is also illustrated in Figure 11, a comparison of circuit sizes between the first release of *CBMC-GC v0.8* [Hol+12] in 2012, its successor *CBMC-GC v0.9* [Fra+14] from 2014, and the *current* version described in this thesis is given. Moreover, the improvement between the initial and the current version of CBMC-GC is shown. The initial release of CBMC-GC provided no gate-level minimization techniques, yet it contained first optimized building blocks. In CBMC-GC v0.9 a first version of the fixed point optimization algorithm, was introduced. The most recent version that implements all techniques and building blocks described in this chapter.

For comparison purposes, we use a selection of applications introduced in Section 2.5. All applications have been compiled from the same source code and optimized with a time limit of 10 minutes on a commodity laptop. We observe that the improvement in building blocks is directly visible in the example applications performing random arithmetic operations and a matrix multiplication, which have been improved up to a factor of two, between the first and the current release of CBMC-GC. These two applications purely consist of arithmetic operations that utilize almost the complete bit-width of the used data types and operations, and thus, barely profit from gate-level optimizations. The resulting circuit sizes for the Hamming distance computation show significant improvements when comparing the first and the current release, yet only marginal improvement in comparison to CBMC-GC v0.9. More complex applications, such as Bubble sort based median computation, which involve noticeably more control flow logic, have been improved by more than a factor

of 10 between the first and the current release of CBMC-GC. Similarly, for the location aware scheduling application, which involves a mix of arithmetic operations and control flow logic, we observe an improvement up to a factor of 3 between the first and the current release. In summary, the proposed fixed point optimization routine is very effective to minimize the number of non-linear gates for a given circuit.

4.5.2 *Compiler Comparison*

We compare CBMC-GC with Frigate [Moo+16] and ObliVC [ZE15], which are the most promising compilers for a comparison, as they create circuits with the least number of non-linear gates (at the time of writing) according the compiler analysis by Mood et al. [Moo+16]. Even though all compilers use different input languages, we ensure a fair comparison by implementing the functionalities using the same code structures (i.e., functions, loops), data types and bit-widths. Moreover, to present a wider variety of applications, we also investigate more integrated example applications, such as the biometric matching (BioMatch) application or floating point operations, which are all described in Section 2.5. All applications have been compiled with the latest available versions of Frigate and ObliVC. For CBMC-GC, we again set an optimization time limit of 10 minutes on a commodity laptop. The resulting circuit sizes and the improvement of CBMC-GC over the best result from related work are presented in Table 5 and partly illustrated in Figure 12. Circuit sizes above one million (M) gates are rounded to the nearest 100,000.

We observe that the current version of CBMC-GC outperforms related compilers in circuit size for almost all applications. For example, the BioMatch application, or scheduling applications improve by more than 25%. Most significant is the advantage in compiling floating point operations, where a 77% improvement can be observed for the dedicated multiplication operation. A similar improvement is achieved for the computation of the Euclidean distance or matrix multiplication (MMul) on floating point and fixed point values. The operations are dominated by bit wise operations and thus, can be optimized with gate-level optimization when compiled from high-level source code. No improvement is observed for the integer based MMul. As discussed above, the MMul compiles into a sequential composition of building blocks, which barely can be improved further.

The Hamming distance computation is well suited to show implementation dependent results and illustrates the challenges of writing source code that compiles into efficient circuits. Here we compare three variants described in more detail in Section 2.5. The tree based computation shows the smallest circuit size, even though it is the most inefficient CPU implementation. This is because a tree-based

Application	CBMC-GC v0.8	CBMC-GC v0.9	CBMC-GC cur	Improvement
Arithmetic 2000	405,640	319,584	253,776	37%
Hamming 320 (reg)	6,038	924	924	85%
Hamming 800 (reg)	15,143	2,344	2,340	85%
Hamming 1600 (reg)	30,318	4,738	4,726	84%
Matrix multiplication, 5x5	221,625	148,650	127,255	43%
Matrix multiplication, 8x8	907,776	600,768	522,304	42%
Median 21, Merge sort	244,720	136,154	61,403	75%
Median 31, Merge sort	602,576	348,761	152,823	75%
Median 21, Bubble sort	112,800	40,320	10,560	91%
Median 31, Bubble sort	349,600	89,280	23,040	93%
Scheduling 56, Euclidean	317,544	169,427	113,064	64%
Scheduling 56, Manhattan	133,192	88,843	62,188	53%

Table 4: Experimental results: circuit sizes in the number of non-linear gates produced by CBMC-GC v0.8 [Hol+12], v0.9 [Fra+14], and the current version for various example applications. Moreover, the improvement between the first and the current version of CBMC-GC is shown.

Application	Frigate	OblivC	CBMC-GC	Improvement
Biometric matching 128	561,218	560,192	404,419	**27.8%**
Biometric matching 256	1,1 M	1,1 M	831,846	**25.7%**
Euclidean 5, int	7,960	7,811	6,235	**20.2%**
Euclidean 20, int	25,675	25,001	23,255	7.0%
Euclidean 5, fix	26,430	26,307	7,834	**70.2%**
Euclidean 20, fix	90,735	90,057	24,559	**72.3%**
Float addition	5,237	5,581	1,201	**77.1%**
Float multiplication	16,502	14,041	3,534	**74.8%**
Hamming 160 (reg)	567	899	449	**20.8%**
Hamming 1600 (reg)	6,546	9,269	4,738	**27.6%**
Hamming 160 (tree)	747	4,929	351	**53.0%**
Hamming 1600 (tree)	8,261	49,569	3,859	**53.3%**
Hamming 160 (naïve)	1,009	4,929	541	**46.3%**
Hamming 1600 (naïve)	10,282	49,569	6,042	**41.2%**
Matrix multiplication 5x5, int	127,477	127,225	127,225	0%
Matrix multiplication 5x5, fix	314,000	313,225	183,100	**41.5%**
Matrix multiplication 5x5, float	2,7 M	2,4 M	626,506	**73.9%**
Scheduling 56, Euclidean	133,216	181,071	113,064	**15.1%**
Scheduling 56, Manhattan	79,008	77,023	58,800	**23.7%**

Table 5: Comparison between Frigate [Moo+16], OblivC [ZE15] and the current version of CBMC-GC. Given are the circuit sizes in the number of non-linear gates when compiling various applications. Marked in bold are significant improvements over the best previous result.

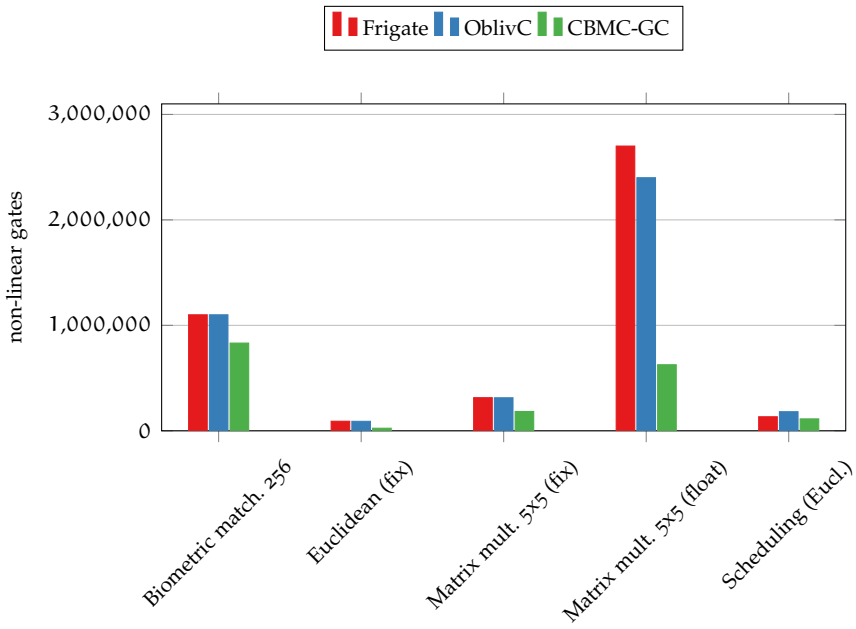


Figure 12: Circuit size comparison of Frigate [Moo+16], OblivC [ZE15] and the current version of CBMC-GC, cf. Table 5.

composition allows to apply adders with small bit-widths for the majority of the bit counting. Comparing the compilers, we also observe differences between the implementations. The tree based and naïve implementation are significantly more optimized, i.e., up to a factor of two, in CBMC-GC than in the other compilers. The implementation optimized for register based computation compiles into a circuit that is also smaller in CBMC-GC than in related work, yet only by 20%. This is because the compilation of the naïve bit counting profits significantly from constant propagation, as only a few bits per expression are required on the gate-level. The register optimized implementation maximizes the number of bits used per arithmetic operation, thus, allows only little gate-level optimization. We remark that Frigate and OblivC compile larger applications noticeably faster than CBMC-GC, yet the circuits created by CBMC-GC are up to a factor of four smaller.

4.6 RELATED WORK

LOGIC SYNTHESIS. Since the development of the first CPUs, logic minimization and optimization have continuously been evolved for logic synthesis in hardware design. Many algorithms and techniques have been developed, e.g., the ESPRESSO logic minimization [Bra+] or DAG rewriting [Mis+06], which are also of relevance for logic synthesis for MPC. An overview of the many existing algorithm is given in [She93].

CIRCUIT MINIMIZATION FOR MPC. In classic logic synthesis, non-linear gates are often cheaper than linear gates, therefore, to the best of our knowledge, minimizing the non-linear complexity has barely been studied in this context. However, in cryptography a minimal number of non-linear gates is not only relevant for MPC but also for zero-knowledge proofs, homomorphic encryption or verifiable outsourced computation. With the practical advancement of these protocols, recently dedicated logic synthesis methods became of relevance. For example, theoretical minimization limits have been studied [BPP00; BU11; TP14] and also first optimization routines have been proposed [BP10].

The only compiler for MPC next to CBMC-GC that employs a complete logic minimization tool-chain is TinyGarble [Son+15], which adapts a commercial logic synthesis tool to the needs of MPC. Other compilers (see Section 2.4), such as Frigate [Moo+16], only provide limited minimization mechanisms, e.g., local constant propagation. ObliVM [Liu+15b] presented a loop rewriting optimization approach for for efficient compilation. An expression rewriting optimization technique has been proposed by Kerschbaum [Ker11; Ker13]. Its core idea is to use knowledge interference between the parties to detect operations that can be locally computed rather than securely. Recently, Kennedy et al. [KKW17] and Laud and Pankova [LP16] independently presented two similar optimization approaches that aim at detecting and fusing duplicated functionalities.

Cryptography primitives have been optimized independently. For example, the AES S-Box has been (hand-)optimized in regard to its non-linear complexity [BP10] and also new cipher designs with a focus on minimal non-linear complexity have been proposed [Alb+15; Alb+16]. Moreover, many arithmetic and control flow building blocks known from hardware design (cf. 4.3) have been hand optimized, e.g., [Dem+15; KSS09; KS08; ZE13].

Further circuit optimization problems are discussed in Chapter 6 and Chapter 7.

5

COMPILING PARALLEL CIRCUITS

Summary: Practical MPC has seen a lot of progress over the past decade. Yet, compared to generic computation, MPC is still multiple orders of magnitude slower. To improve the efficiency of secure computation protocols, we describe a practical and compiler assisted parallelization scheme, exemplary applied to Yao's Garbled Circuits: First, we study how Yao's Garbled Circuits can be parallelized effectively following a fine-grained and coarse-grained parallelization approach. Then we present a compiler extension for CBMC-GC that detects parallelism at the source code level and automatically transforms C code into parallel circuits. These circuits allow more scalable execution on parallel hardware, as we show in an evaluation of three example applications. Finally, by switching the roles of circuit generator and evaluator between both computing parties in the semi-honest model, our scheme makes better use of computation and network resources. This *inter-party parallelization* approach leads to significant efficiency increases already on single-core hardware without compromising security.

Remarks: This chapter is based in parts on our paper – *"Faster Secure Computation through Automatic Parallelization"*, Niklas Büscher and Stefan Katzenbeisser, which appeared in the Proceedings of the 24th USENIX Security Symposium, 2015, Washington DC, USA.

5.1 MOTIVATION AND OVERVIEW

At the time of writing, millions of gates can be computed (garbled) in frameworks implementing Yao's Garbled Circuits on a consumer grade CPU within seconds. Nonetheless, compared with generic computation, Yao's Garbled Circuits protocol is still multiple orders of magnitude slower. Even worse, an information theoretic lower bound on the number of ciphertexts has been identified for gate-by-gate garbling techniques by Zahur et al. [ZRE15], which makes further simplification of computations unlikely. Observing the ongoing trend towards parallel hardware, e.g., smartphones with many-core architectures on a single chip to reduce power consumption, the questions arises, whether Yao's Garbled Circuits can be parallelized. In this chapter, we answer this question positively and describe strategies to garble and evaluate circuits in parallel. In particular, we systematically look at three different levels of *compiler assisted parallelization* that have the potential to significantly speed up applications based on secure computation.

FINE GRAINED PARALLELIZATION. As the first step, we observe that independent gates, i.e., gates that do not provide input to each

other, can be garbled and evaluated in parallel. Therefore, a straight forward parallelization approach is to garble gates in parallel that are located at the same circuit depth, because these are guaranteed to be independent. We refer to this approach as *Fine-Grained Parallelization (FGP)*. We will see that this approach can be efficient for circuits of suitable shape. Nevertheless, the achievable speed-up heavily depends on circuit properties such as the average circuit width, which can be comparably low even for larger functionalities when compiling from a high-level language.

COARSE GRAINED PARALLELIZATION. To overcome the limitations of FGP for inadequately shaped circuits, we make use of high-level circuit descriptions, such as program blocks, to automatically detect larger coherent clusters of gates that can be garbled independently. We refer to this parallelization as *Coarse-Grained Parallelization (CGP)*. We describe how CBMC-GC can be extended to detect concurrency at the source code level to enable the compilation of parallel circuits. Hence, one large circuit is automatically divided into multiple smaller, independently executable circuits. We show that these circuits lead to more scalable and faster execution on parallel hardware. Furthermore, integrating automatic detection of parallel regions into a circuit compiler gives potential users the opportunity to exploit parallelism without detailed knowledge about Boolean circuit based MPC and thus, relieves them of writing parallel circuits.

INTER-PARTY PARALLELIZATION (IPP). Finally, we present an extension to Yao's Garbled Circuits protocol itself (secure against semi-honest adversaries), which balances the computation costs of both parties for parallel circuits. Thus, Inter-Party Parallelization (IPP) allows to profit from parallelization without the necessity of parallel hardware.

In the original protocol (using the point-and-permute optimization [Bea+91; Mal+04]), the garbling party has to perform four times the cryptographic work than the evaluating party. Hence, assuming similar computational capabilities the overall execution time is dominated by the garbling costs. Given the identified coarse-grained parallelism, the idea of the proposed protocol is to divide the work in a symmetric manner between both parties by switching the roles of the garbling and evaluating party to achieve better computational resource utilization without compromising security in the semi-honest model. This approach can greatly reduce the overall runtime. Moreover, we show how IPP and CGP can be combined, hence using load balancing and parallel execution to decrease the runtime even further.

CHAPTER OUTLINE. Next, we describe the basics of parallel circuits and their parallel evaluation in Yao's Garbled Circuits in Sec-

tion 5.2. Then we introduce different parallelization approaches as well as a compiler extension to CBMC-GC in Section 5.3. Moreover, the idea of IPP is presented in Section 5.4. In Section 5.5 an evaluation the presented parallelization approaches in a practical setting alongside example applications is given. Finally, a comparison with related work is given in Section 5.6.

5.2 PARALLEL CIRCUIT EVALUATION

We first discuss the basic notations of sequential and parallel circuit decomposition used throughout this chapter, before describing how Yao's Garbled Circuits protocol can be parallelized.

PARALLEL AND SEQUENTIAL CIRCUIT DECOMPOSITION. In this chapter, we again consider a given functionality $f(x, y)$ with two input bit strings x, y (representing the inputs of the parties) and an output bit string o. Furthermore, we use C_f to denote the circuit that represents functionality f. We refer to a functionality f as *sequentially* decomposable into two *sub-functionalities* f_1 and f_2 iff $f(x, y) = f_2(f_1(x, y), x, y)$.

Moreover, we consider a functionality $f(x, y)$ as parallel decomposable into sub-functionalities $f_1(x, y)$ and $f_2(x, y)$ with non-zero output bit length, if a bit string permutation σ_f exists such that $f(x, y) = \sigma_f(f_1(x, y)||f_2(x, y))$, where $||$ denotes bitwise concatenation.

Thus, functionality f can directly be evaluated by independent evaluation of f_1 and f_2. We observe that the permutation σ_f is only a formal requirement, yet when representing f_1 and f_2 as circuits, the permutation is instantiated by connecting input and output wires at no cost. Furthermore, we note that f_1 and f_2 do not necessarily have to be defined over all bits of x and y. Depending on f they could share none, some, or all input bits. We use the operator $\diamond$ to express a parallel composition of two functionalities through the existence of a permutation σ. Thus, we write $f(x, y) = f_1(x, y) \diamond f_2(x, y)$ if there exists a permutation σ_f such that $f(x, y) = \sigma_f(f_1(x, y)||f_2(x, y))$.

We call a parallelization of f to be *efficient* if the (non-linear) circuit size of the parallelized functionality is roughly equal to the circuit size of the sequential functionality: $size(C_f) \approx size(C_{f_1}) + size(C_{f_2})$. Furthermore, we refer to a parallelization as *symmetric* if sub-functionalities have almost equal circuit sizes: $size(C_{f_1}) \approx size(C_{f_2})$.

Finally, we refer to functionalities that can be decomposed into a sequential and a parallel part as *mixed functionalities*. For example the functionality $f(x, y) = f_3(f_1(x, y) \diamond f_2(x, y), x, y)$ can first be decomposed sequentially in f_3 and $f_1 \diamond f_2$, where the latter part can then be further decomposed in f_1 and f_2. Without an explicit formalization, we note that all definitions can be extended from the dual case f_1 and f_2 to the general case $f_1, f_2, \ldots, f_n$.

PARALLEL CIRCUIT CREATION AND EVALUATION. A decomposition of a circuit into sequential and parallel sub-circuits forms a DAG from inputs to output bits. Parallel sub-circuits can be garbled in any order by one or multiple computing units (threads). This is exemplary illustrated in Figure 13, where three threads are used to garble and two thread to evaluate a circuit. We note that the garbling order has no impact on the security [LP09]. After every parallel decomposition a synchronization between the different threads is needed to guarantee that all wire labels for the next sequential sub-circuit are computed. Multiple subsequent parallel regions with possibly different degrees of parallelism can be garbled, when ensuring synchronization in between.

The circuit evaluation can be parallelized in the same manner. Sequential sub-circuits are computed sequentially, parallel sub-circuits are computed in parallel by different threads. After every parallelization a thread synchronization is required to ensure data consistency.

For efficiency reasons, implementations of Yao's garbled circuits commonly do not need to store the gate id next to the computed garbled table, as their garbling and evaluation order is deterministic. Hence, given a list of garbled tables, the garbled table associated with each gate can be identified by the position in the list. When using parallelization in combination with pipelining (i.e., garbled tables are sent immediately after their generation) the order of garbled tables could be unknown to the evaluator. Hence, a mapping mechanism between gate and garbled table has to be used to ensure data consistency between generator and evaluator. We propose three different variants. First, all garbled tables can explicitly be numbered, which allows an unordered transfer to the evaluator. The evaluator is then able to reconstruct the original order based on the introduced numbering. This approach has the disadvantage of an increased communication cost. Second, garbled tables could be sent in a strict order using thread synchronization. This approach functions without additional communication, yet could lead to an undesirable "pulsed" communication pattern, as threads have to wait for each other. The third approach functions by strictly separating the communication channels for every parallel sub-circuit. This can either be realized by multiplexing within the MPC framework or by exploiting the capabilities of the underlying operating system. Due to the aforementioned reasons, the results described in Section 5.5 are based on an implementation using the latter approach.

5.3 COMPILER ASSISTED PARALLELIZATION HEURISTICS

To exploit parallelism in Yao's protocol, groups of gates that can be garbled independently need to be identified. As described, independent gates can be garbled and evaluated in parallel. However, de-

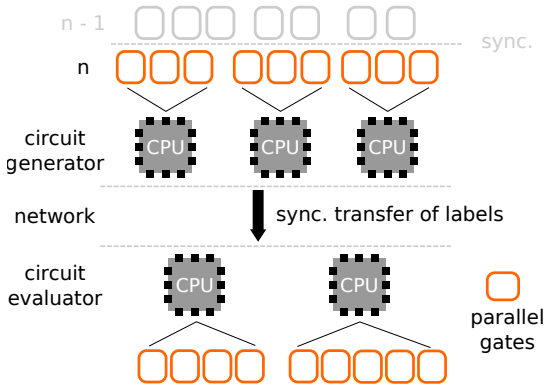


Figure 13: Interaction between a parallel circuit generator and evaluator. The layer n of the presented circuit is garbled and evaluated in parallel. The independent partitions of the circuit can be garbled and evaluated by different threads in any order.

tecting independent, similar sized groups of gates is known as the NP-hard graph partitioning problem [MG90]. The common approach to circumvent the expensive search for an optimal solution is to use heuristics. In the following paragraphs we study a fine- and a coarse-grained heuristic, where the first operates on the gate level and the latter on the source code level.

5.3.1 *Fine-Grained Parallelization (FGP)*

A first heuristic that decomposes a circuit into independent parts is the *fine-grained* gate level approach. Similar to the evaluation of a standard Boolean circuit, gates in garbled circuits are processed in topological execution order. Gates provide input to other gates and hence, can be ordered by the circuit level (depth) when all their inputs are ready or the level when their output is required for succeeding gates. Consequently, gates on the same level can be garbled in parallel [Bar+13; Hus+13]. Thus, a circuit is sequentially decomposable into different levels and each level is further decomposable in parallel with a granularity up to the number of gates in each level. Figure 14 illustrates fine-grained decomposition of a circuit into three levels L1, L2 and L3.

COMPILER-ASSISTED FGP IN YAO'S GARBLED CIRCUITS. With millions of gates garbled and evaluated during the protocol execution, it is useful to identify and annotate the circuit levels already during compilation to achieve an efficient distribution of gates onto threads during protocol runtime. Furthermore, FGP can be improved by considering linear and non-linear gates independently (because they have very different workloads) when distributing them onto all threads, as this enables a more symmetric workload distribution among multiple threads. Consequently, the computation will not be

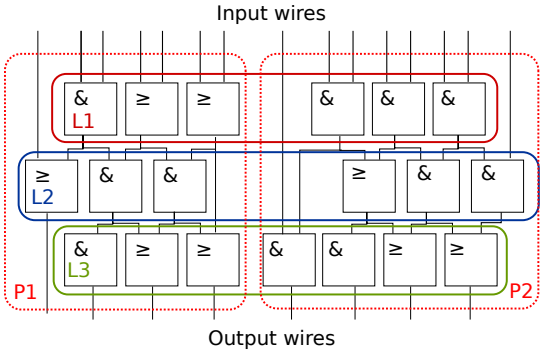


Figure 14: Circuit decomposition. Each level *L1*, *L2* and *L3* consists of multiple gates that can be garbled using FGP with synchronization in between. The circuit can also be decomposed in two coarse-grained partitions *P1* and *P2*.

stalled by threads waiting for other threads. Therefore, each thread gets assigned a similar number of linear and non-linear gates to garble per circuit level. This is exemplary illustrated in Figure 15, where two threads share the task of garbling a circuit layer with a similar workload.

We extended the CBMC-GC compiler with the capability to mark levels and to strictly separate linear from non-linear gates within each level. This information is stored in the circuit description that is then interpreted in a protocol implementation.

Finally, we remark that when using Non-Uniform Memory Access (NUMA) hardware, e.g., multiple cores on a CPU that differentiate between local and shared storage or multiple CPUs on different sockets, the efficiency of FGP can further be improved by fine-tuning the distribution of gates onto threads to maximize the storage proximity of gates with sequential dependencies. In this way, more gates, which are directly connected, can be garbled on the same CPU core with more efficient caching and less communication between different cores.

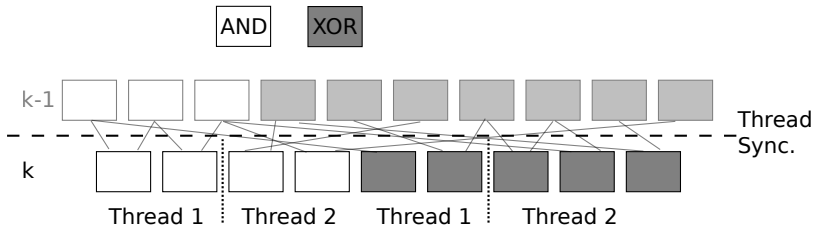


Figure 15: Illustration of the fine grained parallelization approach for one party. Level k is distributed symmetrically on two threads. Each thread is tasked to garble two non-linear gates. In-between two levels, a thread synchronization is needed to ensure data availability.

OVERHEAD. In practice, multi-threading introduces an computational overhead due to thread management and thread synchronization. To decide whether parallelization is useful on a given hardware, it is useful to determine a system dependent threshold τ that describes the minimal number of gates that are required per level to profit from parallel execution. When sharing the workload of less than τ gates onto multiple threads, the performance might actually decrease. Even though τ is circuit dependent, it is a very fast heuristic to decide on the effectiveness of FGP per circuit layer.

For example, in our testbed (see Section 5.5) we observe that at least ~ 8 non-linear gates per core are required to observe first speed-ups. To reach a parallelization efficiency of 90%, i.e., a speed up of 1.8 on 2 cores, at least 512 non-linear gates per core are required, which is not guaranteed for most applications. This limitation can be overcome with Coarse-Grained Parallelization.

5.3.2 *Coarse-Grained Parallelization (CGP)*

Another useful heuristic to partition a circuit is to use high-level functionality descriptions. Given a circuit description in a high-level language, parallelizable regions of the code can be identified using code analysis techniques. The detected code regions allow a parallel code decomposition; independent code parts can then be compiled into *sub-circuits* that are guaranteed to be independent of each other and therefore can be garbled in parallel. We refer to this parallelization scheme as *Coarse-Grained Parallelization (CGP)*. Figure 14 illustrates such a decomposition for an exemplary circuit in two coarse-grained partitions P1 and P2. In the following paragraphs, we introduce a compiler extension for CBMC-GC that automatically produces coarse-grained parallel circuits. Furthermore, we note that FGP and CGP can be combined by utilizing FGP within all coarse partitions.

COMPILER EXTENSION FOR CGP IN YAO'S GARBLED CIRCUITS. Our parallel circuit compiler extension *ParCC* extends CBMC-GC to compile circuits with coarse-grained parallelism. Its core functionality is to identify data parallelism in loops and to decompose the source code in a pre-processing step before its translation onto the circuit level. Conceptually, ParCC detects parallelism within ANSI C code carrying CBMC-GC's I/O annotations and compiles a *global circuit* that is interrupted by one or multiple *sub-circuits* for every parallel code region. The global circuit and all sub-circuits are interconnected by *inner* input and output wires. These are not exposed as inputs or outputs to the MPC protocol, but allow the recombination of the complete and parallel executable circuit. If a parallel region follows the Single-Instruction-Multiple-Data (SIMD) paradigm, it is sufficient

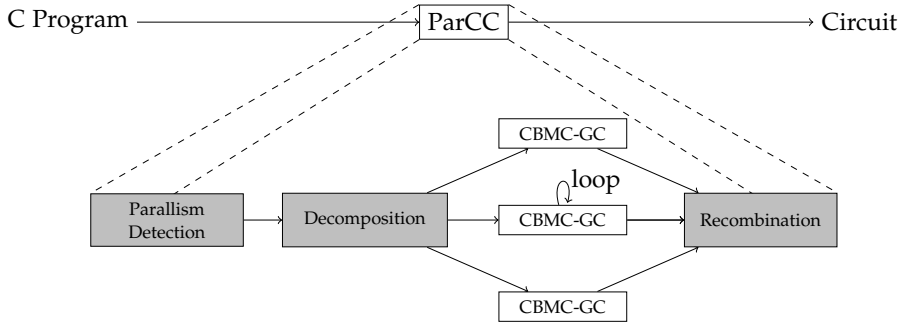


Figure 16: ParCC's compilation pipeline. First, parallelism in the input source is detected. Then, the code is decomposed in parallel and sequential parts. Each part is compiled individually with an unmodified variant of CBMC-GC. Finally, the resulting sub-circuits are recombined into a single circuit with annotated parallelism.

to compile only one sub-circuit to save compilation time as well as storage cost.

Our implementation is built on top of state of the art source code parallelization tools to detect parallelism and to perform code transformations. Namely, the parallellization framework *Pips* [Bon+08] is used to detect and to annotate parallelism with the polyhedral loop optimizer *POCC* [Pou12]. The source-to-source transformation of the annotated code, i.e., the program decomposition described below, is realized with the help of Pips, as well as the static source code analysis toolkit *Frama-C* [Cuo+12]. The complete compilation process, as illustrated in Figure 16, consists of four different steps:

(1) In the first step, parallelism in C code is detected by one of the algorithms provided by Pips and annotated using the *OpenMP* notation [DM98]. OpenMP is the de-facto application programming interface for shared memory multiprocessing programming in C.

(2) The annotated C code is parsed by ParCC in the second step. According to the identified and annotated parallelism, the source code is decomposed using source-to-source techniques into a *global* sequentially executable part, which is interrupted by one or multiple parallel executable *sub*-parts. This is realized by exporting each loop body into an individual function, and providing an interface between the original code location and the newly introduced function in form of CBMC-GC I/O variables. Additionally, OpenMP reduction statements, e.g, `sum`, are replaced with a code template and become part of the global part that is later compiled into an according circuit. Reduction functions require a separate treatment, as they are not embarrassingly parallel. Furthermore, information about the degree of detected parallelism as well as the interface between the global and sub-parts is extracted for later compilation steps.

(3) Given the decomposed source code, the different parts are compiled independently with CBMC-GC. Hence, one global and multiple independent sub-circuits are created.

(4) In the final step information about the mapping of wires between gates in the global and the sub-circuits is exported for use in MPC frameworks. This information is determined based on the I/O variables identified in step (2) and the I/O mapping between wires and I/O variables created for each sub-circuit in step (3). Furthermore, for performance reasons, we distinguish static wires that are shared between parallel sub-circuits and wires that are dedicated for each individual sub-circuit.

COMPILATION EXAMPLE. To illustrate the functionality of ParCC, we discuss the source-to-source compilation steps on a small *fork and join* task, namely computation of the dot product between two vectors **a** and **b** of length n:

$$r = \mathbf{a} \cdot \mathbf{b} = a_0 \cdot b_0 + \cdots + a_{n-1} \cdot b_{n-1}.$$

The source code of the function `dot_product()` using CBMC-GC's I/O notation is presented in Listing 8.

```
1 void dot_product() {
2   int INPUT_A_a[100], INPUT_B_b[100];
3   int res = 0;
4   for(i = 0; i < 100; i++)
5     res += INPUT_A_a[i] * INPUT_B_b[i];
6   int OUTPUT_res = res;
7 }
```

Listing 8: Dot vector product written in C with CBMC-GC input/output notation.

In this example code, two parties provide input for one vector in form of constant length integer arrays (Line 2). A loop iterates pairwise over all array elements (Line 4), multiplies the elements and aggregates the result. In the first compilation step, Pips detects the loop parallelism and annotates this parallel region accordingly. The annotated code is printed in Listing 9, with the OpenMP annotation added in Line 2.

```
1 [...]
2 #pragma omp parallel for reduction(+:res)
3 for(i = 0; i <= 99; i++) {
4   res += INPUT_A_a[i] * INPUT_B_b[i];
5 }
6 [...]
```

Listing 9: Annotation of parallelism as detected and added by the Pips framework after the first compilation step of the dot vector product example.

ParCC parses the annotations in the second compilation step and exports the loop body in a new function named `loop0()`, as it is the first loop encountered during compilation. The exported function is printed in Listing 10.

```
1 void loop0(int INPUT_A_0, int INPUT_A_1, int
      OUTPUT_return)
2 {
3   OUTPUT_LOOP0_return = INPUT_A_0 * INPUT_A_1;
4 }
```

Listing 10: Exported sub-function with CBMC-GC input-output notation.

The function expects two integer variables as input according to the notation of CBMC-GC. The result is returned in form of an (inner) output variable. Note, that during the protocol execution all inner wires are not assigned to any party, instead they connect gates in the global circuit and sub-circuits. Yet, to keep compatibility with CBMC-GC a concrete assignment for the party P_0 is specified. The later exported mapping information is used to distinguish between inner wires and actual input wires of both parties. Moreover, in the same step, the global function `dot_product()`, printed in Listing 11, is transformed by ParCC to replace the loop by an unrolled version of itself.

```
1  void dot_product() {
2    int INPUT_A_a[100], INPUT_B_b[100];
3    int res = 0;
4    int OUTPUT_LOOP0_a[100];
5    int OUTPUT_LOOP0_b[100];
6    int i;
7    for(i = 0; i <= 99; i++) {
8      OUTPUT_LOOP0_a[i] = INPUT_A_a[i];
9      OUTPUT_LOOP0_b[i] = INPUT_B_b[i];
10   }
11   int INPUT_A_LOOP0_res[100];
12   for(i = 0; i <= 99; i++)
13     res += INPUT_A_LOOP0_res[i];
14   int OUTPUT_res = res;
15 }
```

Listing 11: Rewritten function `dot_product()`. The loop has been replaced by inner input/output variables (marked with `LOOP0`).

For this purpose, the two arrays `INPUT_A_a` and `INPUT_B_b`, which are iterated over in the original loop, are now exposed as inner output variables to create the mapping between the global circuit and sub-circuits. Therefore, from Line 4 to Line 10 ParCC added two new output arrays using CBMC-GC's I/O notation that are assigned to the two arrays. Furthermore, an inner input array for the results of the exported sub-functionality is introduced in Line 11. Finally, the re-

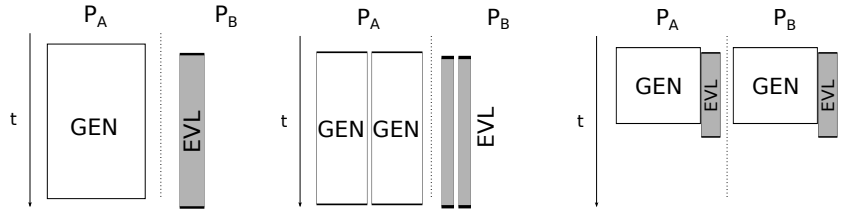
duction statement is substituted by synthesized additions over all intermediate results in Line 13. Multiple parallel parts in a given source code are exported and handled individually by independent decomposition of each part.

