



## Master Thesis

# Design & Evaluation of an Accessible High-Level Language for Advanced Cryptography

**Author(s):**

Aeschbacher, Ulla

**Publication Date:**

2020

**Permanent Link:**

<https://doi.org/10.3929/ethz-b-000411126> →

**Rights / License:**

[In Copyright - Non-Commercial Use Permitted](#) →

This page was generated automatically upon download from the [ETH Zurich Research Collection](#). For more information please consult the [Terms of use](#).

# Master Thesis

Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

---

## Design & Evaluation of an Accessible High-Level Language for Advanced Cryptography

by Ulla Aeschbacher

ETH student ID: 13-916-028

E-mail address: ullaa@student.ethz.ch

Supervisors: Prof. Dr. Friedemann Mattern  
Alexander Viand

Date of submission: March 18, 2020





Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Design & Evaluation of an Accessible High-Level Language for Advanced Cryptography

**Authored by** (in block letters):

*For papers written by groups the names of all authors are required.*

**Name(s):**

Aeschbacher

**First name(s):**

Ulla

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

**Place, date**

Bülach, 16.3.20

**Signature(s)**

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*



# Abstract

More and more private data is being used by companies to offer personally tailored services to users. These services access a lot of private data to provide users with useful utilities. Protecting private data is becoming more important to a lot of people but at the same time, they want to still use these services. Until recently, a user either had to entrust the company with their unencrypted private data or else could not reap the offered benefits. This is bothersome not only for the user but also the company, which does not want to miss out on the user's business.

Advanced cryptography concepts, most importantly Fully Homomorphic Encryption (FHE) and Secure Multi-Party Computation (SMPC), allow privacy-preserving computations, where users can benefit from offloading computation on user-encrypted data to a third party. This allows users to utilise personally tailored services while at the same time not having to give up their privacy. Unfortunately, using these concepts is currently very difficult and requires a lot of cryptographic knowledge from the user. Recent advances have been made to make specific techniques more accessible to the public, but all of them lack features or still require more cryptographic knowledge than an average user would possess. Additionally, there exists no tool that successfully combines the full functionality of both FHE and SMPC.

In this thesis, we introduce Chisel, an easy to use high-level language for advanced cryptography. In contrast, most existing tools are realised as libraries or language extension, which restricts their design. Using a language allows us full freedom to design the most user-friendly, accessible and expressive solution with which the user can specify their advanced cryptography application in a natural, intuitive way. Based on findings from a preliminary study on both our ideas and existing tools, we develop a design for a domain-specific language and implement it with the help of the JetBrains Meta Programming System (MPS). To evaluate the usability of our implementation, we design a validation study that compares Chisel to two existing advanced cryptography tools by doing within-subject A/B testing.



# Acknowledgements

This thesis would not exist in this form without the continuous feedback and enthusiasm from my supervisor Alexander Viand. He was always there for a meeting and our discussions were extremely helpful when I did not know how to proceed. Moreover, his reviews and his knowledge of technical writing helped to improve the readability and clarity of this thesis a lot.

I would like to thank Professor Friedemann Mattern and the whole research group for enabling me to write this thesis and providing me with a workspace in the office. I was very lucky to work in such a welcoming environment and have lots of good discussions over tea or coffee breaks.

My thanks go to everyone that proofread my thesis, notably David and Adrian. A big thank you goes to my parents for reviewing all my work and supporting me during my whole studies. Without them, I would not be where I am today.

Finally, I would like to thank my boyfriend Simon. He proofread my thesis, encouraged me and was always there when I needed support.





# Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>I. Introduction</b>	<b>1</b>
<b>1. Introduction</b>	<b>3</b>
1.1. Motivation . . . . .	3
1.2. Goals . . . . .	4
1.3. Structure . . . . .	5
<b>2. Background</b>	<b>7</b>
2.1. Homomorphic Encryption . . . . .	7
2.2. Secure Multi-Party Computation . . . . .	8
<b>3. Related Work</b>	<b>11</b>
3.1. Fully Homomorphic Encryption Compilers . . . . .	11
3.2. Secure Multi-Party Computation Compilers . . . . .	12
3.3. Verifiable Computing Tools . . . . .	12
3.4. Taxonomy . . . . .	13
<b>4. Methods</b>	<b>15</b>
4.1. Designing Usable Software . . . . .	15
4.2. Evaluating User Experience . . . . .	17
<b>II. Design</b>	<b>21</b>
<b>5. Preliminary Study</b>	<b>23</b>
5.1. Study Design . . . . .	23
5.2. Task Design . . . . .	25
5.3. Hypotheses . . . . .	27
5.4. Demographic . . . . .	27
5.5. How to do a Survey . . . . .	28
5.6. Results . . . . .	29
<b>6. Requirements</b>	<b>33</b>

<b>7. Domain-Specific Languages</b>	<b>37</b>
7.1. Domain-Specific Languages . . . . .	37
7.2. Domain-Specific Language Modeling Tools . . . . .	38
<b>8. Chisel Design</b>	<b>39</b>
8.1. Basic Structure . . . . .	39
8.2. Automation . . . . .	40
<b>III. Implementation</b>	<b>43</b>
<b>9. MPS and Projectional Editors</b>	<b>45</b>
9.1. Projectional Editors . . . . .	45
9.2. Modules in MPS . . . . .	45
9.3. The Language Module . . . . .	46
<b>10. Chisel Implementation</b>	<b>53</b>
10.1. Basic Structure . . . . .	54
10.2. Parties . . . . .	56
10.3. Computations . . . . .	57
<b>IV. Evaluation</b>	<b>59</b>
<b>11. Evaluation of Expressiveness</b>	<b>61</b>
11.1. Task Specification . . . . .	61
11.2. Party Inputs . . . . .	62
11.3. FHE Protocol . . . . .	64
11.4. MPC Protocol . . . . .	65
<b>12. Evaluation of Usability</b>	<b>69</b>
12.1. Research Questions . . . . .	69
12.2. Study Design . . . . .	69
12.3. Task Design . . . . .	70
12.4. Result evaluation . . . . .	71
12.5. Demographic . . . . .	72
12.6. Support . . . . .	72
12.7. Ethics . . . . .	72
12.8. Results . . . . .	73
<b>13. Conclusion</b>	<b>75</b>
13.1. Summary . . . . .	75
13.2. Future Work . . . . .	75

<b>V. Appendix</b>	<b>83</b>
A. Taxonomy	85
B. Exploratory User Survey	87
C. ValidationStudy	95



## Part I

# Introduction

*The beginning is the most important part of the work.*  
— Plato

In this part, we motivate our thesis and describe the goals we aim to achieve. We then give some required background on advanced cryptography concepts. Afterwards, we present an extensive related work section resulting in a taxonomy of advanced cryptography tools. In the end we give a broad overview over prior work on usable security and related areas.



# Chapter 1

## Introduction

### 1.1. Motivation

Today data is becoming increasingly important and is starting to be seen as a valuable asset by companies. Often private data yields the most value. For example, companies use it to improve services by tailoring them to personal preference. Take the Apple Watch, which automatically calls emergency services if it detects a hard fall followed by immobility [62]. On the other hand, using private data introduces new challenges like liability and restrictions due to regulations [59]. For reasons including data leaks and fear of their data being used for surveillance purposes, people are increasingly sensitised to data protection and are less willing to share it openly.

The goal of cryptography is to protect information by making secure communication possible. Advanced cryptography [40, 56] aims to go beyond this and not only protect communication but also computation. We thus achieve privacy-preserving computation, where users share only encrypted data but the same amount of information and insight can be gained. This is done using several techniques, most importantly Fully Homomorphic Encryption (FHE), Secure Multi-Party Computation (SMPC) and Zero-Knowledge Proofs (ZKP). The problem with all of these techniques is that at the moment they are only usable by experts in the field.

These schemes possess a high complexity both in their setup and their configuration. For example, in FHE the user needs to provide a computation circuit and then select the FHE parameters based on the circuit. Both the optimisation of the circuit as well as the selection of the parameters usually have to be done by an FHE expert. Similarly, applying MPC protocols requires a deep understanding of the capabilities and security guarantees of the protocol in question. In addition, choosing the appropriate protocol for an application can be highly non-trivial, and can often only be achieved by an expert.

Recently there have been advancements to make specific techniques more accessible to the public, both for FHE [21, 39, 45, 47], MPC [29, 52, 54] and ZKP [22, 42]. Most of these tools take the form of a compiler that translates high-level code into the required lower-level instructions. Automatic key-generation, automatic noise handling and an informational debug output are other features that are commonly found in these tools. The high-level code is written in an extended version of a language like C++, Python or Haskell. For example, Marble [45] introduces a new all-inclusive type `M` for all data that should be encrypted in the encoded program, as well as functions `analyze` and `evaluate` for analysing computation complexity and benchmarking the FHE computation. For comparison, in



Alchemy [39], the user can call `pt2ct` on a function to get the corresponding ciphertext. Ramparts [47], on the other hand, tries to completely abstract away the cryptographic aspects by allowing the input program to stay the same, automatically handling issues that arise from this.