5.4 INTER-PARTY PARALLELIZATION

In this section, we show how a better load balancing between the computing parties can be achieved in Yao's protocol using a coarse grained circuit decomposition, as created by ParCC. We first illustrate the general idea and then give a detailed protocol extension that allows to balance computation and communication costs between parties, assuming *symmetric efficiently parallelizable* functionalities. We refer to this protocol extension as IPP. Without compromising security, we show in Section 5.5 that the protocol runtime can be reduced in practical applications when using IPP. This is also the case when using only one CPU core per party.

LOAD BALANCING THROUGH IPP. The computational costs that each party has to invest in Yao's protocol secure against semi-honest adversaries is driven by the encryption and decryption costs of the garbled tables as well as their communication costs. Considering the garbling technique with the least number of cryptographic operations, namely Garbled Row Reduction (GRR) combined with free-XOR (see Section 2.2.2), the generator has to compute four ciphertexts per non-linear gate, whereas the evaluator has to compute only one ciphertext per non-linear gate. When considering the communication optimal half-gate approach [ZRE15], the generator has to compute four and the evaluator two ciphertexts per non-linear gate. Assuming two parties that are equipped with similar computational power, a better overall resource utilization would be achieved, if both parties could be equally involved in the computation process. This can be realized efficiently for parallel functionalities by sharing the roles of generator and evaluator, i.e., both parties are equally involved in generating and evaluating the garbled circuit. Thus, the overall protocol runtime could be decreased. Figure 17 illustrates this efficiency gain.

APPLICATIONS OF IPP. IPP is most beneficial in Yao's protocol secure against semi-honest adversaries. Yao's protocol secure against malicious adversaries is commonly build using cut-and-choose techniques, which tend to have a more symmetric workload and thus, requires less load balancing. However, one can imagine many scenarios where the semi-honest adversary model is sufficient and of interest. These are scenarios where either the behavior is otherwise restricted, e.g., limited physical access, or where the parties have sufficient trust into each other. Moreover, IPP can be used in all the scenarios where



(a) Original (b) Independence Analysis (c) Inter-Party Parallelization

Figure 17: The idea and performance gain of IPP visualized. The OT phase and output sharing are omitted. In Figure 17a the sequential execution of Yao's protocol is visualized. Given a parallel decomposition by two circuits representing parallal program regions as displayed in Figure 17b, the protocol runtime can be reduced when sharing the roles generator and evaluator, as displayed in Figure 17c.

the parties inputs and seeds could be revealed at a later point to identify cheating parties. Examples might be negotiations (auctions) or games, such as online poker. Another field of application is the joint challenge creation, e.g., factorization in RSA. Using secure computation, two parties can jointly create a problem instance without already knowing the solution. This allows them to create a problem and to participate in the challenge at the same time without a computational advantage. Once a solution is computed, all parts of the secure computation can be verified in hindsight, as shown by Buchmann et al. [Buc+16]. We also note that the core idea of IPP could be applied in other MPC protocols. To profit from IPP, a secure state sharing mechanism is required as well as an asymmetric workload between the parties. One example might be the highly asymmetric STC protocol by Jarecki and Shmatikov [JS07] that uses zero-knowledge proofs over every gate.

In the following two sub-sections we first show how to extend Yao's protocol to use IPP for purely parallel functionalities. In a second step we generalize this approach by showing how mixed functionalities profit from IPP.

5.4.1 *IPP for Purely Parallel Functionalities*

We assume that two parties P_0 and P_1 agree to compute a functionality $f(x, y)$ with x being P_0's input and y being P_1's input. Moreover, we assume $f(x, y)$ to be parallelizable into sub-functionalities $f_0, \ldots, f_n$:

$$f(x, y) = f_0(x, y) \diamond \ldots \diamond f_n(x, y).$$

Given such a decomposition, all sub-functionalities can be computed independently with any MPC protocol (secure against semi-honest adversaries) without any sacrifices towards the security [HL10].

This observation allows us to run two independent executions of Yao's protocol, each for one half of f's sub-functionalities, instead of computing f with a single execution of Yao's protocol. Hence, P_0 could garble one half of f's sub-functionalities, for example $f_{even} = f_0, f_2, \ldots$, and P_0 could evaluate the other half $f_{odd} = f_1, f_3, \ldots$. Vice versa, P_1 could evaluate f_{even} and garble f_{odd}. Applying this approach to Yao's Garbled Circuits, P_0 and P_1 have to incorporate both roles (sender and receiver) during the OT phase of Yao's protocol. In the output phase, both parties have to share their output labels with each other.

ANALYTICAL PERFORMANCE GAIN. As discussed, the computational costs for Yao's protocol are dominated by encrypting and decrypting the garbled tables. Thus, idealizing and highly abstracted, the time spent to perform a computation t_{total} is dominated by the time to garble a circuit t_{garble}. Using GRR with free-XOR, which allows to assume that t_{garble} is approximately four times the time to evaluate a circuit t_{eval}, by symmetrically sharing this task the total time could be reduced to:

$$t'_{total} \approx \frac{(t_{garble} + t_{eval})}{2} \approx \frac{(4 \cdot t_{eval} + t_{eval})}{2} \approx 2.5 \cdot t_{eval}.$$

This result translates to a theoretical speed-up of $t_{total}/t'_{total} = 4/2.5 = 1.6$. When using the half-gate approach the approximate computational speed-up is 1.33.

TRADE-OFF. Investigating the trade-off of IPP, we observe that during the cost intensive garbling and evaluation phase, no computational complexity is added. Particularly, the number of cryptographic operations and messages is left unchanged. However, if both parties provide a different number of input bits, the overall number of OTs could be increased up to half of the total number of input bits. This is only the case if the garbling party in the traditional execution will provide more inputs that the evaluating party. Moreover, when using OT Extension, a constant one-time overhead for the base OTs in the size of the security parameter k is introduced to establish an OT Extension in both protocol directions, cf. Section 2.2.1. A detailed experimental study on the performance gain through IPP is given in Section 5.5.5.

5.4.2 *IPP for Mixed Functionalities*

To exploit IPP in mixed functionalities, a protocol extension is required, allowing to switch from sequential (dedicated roles) to IPP (shared roles) without violating the privacy. Therefore, we introduce the notion of *transferring roles* to securely interchange between IPP and sequential execution.

TRANSFERRING ROLES. The core idea of the transferring roles protocol is to run a secure computation up to the point where the roles of the parties are changed. Then, the state of the protocol is secret shared (using Boolean sharing) between the parties. Afterwards a second secure computation is invoked, using the secret shared state as input for both parties. We describe the full protocol in two steps. First we sketch an insecure protocol, which is then made secure in a second step.

To switch the roles of evaluator and generator during execution, we consider two parties P_0, P_1 and the sequentially composed functionality $f(x, y) = f_2(f_1(x, y), x, y)$. In the following description, f_1 is computed using Yao's protocol with P_0 being generator and P_1 being evaluator, f_2 is computed with reversed roles.

The transfer protocol begins by computing $f_1(x, y)$ with Yao's original protocol. Once f_1 is computed, the roles have to be switched. For this purpose, the parties reveal the intermediate result $o_1 = f_1(x, y)$ to each other by opening the output wires. In the second phase of the protocol, f_2 is computed using Yao's protocol. This time, P_0 and P_1 switch roles, such that P_1 garbles f_2 and P_0 evaluates f_2. The decrypted output bits $o_1 = f_1(x, y)$ are used by P_0 as input to Yao's protocol, i.e., as input to the OT protocol. After garbling f_2, the output is shared between both parties. This protocol resembles a pause/continue pattern and preserves correctness. However, this protocol leaks o_1 to both parties, which violates the privacy of MPC. Therefore, we propose to use an XOR-blinding (Boolean sharing) during the role switch. The full protocol is shown below.

Protocol: Transferring Roles

P_0 and P_1 agree to securely compute the sequentially decomposable functionality $f(x, y) = f_2(f_1(x, y), x, y)$ without revealing the intermediate result $f_1(x, y)$ to either party, where x is P_0's input bit string, y is P_1's input bit string. The protocol consists of two phases, one per sub-functionality.

Phase 1: Secure computation of $f_1(x, y)$

1. f_1 is extended with a XOR blinding for every output bit. Thus, the new output $o_1' = f_1'(x, y||y_r) = f_1(x, y) \oplus y_r$ is calculated by xor-ing the output of f_1 with additional, randomly drawn input bits by the evaluator of f_1.

2. P_0 and P_1 securely compute f_1' using Yao's protocol. We assume P_0 to be the generator. Additional randomly drawn input bits are then input of P_1.

3. The blinded output o_1' of the secure computation is only made visible to the generator P_0. This is realized by trans-

mitting the output wire labels to P_0, but not sharing the decrypted result with P_1.

Phase 2: Secure computation of $f_2(o_1, x, y)$

1. The circuit representing f_2 is extended with a XOR unblinding for every input bit of o_1'. Hence, $f_2'(o_1', x, y, y_r) = f_2(o_1' \oplus y_r, x, y)$.
2. P_0 and P_1 securely compute f_2' using Yao's protocol. We assume P_1 to be the generator. P_0 provides the input o_1' and P_1 provides the input bits for the blinding with y_r.
3. The output of the computation is shared with both parties.

We observe that, informally speaking the protocol preserves privacy, since the intermediate state o_1 is secret shared between both parties. A detailed formal proof on sequentially decomposed functionalities is given by Hazay and Lindell [HL10, page 42ff]. Correctness is preserved due to blinding and unblinding with the equal bit string y_r:

$$\begin{aligned} f_2'(f_1'(x, y||y_r), x, y, y_r) &= f_2'(f_1(x, y) \oplus y_r, x, y, y_r) \\ &= f_2(f_1(x, y) \oplus y_r \oplus y_r, x, y) \\ &= f_2(f_1(x, y), x, y). \end{aligned}$$

Finally, we note that the transferring roles protocol can further be improved for efficiency. Namely, when using the de facto standard point-and-permute optimization [Mal+04], the point-and-permute bits attached to every wire label already form a Boolean sharing between generator and evaluator, cf. [DSZ15]. Using this sharing, it is possible to securely share the intermediate state o_1 without any additional computation or communication. The experimental evaluation given in Section 5.5.1 utilizes this idea.

TRANSFERRING ROLES FOR MIXED FUNCTIONALITIES. With the idea of transferring roles, IPP can be realized for mixed functionalities. In the following paragraphs, we show how to switch from IPP to sequential computation. Switching in the other direction, namely from sequential to IPP can be realized analogously. With protocols to switch in both directions, it is possible to garble and evaluate any functionality that consists of an arbitrary number of sequential and parallel regions.

To show the switch from IPP to sequential computation, we assume a functionality that is sequentially decomposable into a parallel and a sequential functionality:

$$f(x, y) = f_3(f_1(x, y) \diamond f_2(x, y), x, y).$$

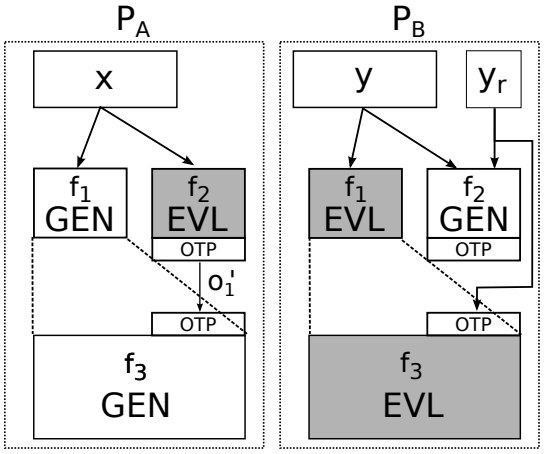


Figure 18: The IPP protocol for a mixed functionality with a switch from parallel to sequential computation. Functionality $f_1 \diamond f_2$ is garbled in parallel using IPP, f_3 is garbled sequentially in combination with f_1. No interaction between parties is shown. The blinded output o_1' of f_1 is only made visible to P_0 and used as additional input for the computation of f_3 using the transferring roles protocol.

Note that f_1, f_2 and f_3 could further be composed of any sequential and parallel functionalities. We observe that f_3 can be merged with f_1 (or f_2) into one combined functionality f_c. Thus, $f(x, y)$ can also be decomposed as $f(x, y) = f_c(f_2(x, y), x, y)$ with f_c being the sequential composition of f_3 and f_1. Given such a decomposition, f_c and f_2 can be computed with alternating roles in Yao's protocol by following the transferring roles protocol. Hence, f_c could be garbled by P_0 while f_2 could be garbled by P_1 to securely compute f.

As a second observation we note that the output of f_2 is not required to start the computation of f_c. Therefore, the computation of f_c can start in parallel to the computation of f_2. This inter-party parallelism can be exploited to achieve further speed-ups. Figure 18 illustrates this approach. Party P_0 garbles f_c and P_1 garbles f_2. The first part of f_c, namely f_1 can be garbled in parallel to f_2. Once the blinded output o_1' of f_2 is computed, the parties can start computing the second part of f_c, namely f_3. Switching from sequential to IPP computation can be realized in the same manner.

We remark that FGP, CGP and IPP can be combined to achieve even further speed-ups. Therefore, every parallel region has first to be decomposed in two parts for IPP. If the parts can further be decomposed in parallel functionalities, these could be garbled using CGP and FGP.

TRADE-OFF. IPP for mixed functionalities has the same trade-off has IPP for parallel functionalities, i.e., to switch from and to IPP in mixed functionalities, additional OTs in the size of the intermediate state are required. Thus, the performance gain through IPP for mixed

functionalities not only depends on the ratio between parallel and sequential regions, but also on the ratio of circuit size and secret shared state. These ratios are application dependent. An experimental evaluation of the trade-off between overhead and performance gain is presented in Section 5.5.5.

5.5 EXPERIMENTAL EVALUATION

To show the performance gains through parallelization, we describe an evaluation and implementation of FGP, CGP and IPP. We begin by introducing a parallel Yao's Garbled Circuits framework named *UltraSFE* and benchmark its performance on a single core in Section 5.5.1. The applications and their circuit descriptions used for benchmarking are described in Section 5.5.2. We evaluate the offline garbling performance of the proposed parallelization techniques in Section 5.5.3, before evaluating the promising CGP in an online setting in Section 5.5.4. Finally, in Section 5.5.5 we benchmark the IPP approach.

5.5.1 *UltraSFE*

UltraSFE is a framework for Yao's garbled circuits that implements CGP, FGP, as well as IPP. To realize efficient parallelization, all data structures, the memory layout, and the memory footprint are all optimized with the purpose of parallelization in mind. All these implementation optimizations are of importance, due to the specific resource requirements of Yao's garbled circuits. When garbling millions of gates per second, memory read and write accesses quickly become a bottleneck. Wire labels in the size of hundred(s) of megabytes per second have to be fetched and written from and to memory in an unaligned manner. Therefore, a reduction of the overall memory footprint leads to better caching behavior, which becomes even more important when using multi core architectures.

UltraSFE is written in C++ using SSE4, OpenMP and Pthreads to realize multi-core parallelization. Conceptually, UltraSFE implements the fixed-key garbling scheme *JustGarble* [Bel+13] and uses ideas from the Java based memory efficient *ME_SFE* framework [HS13], which itself is based on the *FastGC* framework [Hua+11b]. Oblivious transfers are realized with the help of the highly efficient and parallelized OTExtension library written by Asharov et al. [Ash+13]. Moreover, UltraSFE adopts the best known techniques and practical optimizations for Yao's protocol. This includes pipelining, point-and-permute, garbled row reduction, free-XOR, fixed-key garbling, and the half-gate approach [Bel+13; Hua+11b; KS08; Mal+04; Pin+09; ZRE15].

	BCPU [Bar+13]	HCPU [Hus+13]	KSS [KSS12]	GraphSC [Nay+15]	JG [Bel+13]	Ours
gates / second	0.11 M	<0.25 M	0.1 M	0.58 M	8.3 M	8.3 M
clocks / gate	>3500	-	>6500 [Bel+13]	> 1200	~ 110	~ 110
architecture	E5-2609	E5-2620	i7-970	E5-2666 v3	E5-2680 v2	E5-2680 v2

Table 6: Single core garbling speed comparison of different frameworks on circuits with more than 5 million gates. Metrics are the number of *non-linear gates per second* that can be garbled on a single core in millions (M) and CPU *clocks per gate*. All results have been observed on the Intel processor specified in row *architecture*. Note, for HCPU [Hus+13] only circuit evaluation times have been reported on the CPU, the garbling speed can be assumed to be lower, as it requires four times the number of encryptions.

FRAMEWORK COMPARISON. To illustrate the practical performance gains through parallelization schemes in a fair manner, it is necessary to compare the results to a highly optimized single core implementation. To illustrate that UltraSFE is suited to evaluate the scalability of different parallelization approaches, we present a comparison of its garbling performance with other state-of-the-art (at the time of publication) frameworks for CPU architectures in Table 6. Namely, we compare the single core garbling speed of UltraSFE, which is practically identical to the performance of the JustGarble (*JG*) implementation by Bellare et al. [Bel+13], with the parallel frameworks by Barni et al. (*BCPU*) [Bar+13], Husted et al. (*HCPU*) [Hus+13], Kreuter et al. (*KSS*) [KSS12], and *GraphSC* by Nayak et al. [Nay+15]. Note, these results are compared in the *offline* setting, i.e., truth tables are written to memory, rather then sent to the evaluator. This is because circuit garbling is the most cost intensive part of Yao's protocol and therefore the most interesting when comparing the performance of different frameworks. The previous parallelization efforts HCPU and BCPU actually abstained from implementing an *online* version of Yao's protocol that supports pipelining. As metrics we use number of garbled non-linear gates per second and the average number of CPU clock cycles per gate, as proposed in [Bel+13] for a more processor independent benchmark. The numbers are taken from the cited publications and if not given, the clock cycles per gate results are calculated based on the CPU specifications. Even when considering theses numbers only as rough estimates, due to the different CPU types, we observe that UltraSFE performs approximately 1-2 orders of magnitude faster than existing parallelizations of Yao's protocol. This is mostly due to the efficient fixed-key garbling scheme using the AES-NI hard-

	BioMatch	MExp	MVMul
Code size	22 LOC	28 LOC	10 LOC
Circuit size	66 M	21.5 M	3.3 M
Fraction of non-linear gates	25%	41%	37%
# Input bits P_0/P_1	131K/256	1K/1K	17K/1K
Offline garbling time	2.07s	1.136s	0.154s

Table 7: Circuit properties of benchmarked functionalities. Presented are the code size, the overall circuit size in the number of gates, the fraction of non-linear gates that determine the majority of computing costs, the number of input bits as well as the sequential offline garbling time with UltraSFE.

ware extension and a carefully optimized implementation using SSE4. Summarizing, UltraSFE shows competitive garbling performance on a single core, which makes it a very promising candidate to study the effectiveness of parallelization.

5.5.2 *Evaluation Methodology*

To evaluate the different parallelization approaches we use three example applications that have been used to benchmark and compare the performance of Yao's garbled circuits in the past. A detailed description of these can be found in Section 2.5. The benchmarked applications and the chosen configurations are:

- *Biometric matching (BioMatch).* In BioMatch one party matches a biometric sample against the other's party database of biometric templates. In the evaluation we use the squared Euclidean distance as distance function between two samples, a database of size $n = 512$, a degree (number of features) of $d = 4$, and an integer bit-width of $b = 64\,\mathrm{bit}$. These values have been used in previous works on MPC [DSZ15; KSS14].

- *Parallel Modular exponentiation (MExp).* MExp can be used for blind signatures and its parallel version in online signing services. We study a circuit with $k = 32$ parallel instances of modular exponentiation and an integer bit-width of $b = 32\,\mathrm{bit}$.

- *Matrix-vector multiplication (MVMul).* MVMul is a building block for many privacy-preserving applications. We parameterize this task according the size of the matrix $m \times k = 16 \times 16$ and vector of length $k = 16$, as well the integer bit-width of each element $b = 64\,\mathrm{bit}$.

CIRCUIT CREATION. All circuits are compiled twice with CBMC-GC using textbook C implementations, once with ParCC enabled and once without. The time limit for the circuit minimization through CBMC-GC is set to 10 minutes. The resulting circuits and their properties are shown in Table 7. The BioMatch circuit is the largest circuit and has the most input bits. The MVMul garbles in a fraction of a second and thus, is suitable to evaluate the performance of parallelelization on smaller circuits. The MExp circuit shows a large circuit complexity in comparison to the number of input bits. Even so not shown here, we note that the sequential (CBMC-GC) and parallel (ParCC) circuits slightly differ in the overall number of non-linear gates due to the circuit minimization techniques of CBMC-GC, which profit from decomposition.

ENVIRONMENT. As testing environment we used Amazon EC2 cloud instances. These provide a scalable number of CPUs and can be deployed at different sites around the globe. If not stated otherwise, for all experiments instances of type `c3.8xlarge` have been used. These instances report 16 physical cores on 2 sockets with CPUs of type Intel Xeon E5-2680v2, and are equipped with a 10 Gbps ethernet connection. A fresh installation of Ubuntu 14.04 was used to ensure as little background noise as possible. UltraSFE was compiled with `gcc 4.8 -O2` and `numactl` was utilized when benchmarking with only a fraction of the available CPUs. Numactl allows memory, core and socket binding of processes. Results have been averaged over 10 executions.

METHODOLOGY. Circuit garbling is the most expensive task in Yao's protocol. Therefore, we begin by evaluating FGP and CGP for circuit garbling independent of other parts of Yao's protocol. This allows an isolated evaluation of the computational performance gains through parallelization. Following the offline circuit garbling phase is an evaluation of Yao's full protocol in an online LAN setting. This evaluation also considers the bandwidth requirements of Yao's protocol. Finally, we present an evaluation of the IPP approach in the same LAN setting. Therefore, we first evaluate the performance of IPP on a single core, before evaluating its performance in combination with CGP. The main metric in all experiments is the overall runtime and the number of non-linear gates that can be garbled per second.

5.5.3 *Circuit Garbling (offline)*

We begin the evaluation of FGP and CGP by studying the independent task of circuit garbling, which can be executed by the generator offline in a pre-processing phase. In practice the efficiency of any parallelization is driven by the ratio between computational workload

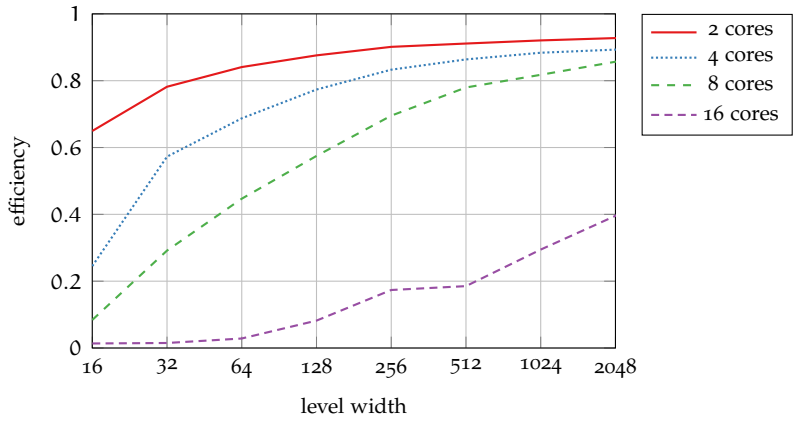


Figure 19: Thread utilization experiment. Shown is the efficiency of FGP for different circuit level widths, i.e., the number of non-linear gates per level. A larger width increases the efficiency of parallelization.

per thread and synchronization between threads. When garbling a circuit with FGP, the workload is bound by the width of each level. When garbling with CGP the workload is bound by the size of parallel partitions. Both parameters are circuit and hence, application dependent.

THREAD UTILIZATION. To get a better insight, we first empirically evaluate the possible efficiency gain for different sized workloads, independent of any application. This also allows to observe a system dependent threshold τ, introduced in Section 5.3.1, which describes the minimal number of gates required per thread to profit from parallelization. Therefore, the following experiment was run in the environment previously described. For level widths $w \in \{2^4, 2^5, \ldots, 2^{10}\}$ we created random circuits of depth $d = 1000$. The width is kept homogeneous in all levels. Furthermore, the wiring between gates on different levels is randomized and only non-linear gates are used. Each circuit is garbled using FGP and we measured the parallelization efficiency, which is the speed-up over the single core performance divided by the number of cores, when computing with different numbers of threads. The results are illustrated in Figure 19.

The experiment shows that on the tested system $\tau \approx 8$ non-linear gates per thread are sufficient to observe first performance gains through parallelization when using CPU cores that are located on the same socket. To achieve an efficiency of 90% approximately 512 non-linear gates per thread are required. The noticeable gap between 8 (1 socket with 8 cores) and 16 cores (2 sockets with 8 cores each) is due to the communication overhead between the sockets. Thus, a significantly larger workload per thread (at least one order of magnitude) is required to profit from further parallelization on a two socket machine.

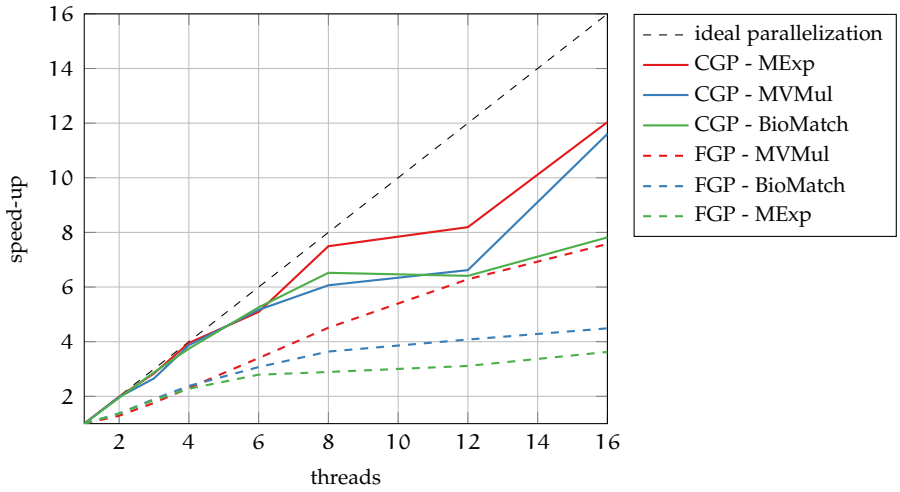


Figure 20: The speed-up of circuit garbling for all three applications when using the FGP, CGP and different numbers of computing threads. CGP significantly outperforms FGP for all applications.

EXAMPLE APPLICATIONS. We evaluate the speed-up of circuit garbling when using FGP and CGP for the three applications BioMatch, MExp and MVMul compiled with ParCC disabled (FGP) and enabled (CGP). The speed-up is calculated in relation to the single core garbling performance given in Table 7. The results, which have been observed for a security level of $k = 128$ bit, are presented in Figure 20. Studying the results for FGP, we observe that all applications profit from parallelization. BioMatch and MExp show very limited scalability, whereas the MVMul circuit can be executed with a speed-up of 7.5 on 16 cores. Analyzing the performance of CGP, we observe that all applications achieve practically ideal parallelization when using up to 4 threads. In contrast to the FGP approach, scalability with high efficiency is observable with up to 8 threads. When scaling to 16 threads (two sockets), significant further speed-ups are noticeable in the MExp and MVMul experiments, with a total throughput of more than 100M non-linear gates per second.

In summary, for all presented applications the CGP approach significantly outperforms the FGP approach regarding scalability and efficiency due to its coarser granularity, which implies a better thread utilization.

CIRCUIT WIDTH ANALYSIS. The limited scalability of FGP is explainable when investigating the different circuit properties. In Figure 21 the distribution of level widths for all circuits are shown when compiled with CBMC-GC.

For the MVMul application, the CBMC-GC compiler produces a circuit with a median level width of 2352 non-linear gates per level, whereas the BioMatch and MExp circuits only show a median width below 100 non-linear gates per level. The major reason for small cir-

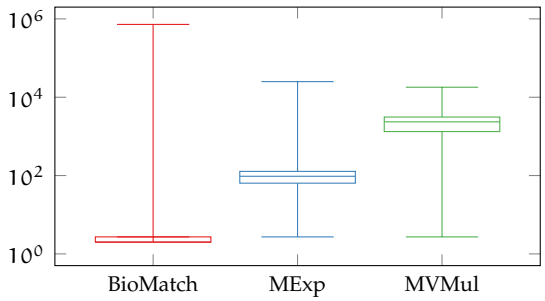


Figure 21: The distribution of non-linear gates per level (circuit width) when compiling the BioMatch, MExp and MVMul applications with CBMC-GC.

cuit widths in comparison to the overall circuit size is that high-level MPC compilers such as CBMC-GC have been developed with a focus on minimizing the number of non-linear gates. Minimizing the circuit depth or maximizing the median circuit width barely influence the sequential runtime of Yao's protocol and is therefore not addressed in the first place. These observations are also a motivation for the next chapter, where we study the depth minimization of circuits. Looking at the building blocks that are used in CBMC-GC, we observe that arithmetic blocks (e.g., adder, multiplier) show a linear increase in the average circuit width when increasing the input size. Integer comparisons have a constant circuit width for any input bit size. However, multiplexers (MUXs), as used for dynamic array accesses and for `if`-statements, have a circuit width that is independent (constant) of the number of choices. Thus, a 2-1 MUX and a n-1 MUX are compiled to circuits with similar sized levels, yet with different circuit depths. Based on these insights we deduce, that the MVMul circuit shows a significantly larger median circuit width, because of the absence of any dynamic array access, conditionals or comparisons. This is not the case with the BioMatch and MExp applications. Considering that every insufficient saturation of threads leads to an efficiency loss of parallelization, we conclude that scalability of FGP is not guaranteed when increasing input sizes.

5.5.4 *Full Protocol (online)*

To motivate that the parallelization of circuit garbling provides advantages in Yao's full protocol with pipelining and assuming a fast network connection, we study the protocol runtime for all applications running on two separated cloud instances in the same Amazon region (LAN setting). We observed an average round trip time of $0.6 \pm 0.3\,\mathrm{ms}$ and a bandwidth of 5.0 ± 0.4 `Gbps` using `iperf`. Following the results of the offline experiments, we only benchmark the more promising CGP approach in the online setting.

		BioMatch	MExp	MVMul
t_{total} [s]	$\kappa = 128$	2.71 ± 0.02	1.43 ± 0.01	0.20 ± 0.00
	$\kappa = 80$	2.56 ± 0.03	1.42 ± 0.01	0.19 ± 0.00
gates/second [M]	$\kappa = 128$	6.23 ± 0.04	6.17 ± 0.05	6.22 ± 0.00
	$\kappa = 80$	6.56 ± 0.07	6.21 ± 0.04	6.43 ± 0.00
bandwidth [Gbps]	$\kappa = 128$	1.48 ± 0.01	1.47 ± 0.01	1.48 ± 0.00
	$\kappa = 80$	0.97 ± 0.01	0.92 ± 0.01	0.95 ± 0.00
t_{input} [s]	$\kappa = 128$	< 0.02	< 0.01	< 0.01
	$\kappa = 80$	< 0.02	< 0.01	< 0.01

Table 8: Single-core performance of Yao's protocol. The runtime (t_{total}), non-linear gate throughput in million *gates per second*, required *bandwidth* and time spent in the input phase (t_{input}), including the OTs when executing Yao's protocol for all applications in a LAN setting.

To precisely measure the speed-up of parallelization, we first benchmark the single core performance of Yao's protocol in the described network environment. Table 8 shows the sequential runtime for all applications using two security levels $\kappa = 80$ bit (short term) and $\kappa = 128$ bit (long term). This runtime includes the time spent on the input as well as the output phase. Furthermore, the observed throughput, measured in non-linear gates per second, as well as the required bandwidth are presented. We observe that for security levels of $\kappa = 80$ and $\kappa = 128$ a similar gate throughput is achieved. Consequently, we deduce that in this setup the available bandwidth is not stalling the computation. We also observe that that the time spent on OTs in all applications is practically negligible ($< 5\%$) in comparison with the time spent on circuit garbling.

In Figure 22 the performance gain of CGP is presented. The speed-up is measured in relation to the sequential total runtime. The timing results show that CGP scales almost linearly up to 4 threads when using $\kappa = 80$ bit labels. Using $\kappa = 128$ bit labels, no further speed-up beyond 3 threads is noticeable. Thus, the impact of the network limits is immediately visible. Five ($\kappa = 80$ bit), respectively three ($\kappa = 128$ bit) threads are sufficient to saturate the available bandwidth in this experiment. Achieving further speed-ups is impossible without increasing the available bandwidth or using different MPC techniques, e.g., IPP, which is evaluated in the next sub-section.

Environment		κ = 80						κ = 128					
		BioMatch		**MExp**		**MVMul**		**BioMatch**		**MExp**		**MVMul**	
Cores	**IPP**	time[s]	S	time[s]	S	time[s]	S	time[s]	S	time[s]	S	time[s]	S
1	none	2.559	1.000	1.423	1.000	0.192	1.000	2.712	1.000	1.485	1.000	0.199	1.000
	2	2.624	0.975	**1.287**	**1.106**	0.206	0.932	2.781	0.975	**1.370**	**1.084**	0.215	0.926
	4	2.556	1.001	**1.285**	**1.107**	0.208	0.923	2.707	1.002	**1.386**	**1.071**	0.218	0.913
2	none	1.384	1.849	0.734	1.939	0.103	1.864	1.497	1.812	0.780	1.904	0.108	1.844
	1	1.524	1.679	**0.686**	**2.074**	0.126	1.524	1.535	1.767	**0.699**	**2.124**	0.134	1.485
	2	1.472	1.738	**0.699**	**2.036**	0.132	1.455	1.516	1.789	**0.726**	**2.045**	0.137	1.453
	4	1.396	1.833	**0.654**	**2.176**	0.124	1.548	1.465	1.851	**0.693**	**2.143**	0.131	1.519
4	none	0.795	3.219	0.395	3.603	0.057	3.368	0.937	2.894	0.450	3.300	0.064	3.088
	1	0.996	2.569	0.426	3.340	0.084	2.286	1.041	2.605	0.452	3.285	0.087	2.287
	2	0.830	3.083	**0.336**	**4.235**	0.085	2.259	**0.874**	**3.103**	**0.356**	**4.171**	0.088	2.261
	4	0.818	3.128	**0.329**	**4.325**	0.081	2.370	**0.856**	**3.168**	**0.341**	**4.355**	0.084	2.369
8	none	0.652	3.925	0.298	4.775	0.045	4.267	0.872	3.110	0.364	4.080	0.048	4.189
	1	0.676	3.786	**0.239**	**5.954**	0.072	2.667	0.947	2.864	**0.303**	**4.901**	0.080	2.488
	2	**0.629**	**4.068**	**0.204**	**6.975**	0.077	2.494	**0.861**	**3.150**	0.342	**4.342**	0.075	2.653
	4	**0.636**	**4.024**	**0.233**	**6.107**	0.070	2.743	0.871	3.114	**0.337**	**4.407**	0.073	2.726
Transferring roles		0.231s		0.076s		0.031s		0.257s		0.082s		0.031s	

Table 9: Evaluation of IPP in a LAN setting. Column *IPP* specifies the number of threads used per core for load balancing. The total protocol runtime is measured in seconds and the speed-up in comparison with CGP is presented in column *S*. Marked in bold are settings, where IPP leads to performance gains. The time spent on the transferring roles protocol is presented in the last row.

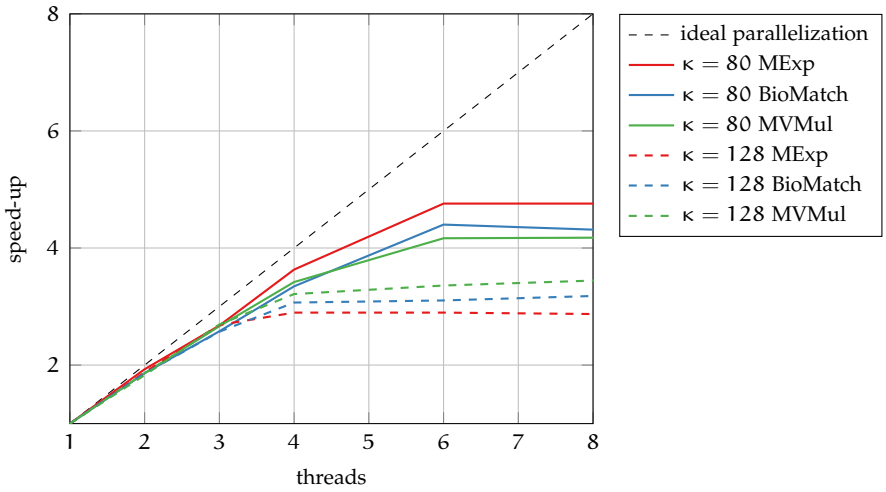


Figure 22: CGP performance in Yao's protocol. Shown is the speed-up of all three applications in the LAN setting with $\kappa = 128$ bit and $\kappa = 80$ bit security.

5.5.5 *Inter-Party Parallelization*

IPP in Yao's protocol is introduced in Section 5.4. Here, we describe and analyze experiments that show IPP in practical settings. The first experiment measures the computational efficiency gain in the same setting as described in Section 5.5.4. In the second experiment the benefits of IPP in a WAN setting with limited bandwidth are presented.

COMPUTATIONAL EFFICIENCY GAIN. In this experiment the raw IPP performance for all example applications, as well as the combination of CGP and IPP techniques is explored. To realize IPP, we use multiple threads per core to utilize the load balancing capabilities of the underlying OS without implementing a sophisticated load balancer. Due to the heterogeneous hardware environment, e.g., unpredictable caching and networking behavior, we evaluated three different workload distribution strategies. The first strategy uses one thread per core and thus only functions with at least two cores. Then, each party has exactly one garbling and one evaluating thread. The second and third strategy use two or four independent threads per core to garble and evaluate at the same time. Moreover, to illustrate that IPP is also beneficial for mixed functionalities and thus a modular concept, all circuits are evaluated using a sequential code block that exposes all inner input and output wires before and after every parallel region. Consequently, all results include the time spent on transferring all required input bits to and from parallel regions. Otherwise applications such as the MVMul application, which is a pure parallel functionality, would profit more easily from IPP. Even though this weakens the results for the example applications, we are

convinced that this procedure provides a better insight into the practical performance of IPP.

The results of this experiment are reported in Table 9. We first observe that only the MExp application significantly profits from IPP. This is due to the small sharing state in comparison to the circuit complexity. For both security levels IPP outperforms the raw CGP approach with an additional speed-up of $10-30\%$ on all cores. The performance of the MVMul applications actually decreases when using IPP. This is because of the large state that needs to be transferred. The performance gain through IPP cannot overcome the newly introduced overhead for the Transferring Roles protocol of 31 ms, which is more than 15% of the sequential runtime.

In summary, parallelizable applications that show a small switching surface (measured in number of bits compared to the overall circuit size) profit from IPP. Thus, IPP is a promising extension to Yao's protocol that utilizes circuit decomposition beyond naive parallelization, independently of other optimization techniques.

BI-DIRECTIONAL BANDWIDTH EXPLOITATION. The second experiment aims towards increasing the available bandwidth by exploiting bidirectional data transfers. Commonly, Ethernet connections have support for full duplex (bi-directional) communication. When using standard Yao's Garbled Circuits, only one communication direction is fully utilized. However, with IPP the available bandwidth can be doubled by symmetrically exploiting both communication channels. This practical insight is evaluated in a WAN setting between two cloud instances of type `m3.xlarge` with 100 ± 10 ms latency and a measured bandwidth of 92 ± 27 Mbps. Each hosts runs two threads (a garbling and a evaluating thread) using only a single core. The results of this experiment are illustrated in Table 10. IPP leads to significant speed-ups of BioMatch and MExp, showing the successful exploitation of bi-directional data transfers. MVMul shows limited performance gains because the time spent on the newly introduced communication rounds for the transferring roles protocol becomes significant. Summarizing, IPP can be very useful for MPC protocols with asymmetric workload in bandwidth limited environments.

5.5.6 *Evaluation Summary*

MPC based on Yao's Garbled Circuits protocol can greatly benefit from compiler assisted parallelization. The FGP approach can be efficient for some circuits, yet its scalability highly depends on the circuit's width. This problem is addressed in more detail in the next chapter, where we study how to compile circuits with a shallow depth (and thus larger width). The CGP approach shows a more efficient parallelization, given suitable, i.e., parallel decomposable, applica-

		BioMatch	MExp	MVMul
$\kappa = 128$	w/o IPP	45.02 ± 0.49s	24.13 ± 0.21s	4.83 ± 0.05s
	w/ IPP	29.94 ± 0.31s	16.05 ± 0.12s	4.66 ± 0.35s
	speed-up	1.50	1.50	1.03
$\kappa = 80$	w/o IPP	30.34 ± 0.62s	14.56 ± 0.21s	4.31 ± 0.23s
	w/ IPP	19.13 ± 0.47s	11.16 ± 0.32s	3.84 ± 0.12s
	speed-up	1.58	1.30	1.12

Table 10: Runtime evaluation of IPP on a single core with limited networking capabilities. Measured is the total protocol runtime, when computing with and without IPP. Furthermore, the speed-up between the two measurements is calculated.

tions. Moreover, we observe that the symmetric workload distribution of IPP leads to runtime improvements in practical setting, even when using only a single CPU core.

5.6 RELATED WORK

We describe related protocol and compiler parallelization approaches for Yao's protocol secure in the semi-honest and malicious model.

PARALLELIZATION IN THE SEMI-HONEST MODEL. Husted et al. [Hus+13] showed a CPU and GPU parallelization with significant speed-ups on both architectures. Their approach is based on the idea that every gate can be garbled independently from all other gates. Afterwards all sequential dependencies are resolved using an additional (XOR) encryption layer. This approach enables very efficient fine-grained parallelization, yet requires additional communication. Therefore, it is very well suited in scenarios where communication is cheap and not a bottleneck. Unfortunately, further bandwidth saving optimizations, such as garbled row reduction, are incompatible.

Barni et al. [Bar+13] proposed a parallelization scheme similar to the here described, which distinguishes between fine- and coarse-grained parallelism. Their approach shows speed-ups for two example applications. However, for coarse-grained parallelism manual user interaction is required to annotate parallelism in handcrafted circuits. Unfortunately, their timing results are hardly comparable with other work, due to the missing implementation of concurrent circuit generation and evaluation, i.e., pipelining, which is required to garble larger circuits.

Nayak et al. [Nay+15] presented a framework named *GraphSC* that supports the parallel computation of graph oriented applications using RAM based secure computation (cf. Section 2.3). GraphSC shows

very good scalability for data intensive computations. Yet, parallelism has to be annotated manually and has to follow a Map Reduce pattern. To exploit further parallelism within different computing nodes of GraphSC, the ideas presented in this chapter could be exploited.

PARALLELIZATION IN THE MALICIOUS MODEL. The "Billion gates" framework by Kreuter et al. [KSS12] was designed to execute large circuits on cluster architectures. The framework supports parallelization in the malicious model using message passing technologies. Frederiksen et al. [FJN14] also addressed the malicious model, yet they targeted the GPU as execution environment. In both cases, the protocol is based upon the idea of *cut-and-choose*, which consists of multiple independent executions of Yao's protocol secure against semi-honest adversaries. This independence enables naive parallelization up to the constant number of circuits required for cut-and-choose. This degree of parallelism cannot be transferred to the semi-honest setting studied in this thesis.

Yet, in recent works on protocol extensions secure against malicious adversaries, e.g., [RR16; Fur+17; LR14; LR15], application parallelization becomes of importance. This is because, these MPC protocols profit from a batched execution of the same functionality, as the number of circuit in a cut-and-choose approach can be reduced. Hence, the CGP approach is a necessity to achieve efficient execution in these protocols when compiling from a high-level language.

PARALLEL COMPILER FOR MPC. Zhang et al. [ZSB13] presented a compiler for distributed secure computation with applications for parallelization. Their compiler converts manually annotated parallelism in an extension of C into secure implementations. Even so the compiler is targeting Arithmetic circuits, it could be used as an additional front-end to the ideas presented in this work.

6

COMPILING DEPTH-OPTIMIZED CIRCUITS FOR MULTI-ROUND MPC PROTOCOLS

Summary: Many practical relevant MPC protocols, such as GMW and SPDZ, have a round complexity that is dependent on the circuit's depth. When deploying these protocols in real-world network settings, with network latencies in the range of tens or hundreds of milliseconds, the round complexity quickly becomes a significant performance bottleneck. In this chapter, we describe a compiler extension to CBMC-GC that optimizes circuits for a minimal depth. We first introduce novel optimized building blocks that are up to 50% shallower than previous constructions. We then present multiple high- and low-level depth-minimization techniques. Our implementation achieves significant depth reductions over hand-optimized circuits (for some applications up to a factor of 2.5). Moreover, evaluating exemplary functionalities in the GMW protocol, we show that depth reductions lead to significant speed-ups in real-world network setting.

Remarks: This chapter is based in parts on our paper – *"Compiling Low Depth Circuits for Practical Secure Computation"*, Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser, which appeared in the Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS), Heraklion, Greece, 2016.

6.1 MOTIVATION AND OVERVIEW

In the previous chapters, we focussed on describing techniques to compile circuits with a minimal number of non-linear gates. However, we also observed that circuits with a shallow depth are of interest for efficient parallelization (cf. Chapter 5). Moreover, one of the first MPC protocols, namely the GMW protocol by Goldreich, Micali and Widgerson [GMW87] and many subsequent protocols, e.g., BGW by Ben-Or et al. [BGW88], Sharemind by Bogdanov et al. [BLW08], SPDZ by Damgard et al. [Dam+12], TinyOT by Nielsen et al. [Nie+12], or the protocol by Furukuwa et al. [Fur+17] have a round complexity that is linear in the circuit depth. Hence, for this class of MPC protocols it is crucial to also consider the circuit depth as a major optimization goal, because every layer in the circuit increases the protocol's runtime by the Round Trip Time (RTT) between the computing parties. This is of special importance, as latency is the only computational resource that has reached its physical boundary. For computational power and bandwidth, parallel resources can always be added. Thus, for applications using the aforementioned multi-round MPC protocols it is commonly more vital to minimize the depth of circuits, rather than improving computational efficiency of the protocols themselves, es-

operation	previous best [SZ13]	this work
n-bit Addition	$2\log_2(n)+1$	$\log_2(n)+1$
n-bit Multiplication	$3\log_2(n)+4$	$2\log_2(n)+3$
m:1 Multiplexer	$\log_2(m)$	$\lceil\log_2(\lceil\log_2(m+1)\rceil)\rceil$

Table 11: Depth of building blocks. Comparison of the depth of the here presented building blocks with the previously known best constructions.

pecially when considering that in theory infinite parallel hardware resources could be added.

We illustrate these thoughts with exemplary practical numbers: The recent MPC protocol by Furukuwa et al. [Fur+17], which provides security against malicious adversaries assuming a honest majority in a three-party setting, can compute more than 1 billion gates on a server CPU with 20 cores. At the same time, the network latency between Asia and Europe[1] is in the range of a hundred milliseconds. In such a setting with a latency of 100 ms, a gate-level parallelism of at least 100 million gates is required to fully saturate the CPUs. Therefore, it is worthwhile to study optimization and compilation techniques for the automatic creation of Boolean circuits with minimal depth.

In this chapter, we describe how to compile circuits with a minimal depth. We implement the techniques as part of a compiler extension to CBMC-GC named *ShallowCC* using a three-fold approach:

First, we describe minimization techniques operating on the source code level. For example, we discuss how aggregations, e.g., sum or minima computation over an array, described in a sequential manner, can be regrouped in a tree structure. This allows to achieve a circuit with a logarithmic instead of a linear depth. We also portray a technique that detects consecutive arithmetic operations and replaces them by a more efficient dedicated circuit (known as Carry-Save Networks (CSNs)) rather than a composition of multiple individual arithmetic building blocks. Second, we present depth- and size-optimized constructions of major building blocks, e.g., adder and multiplexer, required for the synthesis of larger circuits. An overview of these and a comparison with previous constructions is given in Table 11. Third, we adapt CBMC-GC's gate-level optimization methods, described in Section 4.4, to also optimize for depth.

1 Even though the performance of MPC is often evaluated in a LAN setting, a WAN setting is the more natural deployment model of MPC, as mutually distrusting parties are unlikely to have hardware deployed in close proximity.

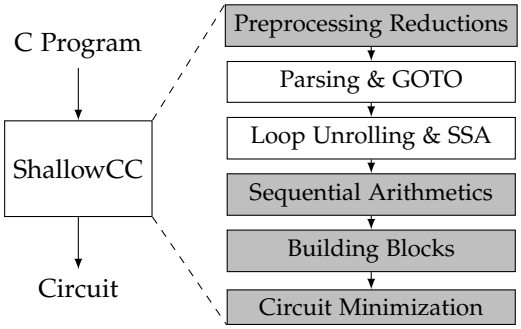


Figure 23: ShallowCC's compilation chain from ANSI C to Boolean circuits. Marked in gray are all modifications to the original compilation chain of CBMC-GC.

CHAPTER OUTLINE. In Section 6.2 we describe the adaptions to CBMC-GC's compilation chain for depth-minimization, including a description of building blocks as well as an adapted fix-point minimization algorithm. A detailed experimental evaluation is given in Section 6.3. Related work is discussed in Section 6.4.

6.2 COMPILATION CHAIN FOR LOW-DEPTH CIRCUITS

The size-minimized circuits generated by CBMC-GC are not necessarily depth-minimized, as both optimization goals can be orthogonal. Therefore, we introduce multiple techniques that deviate from CBMC-GC's compilation chain to create depth-minimized circuits. An implementation of these techniques is realized as a compiler extension to CBMC-GC named ShallowCC. The differences to the compilation chain of CBMC-GC are illustrated in Figure 23 and described in this section.

The extended compilation chain begins with a code preprocessing step to detect and transform reduction statements on the source code level (see Section 6.2.1). The next steps are the same as in the tool chain of CBMC-GC, i.e., the code is parsed and translated into a GOTO program, all bounded loops and recursions are unrolled using symbolic execution and the resulting code is transformed into SSA form. In the fourth step, the SSA form is used to detect and annotate successive arithmetic statements (see Section 6.2.2). Afterwards, all statements are instantiated with depth-minimized building blocks (see Section 6.2.3), before a final gate-level minimization takes place (see Section 6.2.4).

6.2.1 *Preprocessing Reductions*

We refer to a reduction as the aggregation of multiple variables into a single result variable, e.g., the sum of an array. An exemplary reduction is illustrated in the code example in Listing 12. This code com-

```
1 unsigned max_abs(int a[], unsigned len) {
2   unsigned i, max = abs(a[0]);
3   for(i = 1; i < len; i++) {
4     if(abs(a[i]) > max) {
5       max = abs(a[i]);
6     }
7   }
8   return max;
9 }
```

Listing 12: Exemplary function that computes the maximum norm.

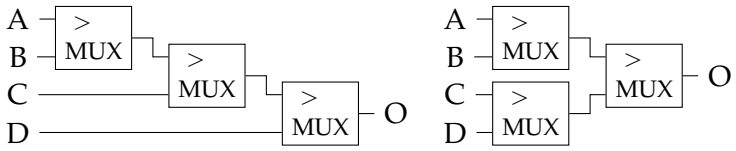
putes the maximum norm of a vector. It iterates over an integer array, computes the absolute value of every element and then reduces all elements to a single value, namely their maximum. A straight forward translation of the maximum computation leads to a circuit consisting of `len` -1 sequentially aligned comparators and multiplexers, as illustrated for four values in Figure 24a. However, the same functionality can be implemented with logarithmic depth when using a tree structure, as illustrated in Figure 24b. Thus, when compiling circuits with minimal depth, it is worthwhile to rewrite sequential reductions. To relieve the programmer from this task, ShallowCC aims at automatically replacing sequential reductions found in loop statements by tree-based reductions.

Since detecting reductions in loop statements is a common task in automatized parallelization, we can reuse compilation techniques presented in Section 5.3.2. Parallelization frameworks are not only very suited to detect parallelism but also to detect sequential reductions, as these can significantly degrade the performance of parallelization. Therefore, we extend the techniques presented in Section 5.3.2 to parse reduction annotations and to rewrite the code with *clang* (source-to-source compilation). For this, we identify the loop range and reduced variable to instantiate a code template that computes the reduction in a tree structure. Such a template is illustrated in Listing 13 for an arbitrary comparator function `cmp()` and datatype `DT`. A similar template can be used for sum or product computations. Albeit being commutative operations, we remark that as in parallelization a developer needs to be aware that an operation reordering can lead to different results when integer overflows occur.

This optimization improves the depth of reductions over m elements from $O(m)$ to $O(\log m)$. To illustrate the effect of this optimization in practice we study an exemplary minimum computation over an array with 100 integers (32-bit). In this small example, the difference in the depth of the compiled circuit is more than one order of magnitude, i.e., the tree based computation can be performed with a depth of 42 non-linear gates, whereas the sequential computa-

```
DT reduction_tree(DT *a, unsigned len, (*cmp)(DT, DT)){
 unsigned i, step = 1;
 while(step < len) {
  for(i = 0; i + step < len; i += (step << 1)) {
   a[i] = (*cmp)(a[i], a[i+step]) ? a[i] : a[i+step];
  }
  step <<= 1;
 }
 return a[0];
}
```

Listing 13: Simplified tree reduction template for an arbitrary array `a` of length `len`, as applied when rewriting reductions in loop statements for an abritary datatype DT.



(a) Sequential circuit (b) Tree circuit

Figure 24: Maximum circuit. Shown are circuit variants to compute the maximum of four values consisting of comparators and multiplexers. The circuit in Figure 24a uses a sequential organization of building blocks, whereas the circuit in Figure 24b follows a tree structure.

tion results in a circuit with a depth of 592 non-linear gates. Further examples are given in Section 6.3.3.

6.2.2 *Sequential Arithmetics and Carry-Save Networks (CSNs)*

In the early 1960s, Carry-Save Addition was introduced for the fast addition of three or more numbers by Earle et al. [Ear65]. The main component of a Carry-Save Addition is the 3:2 Carry-Save Adder (CSA). For three given numbers a, b and c, a CSA computes two numbers x and y, whose sum is equal to the sum of a, b and c, i.e., $a + b + c = x + y$. The key benefit of CSAs is that their computation can be done with significantly less (constant) depth than when computing the actual sum of two out of the three numbers, which requires an adder circuit that has a depth logarithmic in the numbers' bit-widths. Given a 3:2 CSA, the sum of k numbers can be computed by first reducing the k numbers to two numbers using a network of $k - 2$ CSAs, called CSN and then computing the final addition using a standard adder. For example, the addition of four numbers a, b, c, d in such a CSA tree is illustrated in Figure 25.

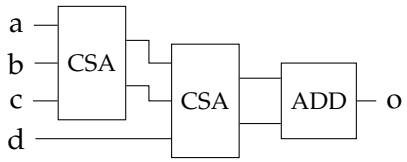


Figure 25: Carry-Save Network. Addition of four numbers using a CSN consisting of two CSAs and one adder (ADD).

CSA CONSTRUCTION. A CSA for three n bit numbers a, b, and c can be constructed using n parallel FAs, cf. Section 4.3, where the sum bits and carry-out bits each form the two numbers x and y. Hence for every bit position $i \in [0, n-1]$, X_i and Y_{i+1} are set to

$$X_i = A_i \oplus B_i \oplus C_i$$
$$Y_{i+1} = (A_i \oplus C_i)(B_i \oplus C_i) \oplus C_i,$$

with $Y_0 = 0$. The sum bit X_i can actually be computed for free and the carry-out bit Y_{i+1} only requires a single non-linear gate. Thus, a 1 bit CSA has depth and size $s^{nX} = d^{nX} = 1$ in MPC. Using a tree based aggregation and exploiting the property that one output of a CSA can be computed with zero non-linear gates, the partial sums x and y for k numbers can be computed with depth $d^{nX}_{CSA}(k) = \lceil \log_2(k) - 1 \rceil$ [SZ13]. Hence, CSNs are efficient circuit constructions for multiple successive arithmetic operations that outperform their individual composition in size and depth.

EXAMPLE. Consider the following lines of code as an example:

```
unsigned a, b, c, d;
unsigned t = a + b;
unsigned sum = t + c + d;
```

A straight forward compilation, as in CBMC-GC, leads to a circuit consisting of three binary adders: `sum = ADD(ADD(ADD(a,b), c), d)`. However, if it is possible to identify that a sum of four independent operands is computed, a CSN as given in Figure 25, could be used instead. In this concrete example, the circuit's depth reduces from 18 to 7 non-linear gates when using 32 bit integers. Hence, an automatic detection and translation of sequential arithmetic operations into CSNs is highly desirable. Detecting these operations on the gate level is feasible, for example with the help of pattern matching, yet impractically costly considering that circuits reach sizes in the range of billions of gates. Therefore, ShallowCC aims at detecting these successive statements before their translation to the gate level.

ALGORITHM. The detection is performed using the SSA form, which allows efficient data flow analyses and as such, also the search for

successive arithmetic operations. We propose a greedy detection algorithm that consists of two parts. First, a breadth-first search from output to input variables is initiated. Whenever an arithmetic assignment is found, a second backtracking algorithm is initiated to identify all preceding (possibly nested) arithmetic operations. This second algorithm stops whenever an operation is found that is not of type +, -, or *. Once all preceding inputs are identified, the initial assignment can be replaced by a CSN. After every replacement, the search algorithm continues its search towards the input variables. Once all relevant assignments have been replaced by a CSN, all now unused expressions are removed (dead code elimination). We note that this greedy replacement approach is depth-minimizing, yet not necessarily size optimal. This is because intermediate results in nested statements may be computed multiple times. A trade-off between size and depth is possible by only instantiating CSNs for non-nested arithmetic statements.

Quantifying the improvements in depth reduction and assuming that the addition of two numbers requires a circuit of depth d^{nX}_{Add}, we deduce that $m > 2$ numbers can sequentially be added with depth $(m-1) \cdot d^{nX}_{Add}$. When using a tree-based structure the same sum can be computed with a depth of $\lceil \log_2(m) \rceil \cdot d^{nX}_{Add}$. When using a CSN, m numbers can be added with a depth of only $\lceil \log_2(m) - 1 \rceil + d^{nX}_{Add}$.

We remark that CSNs can not only constructed for additions but also for any mix of subsequent multiplications, additions and subtractions, because every multiplication internally consists of additions of partial products and a subtraction is an addition with the bitwise inverse, cf. Section 4.3. To illustrate the improvement in practice, for the exemplary computation of a 5×5 matrix multiplication, an improvement in depth of more than 60% can be observed when using a CSN, as shown in Section 6.3.3.

6.2.3 *Optimized Building Blocks*

Optimized building blocks are an essential part when designing complex circuits. They facilitate efficient compilation, as they can be highly optimized once and subsequently instantiated at practically no cost during compilation, cf. Chapter 4. In the following paragraphs, we present building blocks primarily optimized for depth (and only secondary for size) for basic arithmetic and control flow operations.

ADDER. An n-bit *adder* takes two bit strings a and b of length n, representing two (signed) integers, as input and returns their sum as an output bit string s of length $n+1$. The standard adder, described in Section 4.3, is the Ripple Carry Adder (RCA) that consists of a successive composition of n FAs. It has a circuit size and depth $s^{nX}_{RCA} = d^{nX}_{RCA} = O(n)$ that is linear in its bit-width. For faster addi-

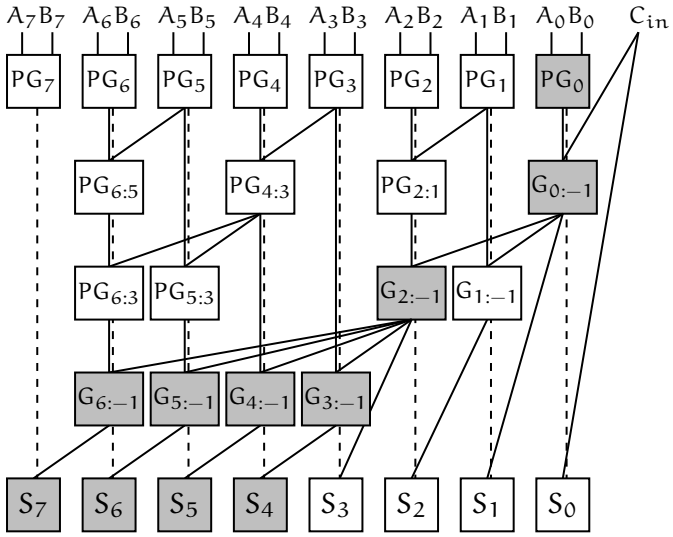


Figure 26: 8-bit Sklansky PPA. Shown are the computed generate and propagate signals. Marked in gray are the longest paths.

tion, Parallel Prefix Adders (PPAs) are widely used in logic synthesis. Their core concept is to use a tree based network to propagate the carry-out bit, which achieves a logarithmic depth under a size trade-off. Various different tree structures have been proposed. Here, we first discuss their general design before comparing two different tree structures relevant for an application in MPC.

In general, the PPA design distinguishes two signals (output bits), named the generate $G_{i:j}$ and the propagate $P_{i:j}$ signal, used to handle the fast propagation of carry-out bits. A generate signal announces whether the prefix sum of the two input strings $a_{i:j}$ and $b_{i:j}$, i.e.,

$$A_iA_{i-1}\dots A_j \;+\; B_iB_{i-1}\dots B_j,$$

will generate a carry-out bit. A propagate signal over the same range announces that a carry-out bit will be propagated if additionally a carry-in bit is set. A complete PPA consists of three parts. First, the initial generate and propagate signals are computed over pairs of input bits as $G_{i:i} = A_i \cdot B_i$ (carry-out bit) and $P_{i:i} = A_i \oplus B_i$ (sum bit). Second, subsequent generate and propagate signals are typically computed in hardware synthesis using the following formula [Haro3]:

$$(G_{i:j},\ P_{i:j}) = (G_{i:k+1} + P_{i:k+1}G_{k:j},\ P_{i:k+1}P_{k:j}).$$

Finally, in the last step each output bit S_i can then be computed as the XOR of the generate and propagate bit $S_i = P_{i:-1} \oplus G_{i-1:-1}$. We observe that the initial and output phase have a constant depth and that the depth of the intermediate phase depends on the structure of the used parallel prefix network. A close look at the generate and propagate signals reveals, that only one of the two can be set for every bit range. Hence, in circuit design for MPC it is very reasonable

to replace the OR ($+$) by an XOR ($\oplus$) in the formula of the generate signal:

$$(G'_{i:j},\ P_{i:j}) = (G'_{i:k+1} \oplus P_{i:k+1}G'_{k:j},\ P_{i:k+1}P_{k:j}).$$

This insight *halves* the delay in every computation step in parallel prefix network, as only one instead of two non-linear gates are required. A proof of the correctness of this substitution is given below.

Theorem 1 *All grouped generate* $G_{i:j} = G_{i:k+1} + P_{i:k+1}\,G_{k:j}$ *and propagate signals* $P_{i:j} = P_{i:k+1}P_{k:j}$ *with* $i \geqslant k \geqslant j$ *used during the binary addition of two numbers* $A = A_nA_{n-1}\ldots A_1$ *and* $B = B_nB_{n-1}\ldots B_1$ *are mutually exclusive.*

Thus, for all $i, j \in \{1, \ldots, n\}$*, with* $i \geqslant j$*:*

$$\overline{P_{i:j}G_{i:j}} = 1. \tag{1}$$

Proof using induction, base case $i = j$*:* For a single pair of bits, the propagate signal is defined as $P_{j:j} = (A_j \oplus B_j)$, the generate signal as $G_{j:j} = A_jB_j$. Using these definitions in Equation 1, we observe that both are mutually exclusive:

$$\overline{P_{j:j}G_{j:j}} = \overline{(A_j \oplus B_j)A_jB_j} = \overline{A_jB_j \oplus A_jB_j} = 1.$$

Induction step ($[i : j] \to [p : j]$, $p > i$)*:* We assume that $\overline{P_{p:i+1}G_{p:i+1}} = 1$ and $\overline{P_{i:j}G_{i:j}} = 1$ hold for $p > i \geqslant j$. By successively applying De Morgan's law, we deduce:

$$\begin{aligned}
\overline{P_{p:j}G_{p:j}} &= \overline{(G_{p:i+1} + P_{p:i+1}G_{i:j})P_{p:i+1}P_{i:j}} \\
&= \overline{G_{p:i+1}P_{p:i+1}P_{i:j} + P_{p:i+1}G_{i:j}P_{p:i+1}P_{i:j}} \\
&= (\overline{G_{p:i+1}P_{p:i+1}P_{i:j}})(\overline{P_{p:i+1}G_{i:j}P_{p:i+1}P_{i:j}}) \\
&= (\overline{G_{p:i+1}P_{p:i+1}} + \overline{P_{i:j}})(\overline{G_{i:j}P_{i:j}} + \overline{P_{p:i+1}}) \\
&=_{BC} (1 + \overline{P_{i:j}})(1 + \overline{P_{p:i+1}}) = 1.
\end{aligned}$$

□

Applying this idea to the Sklansky PPA design, which is the fastest PPA structure according to the taxonomy by Harris [Har03], our construction achieves a depth of $d_{Sk}^{nX}(n) = \lceil \log_2(n) \rceil + 1$ and a size of $s_{Sk}^{nX} = n\lceil \log_2(n) \rceil$ for an input bit length of n and output bit length of $n + 1$.

In Table 12 a depth and size comparison between the standard RCA, the Ladner-Fischer adder, described by Schneider and Zohner [SZ13], the Sklansky PPA, and an alternative PPA structure, namely the Brent-Kung PPA [Har03] is given for different bit-widths. We observe that the RCA provides the smallest size and the Sklansky adder the shallowest depth. The Brent-Kung adder provides a trade-off between size and depth. The here optimized Sklansky and Brent-Kung adder significantly outperform the previous best known depth-minimized construction in size and depth.

	depth d^{nX}				size s^{nX}			
Bit-width	n	16	32	64	n	16	32	64
Ripple-Carry	$n-1$	15	31	63	$n-1$	15	31	63
Ladner-Fischer[SZ13]	$2\lceil\log(n)\rceil+1$	9	11	13	$1.25n\lceil\log(n)\rceil+2n$	113	241	577
Brent-Kung-opt	$2\lceil\log(n)\rceil-1$	7	9	11	$3n$	48	96	192
Sklansky-opt	$\lceil\log(n)\rceil+1$	5	6	7	$n\lceil\log(n)\rceil$	64	160	384

Table 12: Adders. Comparison of circuit size s^{nX} and depth d^{nX} of the standard RCA, the previously best known depth-minimized Ladner-Fischer PPA [SZ13; Dem+15], and the Brent-Kung and Sklansky PPAs with the described generate signal optimization.

SUBTRACTOR. As described in Section 4.3, a subtractor can be implemented using an adder and one additional non-linear gate with the help of the two's complement representation, which is $-a = \overline{a}+1$ with $\overline{a}$ being the inverted binary representation. Consequently, $a-b$ can be represented as $a+\overline{b}+1$. Hence, the subtractor profits to the same degree from the optimized addition.

COMPARATOR. A depth-minimized *equivalence (EQ)* comparator can be implemented by a tree based OR composition over pairwise XOR gates to compare single bits, similar to the size-minimal construction presented in Section 4.3. This yields a depth of $d^{nX}_{EQ}(n) = \lceil\log_2(n)\rceil$ and size of $s^{nX}_{EQ}(n) = n-1$ gates [SZ13].

A depth-minimized *greater-than (GT)* comparator can be implemented in the same way as the size-minimized GT (see Section 4.3) comparator by using the observation that $x > y \Leftrightarrow x-y-1 \geqslant 0$ and returning the carry-out bit. This approach leads to a circuit depth that is equivalent to the depth of a subtractor.

MULTIPLIER. In the size-minimized multiplication (see Section 4.3), n partial products of length n are computed and then added. This approach leads to a quadratic size $s^{nX}_{MUL,s} = 2n^2-n$ and linear depth $d^{nX}_{MUL,s} = 2n-1$. A faster addition of products can be achieved when using CSAs. Such a tree based multiplier consists of three steps: First, the computation of all $n \times n$ partial products, then their aggregation in a tree structure using CSAs, before the final sum is computed using a two-input adder. The first step is computed with a constant depth of $d^{nX}_{PP} = 1$, as only one single AND gate is required. For the last step, two bit strings of length $2n-1$ have to be added. Using the optimized Sklanksy adder, this addition can be realized in $d^{nX}_{Sk}(2n-1) = \lceil\log_2(2n-1)\rceil+1$. The second phase allows many different designs, as the CSAs can arbitrarily be composed. The fastest composition is the Wallace tree [Wal64], which leads to a depth of

	depth d^{nX}				size s^{nX}			
Bit-width	n	16	32	64	n	16	32	64
Standard	$2n-1$	45	93	189	n^2-n	496	2016	8128
MulCSA [SZ13]	$3\lceil\log(n)\rceil+4$	16	19	22	$\approx 2n^2+1.25n\log(n)$	578	2218	8610
Wallace-opt	$2\lceil\log(n)\rceil+3$	11	13	15	$\approx 2n^2+n\log(n)$	512	2058	8226

Table 13: Multipliers. Comparison of circuit depth d^{nX} and size s^{nX} of the school method, the multiplier given in [SZ13] and our optimized Wallace construction.