**Current Problems** However, the features offered by the existing tools are still not sufficient to allow users to efficiently work with advanced cryptographic systems. First of all, users still have to know what techniques are available and which one would be most applicable to their situation before they can choose the most adequate tool. Secondly, most of these tools still require setting certain non-intuitive parameters by hand. Thirdly, a user always needs to learn new functions and parameters for every tool that they want to use. Most importantly though they need to rewrite their code every time their situation or requirements change. For example, assume that Alice wants to compute a function  $f$  on some secret data of hers. To achieve this, she uses Alchemy with the function she has written in Haskell to get FHE. A little while later she notices that it would be helpful if she could also include Bob’s secret data in the computation of  $f$ . This is now an MPC problem, so if she wants to use an existing tool to help with translation she needs to rewrite the whole function  $f$  in the language used by an MPC tool.

## 1.2. Goals

With our tool, we want to aid the user in specifying their real-world application in an intuitive, natural language. Based on these specifications, it should be possible to automatically select the most suitable technique and all further parameters and inputs that are needed. Therefore, users will no longer be required to rewrite their code if their situation changes, instead writing their code only once.

This presents several challenges that we will address in this thesis. We will focus our work more on how to design a user-friendly language and less on how to build an actual compiler. The extension of this language to a fully working system is left for future work. Different techniques sometimes need completely different parameters from the user; we need to ensure that our system can always extract enough information to select the most suitable techniques. Thus, we need to be able to infer most of the information without directly asking the user. In addition, it is not clear from the outset what design features would make such a tool „easy to use“. Hence, we will first try to discover what users desire in these kinds of tools and what users perceive as intuitive design.

## 1.3. Structure

The remainder of this thesis is structured as follows:

**Part 1** Background on advanced cryptography, in-depth related work, our taxonomy of advanced cryptography tools, as well as methods on designing usable software and evaluating it

**Part 2** Design of a preliminary study and resulting requirements for a usable language, introduction to domain-specific languages, as well as design of our language

**Part 3** Introduction to the Meta-Programming System (MPS) and a detailed description of the implementation of our language

**Part 4** Evaluation of our language in terms of expressiveness and usability, concluding with possible directions for further research



# Chapter 2

## Background

We will now give a very high-level overview of homomorphic encryption and secure multi-party computation. This will help the reader understand the rest of this thesis. We assume that the reader knows general encryption schemes.

### 2.1. Homomorphic Encryption

Homomorphic encryption allows computations on encrypted data without requiring a secret key. For example, one party can give encrypted private input to another party that can then do computations upon this encrypted data while never having access to the plaintext. The result of the computation is encrypted with the same key that the input was encrypted with and can thus only be decrypted by the party that gave the input. Additionally, this computation is only polynomially slower than on unencrypted data and encryption as well as decryption have to be reasonably fast.

A homomorphic encryption scheme usually consists of four operations: *Enc*, *Dec*, *KeyGen* and *Eval*. While *Enc*, *Dec* and *KeyGen* are similarly defined to any encryption scheme, *Eval* is where the encrypted inputs are processed and an encrypted output is generated. A scheme is called homomorphic over an operation  $\oplus$ , if the following holds true  $\forall x_1, x_2 \in X$ , where  $X$  is the set of all possible data

$$Dec(Enc(x_1) \oplus Enc(x_2)) = Dec(Enc(x_1 \oplus x_2)) \quad (2.1)$$

Any boolean circuit can be built while only using XOR and AND gates. Similarly any function can be built while only using additions and multiplications, thus for an encryption scheme to be universally applicable, it only needs to support both addition and multiplication.

The first kinds of homomorphic encryption schemes that were found only supported either addition or multiplication and were thus called Partial Homomorphic Encryption (PHE) schemes. One step further were the Somewhat Homomorphic Encryption (SWHE) schemes that supported both operations, but only for a subset of circuits. Leveled Fully Homomorphic Encryption (LFHE) schemes support the evaluation of both operations and thus allow for evaluation of arbitrary circuits, but only of bounded, pre-determined depth. Finally, Fully Homomorphic Encryption (FHE) schemes allow the evaluation of both operations for an unlimited number of times and thus allow for evaluation of arbitrary circuits of unbounded depth.

**Implementations** There is a number of different FHE schemes, for example BGV [13], BFV [14], FHEW [24], CKKS [32] and TFHE [61], as well as libraries implementing these schemes, such as HeLib (BGV, CKKS) [16], HEAAN (CKKS) [32], Palisade (BFV, BGV, CKKS, FHEW) [57] and SEAL (BFV, CKKS) [55].

These schemes all share the same basic characteristics, thus we will not go into further detail here on what differentiates them. One problem they all share is that each ciphertext has some noise that hides the message behind it. If the error is small, the client can use their knowledge of the hidden code to remove the noise, but if the noise gets too large, decryption is hopeless. The noise is mostly increased when two ciphertexts are multiplied. Thus the multiplicative depth of a ciphertext, i.e. the number of times the ciphertext was the result of a multiplication, determines a lot of parameters in an FHE scheme. FHE schemes work by adding Bootstrapping to an LFHE. Bootstrapping is a technique that periodically reduces noise and thus allows an unlimited number of operations to be performed.

## 2.2. Secure Multi-Party Computation

Secure Multi-Party Computation (SMPC) allows a set of parties to jointly compute a function on their private inputs. At the end of the computation each party should obtain its output without acquiring any other information. The protocol is often resistant to dishonest parties. How many dishonest nodes lead to a failure in the protocol depends on the concrete protocol.

There are two different notions of dishonest behaviour. A semi-honest party, also called a honest-but-curious party, acts according to the protocol, but will try to learn as much as they can from the messages they receive. They might also share information between them as long as the protocol allows them to. Malicious parties on the other hand will deviate from the protocol. Some common building blocks used in SMPC are oblivious transfer, secret sharing, garbled circuits and homomorphic encryption. [60]

- **Oblivious transfer** This is a secure two-party computation (S2PK) task, one party is the sender, the other the receiver. It allows the receiver to choose  $k$  of  $n$  secrets of the sender, without revealing which ones they chose, and without learning anything of the remaining secrets.
- **Secret sharing** This is a SMPC protocol, with one party as the dealer while the others are recipients. The dealer breaks a secret value into shares and distributes them among the recipients. Any unqualified group of recipients should learn nothing about the secret, while a qualified group should be able to reconstruct the secret. One kind of secret sharing is threshold sharing, where a group is qualified if they own more than  $t$  shares.[52]
- **Garbled circuits** These again work only for S2PK, one party is the garbler, the other the evaluator. The garbler creates a garbled version of the circuit and encodes their inputs accordingly. The evaluator gets the circuit and inputs, as well as encodings of their inputs via oblivious transfer and evaluates the circuit to compute

an encoded output. This output is then decoded by the garbler and shared with the evaluator.

- **Homomorphic encryption** This was already discussed in the previous section, but we will introduce the notion of multi-user FHE here. There are two types of multi-user FHE. Multi-key homomorphic encryption schemes support computation on ciphertexts that were encrypted using different keys [49]. Threshold homomorphic encryption schemes use an SMPC protocol in which the parties generate a joint public key and each get a secret share of the private key. Thus, the server can compute on the received ciphertexts, and the users can interactively apply a decryption protocol to obtain the decrypted result.



## Chapter 3

# Related Work

There have been several ventures at building programming frameworks for advanced cryptography, mostly in the last decade. In this chapter, we will present the most relevant related work from fully homomorphic encryption, secure multi-party computation and verifiable computing. At the end of this chapter, we compile this body of work into a taxonomy to get a better overview and make comparisons easier.

### 3.1. Fully Homomorphic Encryption Compilers

The first viable fully homomorphic encryption scheme was presented in 2009 [10]. Since then some more mature libraries have been developed, most prominently HELib [16], SEAL [55] and  $\Lambda \circ \lambda$  [39].

One of the first efforts at making FHE more accessible was Armadillo [21], now called Cingulata, where C++ code is transformed into a boolean circuit and then executed using FHE. The biggest limitation was speed, because the circuits generated were large and not well optimised. A few years later, the field has gained traction and there are multiple publications on the topic of accessible FHE.

Chielle et al. [38] published  $E^3$ , which integrated three libraries under one framework. It also enabled users to write extended C++ code which is then compiled into an FHE-compatible circuit and automatically handed to the most suited library. However, the performance relied heavily on user-defined upper bounds for loops and sizes of integers, thus making the tool inaccessible for novice programmers.

The Alchemy [39] compiler published in 2018 is another tool which transforms plaintext Haskell-based programs into corresponding homomorphic ones. Again, the user has to provide performance-critical parameters, making it difficult for non-experts to use.

Marble [45], on the other hand, focuses on automatically selecting all parameters, thus making it easy for everyone to use. However, the lack of advanced optimizations sacrifices performance.

More recently, Ramparts [47] aims for a programmer-friendly system by constructing an optimised circuit from general Julia code, again choosing all parameters automatically. Ramparts also suffers from slow compile-time performance, because their symbolic execution approach slows down compilation.

There have also been some publications focused on machine learning applications, for example nGraph-HE [48], CHET [50] and SEALion [51]. While nGraph-HE still requires the programmer to select HE parameters, CHET and SEALion focus more on accessibility



for people without a cryptography background. Because we will not focus on specific applications, we refer the reader to the respective papers for more information on these tools.

## 3.2. Secure Multi-Party Computation Compilers

MPC schemes have existed since the 1980s [1, 2], and thus developing accessible high-level frameworks for MPC has been considered for a significantly longer time than for FHE.