$d^{nX}_{CSA}(n) = \log_2(n)$ for MPC. Combing all three steps, a multiplication can be realized with depth

$$d^{nX}_{Wa}(n) = d^{nX}_{PP} + d_{CSA}(n) + d^{nX}_{Sk}(2n-1) = 2\lceil\log_2(n)\rceil + 3.$$

In Table 13 we present a comparison of the multipliers discussed above with the depth optimized one presented in [SZ13]. Compared with this implementation, we are able reduce the depth by at least a third for any bit-width.

MULTIPLEXER. We recap Section 4.3, where we described a 2:1 MUX that select between to two data inputs D_0 and D_1 based on a control input C. A MUX over two bit strings requires one single non-linear gate for every pair of input bits by computing each output as $O_i = (D^0_i \oplus D^1_i)C \oplus D^0_i$. Hence, a 2:1 n-bit MUX has size $s^{nX}_{MUX}(n) = n$ and depth of $d^{nX}_{MUX}(n) = 1$. A 2:1 MUX can be extended to a m:1 MUX that selects between m input strings $d^0, d^1, \ldots, d^m$ using $\log_2(m)$ control bits $c = C_0, C_1, \ldots C_{\log(m)}$ by tree based composition described in detail Section 4.3. This circuit construction has a size of $s^{nX}_{MUX\_tree}(m,n) = (m-1)\cdot s^{nX}_{MUX}(n)$ and a depth that is logarithmic in the number of data inputs $d^{nX}_{MUX\_tree}(m,n) = \log_2(m)$.

A further depth-minimized m:1 MUX can be constructed under a moderate size trade-off, by using a design that is similar to a Disjunctive Normal Form (DNF) over all combinations of choice bits. Every conjunction of the DNF encodes a single choice together with the associated data wire. For MPC, this construction leads to a very low depth, because the disjunctive ORs can be replaced by XORs, as all choices are mutually exclusive. For example, a 4:1 MUX is then expressed as

$$O = d^0\overline{C_0}\,\overline{C_1} \oplus d^1\overline{C_0}C_1 \oplus d^2C_0\overline{C_1} \oplus d^3C_0C_1.$$

Consequently the depth of a depth-minimized DNF m:1 MUX, referred to as MUX_{DNFd}, is equivalent to the depth of a single conjunction. A single conjunction can be computed in a tree-based manner, for example one conjunction of a 128:1 MUX can be encoded as $(((d^{127}C_0)(C_1C_2))((C_3C_4)(C_5C_6)))$, which leads to depth that is logarithmic in the number of control bits:

$$d^{nX}_{MUX\_DNFd}(m,n) = \lceil\log_2(\lceil\log_2(m)\rceil+1)\rceil.$$

Choices	depth d^{nX}				size s^{nX}			
	m	8	128	1024	m	8	128	1024
MUX_{Tree}	$\lceil \log(m) \rceil$	3	7	10	$(m-1) \cdot n$	244	4,064	32K
MUX_{DNFd}	$\lceil \log(\lceil \log(m)+1 \rceil) \rceil$	2	3	4	$mn \cdot \lceil \log(m) \rceil$	768	28,672	320K
MUX_{DNFs}	$\lceil \log\lceil \log(m) \rceil) \rceil + 1$	3	4	5	$n \cdot (m+1)$	288	4,128	32K

Table 14: Multiplexers. Exemplary comparison of circuit depth d^{nX} and size s^{nX} of m:1 multiplexers for a different number of inputs m of bit-width $n = 32$ bit.

Unfortunately, a straightforward implementation of m:1 MUX_{DNFd} over data inputs with a width of n-bit that follows the description above has a size that is $\log(m)$ times larger than the size of a tree based MUX:

$$s^{nX}_{MUX\_DNFd}(m, n) = \log(m) \cdot mn.$$

This increase by a logarithmic factor can be quite significant for larger m. Therefore, we describe a second DNF based construction, referred to as MUX_{DNFs}, which offers a more practical size-depth trade-off. The idea is to first compute every conjunction using the one-hot encoder presented in Section 4.3, before AND-gating each "encoded" choice with the data input. This construction has a depth similar to the depth of MUX_DNFd:

$$d^{nX}_{MUX\_DNFs}(m, n) = \lceil \log_2(\lceil \log_2(m) \rceil) \rceil + 1.$$

However, with the separation of control and data inputs, the computation of various combinations of choice bits can be merged more effectively between different conjunctions, e.g., the choices $C_0C_1C_2$ and $\overline{C_0}C_1C_2$ require both the computation of C_1C_2. This reduces the size to:

$$\begin{aligned} s^{nX}_{MUX\_DNFs}(m, n) &= \text{one-hot encoder} + \text{AND data inputs} \\ &= n - 2 + mn \\ &\approx n \cdot (m+1). \end{aligned}$$

In Table 14 a comparison of the three MUXs is given for a different number of inputs m and a typical bit-width of 32 bits. In summary, we improved the depth of MUXs from $O((\log(m))$ to $O(\log(\log(m)))$, which is almost constant in practice, with a moderate increase in size.

6.2.4 *Gate Level Minimization Techniques*

Minimizing the circuit on the gate level is last step in CBMC-GC's compilation chain. For ShallowCC, we left parts, e.g., structural hashing and SAT sweeping, of the fix-point minimization algorithm unmodified, because they do not have a dedicated depth-minimizing

counterpart and both already help to reduce the circuit complexity and thus also contribute to reduce the circuit depth. Instead, we adapt the template based rewriting phase.

Circuit rewriting in CBMC-GC only considers patterns that are size decreasing and have a depth of at most two binary gates. For depth reduction, however, it is useful to also consider deeper circuit structures, as well as patterns that are size preserving but depth decreasing. For example, sequential structures of form $X = A + (B + (C + (D + E)))$ can be replaced by tree based structures of form $X = ((A + B) + C) + (D + E)$ with no change in circuit size. Therefore, in ShallowCC we extend the rewriting phase by several depth-minimizing patterns, which are not necessarily size decreasing. In total 21 patterns changed, resulting in more than 80 patterns that are searched for. Furthermore, circuit rewriting as described in Section 4.4 applies a pattern matching algorithm, which searches for all search patterns. However, to replace sequential structures by tree based structures, it is worthwhile to identify arbitrary sequential compositions of gates without generating all possible search patterns beforehand. Therefore, we replaced the fixed pattern matching algorithm by a flexible and recursive search for sequential structures. For example, instead of having dedicated patterns for $X = A \cdot (B \cdot (C \cdot (D \cdot E)))$ and $X = A \cdot (B \cdot C \cdot (D \cdot E))$, both are matched with the recursive algorithm that replaces all possible sequential structures of non-linear gates that a free of intermediate outputs.

Finally, we modify the termination condition of the fix-point optimization routine such that the algorithm only terminates if no further size *and depth* improvements are made (or a user defined time limit is reached). Moreover, for performance reasons, the rewriting first only applies fixed depth patterns before applying the recursive search for deeper sequential structures.

Quantifying the improvements of individual patterns is challenging. This is because the heuristic approach commonly allows multiple patterns to be applied at the same time and every replacement has an influence on future applicability of further patterns. Nevertheless, the whole set of patterns that we identified is very effective, as circuits before and after gate level minimization differ up to a factor of 20 in depth, see Section 6.3.3.

6.3 EXPERIMENTAL EVALUATION

We evaluate the effectiveness of automatized depth-minimization in a three-fold approach. First, we compare the circuits generated by ShallowCC with circuits that have been optimized for depth either by hand or by using state-of-the-art hardware synthesis tools. Then, we study the effectiveness of different implemented optimization techniques individually. Finally, we show that depth-minimized circuits,

even under size trade-offs, significantly reduce the online runtime of multi-round MPC protocols for different network configurations. We begin by describing the parameters of the benchmarked functionalities.

6.3.1 *Benchmarked Functionalities and their Parameters*

For comparison, we use functionalities that are described in Section 2.5. The applications used to evaluate ShallowCC are:

- *Integer arithmetics.* Due to their importance in almost every computational problem, we benchmark arithmetic building blocks individually. For multiplication we distinguish results for output bit strings of length n and of length $2n$ (overflow free) for n-bit input strings.
- *Floating point arithmetics.* We abstain from implementing hand-optimizing floating point circuits, but instead rely on CBMC-GC's capabilities to compile a IEEE-754 compliment software floating point implementation of addition (*FloatAdd*) and multiplication (*FloatMul*) written in C.
- *Distances.* We study the *Hamming* distance over 160 bit and over 1600 bit. Moreover, we study the *Manhattan* distance for two integers of bit-width 16 and 32 bit, and we also study the squared two dimensional *Euclidean* distance for the same bit-widths.
- *Matrix multiplication (MMul).* MMul is a purely arithmetic task, and therefore a good showcase to illustrate the automatic translation of arithmetic operations into CSNs to achieve very low depth. Here we use a matrix of size 5×5 and 32 bit integers.
- *Oblivious arrays.* Oblivious data structures are a major building block for the implementation of privacy preserving algorithms. The most general data structure is the oblivious array that hides the accessed index. We benchmark the read access to an array consisting of 32 integers of size 8 bit and 1024 integers of size 32 bit.
- *Biometric matching (BioMatch).* As in previous chapters, we use the squared Euclidean distance as distance function between two samples. Moreover, we use a database of size of $n = 32$ and $n = 1024$, and $d = 4$ features per sample with an integer bit-width of 16 bit and an overflow free multiplication.

All applications are implemented using textbook algorithms and are described in detail in Section 2.5. A code example illustrating the simplicty of the implementations is given in Listing 14, which computes the Manhatten distance.

```
1  void manhattan() {
2    int INPUT_A_x, INPUT_A_y, INPUT_B_x, INPUT_B_y;
3    int diff_x = INPUT_A_x - INPUT_B_x;
4    int diff_y = INPUT_A_y - INPUT_B_y;
5
6    if( diff_x < 0)
7      diff_x = -diff_x;
8    if( diff_y < 0)
9      diff_y = -diff_y;
10   int OUTPUT_res = diff_x + diff_y;
11 }
```

Listing 14: Manhattan Distance in ANSI C with variable naming for ShallowCC.

6.3.2 *Compiler Comparison*

We compiled all applications with ShallowCC on an Intel Xeon E5-2620-v2 CPU with a minimization time limit of 10 minutes. The resulting circuit dimensions for different parameters and bit-widths are shown in Table 15 and Table 16. Furthermore, the circuit size is given when compiled with the best size-minimizing compilers, i.e., Frigate [Moo+16] and CBMC-GC v0.9. To evaluate the circuit depths, a comparison with the depth-minimized circuit constructions of [Dem+15] and [SZ13] is shown. The results for [Dem+15], [Moo+16] and [SZ13] are taken from the respective publications.

Comparing the depth of the circuits compiled by ShallowCC with the hand and tool minimized circuits of [Dem+15; SZ13] we observe a depth reduction at least 30% for most functionalities. The only exception are the floating point operations, which do not reach the same depth as given in [Dem+15]. This is because floating point operations mostly consist of bit operations, which can significantly be hand optimized on a gate level, but are hard to optimize when complied from a high-level implementation in C. When comparing circuit sizes, we observe that ShallowCC is compiling circuits that are competitive in size to the circuits compiled by size minimizing compilers. A negative exception is the addition, which shows a significant trade off between depth and size. However, the instantiation of CSNs allows ShallowCC to compensate these trade-offs in applications with multiple connected additions, e.g., the matrix multiplication. In Section 6.3.4 we analyze these trade-offs in more detail. Summarizing the results, ShallowCC is compiling ANSI C code to Boolean circuits that outperform hand crafted circuits and tool optimized circuits in depth, with moderate increases in size. This also illustrates that the here proposed combination of minimization techniques outperforms the classic hardware synthesis tool chain used in [Dem+15].

		size-minimized			depth-minimized				improv.
		Frigate	**CBMC-GC**		**Prev. [Dem+15; SZ13]**		**ShallowCC**		
Circuit	n	s^{nX}	s^{nX}	d^{nX}	s^{nX}	d^{nX}	s^{nX}	d^{nX}	d^{nX}
Building Blocks									
Add $n \to n$	32	31	31	31	232	11	159	**5**	**54%**
Sub $n \to n$	32	31	61	31	232	11	159	**5**	**54%**
Mul $n \to 2n$	32	2,082	4,600	67	2,218	19	2,520	**15**	**21%**
Mul $n \to n$	64	4,035	4,782	67	-	-	4,350	**16**	-
Arithmetics									
Div	32	1,437	2,787	1,087	7,079	207	5,030	**192**	**7%**
MMul 5x5	32	128,252	127,225	42	-	-	128,225	**17**	-
FloatAdd	32	-	2,289	164	1,820	59	2,437	62	-5%
FloatMul	32	-	3,499	134	3,016	47	3,833	54	-14%

Table 15: Compiler comparison. Comparison of circuit size s^{nX} and depth d^{nX} when compiling functionalities with the size-minimizing Frigate [Moo+16] and CBMC-GC v0.9 compiler, the manually depth-minimized circuits given in [Dem+15; SZ13] and the circuits compiled by ShallowCC. Improvements are computed in comparison with the best previous work [Dem+15; SZ13]. The '-' indicates that no results were given. Marked in bold face are cases with significant depth reductions.

		size-minimized			depth-minimized				improv.
		Frigate	**CBMC-GC**		**Prev. [Dem+15; SZ13]**		**ShallowCC**		
Circuit	n	s^{nX}	s^{nX}	d^{nX}	s^{nX}	d^{nX}	s^{nX}	d^{nX}	d^{nX}
Distances									
Hamming-160	1	719	371	9	-	-	281	**7**	-
Hamming-1600	1	4,691	7,521	31	-	-	1,021	**12**	-
Euclidean-2D	16	-	826	47	1,171	29	1,343	**19**	**34%**
Euclidean-2D	32	-	3,210	95	3,605	34	5,244	**23**	**32%**
Manhattan-2D	16	-	187	31	296	19	275	**13**	**31%**
Manhattan-2D	32	-	395	63	741	23	689	**16**	**30%**
Privacy Preserving Protocols									
BioMatch-32	16	-	88,385	1,101	-	-	90,616	**55**	-
BioMatch-1024	16	-	2.9M	35,821	-	-	2.9M	**90**	-
Ob.Array-32	8	-	803	66	248	5	538	**3**	**40%**
Ob.Array-1024	32	-	100,251	2,055	32,736	10	65,844	**4**	**60%**

Table 16: Compiler comparison (continuation). Comparison of circuit size s^{nX} and depth d^{nX} when compiling functionalities with the size-minimizing Frigate [Moo+16] and CBMC-GC v0.9 compiler, the manually depth-minimized circuits given in [Dem+15; SZ13] and the circuits compiled by ShallowCC. Improvements are computed in comparison with the best previous work [Dem+15; SZ13]. The '-' indicates that no results were given. Marked in bold face are cases with significant depth reductions.

6.3.3 *Evaluation of the Optimizations Techniques*

To evaluate the effectiveness of the different optimization techniques, we compiled various example functionalities twice, once with the respective optimization technique enabled and once without. The difference between the two versions for each of the three proposed optimization techniques are shown in Figure 27. We observe improvements in the circuit depth between a factor of 2 and 80 for all functionalities. We also remark that not all optimizations apply to all functionalities, therefore, the improvements (obviously) should be studied with care when transferring the results to other applications. The CSN detection and instantiation shows its strengths for arithmetic functionalities. For example, the 5x5 matrix multiplication shows a depth reduction of 60%, when the optimization is enabled. This is because the computation of a single vector element can be grouped into one CSN. The detection of reductions is a very specific optimization, yet, when applicable, the depth saving can be significant. For example, when computing the minima of 100 integers, a depth reduction of 92% is visible. Similarly, the BioMatch application improves by 92%, which is a factor of 80. Note that in both cases the circuit size itself is unchanged, as only the order of the computation is optimized. Gate level minimization is the most important optimization technique for all functionalities that do not use all bits of every program variable. In these cases constant propagation can be applied, which leads to significant reductions in size and depth, as exemplary shown for the floating point addition and computation of the Hamming distance. In summary, when applicable, the optimization methods significantly improve the circuit depth when compiling with ShallowCC. Moreover, although not shown here, we remark that the the instantiation of a CSNs and the gate level minimization are beneficial for the circuit size.

6.3.4 *Protocol Runtime*

To show that depth-minimization improves the online time of MPC protocols, we evaluate a selection of circuits in the ABY [DSZ15]. The ABY framework provides a state-of-the-art two-party implementation of the GMW protocol [GMW87] secure in the semi-honest model (a detailed protocol description is given in Section 2.2.3). We extended the ABY framework by an adapter to parse CBMC-GC's circuit format. For our experiments, we connected two machines, which are equipped with an AMD FX 8350 CPU and 16GB of RAM, running Ubuntu 15.10 over a 1Gbit ethernet connection in a LAN. To simulate different network environments we made used of the Linux network emulator `netem`.

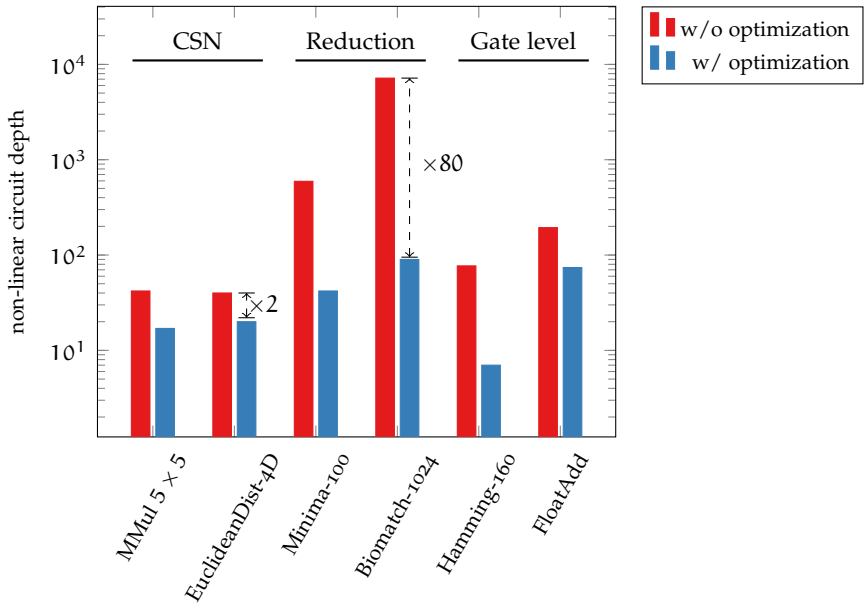


Figure 27: Effectiveness of optimization techniques. Comparison of circuit dimensions when compiled by ShallowCC with different optimization techniques, i.e., CSNs instantiation, rewriting of reduction, or gate level minimization, enabled or disabled.

In this experiment the *online* protocol runtimes of size- and depth-minimized circuits for different RTTs are compared. Netem simulates different RTTs by locally stalling the TCP protocol and thus, the transmission of packets. We ran this experiment for different RTTs, starting with zero delay up to a simulated RTT of 80 ms. In our results, we omit timings for the pre-processing *setup* phase, as this precomputation can take place independently of the evaluated circuits and with any degree of parallelism.

BIOMATCH. The first functionality that we investigate is the BioMatch application with a database of $n = 1024$ entries. Here, we compare the size and depth-minimized circuits generated by CBMC-GC with and without ShallowCC enabled. The resulting circuit dimensions are given in Table 16, and the protocol runtimes averaged over 10 runs are given in Figure 28. We observe speed-ups of the depth-minimized circuit over the size-minimized circuit of a factor between 2 and 400, when increasing the RTT from ~1 ms to 80 ms.

MULTIPLEXER. The second functionality that we evaluate is the array read (MUX). We compiled the read access to an array with 1024 integers of size 32 bit. We compare the tree based MUX, which is the previous best known solution [SZ13], which has depth $d^{nX} = 10$ and size $s^{nX} = 32,736$, with our depth optimized MUX_{DNFd}, which has a depth of $d^{nX} = 4$ and size $s^{nX} = 65,844$, after gate level minimization. Each circuit is evaluated with ABY individually, as well as

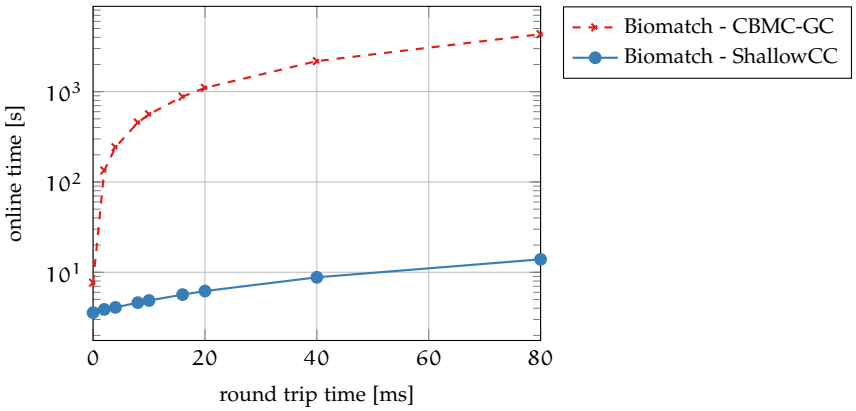


Figure 28: GMW protocol runtime when evaluating a depth- (ShallowCC) and size- (CBMC-GC) minimized biometric matching application. We observe that the depth optimized circuit significantly outperforms the size optimized circuit for any noticeable RTT.

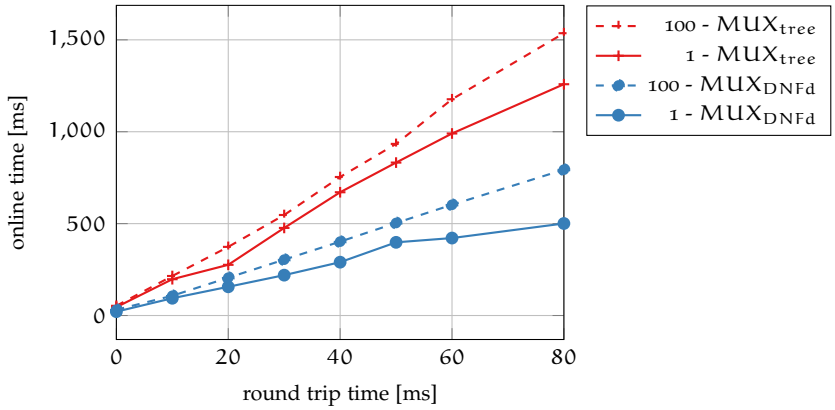


Figure 29: GMW protocol runtime when evaluating the depth-minimized DNF and size-minimized tree 1024:1 MUX for a single and parallel array read access. The DNF based MUX significantly outperforms the tree based MUX for any noticeable RTT.

100 times in parallel. This also allows to investigate whether SIMD parallelism, which is favored in GMW [DSZ15], has a significant influence on the runtime. The resulting online runtimes for both circuits are illustrated in Figure 29. All data points are averaged over 100 runs. We observe that for almost every network configuration beyond 1 ms RTT, the depth optimized circuits outperform their size-optimized counterparts by a factor of 2. The reason for the factor of 2 is, that the GMW protocol requires one communication round for input sharing as well as one round for output sharing, which leads to 6 communication rounds in total for the MUX_{DNFd} and 12 rounds for the MUX_{tree}. Moreover, we observe that the here applied data parallelism shows no significant effect on the speed-up gained through depth reduction.

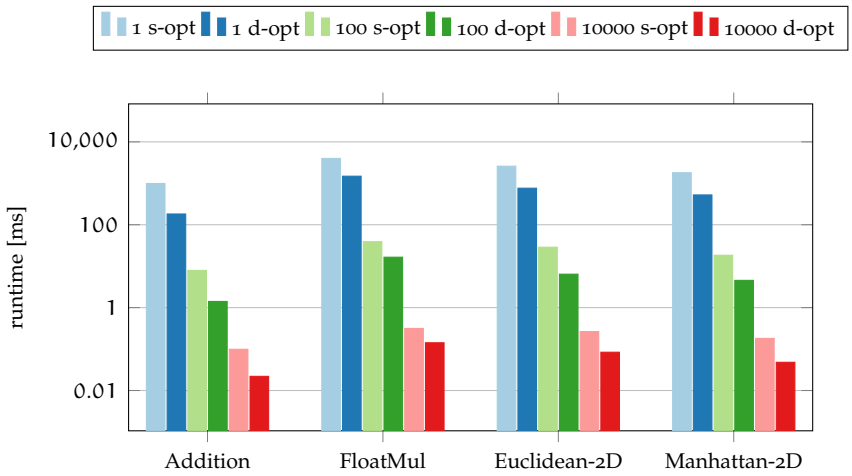


Figure 30: Size-depth trade-off analysis: Shown is the amortized runtime, i.e., the runtime to perform one operation, of 4 functionalities in a network setting with 20 ms RTT for different degrees of parallelism $(1, 100, 10000)$. Each functionality is compiled twice, i.e., size-optimized (s-opt) and depth-optimized (d-opt).

SIZE-DEPTH TRADE-OFF. To further study the trade-off between size- and depth-minimized circuits in multi-round MPC protocols, we evaluate four exemplary functionalities that have been compiled with CBMC-GC and ShallowCC with different degrees of parallelism. The functionalities are a 32 bit integer addition, a floating point multiplication, Euclidean and Manhattan distance computation. The resulting circuit dimensions are shown in Figure 30. Each circuit is evaluated with GMW using ABY in an exemplary network setting with a fixed RTT of 20 ms. Measured is the amortized runtime in ms of a single circuit, when evaluating the circuit individually, 100 or $10,000$ times in parallel. The results are averaged over 100 runs each. We observe that an increasing degree of parallelism significantly decreases the amortized runtime for all functionalities. This is because computation and bandwidth limits have not been reached and thus more gates can be computed and transmitted per communication round. We also observe that for all parallel batch sizes the depth minimized circuits outperform the size optimized circuits, even under significant size-depth trade-offs, e.g., the depth-optimized addition circuit is ≈ 5 times larger than the size-optimized circuit. Hence, even for a coarse grained parallelization degree of $10,000$ functionalities and only a moderate RTT of 20 ms, the trade-off between circuit depth and size is in favor of depth when aiming an optimal runtime. The performance gap increases with an increasing RTT.

In conclusion, the experiments support our introductory statement that depth minimization is of uttermost importance to gain further speed-ups in multi-round MPC protocols. With the ShallowCC extension to CBMC-GC we are capable to achieve these speed-ups by com-

piling depth-minimized circuits for the aforementioned benchmark applications that are up to 2.5 times shallower than hand optimized circuits and up to 400 times shallower than circuits compiled from size optimizing compilers.

6.4 RELATED WORK

The relevance of low non-linear depth for various cryptographic tasks led to a multitude on works that proposed or optimized circuits for cryptographic primitives. For example, Doröz et al. [Dor+14] identified the need for depth-minimized cryptographic primitives, Boyar et al. [BP12] developed a depth-minimized AES S-Box and very recently, Dobrauning et al. [Dob+18] proposed a symmetric cipher name RASTA that aims a minimal non-linear depth.

Schneider and Zohner [SZ13] presented the first depth-minimized building blocks for MPC to achieve a fair comparison between GMW und Yao's Garbled Circuits. Demmler et al. [Dem+15] extended this work and presented a library of depth-minimized circuits exported from a commercial hardware synthesis tool.

General optimization heuristics to minimize the non-linear depth based on rewriting techniques have recently been described by Carpov et al. [CAS18; CAS17]. To the best of our knowledge, beside the low-level synthesis tool chain described in [Dem+15], no compiler or high-level synthesis tool chain has been proposed to compile Boolean circuits with minimal depth for MPC.

Part III

COMPILATION AND OPTIMIZATION FOR HYBRID MPC PROTOCOLS

7

COMPILATION OF HYBRID PROTOCOLS FOR PRACTICAL SECURE COMPUTATION

Summary: In previous chapters, we presented compilation approaches for different MPC protocols, yet always studied them separately. Yet, it is known that multiple protocols using different cryptographic techniques can be combined to more efficient *hybrid* protocols.

In this chapter we present HyCC, a tool chain for automated compilation of ANSI C programs into *hybrid* protocols, supporting size and depth minimized Boolean circuits and size minimized Arithmetic circuits. With HyCC we present a scalable compilation approach, that automatically decomposes an input source code, compiles the decomposed parts, performs local and global optimizations, and finally partitions the compiled application by selecting the most efficient MPC protocols for each part. As a result, our compiled protocols are able to achieve performance numbers that are comparable to hand-built solutions. For the MiniONN neural network (Liu et al., CCS 2017), our compiler improves performance of the resulting protocol by more than a factor of three. Thus, for the first time, highly efficient hybrid MPC becomes accessible for developers without cryptographic background.

Remarks: This chapter is based in parts on our paper – *"HyCC: Compilation of Hybrid Protocols for Practical Secure Computation"*, Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, Thomas Schneider, which appeared in the Proceedings of the 25st ACM Conference on Computer and Communications Security (CCS), Toronto, Canada, 2018 and our paper – *"Scalable Secure Computation from ANSI-C"*, Niklas Büscher, David Kretzmer, Arnav Jindal, Stefan Katzenbeisser, appeared in the Proceedings of the IEEE International Workshop on Information Forensics and Security (WIFS), Abu Dhabi, United Arab Emirates, 2016.

The first paper has been written in the CROSSING collaborative research center and is not solely used in this dissertation. The evaluation section is a joint effort by Daniel Demmler and myself (and thus shared in both theses), the compiler and the protocol selection is my major contribution. The other authors contributed to the implementation and the overall writing of the paper.

7.1 MOTIVATION

In the previous chapters, we studied the compilation for standalone MPC protocols. Albeit their advancements, further optimization is of interest to lower computational and communication requirements of MPC. One solution is to use *hybrid* protocols. Hybrid protocols combine multiple different MPC protocols with the potential to outperform a standalone protocol [DSZ15; Hen+10; KSS14; Liu+17; Nik+13b; PS15]. For example, for an application that consists of numerical com-

putation and a combinatorial problem, it is beneficial to evaluate the former part with an Arithmetic circuit-based MPC protocol, and the latter part with a Boolean circuit-based protocol.

Identifying a (near) optimal choice of MPC protocols for a desired application requires experience with different MPC protocols, their optimizations, their programming models, and conversion costs to securely switch between protocols when performing a hybrid computation. Unfortunately, existing compilers, such as the ones presented in the previous chapters, mostly target only a single class of protocols, e.g., Yao's Garbled Circuits [Mal+04; Son+15] (cf. Chapter 4), the GMW protocol [Dem+15] (cf. Chapter 6), or additive linear secret-sharing-based MPC [BLW08]. Although removing the need of experience in hardware design, all these compilers follow a holistic compilation of an input program rather than supporting automatized decomposition, which is a prerequisite for hybrid MPC. An exception is the TASTY compiler [Hen+10], which compiles to a hybrid protocol consisting of Yao's protocol and additive homomorphic encryption. Yet, TASTY requires specific annotations to mark which protocol is used for each statement. The only other compiler that addresses the compilation of a program using two MPC protocols (Yao's gabled circuits and arithmetic sharing) is *EzPC* [Cha+17]. However, EzPC only provides semi-automation for a domain specific language (DSL), as the input code has to be manually decomposed, array accesses have to be manually resolved into multiplexer structures, and the compiled circuits are left unoptimized. Moreover, EzPC supports only two MPC protocols, which are selected statically and independently of the execution environment, by following a strict set of rules for each expression in the program.

COMPILATION FOR HYBRID MPC. In this chapter, we propose a novel hybrid circuit compiler, named HyCC, that is capable of compiling applications written in standard ANSI C code into an optimized combination of MPC protocols. In contrast to previous work on hybrid compilation, we present a fully automated approach that decomposes the source code, translates the decomposed code into Boolean and Arithmetic circuits, optimizes these circuits, and finally selects suitable MPC protocols for a given deployment scenario, optimizing the selection for a given criterion, such as latency (minimal total runtime), throughput (minimal per-operation runtime), or communication.

Figure 31 illustrates the two major components of this approach. The first component is the (one-time) compilation of the input source code into a decomposed program description in form of circuits. We refer to the different parts of a decomposed program, i.e., the compact logical building blocks a larger application consists of, as *modules*. Each module is compiled into multiple circuit representations. Our

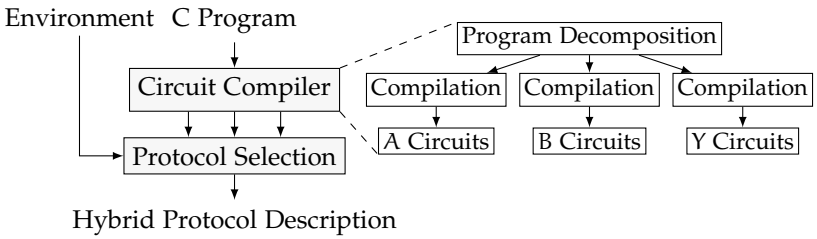


Figure 31: High-level overview of HyCC's compilation architecture. The circuit compiler decomposes an input program and compiles each part into multiple circuit representations. The protocol selection recombines the different parts.

implementation compiles Arithmetic circuits (A), depth-optimized circuits for GMW (B), and size-optimized circuits for Yao's protocol (Y). The second component in HyCC is the protocol selection step in which the most suitable combination of MPC protocols is selected for a decomposed program depending on the computational environment. We note that this protocol selection can be part of an MPC framework and does not necessarily need to be performed during compilation.

OPTIMIZING CIRCUIT COMPILER. MPC is still significantly slower and more expensive than generic plaintext computation in terms of both computation and communication. Thus, a tool chain is required that optimizes the compilation of a program description into an efficient MPC protocol and its corresponding circuits. Even though the optimization of an input program has limits, i.e., an inefficient algorithmic representation cannot automatically be translated into a fast algorithm, a programmer expects the compiler to not only translate every statement of a high-level description of an application or algorithm for a selected target architecture, but also to optimize the given representation, e.g., by removing unnecessary computations. This is of special interest for MPC compilers because code optimization techniques that are too expensive to be applied in traditional compilers become affordable when considering the trade-off between compile time and evaluation costs of the program on the circuit level. For example, in Yao's protocol a 32×32 bit signed integer multiplication requires the evaluation of ~1000 non-linear Boolean gates (cf. Section 4.3), which results in ~5000 symmetric encryptions during the protocol run. Consequently, the removal of any unnecessary operation in MPC is more vital than in traditional compilation, where at most a single CPU cycle is lost per multiplication during program execution. We also observe that optimization techniques performed on the source code level, e.g., constant propagation, are cheaper in computational resources than minimization techniques applied on the gate level after the compilation to circuits.

These observations are reflected in our compiler architecture: Before decomposing the input source code into different parts, a rigorous static analysis is performed to realize constant propagation, detect parallelism, and determine the granularity of decomposition. The optimization then continues on the circuit level, where logic optimization techniques are gradually applied. To achieve a scalable and optimizing compilation, we guide the logic optimization efforts based on the results of static analysis of the source code. For example, loop bodies with a large number of iterations will be optimized with more effort than a piece of code that is only rarely used. Thus, in contrast to classic logic optimization or arithmetic expression rewriting, we make use of structural information given by the programmer in the high-level code.

Summarizing the compiler's functionality, HyCC is capable of compiling optimized Boolean and Arithmetic circuits suiting the requirements of most constant- and multi-round MPC protocols. Our tool chain is highly flexible and independent of the underlying MPC protocols, as only the respective cost models for primitive operations, e.g., addition or Boolean AND, have to be adapted to reflect future protocol developments in MPC.

PROTOCOL SELECTION. Protocol selection is the task of mapping each part of a decomposed program to an MPC protocol representation. The circuits created by our compiler for each module and the mapping of modules into MPC protocols is sufficient to evaluate an application in a hybrid MPC framework. Optimal protocol selection is an optimization problem, where the best mapping is identified in regard to the cost model that considers the cost to evaluate each circuit in the respective MPC protocol as well as the conversion costs between the different representations. The concept of protocol selection has previously been studied independently from compilation in [KSS14; Pat+16]. Kerschbaum et al. [KSS14] investigated protocol selection for a combination of Yao's garbled circuits and additive homomorphic encryption. They conjectured that the optimization problem is NP-hard and proposed two heuristic approaches. First, they presented a transformation of the combinatorial optimization problem into an integer linear programming task by linearization of the cost model. Second, they presented a greedy optimization algorithm, which is capable of optimizing larger functionalities. Pattku et al. [Pat+16] used similar heuristics to optimize the protocol selection for minimal cloud computing costs, i.e., the price to pay a cloud provider to perform a computation.

We follow an approach that is different in multiple aspects. First, we show that the synthesis of an efficient hybrid MPC protocol is not only a protocol selection problem, but also a scheduling problem. Second, in contrast to the work mentioned above, we make use of

structural information in the source code before its translation into circuits. By grouping expressions that perform similar operations, e.g., loops, it becomes possible to perform an exhaustive search over the problem state for many practically relevant applications. Applications that cannot be optimized to the full extent with the available optimization time are approached by a combination of exhaustive search with heuristics.

CONTRIBUTIONS. We present the first complete tool-chain that automatically creates partitioned circuits and optimizes their selection for hybrid MPC protocols from standard ANSI C code, which makes hybrid MPC accessible to non-domain experts. Moreover, we contribute techniques and heuristics for the efficient decomposition of the code, scalable compilation, and protocol selection. As part of these techniques, we propose source code guided optimization, which allow a more scalable optimization of circuits. Finally, we report speedups for our automatically compiled hybrid protocols of more than one order of magnitude over stand-alone protocol compilers, and factor three over previous handmade protocols for an exemplary machine learning application [Liu+17].

CHAPTER OUTLINE. This chapter is organized as follows: Our compilation architecture is presented in Section 7.2, followed by a discussion of protocol selection and partitioning in Section 7.3. We evaluate HyCC in Section 7.4 and discuss related work in Section 7.5.

7.2 THE HYCC MPC COMPILER

Here we describe our hybrid compiler. After introducing the challenges, we provide details on every step of the compilation chain.

7.2.1 *Hybrid Compilation and its Challenges*

We begin with a description of a straight-forward (unoptimized) approach to compile hybrid MPC applications from standard source code in order to illustrate the challenges of achieving efficient hybrid compilation. We will then refine this approach throughout this section and describe a more advanced compilation approach.

An exemplary illustration of the necessary steps for a straightforward compilation is given in Figure 32. First, the input source code is decomposed into multiple parts, henceforth referred to as *modules*. Modules are the finest level of granularity used in the later protocol selection. Thus, all code within a module is guaranteed to be evaluated with the same MPC protocol. We remark that during protocol evaluation this level of granularity is only forming a lower bound. In principle, a program can also be evaluated with only a single MPC

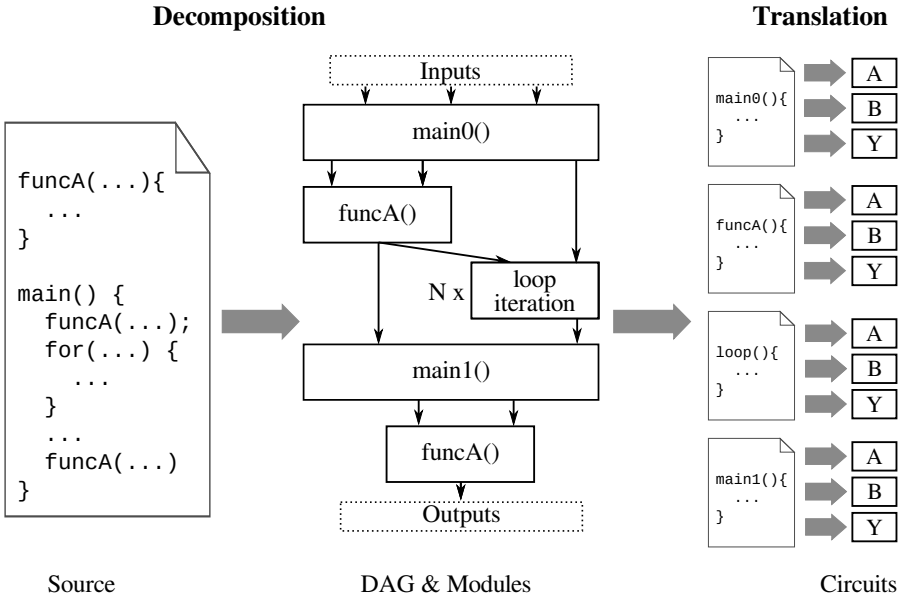


Figure 32: Naïve compilation of hybrid protocols from input source code to a decomposed circuit description. First, the code is decomposed into multiple modules. Then, each module is translated into three different circuit formats.

protocol. The decomposition can be made directly on the source code level or on an intermediate representation of the code, e.g., SSA form. Given a decomposed application description, each module is compiled into the circuit representations for the different MPC protocols forming the hybrid protocol, and then optimized. In HyCC, we consider size-optimized Boolean circuits, required for Yao's protocol (Y), depth-optimized Boolean circuits, required for GMW style protocols (B), and Arithmetic circuits (A). Finally, the hybrid application is synthesized during protocol selection and scheduling, described in detail in Section 7.3.

Multiple challenges (besides the complexity of compiling efficient Boolean or Arithmetic circuits itself) arise when following this straightforward approach. All challenges relate to a trade-off between compilation resources, i.e., time and storage, and compilation result, i.e., circuit size and depth. We describe identified challenges and propose solutions, which motivate our actual compilation architecture:

- *Granularity of decomposition.* Automatically decomposing input code into multiple modules is a non-trivial task, as a fine-grained decomposition limits the possibility of circuit level optimizations and increases the complexity of the computationally expensive protocol selection problem, whereas a coarse-grained decomposition risks to miss the most efficient selection. We tackle this challenge by the use of heuristics based on static analysis of the source code.

```
1  long pow(unsigned b, unsigned exp) {
2    /* Computationally expensive code */
3  }
4
5  void main(){ /* Some code */
6    t1 = pow(x, y);
7    t2 = pow(2, y);
8    /* Some code */
9    unsigned c = 1;
10   if (condition)
11     c += 1;
12   res = pow(x, c);
13   /* Some code */
14 }
```

Listing 15: Example source code to illustrate the conflict between local and inter-procedural optimization.

- *Local versus inter-procedural optimization.* Optimizing an application as a whole or optimizing its modules independently can lead to circuits of different sizes. The former allows more optimizations, whereas the latter is typically more efficient w.r.t compilation because each module will only be compiled and optimized once.
  We illustrate this conflict with the example in Listing 15. This example consists of a function `main()` that performs multiple calls to a function `pow()`, which computes the power of two integers. A function-wise decomposition approach would separate the two functions to compile them independently. However, a careful study of the source code reveals that the `pow()` function is called with a constant argument in Line 7, and with the second argument being either one or two in Line 12, which simplifies the computation of the exponentiation function on the circuit level significantly. An optimizer with an inter-procedural (context-sensitive) approach could detect this fact and optimize the created circuit accordingly. To find a trade-off between modular and holistic optimization, i.e., compile time and circuit size, we rely on static analysis and source code optimization techniques in our compilation framework.

- *Loop handling.* Loops are an essential part of many programs. To create circuits with low complexity, it is best to first unroll (inline) all loop iterations, before translating them into a circuit, as this allows to apply optimizations, such as constant propagation, over all iterations. However, for compilation efficiency, for the exploitation of parallelism, and for a more compact circuit representation, it can be useful to avoid loop unrolling. Therefore, instead of choosing either technique we propose an adap-

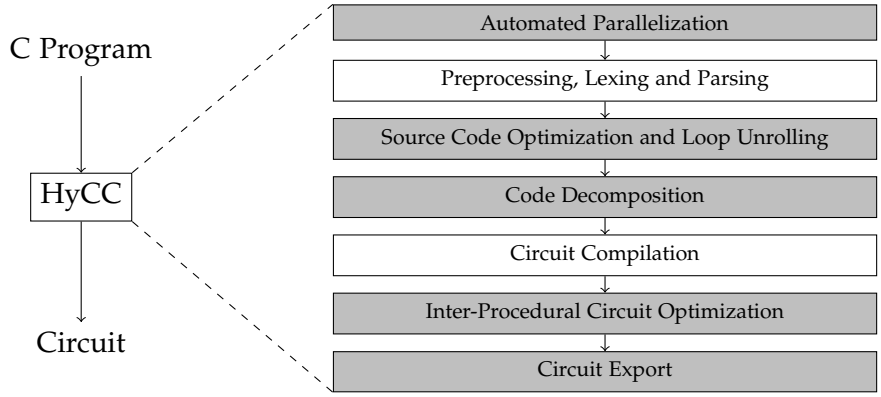


Figure 33: The compilation chain of HyCC. Marked in gray are all new or adapted compilation phases.

tive approach that distinguishes different loop types and then decides for or against loop unrolling.

- *Efficient logic minimization.* Even though we consider the compilation to be a one-time task, which in theory allows to use arbitrary resources, in practice compilation efficiency is of relevance. Optimizing circuits on the gate-level is a resource-consuming task that can become practically infeasible when considering circuits with billions of gates. Therefore, we propose a technique named *source-guided optimization* to optimize circuits under configurable time constraints, by distributing an optimization budget in a controlled manner.

The sketched solutions can be realized using static source code analysis techniques only. This is sufficient because MPC applications have to be bound (finite and deterministic runtime), as they are evaluated independently of the input of the program to avoid any form of information leakage. Using the side-channel free circuit computation model, all possible program paths are visited during protocol runtime and thus can already be studied at compile time.

7.2.2 *Architecture*

We describe our compilation architecture for a resource-constrained environment that expects a source code with a pointer to an entry function f as input, and a compilation and optimization time limit T. The compiler outputs a program description consisting of multiple modules, compiled to different circuit representations, and a direct acyclic dependency graph that describes the dependencies between the different modules. The combination of dependency graph and modules can be used to evaluate the program in a hybrid MPC framework.

The compilation architecture, illustrated in Figure 33 consists of multiple compilation *phases* shown in the next paragraph, which themselves can consist of multiple compilation *passes*. The phases are:

1. *Automated Parallelization:* Automated identification of code blocks that can be evaluated in parallel using external tools.
2. *Preprocessing, Lexing and Parsing:* Construction of an AST from the input code.
3. *Source Code Optimization and Loop Unrolling:* Source-to-source compilation using static analysis.
4. *Code Decomposition:* Decomposition of the input program into multiple modules.
5. *Circuit Compilation:* Compilation of each module into the different circuit representations.
6. *Inter-Procedural Circuit Optimization:* Optimization of Boolean and Arithmetic circuits across multiple modules.
7. *Circuit Export:* Writing the decomposed circuit to a file, ready for reconstruction in protocol selection.

Note that steps 2, 3, and 5 are also part of CBMC-GC's tool chain, whereas the others have been added for the compilation of hybrid protocols. We describe the steps in detail in the following subsections.

Automated Parallelization

Parallel code segments allow efficient compilation and protocol selection. Moreover, most MPC protocols profit from parallelized functionalities. Therefore, their detection is of relevance in compilation for hybrid MPC. As in Chapter 5, we rely on existing automated parallelization tools, e.g., [IJT91; Wil+94], for the detection of *parallel loops*, i.e., loops that have independent loop iterations. These tools are able to detect parallelism and to annotate parallelism using source-to-source compilation techniques, independent of the HyCC compilation chain. In HyCC we follow the same approach as in ParCC, which is described in Section 5.3.2: For annotations, HyCC relies on the OpenMP notation, which is the de-facto application programming interface for shared memory multiprocessing programming in C and supported by most parallelization tools. Specific preprocessing notations, e.g., `#omp parallel for`, are added in the code line before each parallel loop. The annotations are parsed in the next compilation phase.

Preprocessing, Lexing and Parsing

The preprocessing, lexing, and parsing of source code is realized as described in Chapter 3. We remark that, as in CBMC-GC and typical for MPC, the given program has to be bound to avoid leaking information through the program runtime. Furthermore, global variables are not supported, which, however, is an implementation limitation and not a limitation of our approach.

Source Code Optimization and Loop Unrolling

In this compilation step the intermediate code is analyzed and optimized using static analysis. The results are subsequently used as a preparation step for the later code decomposition and parallelization. In detail, to overcome the optimization limits of a (context insensitive) modular compilation, rigorous source code optimization in form of a partial evaluation is performed. Thus, all variables known to be *constant* are propagated, such that every remaining expression (indirectly) depends on at least one input variable (*dynamic variable*).

To achieve an efficient compilation result, partial evaluation requires symbolic execution of the complete source code, which limits compilation scalability. A faster compile time can be achieved, under a (often significant) circuit-size trade-off, when not optimizing across function or loop boundaries. For example, the circuit compiler Frigate [Moo+16] follows this approach. To achieve the best of both worlds, we propose a time-constrained multi-pass optimization routine, which can be interrupted at any point in time. Given sufficient compile time, the iterative approach converges to the same result as a complete context sensitive optimization.

In the first pass, partial evaluation is performed with a local scope, yet not across function or loop boundaries. In the second pass, constants are propagated within every function body and between multiple functions (*inter-procedural constant propagation*), yet not between multiple loop iterations or in recursive function calls to avoid loop unrolling. This form of program specialization can lead to an increase in the code size, as the same function may now appear multiple times with different signatures. For example, in Listing 15, we observe that the `pow()` function is called with none, either of the two, and both arguments being constant. Hence, in this example, two, namely one with the first argument and one with the second argument being constant, additional copies of the function will be introduced (*function cloning*), partially evaluated, and compiled individually.

In the third optimization pass, all (possibly nested) loops are visited. We distinguish three types of loops: Parallel, simple, and complex loops. *Parallel* loops have already been identified in the first compilation phase. We refer to a `for` loop as *simple* if the loop guard is constant and the iterator variable is incremented (or decremented)

in a constant interval and not written inside the loop body. Furthermore, simple loops cannot have `return` or `break` statements. Hence, the loop range of simple loops can be derived without a complete symbolic execution of the loop itself. *Complex* loops are all remaining loops, which require a complete unrolling of all iterations using symbolic execution to determine their termination.

Simple and parallel loops do not need to be unrolled during compilation, as it is sufficient to compile a single circuit for all iterations that is instantiated multiple times within an MPC protocol with the loop iterator variable as input. Nevertheless, similar to function specialization, loop specialization is desirable for an efficient compilation result. Therefore, in HyCC, loops are optimized in an iterative approach. First, all constants that are independent of the loop iterator variable are propagated in the loop body. This allows an effective optimization of multiple loop iterations at the same time. Afterwards, the first iteration of every loop is partially evaluated. In contrast to the previous symbolic execution, the loop iterator variable is now initialized with a constant and can lead to further program specialization. If symbolic execution of the first iteration leads to improvements, i.e., an expression can be evaluated or removed, then the loop becomes a candidate for unrolling. By unrolling the first loop iteration, an estimate on the computational resources required to unroll all iterations can be made. Given sufficient remaining compile-time (and memory), the loop will be unrolled and optimized.

Function and loop specialization may reveal constants relevant for other parts of the code. Therefore, given sufficient remaining compile-time, a further round of partial evaluation is initiated until no further improvements are observed. Finally, a call-graph is exported for usage in the following decomposition. Statements within loops that have been unrolled are enriched with information about their original position in the loop, to re-identify loops and their iterations during decomposition.

Code Decomposition

Identifying a suitable decomposition is the major challenge for efficient protocol partitioning. The task of automated decomposition is to identify which parts of a code should jointly be compiled as one module, which forms the finest level of granularity of protocol selection. Each module has an input and an output interface, where a module can receive input from one or more modules and provide output to one or more modules. We refer to the separation points between two modules as interface. Hence, a decomposed code forms a DAG consisting of modules with interfaces in-between (similar to a call-graph or dependency-graph). The first input and last output interface of the graph are the program input and output variables, respectively.

The overall goal of a useful decomposition is to identify modules of a program that can be evaluated efficiently in a specific circuit representation. A first example of such a heuristic are expressions consisting only of arithmetic statements. Naturally, these should profit from processing in MPC protocols based on Arithmetic circuits. In contrast, control flow operations or comparisons are evaluated more efficiently with Boolean circuit-based protocols. Consequently, arithmetic and combinatorial statements should be in different modules. We follow a multi-pass decomposition approach that starts with the complete source code as a module that is split into more fine-granular modules in every pass.

FUNCTION DECOMPOSITION. Functions already give programs a form of modularization and hence they can be used as natural boundaries for decomposition. Therefore, in the first compilation pass, each function becomes a module, while considering the previously described function specialization. The input interface to a function module consists of the arguments that are read in the function body and assigned to other variables. The output interface are all pointers and variables passed by reference that are written to in the function body, as well as the `return` statement. This form of recursive decomposition leads to three modules per (possibly nested) function call, one module for the callee itself, one for the code before and one after the function call.

Technically, this decomposition becomes challenging when pointers or references are passed to a function. Using the results of the previous (exhaustive) symbolic execution, which involves a pointer analysis, input and output variables can be differentiated, and array sizes can be determined during compile time. We note that dynamic memory management, i.e., memory that is allocated based on (private) input variables, is impossible to be realized in the circuit computation model and is thus outside the scope of circuit compilers.

LOOP DECOMPOSITION. Loops also give code a structure and are therefore a good heuristic for decomposition. Consequently, in the second compilation pass, every module is further decomposed according to its loops, such that every loop iteration becomes its own module, where all variables that are read from an outer scope and the iterator variable form the input interface and all variables that are written to, but defined in an outer scope, form the output interface.

Loops might have been unrolled during code optimization, as described previously. For their re-identification during decomposition, loop iterations are marked as such during loop unrolling. Loops that have not been unrolled during code optimization require a dedicated handling of array accesses before decomposition. Otherwise, an array access that depends on the iterator variable, which is an input

```
1 unsigned scalar = x1 * y1 + x2 * y2;
2 if(scalar > min) {
3   count = count + 1;
4 }
```

Listing 16: Code excerpt to illustrate code decomposition. The scalar product of two two-dimensional vectors is computed and compared to a reference value.

variable after decomposition, would compile into a private array access, which is represented by large multiplexers, cf. Section 4.3. For better efficiency, in HyCC these array accesses are extracted from the loop iteration and placed in the module that encapsulates the iteration. Consequently, these array accesses are evaluated as accesses with publicly known index, and as such without any gates.

Decomposition by loops is especially beneficial for parallel loops, as it allows to derive the placement costs of MPC protocols during protocol selection from the analysis of only one loop iteration.

ARITHMETIC DECOMPOSITION. In the last decomposition pass, connected arithmetic expressions are extracted, as they are candidates for Arithmetic circuits. Therefore, all expressions in each module are visited to extract expressions that purely consists of arithmetic operations (supported by the used MPC protocol). This decomposition is realized as follows: For each module, a data flow dependency graph is constructed from the output to the input interface. Each node in the dependency graph is an elementary expression and an edge represents the data that is computed on. By iterating over all nodes, two sets of sub-graphs are formed. The first contains sub-graphs consisting of connected arithmetic expressions, whereas the second contains sub-graphs consisting of connected remaining expressions. Each sub-graph forms its own module, where edges between the sub-graphs define the respective I/O interfaces. This form of decomposition is illustrated in Figure 34 for the code excerpt given in Listing 16 that computes a scalar multiplication.

We remark that during protocol selection, multiple (or even all) modules can be merged to larger modules, to be jointly evaluated with the same MPC protocol. Finally, the created DAG that represents the modules and their I/O dependencies is exported for the next compilation steps.

Circuit Compilation

The different modules identified in the previous step are compiled separately into two or three circuit representations. Namely, every module is compiled into size-optimized Boolean circuits using the cir-

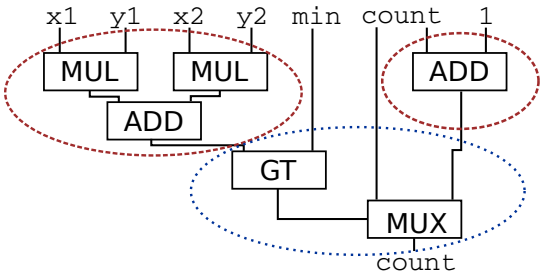


Figure 34: Code dependency graph and decomposition for the code excerpt in Listing 16. Connected statements that can efficiently be expressed as Arithmetic circuits, i.e., addition (ADD) and multiplication (MUL), marked with a red dashed circle, form subgraphs. Statements that profit from a Boolean representation, marked with a blue dotted circle, i.e., greater-than (GT) and multiplexer (MUX), are grouped.

cuit compiler of CBMC-GC (see Chapter 4) and into depth-optimized Boolean circuits using its ShallowCC extension (see Chapter 6). Moreover, every module that can be represented with the supported arithmetic operations, i.e., addition and multiplication (cf. Section 2.2), is also compiled into an Arithmetic circuit using a straight-forward mapping of arithmetic expressions to arithmetic gates. Note that modules representing functions or loops that have not been unrolled are only compiled once.

Inter-Procedural Circuit Optimization

So far, the compiled circuits have only been optimized on the source code level. Yet, it is desirable to also optimize the Boolean circuits on the gate-level, e.g., by removing unused bits (gates) and by propagating constants between modules and circuit types, as applied in CBMC-GC, cf. Section 4.4, and also to subsequently propagate constants in Arithmetic circuits.

The scalability of logic minimization techniques for Boolean circuits is limited, because these techniques are applied in a gate-by-gate manner and some techniques involve computationally expensive operations, such as SAT sweeping. Thus, to distribute the available computational resources, i.e., the optimization time limit T_{opt}, which is the total user specified time T subtracted by the compile time T_{comp}, onto all modules efficiently, we propose a technique named *source code guided optimization*. Over multiple time-constrained optimization passes, the available computing time is distributed using the structural information present on the source code level and the information available from previous optimization passes.

Initially, every module is optimized at least once. For the initial optimization, a fraction λ_{init} ($0 \leqslant \lambda_{init} \leqslant 1$) of the total budget T_{opt} is distributed onto all modules weighted by the number of their non-linear gates. Furthermore, modules originating loops or function

bodies are optimized with an effort that is weighted by the number of their iterations or calls. After the initial optimization pass, the minimized modules are queued by the number of constant inputs, which have been revealed in the previous pass, as well as the absolute improvement in the number of non-linear gates in the previous pass. Based on this order, the most promising module is selected for minimization. After minimizing a module, it is returned to the queue according to the aforementioned criteria and modules with newly revealed constant inputs are reordered. The optimization stops once T_{opt} has be reached or no further improvements are observed. An algorithmic description of the optimization heuristic is given in Algorithm 1.

We selected this heuristic because of two reasons. First, constant propagation is the most cost efficient optimization strategy and should therefore be applied with preference. Second, in the fixed point logic minimization routine (cf. Section 4.4), circuits that have seen significant optimization are more promising for further optimization than circuits with little or none improvement in the previous pass. Therefore, these are given preference.

The described optimization strategy enables the scalable optimization of circuits for standalone protocols and is evaluated in more detail in our paper [Büs+16]. For hybrid compilation we optimize all types of circuits independently with a shared optimization budget. Thus, the different optimizations are performed separately. Yet, the information about identified constant output variables or module outputs is propagated between optimization routines and will also be used to improve the Arithmetic circuits, if all bits of an output variable are identified as constant. We remark that this form of cross propagation maintains functional correctness because all circuits for one module are logically equivalent.

Circuit Export

Once a user-defined compile time has been reached, the optimization routine is stopped and the DAG, consisting of modules with optimized circuit representations and I/O interfaces as well as information about identified parallel loops, is exported. Given the circuits, an MPC framework can choose a protocol selection to perform the computation as described next.

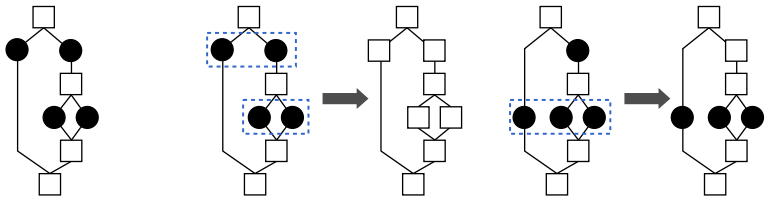
7.3 PROTOCOL SELECTION AND SCHEDULING

In this section, we describe how to determine an optimized scheduling and mapping of the modules that were created during compilation to MPC protocols.

7.3.1 *Problem Definition*

We optimize evaluation costs of a hybrid MPC application by choosing an efficient protocol representation and evaluation order of all modules for a given program description. For this, we present heuristics considering a user-specified cost model. A very interesting use case is the optimization of the protocol's online runtime, yet, various other cost models are also of interest. For example, optimizing the cloud computing costs has been discussed in [Pat+16]. Further examples are the total protocol runtime including or excluding the time spent on preprocessing, depending on the use case of the application, the pure communication costs when considering a constrained network connection, or the power consumption, when considering mobile devices. All these minimization problems can also be formulated as constrained problems, e.g., minimizing the communication costs while keeping the protocol runtime below a user-defined threshold.

The computation and communication costs of a hybrid MPC application depend on the combined costs to evaluate each module in the selected protocol plus the time to convert between modules, when evaluating them with different protocols. However, in contrast to previous works, i.e., [KSS14] and [Pat+16], we observe that the optimization problem, i.e., achieving minimal costs for a given decomposition, is not only a protocol selection problem, but also a scheduling problem. Namely, the evaluation order of parallel modules, i.e., modules without sequential dependencies, can significantly influence the effectiveness of protocol selection, and thus the overall protocol runtime. This is because of the non-linearity of computation and communication costs of MPC protocols (e.g., parallel computations in the program can be performed in the same communication round or packed in the same cryptographic operation), as well as the trade-off that has to be taken into account when converting between different MPC protocols. Figure 35 illustrates this scheduling problem for an example program description and naïve cost model. Namely, Figure 35a shows an exemplary program DAG resembling a computation from inputs (top) to outputs (bottom) with different modules (nodes) in between. For simplicity, we assume that modules illustrated as squares profit from an evaluation in protocol type A (e.g., arithmetic), whereas modules represented by circles profit from a different protocol type B (e.g., Boolean). Furthermore, for illustration purposes, we assume that a conversion between two different protocols is reducing the total evaluation costs if at least three modules are evaluated in the same protocol. The result of an exemplary As-soon-as-possible (ASAP) scheduling followed by a protocol selection is shown in Figure 35b. Two groups of nodes (marked with dashed lines) become a candidate for being evaluated in protocol type B. However, when considering the assumption above, an optimal protocol selection algorithm will pro-



(a) Example DAG with protocol preferences

(b) ASAP scheduling and subsequent protocol selection

(c) Optimal scheduling and subsequent protocol selection

Figure 35: Exemplary DAG with different evaluation schedules and protocol selections described Section 7.3.1.

pose to evaluate all modules with type A, as the conversion is too expensive for only two modules. An optimal scheduling is shown in Figure 35c. In this case, three modules that can jointly be evaluated in protocol type B can be identified during protocol selection and are consequently evaluated in protocol type B.

Thus, we remark that optimal runtime can only be achieved when optimizing both protocol selection and scheduling of modules. Next, we present a formalization of the optimization problem, before presenting optimization routines in the following subsections.

FORMALIZATION. We formalize the cost model and optimization problem as follows. Given is a program description in the form of a DAG G from inputs $i \in In$ to outputs $o \in Out$ with modules $m \in M$ in between. Cost minimization for hybrid MPC consists of two interleaving tasks, namely protocol selection and scheduling. Protocol selection is an assignment that maps every module to an MPC protocol, also referred to as protocol type $t : M \to \{A, B, Y\}$. We denote the set of protocols that represents each module in the respective protocol type with Π^t_M. Moreover, we denote the set of conversion protocols required to convert between adjacent modules evaluated with different MPC protocols with Π^t_C.

Scheduling is the task of assigning an evaluation order to all modules for a given protocol selection. Modules and their conversions form the set of elementary protocols $\Pi^t = \Pi^t_M \cup \Pi^t_C$ that are the atomic units of scheduling. As it is common in scheduling, we use the notion of instructions I, which is the set of protocols that are performed in parallel in hybrid MPC. Furthermore, note that most modules and their conversions have data dependencies to other modules, i.e., module m_2 is dependent on m_1, if the result of m_1 is needed to compute m_2. Therefore, scheduling is the task of creating a sequence of k instructions $(I_1, I_2, \ldots, I_k)$ and assigning protocols to instructions $s : \Pi^t \to I_1, \ldots, I_k$. This assignment must guarantee that every protocol only appears in one instruction, protocols in each instruction are pairwise mutually independent, and the order of protocols

induced through the order of instructions conforms to the dependencies between modules and conversions.

Given a schedule, i.e., an ordered list of instructions IL, the total evaluation cost is the sum of the evaluation costs of all protocols representing a module $\pi_m^t \in \Pi_M^t$ and their respective conversions $\pi_c^t \in \Pi_C^t$ according to IL plus the cost to input values into the protocol, plus the costs to reveal all outputs. In our paper [Büs+18], we study the protocol evaluation costs in more detail and also present a runtime prediction for a given evaluation schedule.

In summary, the goal of optimized protocol selection and scheduling is to minimize the total evaluation cost by choosing a schedule s and protocol selection t. Next, we present approaches to achieve efficient protocol selection and scheduling.

7.3.2 *Protocol Selection in HyCC*

Scheduling and protocol selection are tightly coupled problems, where the latter alone is conjectured to be NP-hard [KSS14; Pat+16]. Therefore, in HyCC we first select an evaluation schedule using an heuristic for a given program decomposition. The schedule is then used in a second step to optimally solve the protocol selection problem.

SCHEDULING. In HyCC, protocol scheduling is performed with respect to the parallelism present on the source code level. Consequently, the identified parallelism, which has been annotated in the program's DAG during compilation, is used to schedule modules in parallel. This explicit scheduling of parallel code structures is necessary, as the straight forward application of an ASAP or similar scheduling algorithm cannot guarantee that parallel code statements will be evaluated in parallel, as shown in Figure 35. Moreover, this approach is beneficial for hybrid MPC, as the MPC protocols, conversion protocols, and their implementations benefit from parallel execution. For example, n sequentially scheduled multiplications in an Arithmetic circuit require n communication rounds, whereas a parallel alignment allows to perform all multiplications in a single communication round, which leads to very different runtimes in any high-latency deployment scenario. Furthermore, parallelization is beneficial for the later protocol selection, as multiple modules can be grouped together and thus, optimized more efficiently.

Besides parallelization, modules are scheduled in an ASAP manner. To combine both strategies in a single algorithm, parallel modules are merged in a single module when creating an ASAP schedule. Afterwards, the merged modules are restored and placed in the same instruction of the evaluation schedule. We leave more advanced scheduling algorithms for future work.

PROTOCOL SELECTION. Even though in the general case protocol selection is conjectured to be NP-hard, given a coarse-grained decomposition, such as the one created by HyCC, an optimal protocol selection can be computed under reasonable computational effort for many practical applications, as we show in Section 7.4.1. This is because the complexity of the protocol selection routine is dominated by the width of the program's DAG rather than its size. Consequently, all applications that only moderately divert in their data and control flow are candidate problems for optimal protocol selection.

To identify the optimal protocol selection for a given DAG G, we apply a straight-forward combinatorial optimization approach by enumerating all possible protocol combinations using dynamic programming. The core concept of the optimization routine is to iteratively optimize the selection of protocols up to a certain module, following the order of modules generated by the instruction list IL. In every step, one module is added and modules that do not have any open outputs, i.e., outputs that are required for subsequent modules, are removed. We refer to the set of modules with open outputs as the working set WS. For every WS, the best selection for every possible protocol combination is computed and stored. When going from WS to the next WS', the best protocol selection to represent the new WS' in every protocol combination is computed by identifying the least cost to compute WS' from any configuration of WS. Thus, the complexity of this optimization approach for a given DAG G with n modules, a maximum width of w, and s different protocol types is in $O(ns^w)$, and thus exponential in the size of the largest working set, i.e., the width of G. Consequently, for a small number of protocol types and for DAGs with moderate widths, the protocol selection problem can be solved optimally in seconds, as evaluated in Section 7.4.1.

ALGORITHM. An implementation of the algorithm is given in Algorithm 2 and described in the following paragraph. The initial WS consists of all inputs of the DAG G. Consequently, the cost to represent a WS in a specific protocol combination is the cost to share each input with the specified protocol (Line 2). Next, the iterative optimization routine is initiated. A module from the ordered G is added to the WS, and completed modules are removed to create the next working set WS' (Line 6). Then, all possible protocol combinations of the next WS' are enumerated. For each of these combinations, the best selection based on all protocol configurations of the previous WS is computed (Line 8). This task is realized in the function `eval_costs()`, outlined in Algorithm 3, which takes as input the two working sets, as well as the desired protocol configuration c' of WS' and a cost table that stores the costs to compute all possible configurations of WS. The costs to evaluate the newly added module, reflecting the protocols in WS and WS' is computed in function `cost_step()`, which

models the evaluation costs of MPC protocols. Once all combinations of WS' are computed, WS is replaced by WS' to add a further module. The algorithm ends, once all modules have been visited, and thus an optimal output sharing has been identified.

SCALABLE PROTOCOL SELECTION. In cases where the DAG exceeds the computationally manageable width, the optimization algorithm can compute the optimal protocol selection for all sub-graphs, which have a width that is solvable. For the remaining sub-graphs, or the combination of multiple sub-graphs, heuristics, such as the hill-climbing heuristic proposed in [KSS14] can be used to search for an optimized selection in the combination of different optimally solved sub-graphs.