One of the earliest approaches to make MPC more accessible to the general public was TASTY [11], which compiles Python-based code into protocols based on partially homomorphic encryption and garbled circuits. One problem is that their proposed high-level language is still very low-level in certain areas, with the user having to explicitly specify the communication between the parties as well as the means of secure computation. Recent work like the ABY framework [23] has a similar goal as TASTY, aiming to combine multiple secure computation schemes, but again it has similar limitations.

Wysteria [20] is an attempt at formalising a functional-style programming language for MPC mixed-mode computations, where some code is executed locally on one machine and some code is executed securely in a joint effort. While the correctness and security guarantees they give in their paper are interesting, the available data types are very restrictive, for example there are no arbitrary-length integer or floats. This makes the tool not applicable for real-world applications.

HyCC [37] makes hybrid MPC accessible to non-domain experts by automatically creating partitioned circuits and optimising the selection of the circuits from standard C code. The tool has several language restrictions, however. For example, there is no support for global variables, floating point operations, or dynamic memory allocation, making it again unusable for real-world applications.

As in FHE, there have been publications focused on machine learning, like PrivPy [54] which aims to provide programmers with an intuitive Python interface, while also providing an MPC backend. Again the main drawback is expressiveness.

## 3.3. Verifiable Computing Tools

Recently, there has also been some development in the field of making verifiable computing more accessible to the public. Z0 [19], Geppetto [22], Buffet [26] and Pinocchio [30] are some of the first high-level language compilers. While these tools still have language restrictions and are sometimes not very accessible, they paved the way for more recent systems like xJSnark [42], which introduces a user-friendly language for non-specialist users. Since this thesis does not focus on verifiable computing, we will not discuss these tools in more detail here.

## 3.4. Taxonomy

We categorised all the existing tools we discussed into a taxonomy, i.e. a classification scheme, to ensure that we have a good overview over their strengths and weaknesses. A condensed version of the taxonomy we developed can be seen in Table 3.1. The complete version of this table can be found in Appendix A. We chose to divide the assessment into five categories:

- **Technique** records which of the three advanced cryptography techniques that we consider - FHE, MPC or VC - the tool supports.
- **Architecture** records how the tool is realised - as a library, an extension or a language - and on what language it is based, if any. This is of interest not only for implementation, but also has subtle effects on the usability, as we discuss in Section 7.1.
- **Documentation** ratings are based on the methodology of a taxonomy of MPC tools [52]. We consider whether or not an academic paper, language documentation, example code and/or documentation for the tool exist and whether or not the tool itself is open source. In the condensed version given here, we summarised these properties into one rating: a full circle ● indicates that the documentation is good, a half-full circle ◐ indicates that the documentation could be improved and an empty circle ○ indicates that documentation is minimal or nonexistent. We chose this property because documentation is one of the most important aspects when it comes to make a tool user-friendly and accessible to non-experts.
- **Automation** tracks which part of the process the tool automates. Possible automated aspects are the selection of FHE vs MPC, library choice, key generation, noise checks, ciphertext parameter generation and circuit generation. We again summarised these properties into one rating in the condensed version, taking into account that some aspects are not applicable for certain techniques, using full, half-filled and empty circles to indicate (near) complete, basic, and minimal automation. Since automation helps to make a tool more user-friendly and accessible to non-experts, this property directly and significantly impacts the overall usability of a tool.
- **Target personas** are the final aspect we recorded. Here we try to categorise the tools based on their intended target audiences. We created a set of user personas with different backgrounds and different requirements and tried to estimate whether or not a tool would be suitable for a given target persona. Of course this only represents our personal impression, as most tools do not explicitly define their target audience. When we were unsure, we tried to err on the side of caution and increased the number of target personas. In the condensed version, the first circle represents a total novice, while the last circle represents a crypto expert.

	Technique			Architecture			Documentation	Automation	Target Personas
	FHE	MPC	VC		Library	Extension Language			
Alchemy	●			Haskell		●	◐	◐	○ ○ ● ● ○ ○
CHET	●					●	○	◐	○ ○ ● ● ● ○
Cingulata	●			C++		●	●	◐	○ ○ ● ○ ○ ○
E3	●			C++		●	●	◐	○ ○ ● ○ ○ ○
Marble	●			C++	●		◐	◐	○ ○ ● ○ ○ ○
NGraph-HE	●			nGraph		●	◐	○	○ ○ ○ ● ● ○
Ramparts	●			Julia	●		○	◐	○ ● ● ● ○ ○
SEALion	●			TensorFlow		●	○	○	○ ○ ○ ● ● ○
TASTY	●	●		Python		●	●	●	○ ○ ○ ○ ● ○
ABY		●		C++	●		◐	●	○ ○ ○ ● ● ○
Frigate		●		C		●	◐	○	○ ● ● ● ○ ○
HyCC		●		C		●	◐	●	○ ○ ● ● ● ○
MP-SPDZ		●		Python		●	◐	◐	○ ○ ○ ● ● ○
OblivM		●		SCVM		●	◐	◐	○ ○ ● ● ● ●
PICCO		●		C		●	◐	○	○ ○ ● ● ○ ○
PrivPy		●		Python		●	○	◐	○ ● ● ○ ○ ○
Wysteria		●				●	●	◐	○ ○ ○ ● ● ○
Buffet			●	C		●	◐	◐	○ ○ ● ● ○ ○
xJSnark			●	Java		●	●	◐	○ ● ● ○ ○ ○
ZØ			●	C#		●	○	●	○ ○ ● ○ ○ ○
zkay			●	Solidity		●	◐	◐	○ ○ ● ● ○ ○

Table 3.1.: Taxonomy of advanced cryptography tools.

# Chapter 4

## Methods

In this chapter, we first give an overview of techniques that have been developed to help design usable software. Then we look at how user experience can be evaluated, with a focus on the cognitive dimensions framework.

### 4.1. Designing Usable Software

There exists a lot of research into how applications can be made more user-friendly. Several methods have been developed to approach the problem of usability in a structured manner and we will present some of them in the following sections.

#### 4.1.1. Usability and Security

Security applications are often thought of having a trade-off between security guarantees and usability. Yee [7] discusses this problem and proposes three recommendations to align security and usability: Both goals must be considered throughout the design process, an agreement between the system's security state and the user's mental model thereof has to be maintained, and security decisions have to be incorporated into the user's workflow. He makes a strong point for security by designation instead of admonition, which makes sure that the system is secure while not asking the user to establish a detailed security policy beforehand. Acar, Fahl, and Mazurek [27] make a similar point and also give key lessons for usable security and how to apply them.

Kainda, Fléchaïs, and Roscoe [12] extend on this and build a security and usability threat model including the critical factors related to either or both of the topics. They then also provide a way to analyse a system based on this model. These papers mostly focus on applications like password managers and firewalls and are thus not directly applicable to our project. Herzog and Shahmehri [9] look at a specific aspect of usability, namely user help techniques, and how they can be employed in a security application. However, this is specific to GUI programs which are out of scope for our project.

In summary, while existing usability guidelines for security software encode important information to keep in mind when designing secure software, they are not directly applicable to our setting of designing a programming language. In the following section, we thus consider related work from an area that is more closely connected to language design.

### 4.1.2. Usability of APIs

Designing a programming language is more closely connected to the design of APIs and thus, it might be possible to apply research into the usability of APIs to our project. In the last thirty years, there has been a lot of research into trying to enhance various aspects of programming. Prior work includes work on usable APIs, libraries and general interfaces. For example, Nielsen [5] gives ten criteria for successful Human-Computer-Interaction (HCI). There has also been research that focuses on the usability of security APIs, for example Johnston, Eloff, and Labuschagne [6] define HCI-S as "the part of a user interface which is responsible for establishing the common ground between a user and the security features of a system" and give six criteria for successful HCI-S, as listed in table 4.1 alongside the original HCI criteria.

Another approach is taken by Gorski and Iacono [28]; they give eleven usability characteristics for security APIs. We list these in table 4.2. There is not a lot of overlap between this approach and the one from Johnston, Eloff and Labuschagne. One similarity is that *Visibility of system status*, *aesthetic and minimalist design* and *convey features* can be mapped to *information obligation*. A second one is that *errors* is included in *case distinction management*. But the rest of the criteria are quite different, with Gorski and Iacono focusing more on outside aspects like *adherence to security principles*, *security prerequisites* and *execution platform*, while the HCI-S criteria focus more on the user with *learnability* and *satisfaction*.

1	Visibility of system status	1	Visibility of system status
2	Aesthetic and minimalist design	2	Aesthetic and minimalist design
3	Error prevention	3	Errors
4	Help users recognise, diagnose, and recover from errors		
5	Recognition rather than recall	4	Convey features
6	Consistency and standards		
7	Flexibility and efficiency of use		
8	Match between system and the real world	5	Learnability
9	User control and freedom		
10	Help and documentation		
		6	Satisfaction

Table 4.1.: Nielsen's ten criteria for successful HCI and the six criteria for successful HCI-S.

1	End-User Protection
2	Case Distinction Management
3	Adherence to Security Principles
4	Testability
5	Constrainability
6	Information Obligation
7	Degree of Reliability
8	Security Prerequisites
9	Execution Platform
10	Delegation
11	Implementation Error Susceptibility

Table 4.2.: Gorski and Iaconos eleven usability characteristics for security APIs.