7.4 EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of HyCC. We study the efficiency of protocol selection, the circuits created by HyCC and their performance in hybrid MPC protocols for various use cases in two different deployment scenarios. The goal of this evaluation is to illustrate that the circuits that were automatically created by HyCC from ANSI C code are comparable to hand-crafted hybrid circuits and significantly more efficient than previous single-protocol compilers. As such, we are able to show that HyCC is simplifying the ease-of-use of hybrid MPC, and thus is a powerful tool to prototype PETs. We remark that the goal of this work is not to outperform dedicated secure computation protocols, which are optimized to achieve maximum efficiency for a specific use case. We begin with an evaluation of the runtime of the protocol selection algorithm presented in Section 7.3.2.

7.4.1 *Protocol Selection*

To illustrate that an exhaustive search is a sufficient solution for the protocol selection problem in most practical cases, we measure the runtime of the protocol selection algorithm in Figure 36. Shown are the runtimes averaged over $k = 10$ executions of a straight forward (unoptimized) implementation running on a commodity laptop for randomly generated graphs with $n = 20$ modules with increasing graph width w. We observe the expected exponential growth in runtime when increasing w. Albeit being a limiting factor of our approach, we remark that to the best of our knowledge all applications considered so far in privacy research have a very small branching factor in their functionalities, which leads to very small w. For example, all use cases in this chapter have a width of at most $w = 3$, which is

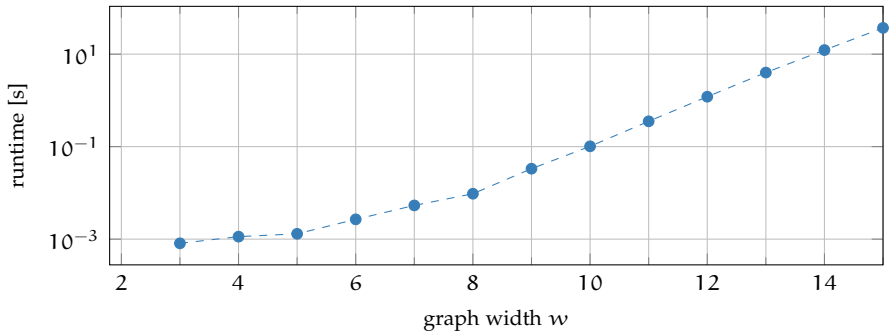


Figure 36: Averaged runtime of the protocol selection algorithm for different graph widths w.

solved in 0.1 seconds and we remark that even larger graphs with a width of $w = 10$ are solved in seconds.

7.4.2 *Use Cases*

Next, we evaluate the generated circuits and protocol selections made by HyCC for different use cases in the ABY framework [DSZ15]. The ABY framework provides state-of-the-art implementations for Yao's protocol, GMW, OT-based additive secret sharing protocols, and the corresponding conversion protocols, which makes it an ideal backend to evaluate the created circuits. For the evaluation, we use applications that illustrate the versatility of HyCC or that have previously been used to benchmark MPC protocols and compilers. A more detailed description of the applications is given in Section 2.5.

EXPERIMENTAL SETUP. All applications are implemented based on textbook algorithms and compiled with HyCC using a total optimization time of $T = 10$ minutes. We used the ABY framework to evaluate the generated circuits on two identical machines with an Intel Core i7-4790 CPU and 32 GB RAM, connected via a 1 Gbps local network, denoted as LAN. To simulate an Internet connection between the MPC parties, denoted as WAN, we use the Linux tool `tc` to set a latency of 50 ms (100 ms RTT) and limit the throughput to 100 Mbps. We set the symmetric security parameter to 128 bit. Running times are median numbers from 10 measurements. The symbol"—" denotes that no values were given or benchmarked.

For all applications the number of non-linear (multiplicative) gates, communication rounds, transferred bytes, and the protocol runtime of the setup phase and of the online phase are measured. For comparison purposes we provide these numbers not only for the best protocol selection, but also for different instantiations of the same functionality, e.g., all modules evaluated in a Boolean circuit-based protocol, or a hybrid of a Boolean circuit and Arithmetic circuit. As before,

we use A to denote an Arithmetic secret sharing based protocol, B for the GMW protocol, and Y for Yao's garbled circuits. We omitted A-only measurements for use cases that include bit-operations (e.g., minimum, comparison), since these are extremely costly in A protocol and therefore not implemented in ABY [DSZ15].

Biometric Matching (BioMatch)

As in previous chapters, we evaluate the BioMatch application, which computes the minimum Euclidean distance as the minimum of the distances from a single coordinate to a list of coordinates. We use databases consisting of $n \in \{1{,}000;\ 4{,}096;\ 16{,}384\}$ samples with dimension $d = 4$, where each coordinate has bit length $b = 32\,\mathrm{bit}$ and show performance results in Table 17.

We compare a hand-built hybrid ABY circuit [DSZ15] with a circuit that is compiled with HyCC. The results show that the circuits that we automatically compiled from a standard ANSI C description achieve the same complexity as the circuits that were hand-built and manually optimized in ABY. Here, a combination of Arithmetic protocols and Yao's protocol (A+Y) achieves the best runtime in all settings. The runtimes in both implementations show a slight variation that is due to variance of the network connection. We remark that the setup phase of the ABY circuit is more efficient, because ABY allows SIMD preprocessing, which is currently not implemented in HyCC.

To show the efficiency gain of hybrid protocols over standalone protocols, we give experiments using B or Y protocols only. These protocols are significantly less efficient and for larger input sizes even exceed the memory resources of our benchmark hardware.

Machine Learning (ML)

Machine learning (ML) has many applications and therefore also many privacy-preserving implementations have been proposed.

SUPERVISED MACHINE LEARNING – NEURAL NETWORKS. We implement CryptoNets [Gil+16] and the MiniONN CNN [Liu+17], which both have recently been proposed to detect characters from the MNIST handwriting data set. Previously these use cases needed to be carefully built by hand, while we achieve even better performance when conveniently compiling easily understandable C source code to a hybrid MPC protocol.

Table 18 shows machine learning performance results. For Cryptonets, HyCC automatically determined A as the best protocol in the LAN setting. When changing the activation function (from the square function to $f(x) = max(0, x)$, known as RELU function), or when changing the number representation (fixed point instead of integer), a hybrid A+Y protocol becomes the fastest option.

For the MiniONN CNN, HyCC proposes to use A+Y, where Y is mainly used to compute the RELU activation function, which results in a hybrid protocol that requires only a third of the online runtime, total runtime, and total communication compared to the original MiniONN protocol [Liu+17]. When expressing the entire MiniONN functionality solely as a Boolean, the circuit created by HyCC consists of more than 250 million non-linear gates. Using Yao's protocol in the LAN setting, sending the corresponding garbled circuit would take more than one minute, assuming perfect bandwidth utilization. Thus, in comparison to all existing Boolean circuit compilers for MPC, i.e., single protocol compilers, HyCC achieves a runtime that is more than one order of magnitude faster.

UNSUPERVISED MACHINE LEARNING – K-MEANS. Clustering is another data mining task, frequently used to identify centroids in unstructured data. We evaluate a textbook k-means algorithm that detects $c = 4$ clusters in 2-dimensional data sets of size $n = 500$ using $i = 8$ iterations and show our results in Table 18. Also in this use case, a hybrid A+Y protocol achieves the best runtime.

Gaussian Elimination

We implement a textbook Gauss solver with partial pivoting for $n \in \{10, 16\}$ equations using a fixed point number representation and present results in Table 19. In all scenarios, HyCC identifies A+Y as the most efficient protocol, where Y is mainly used to compute the row permutations and divisions. Note that due to the significant circuit depth, we did not measure the runtime for Boolean circuits evaluated with the GMW protocol in the WAN setting.

Database Analytics

Performing data analytics on sensitive data has numerous applications and therefore many privacy-preserving protocols and use cases have been studied, e.g., [Bog+15; DHC04]. We study exemplary use cases, where each party provides a database (array) of size n_A and n_B that has two columns each, which are concatenated (merged), leading to a database of size $n = n_A + n_B$, or joined (inner join on one attribute), yielding a database of maximum size $n = n_A \cdot n_B$, and then the mean and variance of one column of the combined database are computed. The performance evaluation is shown in Table 21. We observe that in both use cases, a combination of A+Y achieves minimal runtime in the LAN setting, with the division (and join) being performed in Y. In the WAN setting, Y achieves optimal runtime and minimal online communication.

SUMMARY OF EXPERIMENTS. Summarizing the results obtained in all use cases, we observe that hybrid protocols consisting of *A+Y*, achieve a very efficient runtime in the LAN deployment, whereas Y is often the fastest protocol in the WAN deployment. We observe that the GMW protocol (B) has barely been identified to achieve optimal runtime for any of the benchmark applications. This is because we performed all benchmarks in the function dependent preprocessing model, which is the default setting in ABY, and which allows to garble the circuit in the setup phase. When using a function independent cost model for preprocessing, HyCC identifies *A+B* as the fastest protocol combination in the LAN setting for many applications.

CONCLUSIONS AND FUTURE WORK. In our evaluation we observed that hybrid protocols can significantly outperform standalone protocols. HyCC is capable of automatically synthesizing the required hybrid circuits from a high-level description and selecting them for a given deployment scenario. As such, HyCC is even capable of outperforming certain hand-optimized protocols. Thus, we conclude that HyCC makes hybrid MPC more practical and also accessible to developers without expert-knowledge in MPC.

In future work, we will extend HyCC with floating point operations and integrate more MPC protocols with different cost models. A natural candidate for extension is homomorphic encryption, similar to TASTY [Hen+10]. Another possibility would be integrating trusted hardware environments such as Intel's SGX.

7.5 RELATED WORK

To the best of our knowledge, only a few MPC frameworks and compilers have been presented that support hybrid MPC protocols. TASTY [Hen+10] and L1 [SKM11] combine Yao's garbled circuits with additively homomorphic encryption. Both have compiler support, where the programmer has to manually select the respective protocol per operation. The Sharemind framework [BLR14; BLW08] has been extended to support multiple MPC protocols and provides a compiler for these with a focus on linear secret sharing. Moreover, the user has to manually select the protocol type for operations in Sharemind. The ABY framework [DSZ15] provides state-of-the-art implementations of Yao's Garbled Circuits, the GMW protocol, and additive linear sharing for Arithmetic circuits, as well as efficient conversions between these three protocols (see Section 2.2) in the 2-party setting. ABY^3 [MR18] is a novel framework for hybrid secure 3-party computation with a honest majority. The circuits generated from HyCC can directly be used by ABY and ABY^3. Very recently, Chandran et al. [Cha+17] proposed a solution for hybrid compilation of MPC protocols called EzPC. However, while the main motivation

is similar, our results differ in several key points. In EzPC, a developer needs to invest much more work to *manually* split the input program into suitable modules and needs to *manually* resolve private array accesses into multiplexer-like structures, which hardly goes beyond what's already possible using the underlying ABY framework. Furthermore, EzPC does not apply circuit optimization methods and cannot cope with depth-minimized Boolean circuits, as required for the GMW protocol, which is beneficial for Boolean operations in low-latency networks.

optimize(Modules M, T_{opt}, T_{min}, λ_{init}, λ_m)

1 : **foreach** m **in** M **do**
2 : $\quad t \leftarrow T_{opt} \cdot \lambda_{init} \cdot \frac{\text{size}(m) \cdot \text{occurrences}(m)}{\text{size}(M)}$
3 : $\quad m' \leftarrow$ locally_optimize(m, t)
4 : $\quad M \leftarrow$ update_modules(M, m')
5 : $\quad T_{opt} \leftarrow T_{opt} -$ duration()
6 : **endfor**
7 : $Q \leftarrow$ init_queue(M)
8 : **while** $T_{opt} > 0$ **do**
9 : $\quad m \leftarrow Q$.pop()
10 : $\quad t \leftarrow \min(T_{opt} \cdot \lambda_m, T_{min})$
11 : $\quad$ **if** has_constant_inputs(m) **then**
12 : $\quad\quad m' \leftarrow$ propagate_constants(m, t)
13 : $\quad$ **else**
14 : $\quad\quad m' \leftarrow$ locally_optimize(m, t)
15 : $\quad$ **endif**
16 : $\quad M \leftarrow$ update_modules(M, m')
17 : $\quad Q \leftarrow$ update_queue(Q, m')
18 : $\quad T_{opt} \leftarrow T_{opt} -$ duration()
19 : **endwhile**
20 : **return** M

Algorithm 1: Optimization heuristic. The functions receives all modules M, an optimization time limit T_{opt}, a minimum constant time T_{min}, which is necessary to perform a complete run of a local optimization, and two factors λ_{init} and λ_m describing the fraction of time to be used for the initial and all later optimization runs. Function locally_optimize() performs the fixed point optimization routine described in Section 4.4, whereas function propagate_contants() only propagates constants Boolean or arithmetic values. Function update_queue(m) adds m to the queue and updates all other elements about possible constant inputs and than reorders the elements accordingly.

```
protocol_selection(DAG G, instruction list IL)
1 :  WS ← G.inputs
2 :  foreach c in share_combinations(WS) do
3 :    cost_table[c] ← cost_input_sharing(c)
4 :  endfor
5 :  foreach m in G.modules ordered by IL do
6 :    WS′ ← remove_completed(WS ∪ m)
7 :    foreach c′ ∈ share_combination(WS′) do
8 :      cost_table′[c′] ← eval_costs(WS, WS′, c′, cost_table)
9 :    endfor
10 :   WS ← WS′
11 :   cost_table ← cost_table′
12 :  endfor
13 :  return min(cost_table)
```

Algorithm 2: Algorithm for optimal protocol selection. The algorithm takes as input the DAG of the program with the circuit descriptions of all modules. It returns the protocol cost for the optimal protocol selection.

```
eval_costs(WS, WS′, c′, cost_table)
1 :  min ← ∞
2 :  foreach c ∈ share_combinations(WS)
3 :    cost ← cost_step(WS, WS′, c, c′)
4 :    if cost + cost_table[c] < min then
5 :      min ← cost + cost_table[c]
6 :    endif
7 :  return min
8 :  endfor
```

Algorithm 3: Algorithm to compute the cheapest evaluation cost to compute the next WS' in a specific protocol configuration. The algorithm takes as input the two working sets WS,WS', the designated protocol configuration for WS', denoted as c', as well as a table with the cheapest cost to compute all possible protocol configurations c of WS.

Circuit	Sharing	Non-linear Gates	Interaction Rounds	Setup Phase LAN [ms]	Setup Phase WAN [ms]	Setup Phase Comm. [MiB]	Online Phase LAN [ms]	Online Phase WAN [ms]	Online Phase Comm. [KiB]
min. Euclid ABY [DSZ15] ($n = 1{,}000$)	Y+A	98,936	6	167	2,878	8	**55**	**557**	**1,567**
min. Euclid HyCC ($n = 1{,}000$)	Y+A	98,936	10	175	1,920	8	70	584	1,582
min. Euclid ABY [DSZ15] ($n = 1{,}000$)	B+A	155,879	78	151	2,206	9	73	3,971	1,620
min. Euclid HyCC ($n = 1{,}000$)	B+A	155,879	80	190	3,622	10	131	4,249	1,643
min. Euclid HyCC ($n = 1{,}000$)	Y	3,166,936	3	1,498	10,239	99	1,177	1,789	4,016
min. Euclid HyCC ($n = 1{,}000$)	B	3,497,879	93	550	8,228	107	2,932	7,974	1,725
min. Euclid ABY [DSZ15] ($n = 4{,}096$)	Y+A	405,440	6	420	7,336	34	**211**	**1,234**	**6,416**
min. Euclid HyCC ($n = 4{,}096$)	Y+A	405,440	10	536	5,162	34	330	1,406	6,480
min. Euclid ABY [DSZ15] ($n = 4{,}096$)	B+A	638,855	92	417	8,016	37	303	5,606	6,629
min. Euclid HyCC ($n = 4{,}096$)	B+A	635,020	94	555	4,337	41	689	5,802	6,722
min. Euclid HyCC ($n = 16{,}384$)	Y+A	1,621,952	10	2,239	13,522	112	1,419	4,041	25,920
min. Euclid HyCC ($n = 16{,}384$)	B+A	2,540,935	108	2,286	15,179	164	3,155	11,024	26,883

Table 17: Minimum Euclidean distance benchmarks comparing a hand-built circuit (ABY [DSZ15]) with a compilation from HyCC (best values marked in bold).

Circuit	Sharing	Non-linear Gates	Interaction Rounds	Setup Phase LAN [ms]	Setup Phase WAN [ms]	Setup Phase Comm. [MiB]	Online Phase LAN [ms]	Online Phase WAN [ms]	Online Phase Comm. [KiB]
MiniONN MNIST [Liu+17]	—	—	—	3,580	—	21	5,740	—	651,877
MiniONN MNIST HyCC	B+A	2,275,880	90	1,750	14,469	165	2,689	9,443	35,864
MiniONN MNIST HyCC	Y+A	1,838,120	34	1,825	14,041	150	**1,621**	**5,882**	**35,094**
CryptoNets Square [Gil+16]	—	—	—	0	—	0	297,500	—	381,133
CryptoNets Square HyCC	A	107,570	7	683	10,348	131	**134**	1,359	**2,018**
CryptoNets RELU HyCC	Y+A	195,455	19	784	11,238	134	163	1,297	3,330
CryptoNets RELU HyCC	B+A	195,455	33	735	11,298	134	187	1,917	3,360
CryptoNets Fix-Point HyCC	B+A	195,455	33	765	11,416	134	187	1,910	3,694
CryptoNets Fix-Point HyCC	Y+A	195,455	19	780	11,264	134	162	**1,296**	3,330
k-means HyCC ($n = 500$)	B+A	7,894,592	6,578	3,453	21,887	293	5,917	337,083	**30,473**
k-means HyCC ($n = 500$)	Y+A	4,991,816	125	4,414	21,007	206	**3,748**	**10,503**	38,915

Table 18: Machine Learning Benchmarks comparing with MiniONN [Liu+17] and CryptoNets [Gil+16] (best values marked in bold).

Circuit	Sharing	Non-linear Gates	Interaction Rounds	Setup Phase LAN [ms]	Setup Phase WAN [ms]	Setup Phase Comm. [MiB]	Online Phase LAN [ms]	Online Phase WAN [ms]	Online Phase Comm. [KiB]
Gauss 10×10 HyCC	B+A	555,611	41,305	340	—	29	5,843	—	2,989
Gauss 10×10 HyCC	B	1,158,995	41,829	268	—	23	6,020	—	1,412
Gauss 10×10 HyCC	Y+A	494,215	147	348	2,849	17	256	4,235	1,997
Gauss 10×10 HyCC	Y	1,030,225	3	561	3,850	31	429	631	101
Gauss 16×16 HyCC	B+A	2,516,310	67,920	1,245	—	57	11,182	—	10,031
Gauss 16×16 HyCC	Y+A	2,294,615	243	1,515	8,842	79	1,258	8,126	7,740
Gauss 16×16 HyCC	Y	4,393,173	3	2,445	13,749	134	1,957	2,190	257

Table 19: Gaussian Elimination Benchmarks (best values marked in bold).

Circuit	Sharing	Non-linear Gates	Interaction Rounds	Setup Phase LAN [ms]	Setup Phase WAN [ms]	Setup Phase Comm. [MiB]	Online Phase LAN [ms]	Online Phase WAN [ms]	Online Phase Comm. [KiB]
k-means HyCC ($n = 500$)	B+A	7,894,592	6,578	3,453	21,887	293	5,917	337,083	30,473
k-means HyCC ($n = 500$)	Y+A	4,991,816	125	4,414	21,007	206	3,748	10,503	38,915

Table 20: k-means Clustering Benchmarks (best values marked in bold).

Circuit	Sharing	Non-linear Gates	Interaction Rounds	Setup Phase LAN [ms]	Setup Phase WAN [ms]	Setup Phase Comm. [MiB]	Online Phase LAN [ms]	Online Phase WAN [ms]	Online Phase Comm. [KiB]
DB Merge $500 + 500$ HyCC	B	1,441,732	1,237	593	3,776	44	1,310	63,430	733
DB Merge $500 + 500$ HyCC	B+A	5,395	1,187	29	927	2	144	59,319	56
DB Merge $500 + 500$ HyCC	Y	849,711	3	858	3,619	26	679	886	752
DB Merge $500 + 500$ HyCC	Y+A	4,990	17	22	815	1	**4**	**606**	**30**
DB Join 50×50 HyCC	B	4,429,046	765	1,645	13,312	135	4,219	43,090	2,179
DB Join 50×50 HyCC	B+A	529,526	708	451	5,201	26	564	36,652	6,827
DB Join 50×50 HyCC	Y	2,550,076	3	1,725	8,317	78	1,272	**1,451**	**100**
DB Join 50×50 HyCC	Y+A	443,900	32	472	3,433	23	**435**	2,395	6,705
DB Join 25×200 HyCC	B	8,981,870	767	3,521	26,766	274	9,846	48,937	4,403
DB Join 25×200 HyCC	B+A	1,163,575	708	832	7,085	54	1,202	38,155	13,295
DB Join 25×200 HyCC	Y	5,158,825	3	3,212	15,960	158	2,660	**2,861**	**250**
DB Join 25×200 HyCC	Y+A	937,049	32	927	5,837	47	**942**	3,603	12,861

Table 21: Database Operation Benchmarks (best values marked in bold).

8

COMPILATION FOR RAM-BASED SECURE COMPUTATION

Summary: MPC protocols are powerful privacy enhancing technologies. Yet, their scalability is limited for data intensive applications due to the circuit computation model. Therefore, RAM based secure computation (RAM-SC) has been proposed, which combines MPC with Oblivious RAM (ORAM). Unfortunately, realizing efficient RAM-SC applications by hand is a tedious and error-prone task, which requires expert knowledge in ORAM protocols and circuit design. To make things worse, a multitude of ORAMs with different trade-offs has been proposed. To overcome this entry barrier to RAM-SC, we present a two-fold approach. First, we explore all cost dimensions of relevant ORAMs in various deployment scenarios. Second, we present a fully automatized compilation approach from ANSI C to RAM-SC by extending CBMC-GC. The presented approach analyzes the input source code and extracts relevant information about the usage patterns of all arrays in the code. The results of the analysis are then used to predict the runtime of suitable ORAMs and to identify the ORAM that achieves minimal runtime. Thus, for the first time, RAM-SC also becomes accessible to non-domain experts.

Remarks: This chapter is based on our paper – *"Towards Practical RAM-based Secure Computation"*, Niklas Büscher, Alina Weber, and Stefan Katzenbeisser, which appeared in the Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS), Barcelona, Spain, 2018.

8.1 MOTIVATION AND OVERVIEW

In previous chapters, we observed that almost all MPC protocols have in common that they compute functionalities in the circuit computation model. Thus, to compute a function f, the function has to be represented as Boolean or Arithmetic circuit C_f. Unfortunately, every random memory access, i.e., an access to an array with private index, in this model requires a scan of the complete memory, which renders MPC protocols impractical for any data intensive application. To overcome this performance barrier, Gordon et al. [Gor+12] proposed the idea of RAM-SC (introduced in detail in Section 2.3.2), later refined by Liu et al. [Liu+14], which combines MPC with ORAM [GO96]. Thus, RAM-SC partially performs the same MPC computations, yet every RAM access is evaluated (more efficiently) using an ORAM protocol. ORAMs obfuscate each RAM access by producing a sequence of physical accesses that is indistinguishable to a random access pattern.

Many ORAMs using a wide range of constructions have been proposed, e.g., [KLO12; LO13; Shi+11; Ste+13], and recently also new ORAMs optimized for RAM-SC have been presented, e.g, optimized

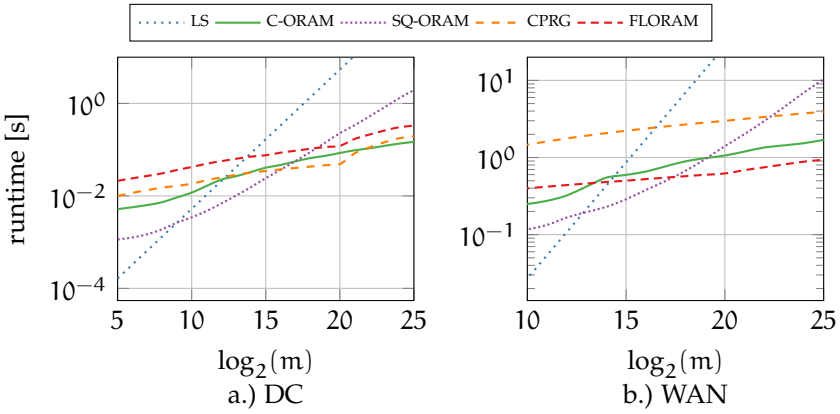


Figure 37: ORAM access time in RAM-SC. Runtime in seconds for different ORAMs to perform a $b = 32\,\text{bit}$ write access in RAM-SC for arrays with different numbers of elements $\log_2(m)$. Figure 37a illustrates the results for a high connectivity network, e.g., datacenter setting with 1.03 Gbit bandwidth and 0.5 ms latency. In Figure 37b the results for a WAN setting with 200 Mbit bandwidth and 50 ms latency is shown.

Tree ORAM [Gen+13], SCORAM [Wan+14], Circuit ORAM (C-ORAM) [WCS15], optimized Square-Root ORAM (SQ-ORAM) [Zah+16], and Function-secret-sharing Linear ORAM (FLORAM) [Ds17a]. Though RAM-SC is asymptotically more efficient than MPC, it is almost impossible to identify a suitable ORAM that achieves optimal runtime by hand due to their complex cost models. Yet, the runtimes can differ by multiple orders of magnitude depending on the ORAM choice.

For instance, the array size influences the ORAM choice. Namely, all ORAMs have different ranges of use, as illustrated in Figure 37a, where the (simulated) time to access a block of size 32 bit for ORAMs of different sizes is shown. An access using circuit only techniques, i.e., Linear Scan (LS), is very efficient for arrays of at most $m = 2^9$ elements, then from $m = 2^{11}$ to $m = 2^{15}$ elements, SQ-ORAM is more efficient, yet being outperformed for $m > 2^{15}$ elements by both FLORAM variants.

Hence, for an optimal decision it is necessary to consider the number of elements, the number of accesses, and the size of the accessed elements. Yet, also the distinction between read or write accesses and the distinction between accesses with private index or public index is relevant. In the latter case, the array is accessed under encryption yet the accessed position is leaked. For example, array accesses with publicly known index can be performed at little cost in FLORAM, but have to be performed with costs similar to an access with private index in C-ORAM. Additionally, the RAM initialization pattern within the program influences the RAM-SC runtime. Moreover, each ORAM has its own set of parameters that can be selected during instantiation. While most parameters depend on the security parameter, some

parameters, e.g., the recursion depth and the packing factor, have an impact on the overall performance of the ORAM construction. Also, the environment in which the protocol is executed has to be taken into account, as all ORAMs have different communication and computation patterns. This includes the properties of the network connection (bandwidth and latency), but also the computational power of the executing hardware. The impact of the network environment is illustrated in Figure 37a and Figure 37b, where the difference between a high-connectivity setup and a typical Internet like (WAN) setup is illustrated. We observe noticeable differences in the range of use of all schemes, when changing from the DC to the WAN setting. Concluding, for an optimal ORAM choice it is necessary to consider all aforementioned parameters. Up to know it is a tedious task for a developer to create an efficient RAM-SC program, as this requires an array usage statistic of the input program and in depth knowledge about ORAMs and their deployment costs.

COMPILER FOR RAM-SC. To make RAM-SC accessible for non-domain experts, we present an automatized framework that analyzes which ORAMs (if at all) should be used to achieve optimal runtime for a RAM-SC program with a given number of array accesses and a deployment scenario. Moreover, by implementing the framework in CBMC-GC, we provide a compile-chain from generic ANSI C code into a RAM-SC program.

In contrast to previous work, such as SCVM [Liu+14] and its successor ObliVM [Liu+15b], which statically decide for or against a single ORAM, our approach is aware of all aforementioned cost dimensions of RAM-SC. Namely, we automatically identify all array accesses (individually for each array in the input code), determine an optimal ORAM choice depending on the access pattern, which includes the optimal selection of ORAM parameters, and automatically partition the code into circuit based computations and ORAM accesses.

For this purpose, we revisit C-ORAM, which is the most efficient tree-based ORAM optimized for MPC, SQ-ORAM, and FLORAM, which both have been developed to outperform C-ORAM for mid-sized arrays, to develop a library with gate-precise costs models. This library allows runtime estimations for arbitrary ORAM sizes, access patterns, and deployment scenarios within seconds, which is multiple orders of magnitude faster than benchmarking all ORAMs in an actual deployment scenario. Using a modular composition, the library can be adapted to future ORAMs with ease and circuit building blocks, e.g., oblivious shuffle, can be replaced once faster constructions are known. Moreover, the library can compute multiple different cost metrics, e.g., to determine which ORAM has minimal communication complexity in a given scenario. As a side-product of our studies, we present practical optimizations for all ORAMs that

reduce the runtime for each access of up to a factor of two. Furthermore, we present the first extensive study of RAM-SC runtimes for different real world deployment scenarios and show that the use of ORAMs over purely circuit based computations is often only useful for arrays larger than one could assume.

CHAPTER OUTLINE. We study the different ORAMs and propose optimizations in Section 8.2, before describing the compiler in Section 8.3. Afterwards, an evaluation of our approach is given in Section 8.4. Finally, we briefly discuss related work in Section 8.5.

8.2 ANALYSIS AND OPTIMIZATION OF ORAMS FOR SECURE COMPUTATION

In order to precisely determine the best suiting ORAM for a RAM-SC application, in this section we revisit the most efficient ORAMs for RAM-SC to establish gate-precise cost models. These models allow the approximation of runtime costs in any RAM-SC deployment, which forms the basis for the optimizing compiler in Section 8.3. Since RAM accesses are basic primitives for any algorithm, we also propose gate-level optimizations for all ORAMs. We begin with a description of implementation pitfalls observed in previous implementations, which can lead to inefficient RAM-SC.

8.2.1 *Pitfalls of ORAM implementations for MPC*

ORAMs are complex cryptographic primitives, and thus substantial engineering effort is necessary to translate them in efficient circuit representations as required for RAM-SC. Consequently, the majority of ORAM implementations in MPC is written in high-level languages for MPC and translated using compilers for MPC. Unfortunately, due to the lacking maturity of tools, compilers, and programming paradigms, a straight-forward high-level implementation does not automatically translate into an efficient circuit description. Thus, while revising the ORAMs and their implementations we identified the following inefficiencies and provide hints for future implementations.

OVERALLOCATION OF INTERNAL VARIABLES. Some MPC compilers use fixed bit-widths for all program variables. For example, leaf identifiers for any tree based ORAM scheme can be represented as bit strings of $\log(m)$ bits. Consequently, for small to medium numbers of elements m, e.g., $m < 2^{32}$, a fixed integer bit-width of $32\,\mathtt{bit}$, introduces a noticeable overhead in the number of used gates, which also propagates to subsequent (possibly recursive) computa-

tions. Therefore, it is preferable to either use optimizing compilers, such as CBMC-GC [Hol+12] or to adjust the bit-width accordingly.

INSUFFICIENT CONSTANT PROPAGATION. Constants are not always properly identified and propagated by some compilers, especially between multiple functions, which can result in cascading effects of significant circuit size. This especially concerns temporary variables in conditional blocks, which can be expressed by wires without any gate costs, but are often multiplexed with all other variables in the conditional.

DUPLICATED MULTIPLEXER BLOCKS. Conditional blocks are represented by multiplexers on the circuit level. When using `if/else` statements that write the same variable (with different values), some compilers introduce duplicated multiplexer blocks, one for each write. However, both can be merged into a single conditional write, which results in a smaller circuit. We are not aware of compilers explicitly optimizing this use case, yet compilers with logic minimizers that support structural hashing and circuit rewriting partly achieve this goal (cf. Chapter 4).

BOUND CHECKING. The most recent ORAM implementations for MPC [Ds17a; Zah+16] perform an inefficient out-of-bounds check for each array access. To limit the accessible range, indexes are masked using a modulo computation, which requires a significant number of gates. While there is no perfect solution to this problem, as there is no unified error handling approach in MPC, several other and more efficient approaches exist. For example, an MPC compiler that is able to identify that all accesses are within their bounds can be used (if possible), a faster masking scheme can be used, or for some schemes the ORAM's size can be increased to the next power of two without a noticeable loss in runtime.

FIXED PARAMETERS FOR RECURSIVE ORAM STRUCTURES. Most of the existing ORAM schemes use recursive techniques to store the position map. Hence, these ORAM schemes are parameterized to configure the maximum recursion depth or the packing factor, i.e., number of addresses per bucket. These parameters are often hard-coded into the ORAM's algorithm and cannot be changed dynamically to achieve optimal costs. However, it is preferable to select them optimally during compilation. For example, the access to an array with $m = 2^9$ elements using a bit-width of $b = 32$ bit can reduce the access time in C-ORAM by 44%, when using an optimal choice of parameters in a DC environment.

8.2.2 *Circuit Models and Optimized ORAM Construction for MPC*

To determine an optimal ORAM choice for RAM-SC, we develop circuit, computation and communication models for all schemes, which are composed of hand-crafted circuit building blocks, e.g., conditional swap, adder, or shuffle. Using a modular construction of all ORAM schemes allows to adapt to future improved building blocks, to recombine different ORAM schemes (e.g., for the recursive position map), and to evaluate different implementation options.

The developed models are based on the papers and their implementations [Ds17a; WCS15; Zah+16] and precisely consider the number of non-linear gates, the communication complexity (rounds and bandwidth), and auxiliary computation costs, i.e., computations performed outside of secure computation. We use optimal bit-widths for variables and avoid the earlier described pitfalls. As this is purely an engineering task, we do not elaborate on the created models, but rather focus on their optimization. We begin with a study of the trivial circuit solution.

TRIVIAL CIRCUIT SOLUTION. Traditionally, MPC compilers translate a dynamic array access into a linear scan (LS) of the complete memory to hide which position was actually accessed. The most efficient MPC circuit construction for LS read is based on a multiplexer tree that bit-wise encodes the accessed index over the stages of the tree. For write accesses a decoder of $m - 1$ non-linear gates is used to convert the index to a so called One-Hot Code, where each bit of the decoders' output is connected to a multiplexer, which selects either the element to write or the previous data. These circuit constructions are described in detail in Section 4.3. In contrast to ORAM schemes, the elements are not shared between the parties but reside inside the garbled circuit. Hence, while LS has a significant circuit size for a growing numbers of elements, it is very efficient in case of networks with high latencies and smaller array sizes, as accesses can be performed in zero rounds and without any initialization.

We note that LS is also used in other ORAM schemes, e.g., to read the stash, buckets, or one of the recursive layers. In this cases it might be necessary to perform a LS using an equivalence comparator on additional metadata, as the data might be unordered.

C-ORAM. C-ORAM [WCS15] was presented to reduce the costs of the most practical ORAM, i.e., Path ORAM, in RAM-SC. This goal has been realized by optimizing the eviction algorithm for minimal circuit size. C-ORAM is known to achieve almost optimal asymptotic cost, and is thus the best suiting ORAM scheme for larger arrays. Unfortunately, C-ORAM suffers from high initialization costs, as each element has to be initially written in an ordinary ORAM access. Fur-

thermore, C-ORAM is a multi-round protocol, where the number of communication rounds is dominated by the recursive structure of the scheme. Nevertheless, accesses to physical blocks can be performed using the soldering approach, which only requires to transmit the computed public indices. Explained in more detail in Section 2.3.2, soldering functions by continuing the circuit garbling and evaluation by using wire the labels stored at a position that is revealed to parties.

The most recent implementation of C-ORAM [Ds17a] that we are aware of has been implemented with OblivC, which neither optimizes the bit-width of internal variables nor thoroughly eliminates unnecessary multiplexer blocks. This has a significant impact on the number of gates required for the eviction algorithm, where for example variables with bit-width $\log(\log(m)+1)$ are sufficient to represent the tree height. Additionally, an inefficient implementation of LS is used. Furthermore, the ReadAndRemove() operation used in all tree ORAMs to read and move a path into the stash, can be optimized such that only necessary metadata and isDummy flag is accessed, rather than all metadata. The existing implementation uses a large multiplexer construction over the whole data block, which can thus be partly omitted. This is because the leaf identifier and virtual index are known beforehand and will be overwritten with a new random value anyways when the block is written back to the stash.

SQ-ORAM. Optimized SQ-ORAM [Zah+16] has been proposed to outperform C-ORAM for moderate array sizes, albeit being asymptotically less efficient. For small numbers of elements the circuit complexity is (surprisingly) small, as the major costs stem from the scan of the stash, i.e., the temporary cache, whose publicly known size is of at most $\sqrt{m}$. SQ-ORAM has a substantially more efficient initialization phase in comparison to C-ORAM. Physical blocks are efficiently accessed using the soldering approach. However, similar to C-ORAM, the number of communication rounds depends on the number of recursive position maps, which is in $\log_c(m)$ with c being the packing factor.

The implementation of Square-Root ORAM in Obliv-C was done by the original authors of the paper, is highly optimized, and is, to the best of our knowledge, the most efficient implementation of this scheme. For their construction the same low-level optimizations as described for C-ORAM can be applied, while the LS is already using the most efficient version.

FLORAM. FLORAM is the most recent ORAM scheme for RAM-SC. Based on PIR techniques, $O(m)$ server computations are required per access, however, these are performed outside secure computation and lead to very low communication complexity. For the generation of the FSS, the FLORAM algorithm requires $2 \cdot \log_2(m)$ AES encryptions

that have to be computed inside a circuit, which consists of ≈ 5000 non-linear gates each. Being a constant round protocol, FLORAM has a huge advantage over the other ORAMs in high latency settings. Furthermore, in contrast to other ORAMs, it is possible to efficiently perform semi-private accesses with little costs, as the physical addresses of the elements correspond to the virtual addresses used. The implementation of FLORAM uses inefficient modulo operations to compute the element position inside its 128 bit data blocks, which requires additional 6000 non-linear gates upon each access. This checks can be omitted, when using a packing factor c that is a power of two, which is the case when using standard data types.

FCPRG. The CPRG optimization for FLORAM was proposed to remove the expensive computation of the many AES encryptions within MPC, so that both parties are able to compute the encryptions locally and only input their results into the secure computation for each stage of the FSS tree. Hence, it introduces a trade-off by reducing the computational effort within the MPC protocol, yet turns the constant round protocol into a multi-round protocol with $O(\log_2(m))$ rounds. The implementation of the FCPRG scheme can be optimized in the same manner as the original FLORAM.

OPTIMAL PARAMETER SELECTION FOR RECURSIVE ORAMS. Most ORAM schemes come with a set of parameters that can be selected for every instantiation. For example, while maintaining the same level of security, larger buckets in tree based ORAMs allow to use a smaller stash [Wan+14], which influences the resulting circuit complexity and thus RAM-SC runtime. Therefore, for an optimal ORAM instantiation in RAM-SC it is desirable to identify optimal parameters. These parameters, i.e., bucket size, stash size, number of levels in ORAMs with recursive position maps, and the eviction strategy are (often) constrained by the desired security level, as well as the failure probability (overflow of the stash). Fortunately, for most ORAM schemes, safe parameter ranges for different security configurations have been proposed [Ste+13; WCS15; Wan+14]. Within these ranges, we solve the combinatorial optimization problem by exhaustive search over the parameter space, which can be performed in seconds for all schemes.

Although we only described optimizations that lead to constant improvements, in Section 8.4.2 we observe gate reductions of practical relevance, namely up to 70.7% for C-ORAM, 17.9% for SQ-ORAM, and up to 35.6% for FCPRG, when implementing the above-mentioned optimizations.

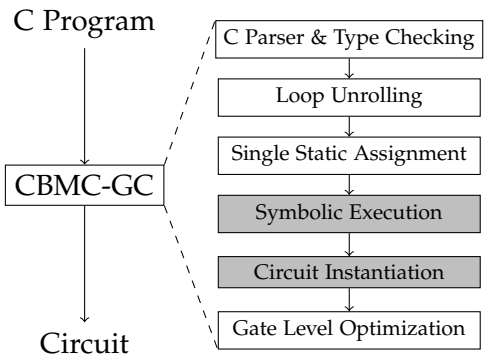


Figure 38: The compilation chain of CBMC-GC with modifications marked in gray to compile RAM-SC programs.

8.3 AUTOMATIZED RAM-SC

In order to facilitate the broad usage of RAM-SC, we present an automatized compilation approach from ANSI C to RAM-SC that is able to detect dynamic memory accesses in a high-level input language and that places the corresponding arrays into ORAMs without the need of any interaction, e.g., by annotations, from the programmer.

To achieve this goal, we follow a two-step approach. First, an input code analysis and transformation is performed that identifies arrays and enumerates array usage statistics. Second, an optimizer is invoked that identifies a suitable scheme for each array in the input code for a selected runtime environment, using the analysis result of the first step, as well as the cost models developed in Section 8.2.

8.3.1 *Input Code Analysis and Transformation*

To transform an input source code into a RAM-SC program, a naïve compilation can be performed by iterating over the abstract syntax tree of the input source code and by translating each array and access into an equivalent RAM access. However, this approach leads to very inefficient RAM-SC programs, as not every access requires full memory trace-obliviousness. For example, arrays can also be accessed purely with public indexes or with a mix of public and private indices. Moreover, the number of accesses, as well as the initialization of the array, play an important role for the performance of RAM-SC (cf. Section 8.4.1). Also the order of accesses is of relevance, e.g., in the case of semi-private accesses, the stash size in FLORAM only depends on the number of writes. Therefore, for an optimized compilation it is important to create precise array usage statistics.

We implemented such a more advanced compilation approach for CBMC-GC, which provides the most powerful Symbolic Execution (SE), required for the analysis, of all currently available compilers for MPC. Its powerful constant propagation performed on the source-

code level allows to separate private and semi-private array accesses. Recapping Chapter 3, CBMC-GC unrolls the input program and translates it into a SSA form. This form is then used for a SE, also referred to as Expression Simplication, of the source code. During SE, every expression of the unrolled code is visited and partial evaluation is performed. In this chapter, we extend the SE interface for array accesses to i.) maintain a list of all allocated arrays, ii.) track each access, and iii.) distinguish semi-private and private accesses.

This approach allows to create a detailed usage statistic for each array, which consists of array size m, element bit-width b, an enumeration of all (semi-)private reads and writes, and an initialization pattern. Namely, we distinguish the case that an array is initialized i.) by only one party, ii.) by using only public indices, e.g., by iterating over the array, or iii.) in a random manner purely based on private writes.

To compile a RAM-SC program the existing LS interface, which is CBMC-GC's traditional approach to handle array accesses, is overwritten, such that each array read or write is replaced by input and output wires of the circuit. Using this approach the compiler does not need to be aware of the concept of RAM-SC, as it is only concerned about the computations performed in the circuit model. Consequently, the remaining code is compiled into a circuit using the existing compilation chain of CBMC-GC, which is illustrated in Figure 38. This ensures to profit from all implemented gate-level optimizations. To execute a compiled RAM-SC program, the inputs and outputs have to be connected to ORAM client circuits, which are selected in the second compilation step.

8.3.2 *Optimal ORAM Selection*

Given a detailed array access description, an optimizing compiler should select the ORAM that achieves minimal costs, e.g., provides optimal runtime. For this purpose, our compiler computes a model of all ORAM schemes for a given array description with the help of the ORAM library developed in Section 8.2. Afterwards, the secure ORAM parameter space is identified for a desired security level. Finally, this combinatorial optimization problem is solved by enumerating the complete search space, consisting of all ORAMs and their possible configurations, which is manageable in seconds on commodity hardware. The optimal choice depends on the applied evaluation metric, which currently is either the runtime or the number of transferred bits. Next, we describe how to predict the runtime in RAM-SC, which requires additional input to the compiler, and remark that these ideas can also be transferred to other metrics, e.g., cloud computing costs (cf. [Pat+16]) with little engineering effort.

RUNTIME ESTIMATION. Using the library developed in the previous section, the runtime of all RAM accesses within a RAM-SC program can be estimated efficiently for a computing environment specified by the developer. Namely, taking the type of array usage description and the security parameter κ into account, the library returns a gate count, the number of communication rounds, the number of OTs, and additional local costs, e.g., such as the FSS evaluation for FLORAM. The environment is described by three parameters, i.e., the computational power (as the non-linear gate throughput, the number of OTs that can be performed per second, and the time to evaluate a FSS scheme), the available bandwidth, and the round trip time.

For runtime approximation we assume that the computing time is linear in the number of non-linear gates and the number of OTs, which is a reasonable assumption as in practice both depend on the throughput of the AES-NI hardware extension. Thus, assuming perfect resource allocation and parallel generation of garbled tables and their transmission (known as streaming), the runtime is estimated as the sum of the time until the last gate has been evaluated (assuming a constant garbling throughput) by the circuit evaluator, the time to perform OTs with OT Extension (assuming a constant OT throughput), and number of communication rounds times the latency. The runtime for the circuit initialization can be estimated in a similar manner.

Although simplifying the RAM-SC computation, we only observed moderate deviations in a lab setting (a detailed experimental study is given in [Web17, 49 ff.]) that are decreasing with increasing RAM size, when comparing to experimentally measured runtimes. These deviations are especially acceptable as only the relation between different ORAM schemes is of major relevance.

OPTIMIZING MULTIDIMENSIONAL ARRAYS. Multidimensional arrays can be represented in a single or in multiple (hierarchical structured) ORAMs, where one ORAM scheme is used per dimension. The latter can be more efficient, if one dimension is predominately accessed using static indexes. Therefore, our compiler studies both cases, i.e., using multiple or a singular ORAM separately to identify the optimal choice.

8.4 EXPERIMENTAL EVALUATION

We give a threefold evaluation of our approach for automatized RAM-SC. First, we evaluate the parameter space that influences the choice for a suitable ORAM when implementing a RAM-SC application. Second, we study the circuit optimizations presented in Section 8.2.2. Finally, we illustrate the compilation approach introduced in Section 8.3 for an exemplary use case.

EXPERIMENTAL SETUP. Our evaluation is based on the runtime estimation, described in the previous section. Assuming a state of the art implementation of Yao's protocol, cf. Section 2.2.2, and a commodity CPU, at least 10 million (M) non-linear gates can be garbled per second per core (fixed-key garbling [Bel+13]), where two wire labels per non-linear gate have to be transmitted (cf. two halve gates [ZRE15]). We use a security level of $\kappa = 80$ bit. Thus, each label has length $\kappa_{gc} = 80$ bit. The computation of XOR is assumed to be for free (free-XOR [KS08]). Similarly, we assume an efficient OT Extension implementation with a throughput of 10 millions (correlated) OTs per seconds [Ash+13]. Two values with length $\kappa_{ot} = 80$ bit have to be transmitted per OT. We remark that in practice, the computation throughput could be experimentally determined in the executing environment for better accuracy, yet also observe that these (conservative) estimates, easily exceed the capacity of a 1 Gbit link. The time to compute Base OTs is left of out scope, as these only need to be computed once and have practically negligible costs for any larger RAM-SC application. The computational effort for the local computations in FLORAM are taken from the evaluation and implementation described in [Ds17a], assuming a parallelization onto four cores.

We investigate three exemplary network settings. First, for comparison purposes with [Zah+16] we use a data center (DC) setting, a scenario with 1.03 Gbit connectivity a low latency 0.5 ms. Second, a local area network (LAN) scenario, typical for the internal network of a larger company, with a 1 Gbit bandwidth and 5 ms latency is studied. Finally, we study a wide area network (WAN) setting as it can be found in nowadays Internet, i.e., servers located on different continents with 200 Mbit bandwidth and 50 ms latency.

8.4.1 *RAM-SC Parameter Dimensions*

We give a quantitative evaluation of the different parameter dimensions of ORAM schemes. The results of this analysis are given in Figure 39 and Figure 40, where the average ORAM access runtime or the initialization costs is shown for different network settings, block sizes b, and number of accesses n.

NETWORK SETTINGS. In the first column of Figure 39, the runtime to perform a typical integer access with $b = 32$ bit for different ORAM sizes m is shown in the three different network settings without considering initialization costs. We observe that for latencies above or equal to 5 ms (LAN), LS is superior to all other schemes for ORAM sizes of up to $m \approx 2^{12}$ elements, afterwards, FLORAM becomes more efficient. The efficiency of LS and FLORAM stems from the fact that they are constant (or zero) round protocols, whereas the other recursive schemes are multi-round protocols. SQ-ORAM out-

performs the other schemes for mid-sized RAM sizes, yet its advantages decreases with increasing latency.

BLOCK SIZE. The runtime of a single ORAM access without considering initialization costs for three different block sizes, namely $b = 64, 128, 1024$ bit, in the DC and WAN setting is shown in the first and second column of Figure 40. In general we observe that the optimal range of all ORAM schemes shifts towards smaller RAM sizes with only marginal changes in their relation to each other. Moreover, with increasing block sizes LS becomes more inefficient, because all blocks are scanned to the full extent for every access.

NUMBER OF ACCESSES AND INITIALIZATION AMORTIZATION. The ORAM schemes have different initialization costs, which have not been considered in the previous analyses. Shown in the second column of Figure 39 is the total time to initialize a RAM with m values and to perform n accesses afterwards in the DC setting. We note that the total runtime is given in seconds for Figure 39i and Figure 39ii and days for Figure 39iii. We observe that LS and FLORAM have none or negligible initialization costs, whereas SQ-ORAM and C-ORAM require a certain number of accesses to amortize their asymptotic costs. In Figure 39i and Figure 39ii, the amortization of SQ-ORAM's initialization costs is shown, which is achieved with $n \ll m$ accesses. Whereas C-ORAM requires almost $n \approx m$ accesses to amortize its initialization as shown in Figure 39iii, albeit being around 10 times faster per access than the second best ORAM, i.e., FCPRG, with a total amortization time of 2900 days.

SUMMARY. For small block sizes and elements, LS is the recommendation of choice in any network setting, SQ-ORAM is effective in fast networks and for larger block sizes, yet has a very short range of use that must be carefully studied before deployment. In all other settings, FLORAM is the most promising ORAM. With its constant rounds and the ability to parallelize the server workload, it is significantly less constrained by network resources, which is the most limiting factor for the other ORAMs. In fast networks FCPRG slightly outperforms FLORAM, but also has a comparably high round complexity (logarithmic to the power of two, and not logarithmic to the packing factor c, as SQ-ORAM and C-ORAM). We were unable to identify a scenario where C-ORAM amortizes its high initialization costs with less than one month total runtime to outperform FLORAM or FCPRG.

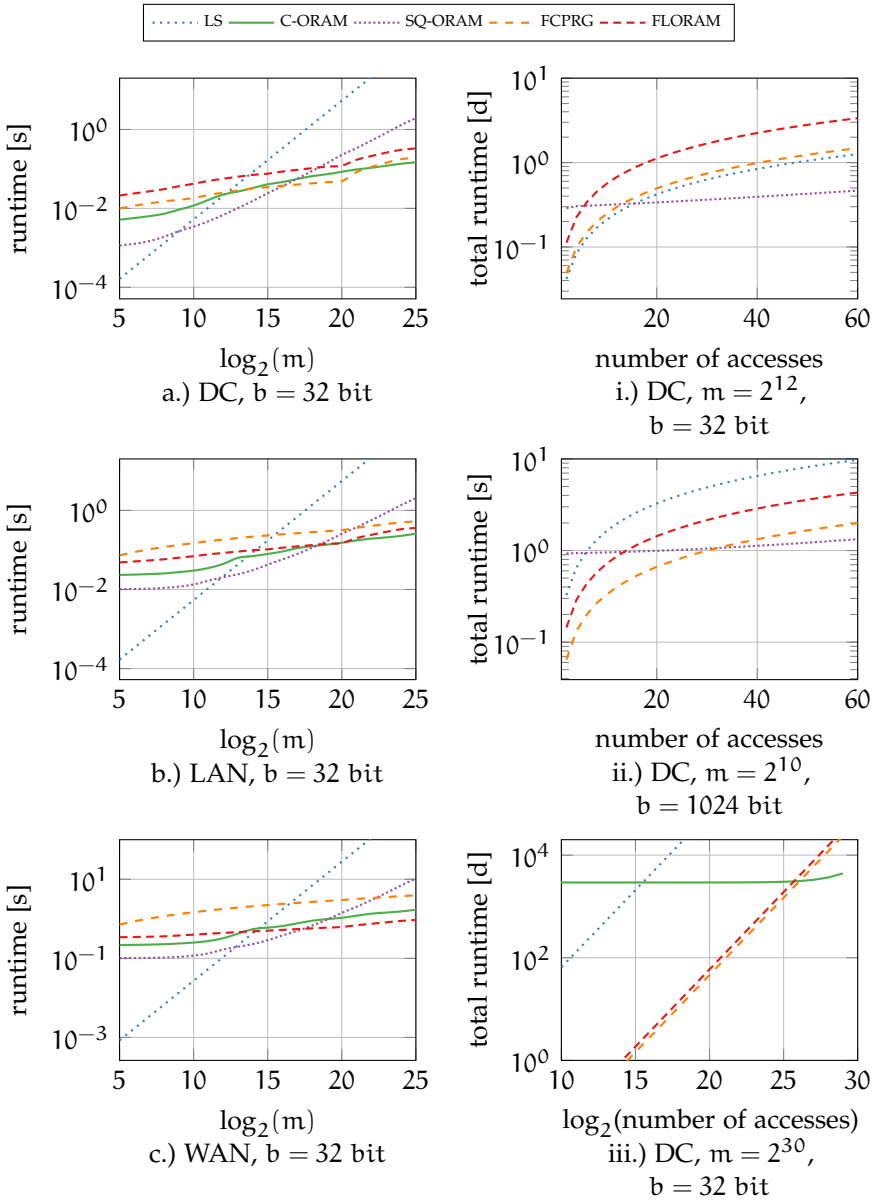


Figure 39: Network settings and initialization costs in RAM-SC. In the first column the average runtime for one RAM-SC access is shown in different network settings and for different sizes m. In the second column the time to perform the first number n of accesses to an ORAM is shown.

8.4.2 *ORAM Optimizations*

We evaluate the ORAM optimizations presented in Section 8.2.2 by comparing the optimized ORAMs with the latest implementation given in [Ds17a] in the number of non-linear gates. The resulting circuit sizes are shown for an exemplary single write access for elements of size $b = 32$ bit and different ORAM sizes m in Figure 41. We observe that the break-even points between different schemes shift

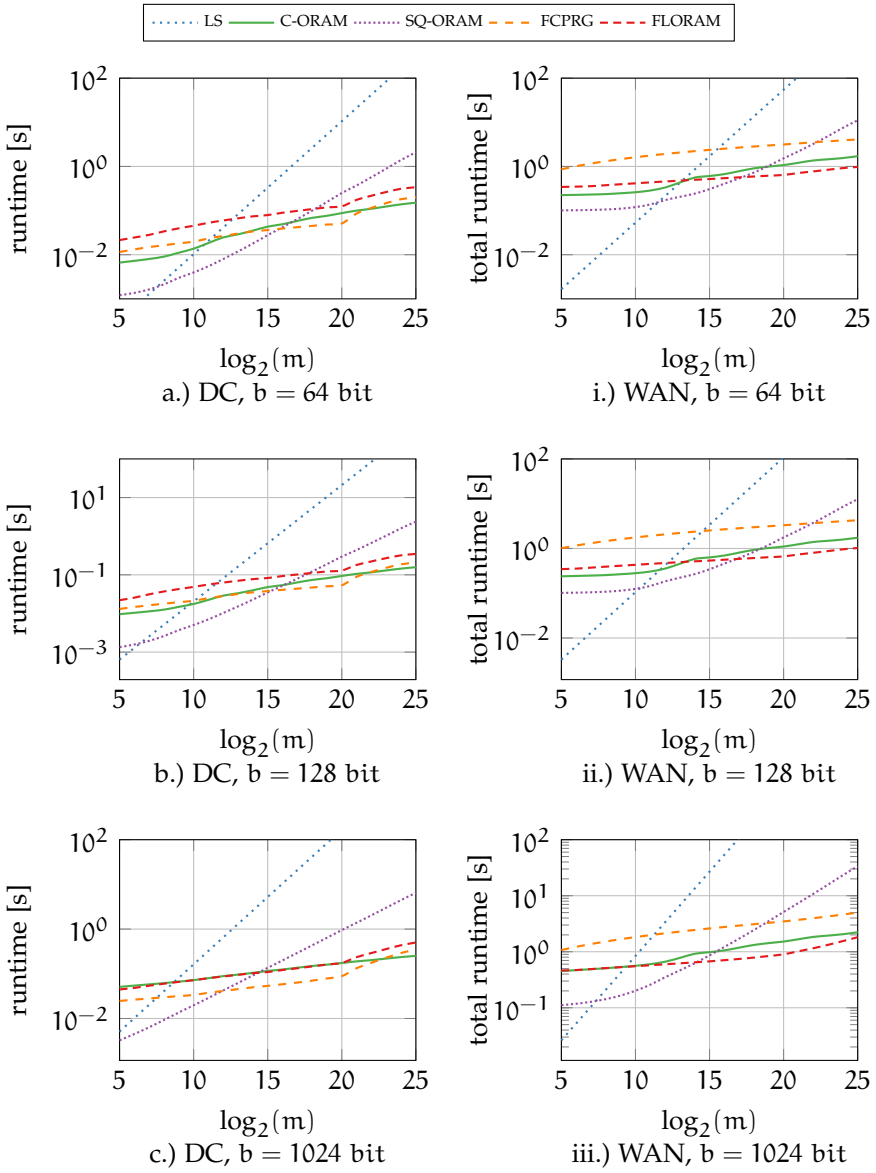


Figure 40: RAM-SC access runtimes for different block sizes. Illustrated is the runtime of a one averaged RAM-SC access in seconds for ORAMs of different size m and different block size b in two network settings.

when comparing both figures. For example, both FLORAM variants outperform LS for a lager number of elements than previously assumed. The improvements of the individual schemes are discussed in the following paragraph.

We observe a difference in form of a factor of two in the number of (non-linear) gates between the optimized LS and the LS based on equality comparators, as it has often been used in the past. This has a noticeable impact on the break-even points with the other ORAM schemes, as LS is more efficient than previously assumed. The difference between the two LS implementations becomes smaller with

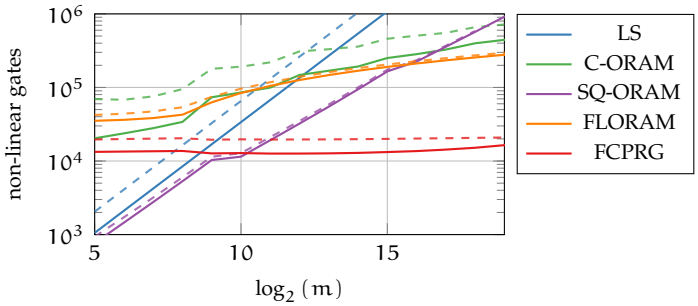


Figure 41: Circuit Optimization. Comparison of the circuit size (in the number of non-linear gates) between the ORAM schemes for RAM-SC in [Ds17a], illustrated with dotted lines, and the optimized circuits described in 8.2.2, illustrated with solid lines, for one write access of bit-width $b = 32$ bit and different array sizes m.

an increasing block size. The circuit size of C-ORAM is reduced by $40\% - 70\%$. Yet, we remark that the difference between the two implementations slightly decreases when increasing m, as all overly allocated resources are decreasingly used. The existing SQ-ORAM implementation is already highly optimized and therefore, only marginal improvements are observed, i.e., for up to $m = 2^{11}$ elements, on average 12.5% non-linear gates are saved. We only observe marginal improvements for FLORAM with savings of up to 20.8% in non-linear gates. This is because the majority of FLORAMs circuit consists of already highly optimized AES circuits. This is not the case in FCPRG, where only two AES circuits are used per access and therefore, an improvement of up to 35.7% of non-linear gates is observed.

8.4.3 *Use Case – Dijkstra Shortest Path Algorithm*

We illustrate our compilation approach for an exemplary use case that has previously been studied in RAM-SC research, namely Dijkstra's single-source shortest path algorithm [Liu+14; Liu+15b]. One party inputs a set of weighted edges between the nodes in the graph, representing the distances, as a two-dimensional array (INPUT_A_e) and the other party inputs the source and destination node, represented by the indices of the respective nodes. The algorithm, given in Listing 17, consists of multiple arrays that are accessed in a semi- and private manner.

In the first step of the compilation, constants are propagated, such that unnecessary array access are removed, e.g., Line 9. Afterwards, the array usage statistic is generated, which is illustrate for $m = 8$ nodes in Table 22. The code uses two one dimensional arrays, namely an array to store visited nodes (vis) and an array to store the shortest path to the source node (dis), as well as the two dimensional input array that stores the distances between nodes (INPUT_A_e). Shown

		private		semi-private	
array	**b**	**init**	**read**	**write**	**read**	**write**
visited (vis)	8	false	0	8	64	8
distance (dis)	32	false	106	0	105	72
inner edges(INPUT_A_l1)	16	true	0	0	64	0
outer edges (INPUT_A_l0)	128	true	8	0	0	0

Table 22: Exemplary array usage statistics for $m = 8$. Statistics gathered by the compiler extension after symbolic execution.

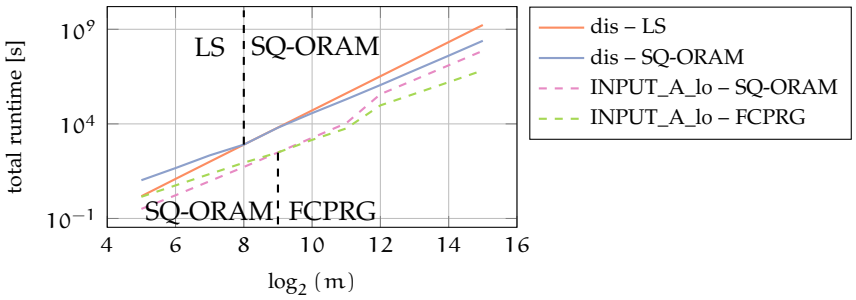


Figure 42: *Total runtime for all accesses to the arrays* `dis` *and* `INPUT_A_l0` *in Dijkstra's algorithm.*

is the analysis result when separating the two dimensions. The inner dimension of the array is always accessed using a public index, whereas the outer dimension is accessed with private indeces only. Moreover, the arrays vis and dis are first written during the algorithm, whereas, the weighted graph is already pre-initialized with values from Party A. In the next compilation step, the statistics are handed to the ORAM library. The runtime estimated by the ORAM library in the DC setting for the two most compute intensive arrays is illustrated in Figure 42 for different graph sizes m. We note that the compiler is only able to compute absolute array usage statistics, yet not parametrized formulas. Therefore, the results are based on multiple compiler runs, one for each size m. Shown is the total runtime in seconds to perform all semi- and private array accesses for the two most efficient ORAM choices for each array. The array dis is best stored as a LS for up to $m = 2^8$ nodes, then SQ-ORAM becomes most efficient. For the INPUT_A_e array, a decomposition in two dimensions l0 and l1 is more efficient than placing it in a single ORAM. For $m \leqslant 2^9$ a SQ-ORAM representation of the outer array dimension INPUT_A_e_l0 is most efficient. Albeit being a small array, the significant block size to store the second layer of the array makes LS inefficient. For $m > 2^9$ nodes, FCPRG becomes most efficient.

We observe, that even for simple algorithms, an automatized approach is highly beneficial, as many factors need to be considered

```
#define M 128

typedef struct {short m[M][M];} Graph;

int main (Graph INPUT_A_e, int INPUT_B_s, int INPUT_B_d) {
  char vis[M]; // inidicates if node has been visitied
  int dis[M]; // current smallest distance from src to dst
  for(int i = 0; i < M; i++) {
    vis[i] = 0; dis[i] = 0;
  }
  vis[INPUT_B_s] = 1;
  for(int i = 0; i < M; i++) {
      dis[i] = INPUT_A_e.m[INPUT_B_s][i];
  }

  for(int i = 0; i < M; i++) {
    int minj = -1;
    for(int j = 0; j < M; j++) {
      if(!vis[j] && (minj < 0 || dis[j] < dis[minj]))
        minj = j
    }
    vis[minj] = 1;
    for(int j = 0; j < M; j++) {
      if(!vis[j] &&
        (dis[minj]+INPUT_A_e.m[minj][j] < dis[j])) {
        dis[j] = dis[minj] + INPUT_A_e.m[minj][j];
      }
    }
  }
  return dis[INPUT_B_d];
}
```

Listing 17: Dijkstra's shortest-path algorithm. Source code is based on [Liu+14] using CBMC-GC's input annotations.

when manually selecting ORAMs. In total we observe runtime of more than an hour for a moderately sized array, e.g., 2^{10}.

8.5 RELATED WORK

The first compiler that combines ORAMs and MPC, named SCVM, has been proposed by Liu et al. [Liu+14]. In follow-up works, Liu et al. presented the ObliVM [Liu+15b] compiler targeting practical RAM-SC, and adapted this compiler for the needs ORAM supported hardware synthesis [Liu+15a]. These compilers translate a domain-specific or annotated language that compiles specially marked arrays into RAM-SC programs using a single ORAM type. Although simplifying the development effort for RAM-SC, the developer is still required to have expert knowledge in ORAMs. The OblivC compiler by Zahur and Evans [ZE15] allows to jointly compile public and private computations, and has therefore been used to implement many

ORAM protocols, e.g., SQ-ORAM and FLORAM. However, it does not primarily target RAM-SC and therefore does not provide any form of automatization for RAM-SC.

Related to the work on RAM-SC, is the work on structured memory accesses in MPC. For example, Zahur and Evans [ZE13] as well as Keller and Scholl [KS14] have studied dedicated data structures, such as oblivious stacks or queues that can outperform the generic ORAM solution for applications with the according access pattern.

9 CONCLUSION AND FUTURE WORK

The growing complexity in information technology has a sincere impact on the security of our systems and hence also on our privacy. We further observe that the complexity of newly proposed protection mechanisms is also continuously increasing. An example are the RAM-SC protocols studied in the previous chapter, which combine MPC protocols with ORAMs. Their description individually fills research papers. Thus, implementing secure applications and privacy-preserving protocols becomes a challenging task, which limits their widespread use.

In principle there are two solutions that allow to handle this level of complexity, namely *abstraction* and *automatization*. We contribute to both in this thesis, while mostly concentrating on the latter. We make use of abstraction in the sense that we follow a strict separation between compilation and protocol framework. This allows to use compilation goals that are based on abstract cost models rather than concrete protocols implementations. On the automatization side, we present a variety of compilation approaches and tools that automatize the task to create PETs based on MPC protocols. By compiling for four different protocol types, i.e., constant- and multi-round MPC protocols over Boolean circuits, hybrid protocols and RAM-SC protocols from the same input language, we introduce a degree of automatization that not only allows to develop applications for different classes of MPC protocols but also enables their rapid prototyping. This prototyping also allows to evaluate whether generic MPC protocols are a sufficient for a desired application or whether dedicated protocols need to be designed. In many cases we are able to outperform previous (hand-made) applications and thus improve the efficiency of applied MPC.

FUTURE WORK

Although we present compilation approaches for a multitude of directions, there are many open ends that require future work and that are outlined in the next paragraphs.

BOOLEAN CIRCUIT OPTIMIZATION. The limits of Boolean circuit minimization in the number of non-linear circuit size or circuit depth in both theory and practice are either unknown or have only recently been studied. For example, it is known that every Boolean function (one output bit) with $l \leqslant 5$ inputs can be computed with

at most $s^{nX} = l - 1$ non-linear gates [TP14]. Although the non-linear complexity of a random Boolean function with l inputs is at least $s^{nX} = 2^{l/2} - O(l)$ with high probability [BPP00], the first concrete function with non-linear complexity larger than $l - 1$ has only been found in 2018 [CTP18]. Upper bounds for the side-depth trade-offs of Boolean circuits have been studied in [BB94], where it has been shown that every circuit with complexity n has an equivalent circuit of depth $O(\log(n))$. Yet to the best of our knowledge, there is little known about the practical implications of this result, i.e., whether efficient conversion algorithms with minimal or moderate size trade-off exist. Moreover, there is mostly preliminary work on dedicated heuristics for minimizing the non-linear complexity [BP10; CAS18]. Extending these theoretic results for practical relevant functions and mapping these into a synthesis framework is a promising task to explore the limits of practical circuit minimization for MPC.

FURTHER COMPILATION TARGETS. We identify four further possible compilation targets, in which our work could be extend:

First, we only focused on the compilation of the private functionality. Yet, an extension towards mixed-mode frameworks that support both public and private computations, e.g., [ZE15; Liu+15b], is of interest to create a wider range of applications. Fur this purpose, safe type systems have already been proposed, e.g., [RHH14], and also optimization techniques for mixed-mode applications have been presented, e.g., [Ker11]. Yet, the combination of mixed-mode computation and advanced circuit optimization has not been investigated so far.

Second, during this thesis a different class of protocols became of relevance, namely protocols operating on look-up tables [DK10; Ish+13; Des+17]. Thus, instead of using gates with two inputs, gates with more inputs, e.g., 8 bit input for the AES S-Box, are used. Compilation for look-up tables is known in hardware synthesis and a first compilation approach using these techniques is given in [Des+17].

Third, another compilation target are protocols with richer set of operations. So far, we focused on the compilation towards elementary circuit operations. Nevertheless, a richer standard library with more advanced protocols, e.g., sorting protocols, could be interesting for the research on automatized algorithmic transformations. For example, the Sharemind [BLW08] MPC framework provides dedicated protocols for a very broad range of functionalities. The automatized use of these dedicated protocols can lead to faster protocol execution.