## 4.2. Evaluating User Experience

In this section, we first describe the cognitive dimensions framework as one of the most used tools for evaluating the usability of programming tools. Then we give two example areas in which multiple UX evaluations have been done: SSL tools and smart contract languages.

### 4.2.1. Cognitive Dimensions Framework

The cognitive dimensions framework was first introduced by Green [3] as a way to evaluate the usability of programming tools. The core idea behind it is to provide a way to compare the form and structure of a language.

One example of a cognitive dimension is viscosity, i.e. how much a system resists local changes. A statically typed language, for instance, is more viscous than a dynamically typed one, which reduces typing errors and increases readability. But a more viscous system usually also cause more work for the user. This example illustrates how these dimensions are often not strictly rated on a good to bad scale, but instead only help to categorise and compare different languages.

Together with Blackwell, he later published a generic questionnaire with questions covering all the cognitive dimensions [4]. In 2003, Clarke tried to use this framework but felt that the original dimensions did not always have a clear distinction and that the names would not translate well to an industrial setting. Thus, he used the same methodology but developed his own set of cognitive dimensions and appropriate questions [8].

One example of a newly added dimension is the working framework, which describes how much of the context the user has to have in mind while writing code. These cognitive dimensions are listed in table 4.3, together with Green’s original dimensions.

A concrete implementation of a questionnaire to evaluate the usability of a security API is given by Wijayarathna, Arachchilage, and Slay [36]. They base their questions on Clarke’s cognitive dimensions, but also take into account Gorski and Iacono [28]’s eleven characteristics, as well as the ten rules for a good cryptography API given by Green and Smith [25]. They added three new dimensions: *Hard to Misuse*, *End-User Protection* and *Testability*. This resulted in a questionnaire with 45 questions in fifteen dimensions. We used this questionnaire as a base for our own questionnaire section on cognitive dimensions.

1	Visibility & Juxtaposition		
2	Diffusenes		
3	Hard Mental Operations		
4	Error Proneness		
5	Hidden Dependencies		
6	Provisionality		
7	Secondary Notion		
8	Viscosity	1	API Viscosity
9	Closeness of Mapping	2	Domain Correspondence
10	Role Expressiveness	3	Role Expressiveness
11	Progressive Evaluation	4	Progressive Evaluation
12	Premature Commitment	5	Premature Commitment
13	Consistency	6	Consistency
14	Abstraction Management	7	Abstraction Level
		8	API Elaboration
		9	Work-Step Unit
		10	Working Framework
		11	Learning Style
		12	Penetrability

Table 4.3.: Blackwell and Green’s original 14 cognitive dimensions as well as Clarkes 12 cognitive dimensions.

Lewis [33] argues that the cognitive dimensions style approach to language design will be impactful, while generalisations based on quantitative data will be much less informative. Stefik and Hanenberg [34], on the other hand, calls for such empirical investigations, claiming that most of the current claims are not based on sound evidence, including the cognitive dimensions framework mentioned above. We nevertheless plan to use the cognitive dimensions framework in parts of our project, because we feel that even in the absence of sound statistical evidence, it grants us insight into how a certain language resonates with participants.

A similar project to ours, just for statistical analysis instead of advanced cryptography, is pursued by Jun et al. [53]. They present the high-level language Tea, with the goal that the user is able to give a high-level specification of their study design. The system then automatically chooses an appropriate statistical test and executes it.

### 4.2.2. Evaluation of SSL Tools

Studies [15, 17] show that using SSL is extremely error-prone. Fahl et al. [18] investigate why developers deploy non-validating certificates and found that accidental misconfiguration was the reason in about a third of the cases. One reason a lot of those developers gave was that they were confused about SSL configuration. Tools like Let's Encrypt [46] and Certbot are trying to increase the usage and usability of Transport Layer Security (TLS). Tiefenau et al. [58] found in a randomised control trial that Certbot increases usability, especially for less experienced participants. This result highlights the necessity of usable tools in order for security concepts to be adopted by the general public.

### 4.2.3. Evaluation of Smart Contract Languages

Blockchain, a data structure maintained by a peer-to-peer network, is a term that gets more and more attention as different use cases are discovered. One of these new use cases is the application of smart contracts, with which agreements between multiple parties can be enforced without having to rely on a trusted third party.

Recently there has been some research on smart contract programming languages like Solidity, Pact and Liquidity. Parizi, Amritraj, and Dehghantanha [43] compare all three languages in terms of usability and security vulnerability. They found that the language that was rated the most usable, Solidity, was also the one where developers left behind the most bugs and security vulnerabilities. Solidity has also been studied by other groups [31, 41, 44] and found to have lots of easily missed vulnerabilities. These papers highlight the main problems that exist in smart contract languages today: unexpected behaviour and bad or non-existent exceptions. They prove that a lot of the problems can be solved by running a static analyser over the high-level code. This confirms that a usable language that is still secure can only be achieved with a detailed and sophisticated language design.





## Part II

# Design

*Good design is obvious. Great design is transparent.*  
— Joe Soparano

This part describes our design and explains the design decisions we made. It starts with the description of a preliminary survey we conducted to discover the issues people have with existing tools and to test our first ideas. Based on the results of this survey we formulate the requirements a useful tool should fulfil. Next, we explain why we chose to use a domain-specific language for our tool. Finally, we specify the design of our new language Chisel.



# Chapter 5

## Preliminary Study

A good way to start the design process is to conduct a study on what users actually look for in an advanced cryptography tool. The main goal of our study was to identify how people from different backgrounds prefer to define their needs when designing a cryptographic application. We also aimed to discover what aspects of a language make it intuitive to use and easy to understand. This study should also show that existing solutions still have a lot of potential for improvements. In the end, this study will help us to outline the perfect tool and specify a list of requirements such a tool has to fulfil. Additionally, we present guidelines for study design that we learned from doing this survey.

### 5.1. Study Design

We want to test the users' understanding of the code. Even simple programs with very few lines of code can exhibit the tool- and cryptography-specific issues that we want to investigate. Therefore, asking users questions about short code snippets is a feasible way to test their understanding. This then gives us some indication of how easy and straightforward certain tools are.

We decided on comparing code snippets from existing tools. By testing the user on specific aspects of the code we can see where people encounter problems and what these problems are. Additionally, we also wanted to bring in our ideas of how a good tool might look, so we included code snippets written in a pseudo-language to represent those. Asking questions about these tools will give us initial feedback on our ideas and show us whether we are on the right track.

For existing tools, we decided on using Cingulata for FHE and HyCC for MPC. Cingulata is a compilation chain compiling C++ applications to work on encrypted data. We chose Cingulata because the C++ part of the code where the computation is defined is easy to understand, while the config file that generates keys, defines inputs and does the encryption and decryption is quite difficult to understand. This difference in difficulty between the computation and the rest is something we noticed with lots of tools and makes it hard for users to specify the security aspect of their protocol. By asking questions about both areas we can confirm this belief and thus show an area where improvements can be made.

HyCC is a toolchain for automated compilation of C programs into hybrid MPC protocols. We chose HyCC for similar reasons as Cingulata, as there is again a discrepancy between the difficulty of understanding the computation and understanding the cryptographic concepts going on around it. HyCC is even more extreme than Cingulata in that regard.

The computation is written in an easy to understand C file that does not even import any specific library. On the other hand, everything that has to do with the cryptographic aspect of the protocol is specified in a hard to understand makefile. This includes how the parties and inputs are defined, what part of the computation is secure, which party can see which variable, etc. Having two tools with different levels of this problem will help us gauge how this difference affects the comprehensibility of a tool.

Our ideas for FHE and MPC tried to address the problems displayed by the existing tools, most importantly we wanted to be more transparent with party and input definition and the security aspects of the computation. How good these tools perform in the survey will show us whether we are going in the right direction. To help us figure out the best study design, we did a pre-study, in which we included two different ideas for both FHE and MPC. We then carried the more well-liked one over to the main study, to make sure we got the best contrast between existing tools and what could be achieved.

The FHE idea was named AcTool, where our main goal here was to fit every part of the protocol in one easy to understand file. Both Cingulata and HyCC and with them most existing tools need at least two files to specify a protocol. Most of the time, they single out the computation from the rest. This leads to more work for the user, as well as having a detrimental effect on clarity and noticing errors quickly. Being able to see the whole protocol at a glance helps remove these problems.

The MPC idea was named MpcM, this code was split into one snippet per party to better understand how the parties worked together. Most existing tools are benchmarking tools that only simulate the concept of multi-party computation on a single computer, and how they would work in an actual application setting is left open. Users often struggle with this, as they cannot fathom why they would use a tool that is not usable for a concrete application. By specifying a snippet for each party as well as how they communicate, we made sure that this tool clearly indicates how it is used in a real-life application.

Our already mentioned pre-study helped us to test the usefulness of the results, and as a consequence, we removed questions that did not produce meaningful answers. We learned a lot about how a study should be designed to be appealing for a person to fill out while still giving us the information we care about. Our main changes were to shorten the survey considerably as well as to adapt a lot of the questions into multiple-choice questions. Multiple-choice questions are easier for the user to fill out, and are also easier to quantify into statistical results. An overview of what we learned about study design is given in Section 5.5.

At the end of the study, we first asked the user to provide us with some personal information like gender, age and background. We wanted to gauge if we hit our target demographic of people with programming experience but not cryptography knowledge. To do that, we asked for the users total years of programming experience as well as if they had any security knowledge and if yes, if that was helpful. In the end, the user could also leave any comments they might have.

## 5.2. Task Design

For each tool we wrote a small code snippet doing a similar computation, to make sure that the difficulty of the computation did not impact our comparison of the tools. The computations were very simple, on the level of averaging an array or doing an addition of two integers. Our code was well documented with descriptive variable and method names as well as comments. For each code snippet, we first asked the participants to describe what the code was doing. With this question, we on one hand wanted to get a first impression of how well people understand the code, and on the other hand learn what they notice while reading the program. For example, we were interested in how many people would mention security or cryptography in their answers. As a follow-up to the first question, the second question asked the participants to rate how confident they were in their answer on a scale from 1 (just a guess) to 5 (very confident). This question, even more so than the first one, gives us insight into how understandable the code is.

Next, we asked the participants to rate how important the comments were on a scale from 1 (not necessary) to 5 (essential). Our goal here was to again determine how easy to read the code is. As the last question, we asked about which part of the code was least clear. This was a straight-forward question to identify the parts of the code where users struggled most. It would also be interesting to see whether that is also where they would make the most errors.

Then we created five true-or-false statements for each tool that tested the understanding of the computation as well as the security aspects of the code. These multiple-choice questions were a good way to quantify the usability of a tool. We use an example code snippet shown in Listing 5.1 to show what we wanted to achieve with these questions, following it are the true-or-false statements corresponding to it.

---

```
//define vector-like structure
typedef struct
{
    int x_1, x_2, x_3, x_4;
} Vec4;

//function to compute
int average(Vec4 x)
{
    int sum = x.x_1 + x.x_2 + x.x_3 + x.x_4;
    return sum / 4;
}

int mpc_main(Vec4 INPUT_A_firstSecretInput, int INPUT_B_secondSecretInput)
{
    //compute result
    return average (INPUT_A_firstSecretInput) + INPUT_B_secondSecretInput;
}
```

---

---

```
# define locations of references
CBMC_GC = ../../bin/cbmc-gc
CIRCUIT_SIM = ../../bin/circuit-sim
ABY_CBMC_GC = ../../ABY/build/bin/aby-hycc

# define location of function
mpc_main.circ: main.c
    $(CBMC_GC) $^

# what to run
.PHONY: clean run-sim

# clean output files
clean:
    rm -f mpc_main.circ mpc_main.stats

# define inputs for simulation and run it
run-sim:
    @ $(CIRCUIT_SIM) mpc_main.circ --spec "
        INPUT_A_firstSecretInput := {x_1: 2; x_2: 2; x_3: 4; x_4: 4};
        INPUT_B_secondSecretInput := 5;
    print;"
```

---

Listing 5.1: The Two HyCC Code Snippets

- Please check all the statements that you think are true.
  - "mpc\_main" returns 3.
  - At some point, this code computes the average of two numbers.
  - The call to "average" returns 3.
  - There are 2 parties in this computation.
  - "INPUT\_A\_firstSecretInput" and "INPUT\_B\_secondSecretInput" come from two different parties.

The first three questions were to test the understanding of the computation as well as if inputs could correctly be pinpointed. With the fourth and fifth question, we wanted to test the understanding of the cryptography behind the scenes.

Before conducting our study, we set up some hypotheses about what we expected as the result of the study. One of our hypothesis was that questions about the computation only (like question two and three above) will be answered mostly correctly because they only require an understanding of C/Java/etc. The questions also taking into account input (like question one above) would be more difficult, depending on how transparent the tool was with the declaration and assignment of those. Here we expected to see a difference between the existing tools and our ideas, as we especially put a focus on making inputs more clear. We expected questions about cryptography (like questions four and five above) to perform the worst in existing tools. They often require the user to know exactly where to look for the answer, a feat not possible without extensive knowledge of the tool.

Because we did not require any previous knowledge of security, this missing background could also lead to difficulties here. This was our second focus in designing our ideas, so we expected them to perform better than existing tools.

The whole survey is added in Appendix B. The following section gives the full list of hypotheses we came up with before the study. These helped us to assess the situation and our assumptions after the study.

## 5.3. Hypotheses

- (H1) Our tools (AcTool and MpcM) have a higher total correctness score than existing tools (Cingulata and HyCC).
- (H2) FHE tools (Cingulata and AcTool) have a higher total correctness score than MPC tools (HyCC and MpcM).
- (H3) For all tools, the computation correctness score is higher than the security correctness score.
- (H4) Most people have their worst total correctness score in Cingulata.
- (H5) If people have a high total correctness score in Cingulata, they also have a high total correctness score in the other tools.
- (H6) Existing tools (Cingulata and HyCC) generally are not clear about the number of parties.
- (H7) People are more confident of their assessment of what the code does with our ideas (AcTool and MpcM).
- (H8) The comments are considered more useful with existing tools (Cingulata and HyCC).
- (H9) People who are confident in their assessment of what the code does did not find the comments useful.
- (H10) Cryptography background does not improve the total correctness score.
- (H11) Programming experience does improve the total correctness score.

## 5.4. Demographic

For our pre-study, we recruited eight people from our research group and friend circle to give feedback on the design of the study, as they were the most accessible. For our main study, we recruited our participants by sending emails to students of ETHZ as well as via a mailing list of German students. Because we did not want to assess people's programming skills, we needed all of them to be able to recognise common structures like a loop. We



thus aimed our study at computer science master students, for which this knowledge can be reasonably assumed. We did not require any knowledge about cryptography, because the existing tools, as well as our tool-mockups, are all aimed at people not familiar with cryptography.

This study did not need approval from the Ethics Commission according to ETH regulations, because it is an online survey which collects only opinions and does not concern personal data. We got nineteen responses in total, of which sixteen are students. The following Table 5.2 shows an overview of our participants.

	Number	Percent		Years
<b>Gender</b>			<b>Age</b>	
Female	0	0%	Min	17
Male	18	95%	Max	29
No answer	1	5%	Median	23.5
<b>Security Experience</b>			<b>Programming Experience</b>	
Yes	8	42%	Min	0
No	10	53%	Max	12
No answer	1	5%	Median	6

Table 5.2.: Participants' Demographic (N=19)

## 5.5. How to do a Survey

We now present the guidelines we learned from the research described in Section 4.2, as well as from doing a survey ourselves. Most of these insights come from our pre-study, and we were able to incorporate these into our main study, as well as to some extent in our validation study, presented in Chapter 12.

- (I1) Keep it under 30 minutes, and try to give an accurate time frame upfront. We gave a time frame of 20 minutes, but some people reported needing more than double that. This leads to discontent and some people might break off the study after the advertised time. Also, the longer the study, the less people are motivated to do it, so keeping it short helps to get more responses.
- (I2) Start with easy questions, otherwise people might be intimidated and not take the survey. Our pre-study started with an open-ended question, which some people mentioned was quite the turnoff, because they could not answer it. Starting with easy questions encourages the participant to continue with the study.
- (I3) Use multiple-choice or Likert-scale questions as often as possible. They have two advantages over text-answered questions: first they make it easier to answer and thus more appealing to users, and second they enable the generation of statistics. We had a lot of text questions in our pre-study and could often not get comparable results.

- (I4) Do not pose comparison questions at the end, the user will not remember the whole survey. We included a question at the end of our pre-study that asked the participant to name his favourite tool. The results suggest that we did not get much insight from this, as people did not remember what they liked or disliked about specific tools. We thus opted to include questions about likes and dislikes after every tool and compare them that way.
- (I5) Pose similar questions in different parts of the survey, for easy comparisons later. Our main study had five multiple-choice questions for each tool. Having similar questions throughout the different tools meant that we could compare the user's understanding much better.

## 5.6. Results

One participant answered questions only for one of the four code snippets, and three more participants answered only for three of the four. Thus for Cingulata, we had sixteen answers, for AcTool we had nineteen answers, for HyCC we had eighteen answers and for MpcM we had seventeen answers.

In the following paragraphs, we will highlight some of the most interesting results we got from the study. Because we had a quite small group of participants, a lot of the results are not statistically significant, but still give a good insight.

In Figure 5.3a, we present the results of the multiple-choice questions. The average person got 3.4/5 points in Cingulata, 4.4/5 points in AcTool, 3.1/5 points in HyCC and 3.8/5 points in MpcM. A first observation is that on average questions about our tools were answered better (4.1/5) than those about existing tools (3.2/5), thus validating hypothesis (H1). A comparison between FHE tools and MPC tools also reveals that our hypothesis (H2) was correct: FHE tools have a higher total correctness score of 3.9/5 versus the MPC total correctness score of 3.4/5.

Generally, computation questions have a higher score with 3.9/5 versus security questions with a score of 3.5/5. But there are two tools (Cingulata and MpcM) where security questions have a higher score than computation questions, thus we can not validate (H3). One possible explanation is that for some computation questions, the participant also had to parse the inputs correctly, which is more about understanding cryptography than computation. For example, the first question in Section 5.2 about the result of the computation can only be answered correctly if one correctly identifies the inputs.

One noticeable gap can also be seen between computation and security questions about HyCC. This can be explained by the fact that HyCC takes a complete obfuscation approach, meaning that the user does not know how/what/when the computation happens. While there are clear advantages to this approach, because the user does not need any knowledge of how such a computation should be set up, this also caused great confusion for the user. These results seem to support our theory that clarity is a big problem in existing tools, and our approach to make a more well laid-out and transparent tool is working.