Finally, in Chapter 8, we observed that RAM-SC is only at the verge of being practical. Even in fast networks, RAM accesses create noticeable costs. As it is impossible to perform a RAM access faster than the latency, the round complexity becomes a sincere bottleneck in any intercontinental deployment scenario. Consequently, parallelization for

RAM-SC [BCP16; Lau15; Nay+15] with compiler support becomes necessary to overcome the performance barrier of multi-round RAM-SC protocols.

INFORMATION LEAKAGE AND DEVELOPER SUPPORT. To the best of our knowledge, no compiler for MPC exists that provides automatized feedback or implementation recommendations for performance or security. For example, unintended information leakage in MPC cannot be identified by traditional compilation methods, yet can be a sincere security problem. Thus, an open research questions is whether possible information leaks could automatically be detected by compilers and reported to the developer. Similarly, it would be interesting to see, if MPC specific inefficiencies could be detected and automatically replaced by more efficient implementations. This would not only allow algorithmic optimizations, but also MPC specific optimizations, e.g., the use of slow floating point operations could be investigated during compile time to evaluate whether fixed-point computations would be a correct and more efficient alternative.

VERIFICATION. With the increasing number of protocols and their complexity, also the compilers for MPC became more complex, cf. Chapter 7. Therefore, verification techniques become of importance. A first step has been made in [Alm+17], where a part of the compilation chain is formally verified. However, the verification of an optimizing circuit compilation chain is still an open problem.

In conclusion and albeit the many open topics, I sincerely hope that this thesis could contribute a tiny piece to the enormous task of making the powerful world of modern cryptography accessible to non-domain experts.

[Afs+15] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. "How to Efficiently Evaluate RAM Programs with Malicious Security." In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Vol. 9056. Lecture Notes in Computer Science. Springer, Heidelberg, 2015, pp. 702–729.

[Alb+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. "Ciphers for MPC and FHE." In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Vol. 9056. Lecture Notes in Computer Science. Springer, Heidelberg, 2015, pp. 430–454.

[Alb+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity." In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Vol. 10031. Lecture Notes in Computer Science. Springer, Heidelberg, 2016, pp. 191–219.

[Alm+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. "A Fast and Verified Software Stack for Secure Function Evaluation." In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. ACM Press, 2017, pp. 1989–2006.

[Ara+17] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. "Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier." In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2017, pp. 843–862.

[Ash+13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. "More efficient oblivious transfer and extensions for faster secure computation." In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. ACM Press, 2013, pp. 535–548.

[Ata+04] Mikhail J. Atallah, Marina Bykova, Jiangtao Li, Keith B. Frikken, and Mercan Topkara. "Private collaborative forecasting and benchmarking." In: *ACM WPES 04: Workshop on Privacy in the Electronic Society*. ACM Press, 2004, pp. 103–114.

[AL07] Yonatan Aumann and Yehuda Lindell. "Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries." In: *TCC 2007: 4th Theory of Cryptography Conference*. Vol. 4392. Lecture Notes in Computer Science. Springer, Heidelberg, 2007, pp. 137–156.

[Bar+13] Mauro Barni, Massimo Bernaschi, Riccardo Lazzeretti, Tommaso Pignata, and Alessandro Sabellico. "Parallel Implementation of GC-Based MPC Protocols in the Semi-Honest Setting." In: *Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM, and 6th International Workshop, SETOP*. Vol. 8247. Lecture Notes in Computer Science. Springer, 2013, pp. 66–82.

[Bea92] Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization." In: *Advances in Cryptology – CRYPTO'91*. Vol. 576. Lecture Notes in Computer Science. Springer, Heidelberg, 1992, pp. 420–432.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. "The Round Complexity of Secure Protocols (Extended Abstract)." In: *22nd Annual ACM Symposium on Theory of Computing*. ACM Press, 1990, pp. 503–513.

[Bea+91] Donald Beaver, Joan Feigenbaum, Joe Kilian, and Phillip Rogaway. "Security with Low Communication Overhead." In: *Advances in Cryptology – CRYPTO'90*. Vol. 537. Lecture Notes in Computer Science. Springer, Heidelberg, 1991, pp. 62–76.

[Bel+13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. "Efficient Garbling from a Fixed-Key Blockcipher." In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2013, pp. 478–492.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)." In: *20th Annual ACM Symposium on Theory of Computing*. ACM Press, 1988, pp. 1–10.

[Ber] *Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 30916*. http://www.eecs.berkeley.edu/~alanmi/abc/.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. "The Security Impact of a New Cryptographic Library." In: *Progress in Cryptology - LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America*. Vol. 7533. Lecture Notes in Computer Science. Springer, Heidelberg, 2012, pp. 159–176.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs." In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*. Springer, 1999, pp. 193–207.

[Bil+11] Igor Bilogrevic, Murtuza Jadliwala, Jean-Pierre Hubaux, Imad Aad, and Valtteri Niemi. "Privacy-preserving activity scheduling on mobile devices." In: *ACM CODASPY 11: First Conference on Data and Application Security and Privacy*. ACM Press, 2011, pp. 261–272.

[BB04] Per Bjesse and Arne Borälv. "DAG-aware circuit compression for formal verification." In: *ICCAD 04: International Conference on Computer-Aided Design*. IEEE Computer Society / ACM, 2004, pp. 42–49.

[BG11] Marina Blanton and Paolo Gasti. "Secure and Efficient Protocols for Iris and Fingerprint Identification." In: *ESORICS 2011: 16th European Symposium on Research in Computer Security*. Vol. 6879. Lecture Notes in Computer Science. Springer, Heidelberg, 2011, pp. 190–209.

[BLR13] Dan Bogdanov, Peeter Laud, and Jaak Randmets. "Domain-polymorphic language for privacy-preserving applications." In: *ACM PETShop 13: Workshop on Language Support for Privacy-Enhancing Technologies*. ACM Press, 2013, pp. 23–26.

[BLR14] Dan Bogdanov, Peeter Laud, and Jaak Randmets. "Domain-Polymorphic Programming of Privacy-Preserving Applications." In: *ACM PLAS@ECOOP 14: Ninth Workshop on Programming Languages and Analysis for Security*. ACM Press, 2014, p. 53.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. "Sharemind: A Framework for Fast Privacy-Preserving Computations." In: *ESORICS 2008: 13th European Symposium on Research in Computer Security*. Vol. 5283. Lecture Notes in Computer Science. Springer, Heidelberg, 2008, pp. 192–206.

[Bog+15] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. "How the Estonian Tax and Customs Board Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation." In: *FC 2015: 19th International Conference on Financial Cryptography and Data Security*. Vol. 8975. Lecture Notes in Computer Science. Springer, Heidelberg, 2015, pp. 227–234.

[Bog+09] Peter Bogetoft et al. "Secure Multiparty Computation Goes Live." In: *FC 2009: 13th International Conference on Financial Cryptography and Data Security*. Vol. 5628. Lecture Notes in Computer Science. Springer, Heidelberg, 2009, pp. 325–343.

[Bon+08] Uday Bondhugula, Albert Hartono, Jagannatan Ramanujam, and Ponnuswamy Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer." In: *ACM PLDI 08: Conference on Programming Language Design and Implementation*. ACM Press, 2008, pp. 101–113.

[BB94] Maria Luisa Bonet and Samuel R. Buss. "Size-Depth Tradeoffs for Boolean Fomulae." In: *Information Processing Letters* 49.3 (1994), pp. 151–155.

[BP10] Joan Boyar and René Peralta. "A New Combinational Logic Minimization Technique with Applications to Cryptology." In: *SEA 10: 9th International Symposium on Experimental Algorithms*. Springer, Heidelberg, 2010, pp. 178–189.

[BP12] Joan Boyar and René Peralta. "A Small Depth-16 Circuit for the AES S-Box." In: *SEC 2012: 27th IFIP TC 11 Information Security and Privacy Conference*. Vol. 376. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 287–298.

[BPP00] Joan Boyar, René Peralta, and Denis Pochuev. "On the multiplicative complexity of Boolean functions over the basis (cap, +, 1)." In: *Theoretical Computer Science* 235.1 (2000), pp. 43–57.

[BCP16] Elette Boyle, Kai-Min Chung, and Rafael Pass. "Oblivious Parallel RAM and Applications." In: *TCC 2016-A: 13th Theory of Cryptography Conference, Part II*. Vol. 9563. Lecture Notes in Computer Science. Springer, Heidelberg, 2016, pp. 175–204.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function Secret Sharing." In: *Advances in Cryptology – EUROCRYPT 2015, Part II*. Vol. 9057. Lecture Notes in Computer Science. Springer, Heidelberg, 2015, pp. 337–367.

[Bra+] Robert K Brayton, Gary D Hachtel, Curt McMullen, and Alberto Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media. ISBN: 089838-164-9.

[BU11] David Buchfuhrer and Christopher Umans. "The complexity of Boolean formula minimization." In: *Journal of Computer and System Sciences* 77.1 (2011), pp. 142–153.

[Buc+16] Johannes A. Buchmann, Niklas Büscher, Florian Göpfert, Stefan Katzenbeisser, Juliane Krämer, Daniele Micciancio, Sander Siim, Christine van Vredendaal, and Michael Walter. "Creating Cryptographic Challenges Using Multi-Party Computation: The LWE Challenge." In: *ACM AsiaPKC@AsiaCCS 16: International Workshop on ASIA Public-Key Cryptography*. ACM Press, 2016, pp. 11–20.

[Büs+16] Niklas Büscher, David Kretzmer, Arnav Jindal, and Stefan Katzenbeisser. "Scalable secure computation from ANSI-C." In: *IEEE WIFS 16: International Workshop on Information Forensics and Security*. IEEE Computer Society Press, 2016, pp. 1–6.

[Büs+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. "HyCC: Compilation of Hybrid Protocols for Practical Secure Computation." In: *ACM CCS 18: 25th Conference on Computer and Communications Security*. ACM Press, 2018, pp. 847–861.

[CTP18] Cagdas Calik, Meltem Sonmez Turan, and Rene Peralta. *The Multiplicative Complexity of 6-variable Boolean Functions*. Cryptology ePrint Archive, Report 2018/002. https://eprint.iacr.org/2018/002. 2018.

[CAS17] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. *A multi-start heuristic for multiplicative depth minimization of boolean circuits*. Cryptology ePrint Archive, Report 2017/483. http://eprint.iacr.org/2017/483. 2017.

[CAS18] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. "A Multi-start Heuristic for Multiplicative Depth Minimization of Boolean Circuits." In: *IWOCA 17: 28th International Workshop On Combinatorial Algorithms*. Vol. 10765. Lecture Notes in Computer Science. Springer, 2018, pp. 275–286.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. "Armadillo: A Compilation Chain for Privacy Preserving Applications." In: *SCC@ASIACCS 15: 3rd International Workshop on Security in Cloud Computing*. ACM Press, 2015, pp. 13–19.

[Cha+17] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. *EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation for Machine Learning*. Cryptology ePrint Archive, Report 2017/1109. https://eprint.iacr.org/2017/1109. 2017.

[Cho+12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. "Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces." In: *Topics in Cryptology – CT-RSA 2012*. Vol. 7178. Lecture Notes in Computer Science. Springer, Heidelberg, 2012, pp. 416–432.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs." In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004*. Springer, 2004, pp. 168–176.

[CKY03] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. "Behavioral consistency of C and verilog programs using bounded model checking." In: *DAC 03: Proceedings of the 40th Design Automation Conference*. ACM, 2003, pp. 368–371.

[CDN15] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015. ISBN: 978-110704-305-3.

[CPS18] Eric Crockett, Chris Peikert, and Chad Sharp. "ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easY." In: *ACM CCS 18: 25th Conference on Computer and Communications Security*. ACM Press, 2018, pp. 1020–1037.

[Cuo+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C - A Software Analysis Perspective." In: *SEFM 2012: International Conference on Software Engineering and Formal Methods*. Springer, Heidelberg, 2012, pp. 233–247.

[DM98] Leonardo Dagum and Ramesh Menon. "OpenMP an industry standard API for shared-memory programming." In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. ISSN: 1070-9924.

[DK10] Ivan Damgård and Marcel Keller. "Secure Multiparty AES." In: *FC 2010: 14th International Conference on Financial Cryptography and Data Security*. Vol. 6052. Lecture Notes in Computer Science. Springer, Heidelberg, 2010, pp. 367–374.

[DN03] Ivan Damgård and Jesper Buus Nielsen. "Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption." In: *Advances in Cryptology – CRYPTO 2003*. Vol. 2729. Lecture Notes in

Computer Science. Springer, Heidelberg, 2003, pp. 247–264.

[Dam+09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. "Asynchronous Multiparty Computation: Theory and Implementation." In: *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*. Vol. 5443. Lecture Notes in Computer Science. Springer, Heidelberg, 2009, pp. 160–179.

[Dam+12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption." In: *Advances in Cryptology – CRYPTO 2012*. Vol. 7417. Lecture Notes in Computer Science. Springer, Heidelberg, 2012, pp. 643–662.

[Dam+16] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. "Confidential Benchmarking Based on Multiparty Computation." In: *FC 2016: 20th International Conference on Financial Cryptography and Data Security*. Vol. 9603. Lecture Notes in Computer Science. Springer, Heidelberg, 2016, pp. 169–187.

[Dar+81] John A Darringer, William H Joyner, C Leonard Berman, and Louise Trevillyan. "Logic Synthesis Through Local Transformations." In: *IBM Journal of Research and Development* 25.4 (1981).

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation." In: *ISOC Network and Distributed System Security Symposium – NDSS 2015*. The Internet Society, 2015.

[Dem+15] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. "Automated Synthesis of Optimized Circuits for Secure Computation." In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. ACM Press, 2015, pp. 1504–1517.

[Des+17] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. "Pushing the Communication Barrier in Secure Computation using Lookup Tables." In: *ISOC Network and Distributed System Security Symposium – NDSS 2017*. The Internet Society, 2017.

[Dob+18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. *Rasta: A cipher with low ANDdepth and few ANDs per bit*. Cryptology ePrint

Archive, Report 2018/181. https://eprint.iacr.org/2018/181. 2018.

[Ds17a] Jack Doerner and abhi shelat. "Scaling ORAM for Secure Computation." In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. ACM Press, 2017, pp. 523–535.

[Ds17b] Jack Doerner and abhi shelat. *Scaling ORAM for Secure Computation*. Cryptology ePrint Archive, Report 2017/827. http://eprint.iacr.org/2017/827. 2017.

[Dor+14] Yarkin Doröz, Aria Shahverdi, Thomas Eisenbarth, and Berk Sunar. "Toward Practical Homomorphic Evaluation of Block Ciphers Using Prince." In: *Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC*. Springer, Berlin, Heidelberg, 2014, pp. 208–220.

[DHC04] Wenliang Du, Yunghsiang S. Han, and Shigang Chen. "Privacy-Preserving Multivariate Statistical Analysis: Linear Regression and Classification." In: *Fourth SIAM International Conference on Data Mining*. SIAM, 2004, pp. 222–233.

[Ear65] JG Earle. "Latched carry-save adder." In: *IBM Technical Disclosure Bulletin* (1965).

[Erk+09] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. "Privacy-Preserving Face Recognition." In: *PETS 09: 9th International Symposium on Privacy Enhancing Technologies*. Springer, Heidelberg, 2009, pp. 235–253.

[Fra+14] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations." In: *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*. Springer, 2014, pp. 244–249.

[FJN14] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. "Faster Maliciously Secure Two-Party Computation Using the GPU." In: *SCN 14: Security and Cryptography for Networks - 9th International Conference*. Springer, Cham, 2014, pp. 358–379.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. "Efficient Private Matching and Set Intersection." In: *Advances in Cryptology – EUROCRYPT 2004*. Vol. 3027. Lecture Notes in Computer Science. Springer, Heidelberg, 2004, pp. 1–19.

[Fur+17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. "High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority." In: *Advances in Cryptology – EUROCRYPT 2017, Part II*. Vol. 10211. Lecture Notes in Computer Science. Springer, Heidelberg, 2017, pp. 225–255.

[Gaj+12] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 2012. ISBN: 079239-194-2.

[Gen+13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. "Optimizing ORAM and Using It Efficiently for Secure Computation." In: *PETS 13: International Symposium on Privacy Enhancing Technologies*. Vol. 7981. Lecture Notes in Computer Science. Springer, 2013, pp. 1–18.

[Gil+16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy." In: *ICML 16: 33nd International Conference on Machine Learning*. JMLR.org, 2016, pp. 201–210.

[Gil99] Niv Gilboa. "Two Party RSA Key Generation." In: *Advances in Cryptology – CRYPTO'99*. Vol. 1666. Lecture Notes in Computer Science. Springer, Heidelberg, 1999, pp. 116–129.

[Gol91] David Goldberg. ""What Every Computer Scientist Should Know About Floating-Point Arithmetic"." In: *ACM Computing Surveys* 23.3 (1991), p. 413.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design (Extended Abstract)." In: *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1986, pp. 174–187.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *19th Annual ACM Symposium on Theory of Computing*. ACM Press, 1987, pp. 218–229.

[GO96] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." In: *Journal of the ACM* 43.3 (1996), pp. 431–473.

[Gor+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. "Secure two-party computation in sublinear (amortized) time." In: *ACM CCS 12: 19th Conference on Computer and Communications Security*. ACM Press, 2012, pp. 513–524.

[Gue+15] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. "Fast Garbling of Circuits Under Standard Assumptions." In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. ACM Press, 2015, pp. 567–578.

[Har03] David Harris. "A taxonomy of parallel prefix networks." In: *Signals, Systems and Computers, Thirty-Seventh IEEE ASILOMAR conference*. IEEE Computer Society Press, 2003.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010. ISBN: 978-3-642-14302-1.

[HS13] Wilko Henecka and Thomas Schneider. "Faster secure two-party computation with less memory." In: *ASIACCS 13: 8th ACM Symposium on Information, Computer and Communications Security*. ACM Press, 2013, pp. 437–446.

[Hen+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. "TASTY: tool for automating secure two-party computations." In: *ACM CCS 10: 17th Conference on Computer and Communications Security*. ACM Press, 2010, pp. 451–462.

[Hol+12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. "Secure two-party computations in ANSI C." In: *ACM CCS 12: 19th Conference on Computer and Communications Security*. ACM Press, 2012, pp. 772–783.

[HKE12] Yan Huang, Jonathan Katz, and David Evans. "Quid-Pro-Quo-tocols: Strengthening Semi-honest Protocols with Dual Execution." In: *2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2012, pp. 272–284.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. "Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose." In: *Advances in Cryptology – CRYPTO 2013, Part II*. Vol. 8043. Lecture Notes in Computer Science. Springer, Heidelberg, 2013, pp. 18–35.

[Hua+11a] Yan Huang, Lior Malka, David Evans, and Jonathan Katz. "Efficient Privacy-Preserving Biometric Identification." In: *ISOC Network and Distributed System Security Symposium – NDSS 2011*. The Internet Society, 2011.

[Hua+11b] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. "Faster Secure Two-Party Computation Using Garbled Circuits." In: *20th USENIX Security Symposium*. USENIX Association, 2011.

[Hua+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. "Amortizing Garbled Circuits." In: *Advances in Cryptology – CRYPTO 2014, Part II*. Vol. 8617. Lecture Notes in Computer Science. Springer, Heidelberg, 2014, pp. 458–475.

[Hus+13] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. "GPU and CPU parallelization of honest-but-curious secure two-party computation." In: *ACSAC 13: Annual Computer Security Applications Conference*. ACM Press, 2013, pp. 169–178.

[IR89] Russell Impagliazzo and Steven Rudich. "Limits on the Provable Consequences of One-Way Permutations." In: *21st Annual ACM Symposium on Theory of Computing*. ACM Press, 1989, pp. 44–61.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. "Semantical interprocedural parallelization: an overview of the PIPS project." In: *International Conference on Supercomputing (ICS'91)*. ACM, 1991, pp. 244–251.

[Ish+03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. "Extending Oblivious Transfers Efficiently." In: *Advances in Cryptology – CRYPTO 2003*. Vol. 2729. Lecture Notes in Computer Science. Springer, Heidelberg, 2003, pp. 145–161.

[Ish+13] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. "On the Power of Correlated Randomness in Secure Computation." In: *TCC 2013: 10th Theory of Cryptography Conference*. Vol. 7785. Lecture Notes in Computer Science. Springer, Heidelberg, 2013, pp. 600–620.

[JW16] Zahra Jafargholi and Daniel Wichs. "Adaptive Security of Yao's Garbled Circuits." In: *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*. Vol. 9985. Lecture Notes in Computer Science. Springer, Heidelberg, 2016, pp. 433–458.

[JW05] Geetha Jagannathan and Rebecca N. Wright. "Privacy-preserving distributed k-means clustering over arbitrarily partitioned data." In: *ACM SIGKDD 05: International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2005, pp. 593–599.

[JS07] Stanislaw Jarecki and Vitaly Shmatikov. "Efficient Two-Party Secure Computation on Committed Inputs." In: *Advances in Cryptology – EUROCRYPT 2007*. Vol. 4515. Lecture Notes in Computer Science. Springer, Heidelberg, 2007, pp. 97–114.

[KP08] Stefan Katzenbeisser and Milan Petkovic. "Privacy-Preserving Recommendation Systems for Consumer Healthcare Services." In: *ARES 08: Third International Conference on Availability, Reliability and Security*. IEEE Computer Society Press, 2008, pp. 889–895.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "Actively Secure OT Extension with Optimal Overhead." In: *Advances in Cryptology – CRYPTO 2015, Part I*. Vol. 9215. Lecture Notes in Computer Science. Springer, Heidelberg, 2015, pp. 724–741.

[KS14] Marcel Keller and Peter Scholl. "Efficient, Oblivious Data Structures for MPC." In: *Advances in Cryptology – ASIACRYPT 2014, Part II*. Vol. 8874. Lecture Notes in Computer Science. Springer, Heidelberg, 2014, pp. 506–525.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. "An architecture for practical actively secure MPC with dishonest majority." In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. ACM Press, 2013, pp. 549–560.

[KKW17] W. Sean Kennedy, Vladimir Kolesnikov, and Gordon T. Wilfong. "Overlaying Conditional Circuit Clauses for Secure Computation." In: *Advances in Cryptology – ASIACRYPT 2017, Part II*. Vol. 10625. Lecture Notes in Computer Science. Springer, Heidelberg, 2017, pp. 499–528.

[Ker11] Florian Kerschbaum. "Automatically optimizing secure computation." In: *ACM CCS 11: 18th Conference on Computer and Communications Security*. ACM Press, 2011, pp. 703–714.

[Ker13] Florian Kerschbaum. "Expression rewriting for optimizing secure computation." In: *ACM CODASPY 13: Third Conference on Data and Application Security and Privacy*. ACM Press, 2013, pp. 49–58.

[KSS14] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. "Automatic Protocol Selection in Secure Two-Party Computations." In: *ACNS 14: 12th International Conference on Applied Cryptography and Network Security*. Vol. 8479. Lecture Notes in Computer Science. Springer, Heidelberg, 2014, pp. 566–584.

[Kil88] Joe Kilian. "Founding Cryptography on Oblivious Transfer." In: *20th Annual ACM Symposium on Theory of Computing*. ACM Press, 1988, pp. 20–31.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. "Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima." In: *CANS 09: 8th International Conference on Cryptology and Network Security*. Vol. 5888. Lecture Notes in Computer Science. Springer, Heidelberg, 2009, pp. 1–20.

[KS08] Vladimir Kolesnikov and Thomas Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications." In: *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*. Vol. 5126. Lecture Notes in Computer Science. Springer, Heidelberg, 2008, pp. 486–498.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. "Billion-Gate Secure Computation with Malicious Adversaries." In: *21th USENIX Security Symposium*. USENIX Association, 2012, pp. 285–300.

[Kre+13] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin R. B. Butler. "PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation." In: *22th USENIX Security Symposium*. USENIX Association, 2013, pp. 321–336.

[Krü+17] Stefan Krüger et al. "CogniCrypt: supporting developers in using cryptography." In: *32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*. IEEE Computer Society, 2017, pp. 931–936.

[Kue04] Andreas Kuehlmann. "Dynamic transition relation simplification for bounded property checking." In: *ICCAD 04: International Conference on Computer-Aided Design*. IEEE Computer Society / ACM, 2004, pp. 50–57.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. "On the (in)security of hash-based oblivious RAM and a new balancing scheme." In: *23rd Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 2012, pp. 143–156.

[Lau15] Peeter Laud. "Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees." In: *PoPETs 15: Proceedings of Privacy Enhancing Technologies* 2015.2 (2015), pp. 188–205.

[LP16] Peeter Laud and Alisa Pankova. "Optimizing Secure Computation Programs with Private Conditionals." In: *ICICS 16: 18th International Conference on Information and Communications Security*. Vol. 9977. Lecture Notes in Computer Science. Springer, 2016, pp. 418–430.

[Ler06] Xavier Leroy. "Formal certification of a compiler backend or: programming a compiler with a proof assistant." In: *ACM POPL 06: 33rd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54.

[Lin16] Yehuda Lindell. "Fast Cut-and-Choose-Based Protocols for Malicious and Covert Adversaries." In: *Journal of Cryptology* 29.2 (2016), pp. 456–490.

[LP09] Yehuda Lindell and Benny Pinkas. "A Proof of Security of Yao's Protocol for Two-Party Computation." In: *Journal of Cryptology* 22.2 (2009), pp. 161–188.

[LP15] Yehuda Lindell and Benny Pinkas. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries." In: *Journal of Cryptology* 28.2 (2015), pp. 312–350.

[LR14] Yehuda Lindell and Ben Riva. "Cut-and-Choose Yao-Based Secure Computation in the Online/Offline and Batch Settings." In: *Advances in Cryptology – CRYPTO 2014, Part II*. Vol. 8617. Lecture Notes in Computer Science. Springer, Heidelberg, 2014, pp. 476–494.

[LR15] Yehuda Lindell and Ben Riva. "Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries." In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. ACM Press, 2015, pp. 579–590.

[Liu+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. "Automating Efficient RAM-Model Secure Computation." In: *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2014, pp. 623–638.

[Liu+15a] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation." In: *ASPLOS 15: Twentieth International*

Conference on Architectural Support for Programming Languages and Operating Systems. ACM Press, 2015, pp. 87–101.

[Liu+15b] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. "ObliVM: A Programming Framework for Secure Computation." In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2015, pp. 359–376.

[Liu+17] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "Oblivious Neural Network Predictions via MiniONN Transformations." In: *ACM CCS 17: 24th Conference on Computer and Communications Security*. ACM Press, 2017, pp. 619–631.

[LO13] Steve Lu and Rafail Ostrovsky. "Distributed Oblivious RAM for Secure Two-Party Computation." In: *TCC 2013: 10th Theory of Cryptography Conference*. Vol. 7785. Lecture Notes in Computer Science. Springer, Heidelberg, 2013, pp. 377–396.

[Mal+04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. "Fairplay - Secure Two-Party Computation System." In: *13th USENIX Security Symposium*. USENIX Association, 2004, pp. 287–302.

[MG90] Carolyn McCreary and Helen Gill. "Efficient Exploitation of Concurrency Using Graph Decomposition." In: *International Conference on Parallel Processing*. 1990, pp. 199–203.

[MCB06] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. "DAG-aware AIG rewriting a fresh look at combinational logic synthesis." In: *DAC 06: 43rd Design Automation Conference*. ACM Press, 2006.

[Mis+06] Alan Mishchenko, Satrajit Chatterjee, Robert K. Brayton, and Niklas Eén. "Improvements to combinational equivalence checking." In: *ICCAD 06: International Conference on Computer-Aided Design*. IEEE Computer Society / ACM, 2006, pp. 836–843.

[MR18] Payman Mohassel and Peter Rindal. "ABY^3: A Mixed Protocol Framework for Machine Learning." In: *ACM CCS 18: 25th Conference on Computer and Communications Security*. ACM Press, 2018, pp. 35–52.

[MLB12] Benjamin Mood, Lara Letaw, and Kevin Butler. "Memory-Efficient Garbled Circuit Generation for Mobile Devices." In: *FC 2012: 16th International Conference on Financial Cryptography and Data Security*. Vol. 7397. Lecture Notes in Computer Science. Springer, Heidelberg, 2012, pp. 254–268.

[Moo+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin R. B. Butler, and Patrick Traynor. "Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation." In: *IEEE EuroS&P 16: European Symposium on Security and Privacy*. IEEE Computer Society Press, 2016, pp. 112–127.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN: 155860-320-4.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. "Privacy preserving auctions and mechanism design." In: *1st ACM Conference on Electronic Commerce*. ACM Press, 1999, pp. 129–139.

[Nay+15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. "GraphSC: Parallel Secure Computation Made Easy." In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2015, pp. 377–394.

[NS07] Janus Dam Nielsen and Michael I. Schwartzbach. "A domain-specific programming language for secure multiparty computation." In: *ACM PLAS 07: Workshop on Programming Languages and Analysis for Security*. ACM Press, 2007, pp. 21–30.

[Nie+12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. "A New Approach to Practical Active-Secure Two-Party Computation." In: *Advances in Cryptology – CRYPTO 2012*. Vol. 7417. Lecture Notes in Computer Science. Springer, Heidelberg, 2012, pp. 681–700.

[Nik+13a] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. "Privacy-preserving matrix factorization." In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. ACM Press, 2013, pp. 801–812.

[Nik+13b] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. "Privacy-Preserving Ridge Regression on Hundreds of Millions of Records." In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2013, pp. 334–348.

[Pat+16] Erman Pattuk, Murat Kantarcioglu, Huseyin Ulusoy, and Bradley Malin. "CheapSMC: A Framework to Minimize Secure Multiparty Computation Cost in the Cloud." In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, Cham, 2016, pp. 285–294.

[Pin+09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. "Secure Two-Party Computation Is Practical." In: *Advances in Cryptology – ASIACRYPT 2009*. Vol. 5912. Lecture Notes in Computer Science. Springer, Heidelberg, 2009, pp. 250–267.

[Pou12] Louis-Noel Pouchet. *Polyhedral Compiler Collection (PoCC)*. 2012.

[PS15] Pille Pullonen and Sander Siim. "Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations." In: *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable*. Vol. 8976. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2015, pp. 172–183.

[Rab81] Michael O. Rabin. *How to exchange secrets by oblivious transfer*. Tech. rep. Technical Report TR-81, AikenComputation Laboratory, Harvard University, Cambridge, MA, 1981.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations." In: *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2014, pp. 655–670.

[Ria+18] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. "Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications." In: *ASIACCS 18: 13th ACM Symposium on Information, Computer and Communications Security*. ACM Press, 2018, pp. 707–721.

[RR16] Peter Rindal and Mike Rosulek. "Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution." In: *25th USENIX Security Symposium*. USENIX Association, 2016, pp. 297–314.

[Rob58] James E Robertson. "A new class of digital division methods." In: *IRE Transactions on Electronic Computers* 3 (1958), pp. 218–222.

[SZ13] Thomas Schneider and Michael Zohner. "GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits." In: *FC 2013: 17th International Conference on Financial Cryptography and Data Security*. Vol. 7859. Lecture Notes in Computer Science. Springer, Heidelberg, 2013, pp. 275–292.

[SK11] Axel Schröpfer and Florian Kerschbaum. "Forecasting Run-Times of Secure Two-Party Computation." In: *QEST 11: Eighth International Conference on Quantitative Evaluation of Systems*. IEEE Computer Society Press, 2011, pp. 181–190.

[SKM11] Axel Schröpfer, Florian Kerschbaum, and Günter Müller. "L1 - An Intermediate Language for Mixed-Protocol Secure Computation." In: *IEEE COMPSAC: 35th Annual International Computer Software and Applications Conference*. IEEE Computer Society Press, 2011, pp. 298–307.

[She93] Naveed A. Sherwani. *Algorithms for VLSI physical design automation*. Kluwer Academic Publishers, 1993. ISBN: 079239-294-9.

[Shi+11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. "Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost." In: *Advances in Cryptology – ASIACRYPT 2011*. Vol. 7073. Lecture Notes in Computer Science. Springer, Heidelberg, 2011, pp. 197–214.

[Soc85] IEEE Computer Society. *IEEE-754 Standard for Binary Floating-Point Arithmetic*. 1985.

[Son+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits." In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2015, pp. 411–428.

[Spd] *SPDZ-2 Compiler as part of the SPDZ framework*. https://github.com/bristolcrypto/SPDZ-2. 2017.

[Ste+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. "Path ORAM: an extremely simple oblivious RAM protocol." In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. ACM Press, 2013, pp. 299–310.

[TP14] Meltem Sönmez Turan and René Peralta. "The Multiplicative Complexity of Boolean Functions on Four and Five Variables." In: *Lightweight Cryptography for Security and Privacy - Third International Workshop, LightSec 2014*. Vol. 8898. Lecture Notes in Computer Science. Springer, 2014, pp. 21–33.

[VC03] Jaideep Vaidya and Chris Clifton. "Privacy-preserving *k*-means clustering over vertically partitioned data." In: *ACM SIGKDD 03: International Conference on Knowledge*

Discovery and Data Mining. ACM Press, 2003, pp. 206–215.

[Wal64] Christopher S Wallace. "A suggestion for a fast multiplier." In: *IEEE Transactions on Electronic Computers* 1 (1964).

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound." In: *ACM CCS 15: 22nd Conference on Computer and Communications Security*. ACM Press, 2015, pp. 850–861.

[Wan+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. "SCORAM: Oblivious RAM for Secure Computation." In: *ACM CCS 14: 21st Conference on Computer and Communications Security*. ACM Press, 2014, pp. 191–202.

[Web17] Alina Sophie Weber. *Compiler for RAM-based Secure Multiparty Computation*. Technische Universität Darmstadt, Germany, 2017.

[Wil+94] Robert P. Wilson et al. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers." In: *ACM SIGPLAN Notices* 29.12 (1994), pp. 31–37.

[Yao82] Andrew Chi-Chih Yao. "Protocols for Secure Computations (Extended Abstract)." In: *23rd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1982, pp. 160–164.

[Yao86] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)." In: *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1986, pp. 162–167.

[ZE13] Samee Zahur and David Evans. "Circuit Structures for Improving Efficiency of Security and Privacy Tools." In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2013, pp. 493–507.

[ZE15] Samee Zahur and David Evans. *Obliv-C: A Language for Extensible Data-Oblivious Computation*. Cryptology ePrint Archive, Report 2015/1153. http://eprint.iacr.org/2015/1153. 2015.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. "Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates." In: *Advances in Cryptology – EUROCRYPT 2015, Part II*. Vol. 9057. Lecture Notes in Computer Science. Springer, Heidelberg, 2015, pp. 220–250.

[Zah+16] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. "Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation." In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2016, pp. 218–234.

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. "PICCO: a general-purpose compiler for private distributed computation." In: *ACM CCS 13: 20th Conference on Computer and Communications Security*. ACM Press, 2013, pp. 813–826.