In Cingulata, the first integer in their input specification is the byte length of the rest of the inputs. This is not clear from the code and could easily be interpreted as an

additional input. The multiple-choice question about the final result of the computation thus performed very badly, with only 44% correct answers. For comparison, AcTool had 95% and HyCC 94% correct answers to the same question. This validates our theory that there is room for improvement in how inputs are specified.

We counted for which tool people had their worst correctness score. If multiple tools had an equally low score, we added an adequate percentage to the count. Overall, the tool that most people had their worst total correctness score in was HyCC with 49%, followed by Cingulata with 34%. Hypothesis **(H4)** is thus invalidated.

In Figure 5.3c we plot the total correctness score in one tool versus the total correctness score in the rest of the tools. We notice a slight trend that if people have a high total correctness score in Cingulata, they have a lower total correctness score in the rest of the tools. When looking at this trendline for all tools, we notice that it is the same for HyCC, while for AcTool and MpcM it is reversed, where a high total correctness score in one tool correlates with a high total correctness score in all other tools. This does invalidate hypothesis **(H5)** but is a very interesting result nevertheless.

In Figure 5.3f we plot the percentage of people who correctly answered the question about how many parties are involved. For Cingulata, AcTool and MpcM this question was answered mostly correctly with 88%, 95% and 94% respectively. For HyCC, only 33% could answer correctly that there were two parties involved in the computation. This can again be explained with HyCC's approach to hide as much as possible from the user. The only way to know that there are two parties involved is to read the documentation and know that there are always two parties. This shows us that important information like the number of parties should be clearly visible at all times and validates hypothesis **(H6)**.

In Figure 5.3e, we show what difficulties were the most prevalent for each tool. For three of the tools, obfuscation was mentioned the most, i.e. people struggled with "magic" functions like `revealTo()`. In contrast, for HyCC it was the programming language. We can explain this with the makefile they use for input definition and running, which not a lot of people know how to read. We notice that AcTool had the least problems by far, with seven people even having no unclarities at all. Also, it is interesting to note that security does not seem to be a problem for people. Cingulata has the highest mention of difficulties with the input, which probably comes back to the already mentioned problems with Cingulatas input specification.

In Figure 5.3b we plot how confident people felt about their assessment of what the code does as well as how useful they found the comments. We see that people are most confident with AcTool (4.1/5) but followed by HyCC (3.9/5), which invalidates our hypothesis **(H7)**. The comments were most important with Cingulata (3.2/5), but again followed by one of our tools MpcM (3.1/5), thus again invalidating our hypothesis **(H8)**. What can be clearly seen though is that confidence and comments are reversely correlated and thus validate our hypothesis **(H9)**.

Of the 8 people that indicated they had a security background, only one indicated they found it helpful, thus validating our hypothesis **(H10)**.

In Figure 5.3d we plot programming experience versus their total correctness score. We notice a slight trend that more years of programming experience also lead to a higher total correctness score, thus validating our hypothesis **(H11)**.

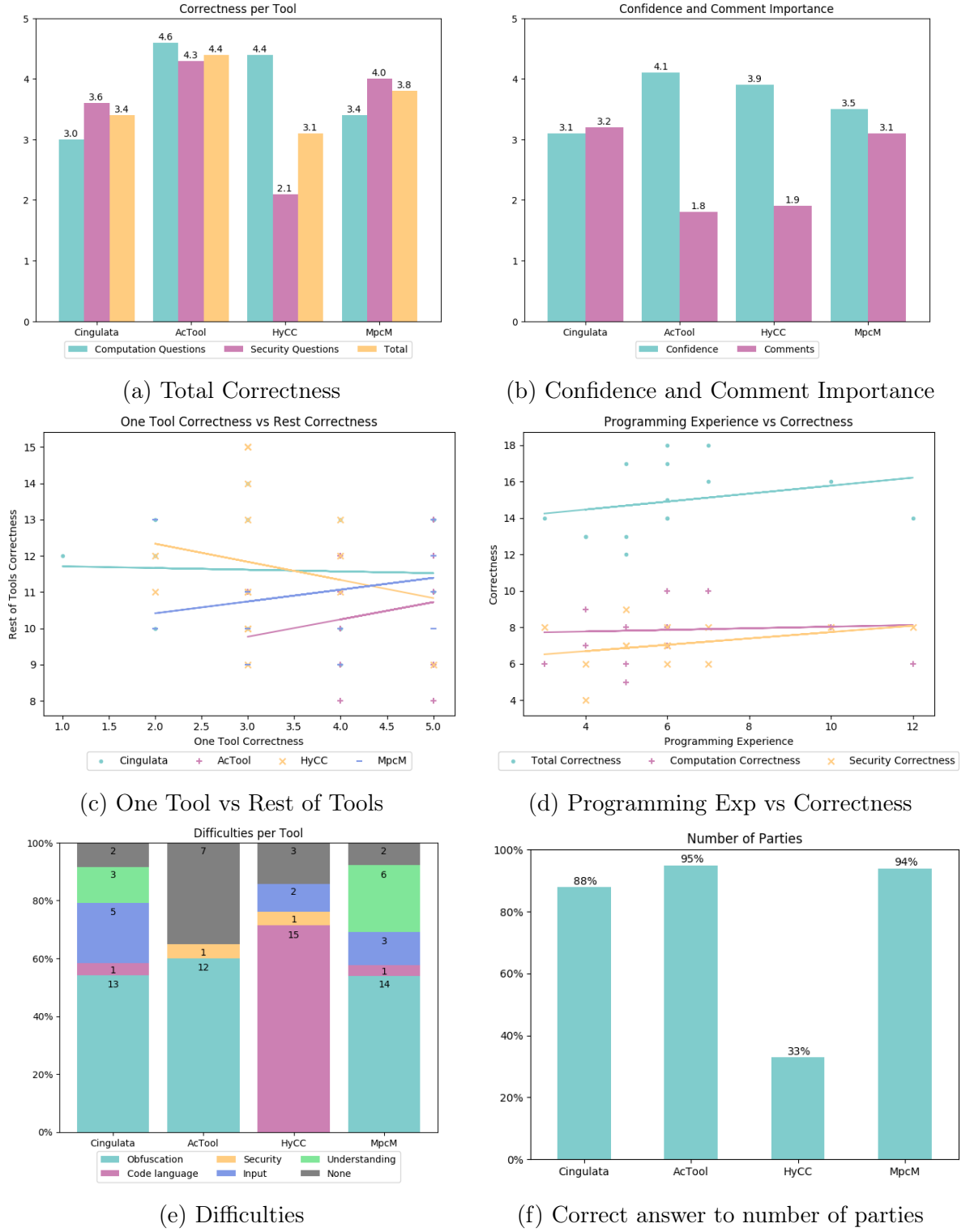


Figure 5.3.: Main results from the user survey

All hypothesis and if they could be validated can be found in table Table 5.4.

	Validated	Invalidated
(H1) Our tools have a higher total correctness score than existing tools.	●	
(H2) FHE tools have a higher total correctness score than MPC tools.	●	
(H3) For all tools, the computation correctness score is higher than the security correctness score.		●
(H4) Most people have their worst total correctness score in Cingulata.		●
(H5) If people have a high total correctness score in Cingulata, they also have a high total correctness score in the other tools.		●
(H6) Existing tools generally are not clear about the number of parties.	●	
(H7) People are more confident of their assessment of what the code does with our ideas.		●
(H8) The comments are considered more useful with existing tools.		●
(H9) People who are confident in their assessment of what the code does did not find the comments useful.	●	
(H10) Cryptography background does not improve the total correctness score.	●	
(H11) Programming experience does improve the total correctness score.	●	

Table 5.4.: All hypothesis and their outcome

# Chapter 6

## Requirements

Based on the results of our preliminary study, we compiled a list of requirements which a useful tool has to fulfil. This tool is aimed at people with a solid understanding of programming, but only surface-level knowledge of cryptography. It should guide the user to write a safe cryptographic protocol that adheres to the user's specifications.

**(R1) Single View**

Everything is defined in a single file and in a single language. Juggling multiple files is a big hassle for the user and introduces unnecessary complexity. Using multiple programming languages would raise the bar on what a user needs to know. Having to switch between different programming languages leads to more clerical errors.

**(R2) Explicit Protocol Participants**

A protocol describes a system with multiple entities that transmit information and perform functions. The combination of users unfamiliar with a tool and implicit definition of important protocol information is very dangerous. For example, the user could unknowingly have set the number of parties to the default. This can lead to errors and also to confusion in the user. Thus, we aimed for explicit definition of all important aspects of a protocol. The first of which is the entities in a protocol. When specifying a protocol, the user usually knows how many of these entities, also called parties, are taking part. Thus they should be able to specify these parties and be able to access this information easily.

**(R3) Explicit Computation Participants**

A computation describes one of the functions that one or more parties are performing. If multiple parties are taking part, a distributed computing scheme has to be employed and depending on the privacy of the data used in the computation, a cryptographic protocol, e.g. a multi-party computation, has to be employed. Furthermore, a computation on one party can be done locally as usual. Thus how many and which parties take part in a computation is one of the defining factors of how this computation will be done. The same reasons for explicit definition apply here as they did in requirement **(R2)**. This information should thus be easily specified and readily available to be accessed.

**(R4) Explicit Output Parties**

A computation usually has an output. In a distributed computing scheme not everyone should necessarily have access to this result, especially in cryptographic settings. The same reasons for explicit definition apply here as they did in requirement **(R2)**. Thus how many and which parties receive the plaintext output of a computation should be easily restricted and readily available to be accessed.

**(R5) Explicit Input Definition**

Input describes any kind of data a party gives as an argument into the protocol. The same reasons for explicit definition apply here as they did in requirement **(R2)**. For each input, it should be obvious what its value is and which party it came from. One of the main results we got from the preliminary study is that users can not easily recognise the inputs. This often makes the user feel overwhelmed and leads to errors in the computation.

**(R6) Explicit Secrecy**

An input can be defined as public or secret. Being secret means that no other party should ever see the plaintext version of this input. The user usually knows which party wants to keep what input private and what input can be shared openly. For each input, it should thus be easy to make it either secret or public.

**(R7) Security Flexibility**

To allow more complex protocols, every type of input should be able to be defined as secret. Most existing tools limit their secret inputs to numbers or arrays of numbers. This is a huge restriction and leads to a lot of hassle for the user. By making sure all data types, for example also user-defined classes, can be marked as secret we allow for maximum flexibility and expressiveness.

**(R8) Adaptability**

A problem we noticed in some tools is that if the user wants to change a small part of their specification, they have to change a lot of existing code. In the worst case, the change in specification leads to a protocol that is not expressible in that tool and thus the user has to completely switch to a different tool and begin the whole process all over. Adapting a protocol to different specifications should be simple and not require more rewriting than necessary.

**(R9) Conformity**

Following established conventions is important. One such convention is to employ a widely used high-level programming language for the computation to make sure that specifying the computation is as effortless as possible. We have seen in the preliminary study that this helps the user quickly familiarize themselves with the tool. This also lowers the bar for the user and helps them to quickly feel comfortable in using the tool.

**(R10) Modularity**

Reusing code is one of the basic principles of programming. But this can only be achieved if functions can be declared and then called in multiple contexts. Thus declaring multiple interacting functions, here called computations, should be possible. Each of these computations should be able to independently declare the parties that take part in it and the parties that get access to the result. This also allows users to single out computations that can be done locally on one party. Because local computations do not have to adhere to a cryptographic protocol, these computations can be done much faster, thus this modularity also leads to improved performance.

**(R11) Accessibility**

Learning the tool is easy and novice users can write simple programs very quickly. This is achieved due to the tool being intuitive and self-explanatory. Also, extensive documentation on all features with examples is provided to facilitate users to learn advanced features.

**(R12) Efficiency**

The most expressive and accessible tool is not useful if it cannot be run in an adequate amount of time. Compile-time should be independent of runtime, meaning that the compile-time does not depend on the inputs to the protocol. Running the protocol should be easy and not require any cryptography knowledge. Runtime should be polynomial and be similar to what a competent user can achieve with an existing tool.

**(R13) Interoperability**

A tool will not find acceptance if it cannot be used in combination with other common appliances around it. Thus, integrating the tool into existing workflows should be easy, for example being able to easily build a docker image.





## Chapter 7

# Domain-Specific Languages

There are many possible ways a tool for advanced crypto can manifest itself. For example, it could be a library, a compilation chain, a framework or also a completely new language. In this section, we now give the reasons why we chose to write a new language and give an overview of so-called domain-specific languages. We also present different approaches to building a DSL and motivate our decision to use a projectional editor.

### 7.1. Domain-Specific Languages

A Domain-Specific Language (DSL) is a language that is built for a specific purpose. It contains elements that directly fit the logic of the problem space instead of providing general building blocks. It is designed for domain experts, which are often not programmers. Thus, a DSL focuses more on usability and notation and less on aspects like Turing-completeness or totality.

Examples of DSLs include SQL, HTML, UML and  $\text{\LaTeX}$ . These are all languages that can only be applied to very specific problems, like database-queries for SQL or word-processing for  $\text{\LaTeX}$ . But in that domain, they are very powerful.

In general, a DSL has several advantages over a general programming language (GPL). Because their syntax is more specific to the domain, it is easier for the user to specify their problem. The restricted problem space allows the developer to force more restrictions on the user to make sure that the user makes fewer errors and the resulting program is safe. When errors happen, the error messages can be more specific and thus help the user to fix their problems faster. Lastly, because the concepts are already familiar to the target group of domain experts, learning a DSL is much easier than learning a GPL [35].

In addition to these advantages, using a DSL gives us the maximum amount of freedom in designing the most user-friendly, accessible and expressive tool that we can think of. A library is bound to a single language and its compiler, thus restricting how far we can modify common high-level programming abstractions like classes or methods. A framework allows for more flexibility and can provide its own compiler, but if we want to make sure a user is adhering to safety principles, it might limit the user in what they can express. In the end, using a DSL makes sure that for each design decision we make, we choose the best possible path without being held back by unnecessary limitations.

## 7.2. Domain-Specific Language Modeling Tools

There are different approaches to building a DSL. We will present the possibilities here and give reasoning as to why we chose a projectional editor in the end. A first possibility is to do everything yourselves by defining a lexer and parser grammar, an interpreter and an editor. This has the advantage of having total control over every aspect of the DSL, but the necessary effort is very high. A better alternative is to use a Domain-Specific Modeling Tool (DSML), which helps to define a DSL by automating or facilitating certain steps in the process. There are three kinds of DSMLs, each supporting another kind of DSL: graphical, textual or projectional.

Graphical languages show the program as a pictorial depiction, for example diagrams, tables or trees. These kinds of languages seem more approachable to non-programmers but are less flexible than textual or projectional languages. There exist several tools for defining graphical languages, like the Graphical Modeling Framework (GMF), the Eclipse Modeling Tools, the browser-based Generic Modeling Environment or the Modeling SDK for Visual Studio. Because we want our tool to be as expressive as possible, a graphical language would not work for the vision we have of our tool.

Textual languages are the most common approach and the vast majority of DSLs are textual languages. They are easy for programmers to grasp and can be used for all sorts of applications. Tools for creating textual languages include XText, textX or the Spoofax Language Workbench. Because we want our tool to be as user-friendly as possible, we want to make sure to present the code in a structured and clearly laid out way, thus a textual language also would not work for us.

Projectional editors show a projection of the underlying file. While you edit on a character-by-character level in a textual editor, you edit on a concept-by-concept level in a projectional editor. This means that even though it looks like text, the changes you make affect the underlying file in a different way than what you perceive. This gives you the intuitiveness of a graphical editor but combined with the flexibility of a textual editor. There are not many tools available, the most prominent being the Meta Programming System (MPS).

A projectional editor combines the advantages of both graphical and textual editors and thus gives us all the properties we need to build a user-friendly and expressive language. Additionally, MPS is a well-established feature-rich editor that has an active community, good tutorials and introductory books. This makes it the ideal workbench to use in this project.

# Chapter 8

## Chisel Design

The design of our tool, Chisel<sup>1</sup>, is shaped by the requirements established from the preliminary study, and the previous design decision to employ a DSL. The language structure that is described in this chapter is achieved by design decisions based on the broad insights into advanced cryptography that we gained from research into related work as well as our preliminary study.

### 8.1. Basic Structure

Our first design decision based on requirement **(R2)** was to make the concept of a party explicit. As already mentioned in the requirements, the implicit definition of important protocol information is very dangerous for users unfamiliar with a tool, because it can lead to errors and also to confusion in the user. A party should be an object with a name, and a group of data that this party owns.

Our second design decision based on requirement **(R3)** and **(R4)** was to divide parties and computations into two parts. Using a domain-specific language allows us to do this, while still keeping all specification in a single file, thus fulfilling requirement **(R1)**. By allowing an arbitrary number of computations, we make the whole protocol modular and fulfil requirement **(R10)**.

**Party** A party is identified via a unique name so that it can be referenced as an executing party or result party. Each party can then specify data, for example an array of integers. But a party can also define their own classes, and then use them to specify data, for example an array of people. This data can be accessed by a computation and used as input. Each input can be accessed via the party's name. This way we instantly know where the input came from and can easily look up the value, thus fulfilling requirement **(R5)**. Each input is public by default, but it can be defined to be secret, which means that other parties should not get unencrypted access to this value without explicit consent. This definition of secret data allows users to indicate that their data is only to be used in a secure computation. This fulfils requirements **(R6)** and **(R7)**. If a party gets the result values of a computation, then this value can also be accessed via the party's name. This allows a protocol to span over multiple computations, thus providing modularity and performance improvements, as explained in requirement **(R10)**.

---

<sup>1</sup>Chisel, as in the chisel tool to work marble, as in the Marble library that will in the future be added as the cryptographic background library.

**Computation** A computation is identified via a unique name so that they can be called from other computations. A computation defines a set of executing parties, which includes all parties that actively take part in the computation. Each computation also defines a set of result parties, which includes all parties that get access to the unencrypted results. These two sets are the backbone of the decision if this computation has to be done as a secure computation, i.e. with advanced cryptography, or not. The executing parties are not to be confused with parties that give input into the computation, as they are not related. For example, in an FHE protocol, one party gives input and the other party does the computation. Neither is this set to be confused with the set of result parties, as they are also not related. A party taking part in the computation does nothing more than lending its computation power. Being a result party of a computation means that from now on you can access these results as if they were your own data. These two sets allow the flexibility to define both FHE and MPC protocols, thus fulfilling requirement **(R8)**.

If the computation accesses data that is defined to be secret, it has to be defined as being cryptographic, which means that no secret inputs will be leaked. Depending on the number of computation parties it will run as an FHE or MPC protocol. Of course, we cannot prevent parties from deducing directly from the output. For example, if the computation was the sum of two secret inputs from two parties, each party could deduce the other parties input from the result.

Each computation is then a series of statements, that can be written in any existing high-level language, thus fulfilling requirement **(R9)**. This series of statements most likely ends in one or more return statements. These return statements mark the output values of the computation. A computation is not required to return any value, e.g. for testing. Having the option to define several return values allows for more freedom and flexibility for the user.

Statements can access secret and public input data via the parties name. All data from all parties are accessible, but parties can make restrictions on whom they trust to be a result party in a computation accessing their secret data. Computations can also access results from previous computations, these are handled as secret if they come from a computation that was marked as cryptographic, otherwise they are handled as public variables.

## 8.2. Automation

To help the user write a safe and sound protocol, we want to automate certain decisions the user otherwise would have to make. This also simplifies the user's life, as some of these are based on cryptographic principles that most people do not know enough about to make an informed decision.

### 8.2.1. Automatic choice of executing parties

The choice of which parties are executing a computation is difficult for a user with no cryptographic background. Are more parties always better? Or is a local computation on one party the best option? If not chosen correctly, it can lead to a massive decline in

performance as well as lost security guarantees. We thus automate this choice based on cryptographic principles, providing the user with an appropriate set of parties. For users that want to experiment or experts that know what they are doing, we still allow them to overwrite this suggestion. These suggestions can be simple, like to use all parties for a computation, or more involved by taking into account each parties computation power and input sizes.

### 8.2.2. Unsafe computation detection

Writing unsafe computations can happen very easily in some existing tools because the difference between secure and non-secure computations is not made clear. A user could forget that a certain data input is actually secret and mistakenly compute with it in plaintext. We want to make sure that such oversights happen as little as possible, and thus try to detect unsafe computations and suggest possible remedies to the user. For example, assume a computation accesses secret data of one party but executes on another. In this case, we suggest to remove the secret input and thus make the computation not require cryptography anymore. A second suggestion is to replace the executing party with the input party in order to make the computation local. A third suggestion would be to make the computation cryptographic and thus guaranteeing that the executing party does not see the secret input.

To determine, which computations currently need changing, we follow the decision tree shown in Figure 8.1. For simplicity, all computations happening on more than one party are automatically determined to be cryptographic, even though there are situations where a simple distributed system would be fine. As our work focuses on cryptographic applications, we believe this assumption to be acceptable. Turquoise hexagons indicate our decision if this computation should be done as a cryptographic protocol or not. Having a crypto modifier on a computation that does not require one leads to a warning, while not having a crypto modifier on a computation that requires one leads to an error. Purple arrows indicate the possible fixes that are applicable if the computation is currently not in the correct state.

### 8.2.3. Division detection

Doing a division involving one or more secret inputs is very expensive because it cannot be built from addition and multiplication, which are the two operations that are available in FHE and MPC. It thus has to be done with a deep and complex binary circuit with a runtime that is way above the rest of the operations. Such divisions can thus easily lead to performance problems, so we issue a warning to ensure the user is aware of this and is certain of their decision to use a division. We do not want to prohibit the usage of division, as that would restrict the user too much.

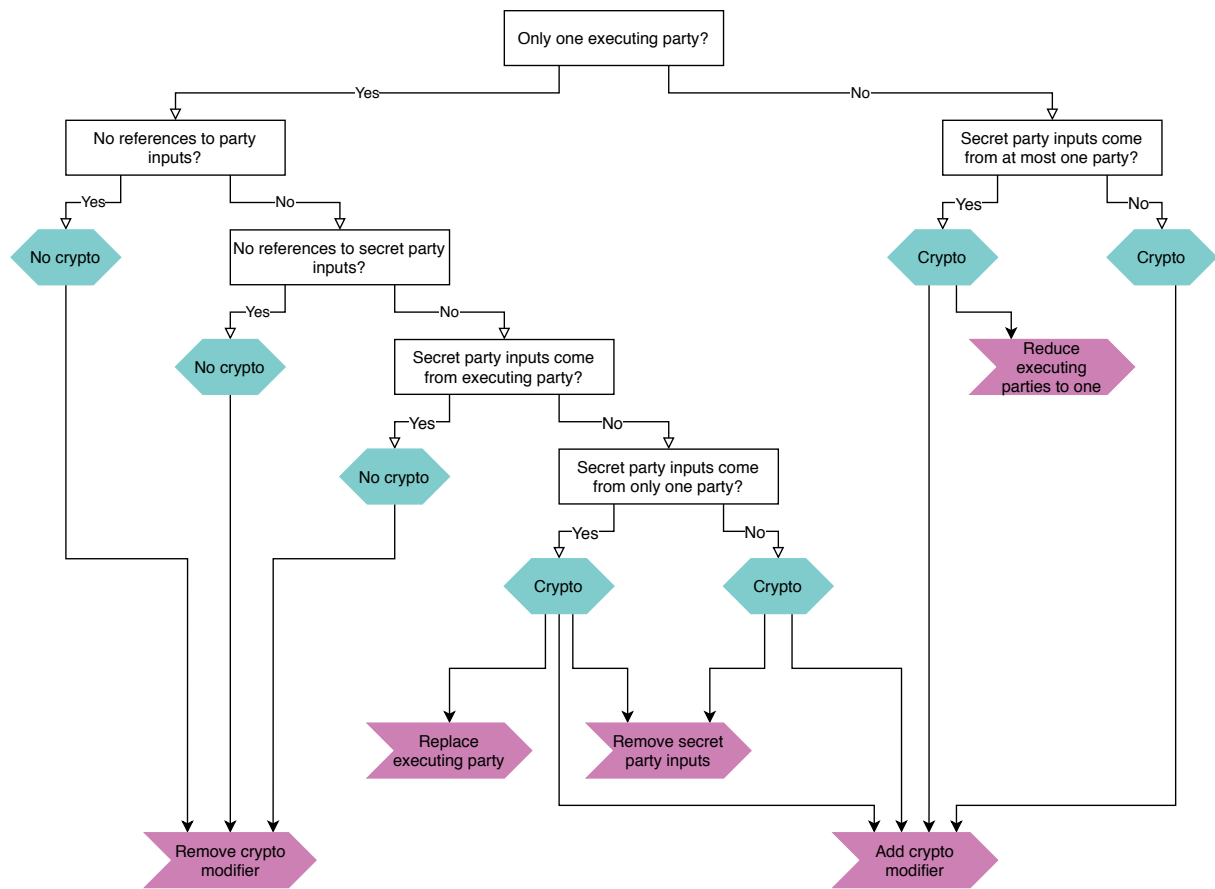


Figure 8.1.: Decision Tree for Computations

## Part III

# Implementation

*The most important property of a program is whether it accomplishes the intention of its user.*  
— Sir Charles Antony Richard Hoare

In this part, we will first give more information about projectional editing and its advantages and drawbacks. Then we go through the most important information on how a language is defined in the domain-specific language tool MPS. The second chapter presents how MPS was used to implement Chisel as a domain-specific language.





## Chapter 9

# MPS and Projectional Editors

In this chapter, we will first give more information about projectional editing and its advantages and drawbacks. Then we go through the most important information on how a language is defined in the domain-specific language tool MPS.

### 9.1. Projectional Editors

As we already mentioned in Section 7.2, MPS works with projectional editors. In a traditional programming language, the user writes his program in a text editor. A compiler then uses lexers and parsers to read it and transform it into an Abstract Syntax Tree (AST), the data structure that represents a program. This tree starts with a root node, and every node can contain an arbitrary amount of child nodes. In MPS, the user instead directly writes the ASTs of his programs using the projectional editor.

A projectional editor has one notable advantage: because there is no parsing involved, the user can express his program in ways other than text. For example, the language designer could utilise tables or diagrams to make it easier for users to specify their program. Another advantage of not relying on parsers is that any combination of languages is syntactically valid. With projectional editors, one can target non-professional developers by providing form-like notations, which make it clear to the user where exactly they should input their information. One can also offer multiple notations for the same language, targeted at different users or use-cases. Lastly and most important for our project, it is much easier to extend an existing language by adding some new AST concepts, instead of having to extend the lexer and parser.

### 9.2. Modules in MPS

MPS structures its files into *models*, which are then grouped together into *modules*. There are four different kinds of modules: solutions, languages, devkits and generators.

- There exist several types of *solutions*. Sandbox solutions contain end user code. Runtime solutions contain code other modules depend on. Plugin solutions extend the functionality of the Integrated Development Environment (IDE), like adding a new menu entry.
- The actual concepts making up the language are defined in the *language* module. We will look at this module in more detail in the next section.

- *Devkits* bundle several languages into one item, so that they can, for example, be imported easier.
- *Generators* define how one language should be transformed into another. Writing a generator is a time-consuming and complicated process which would go beyond the scope of this thesis. Because we did not write a generator for our language, we will also not go into further details here.

## 9.3. The Language Module

The language module contains 17 different models, each of which defines a corresponding aspect of the language. We will look at the seven most fundamental models in the following subsections, which are the ones we used to define Chisel.

### 9.3.1. Structure

The structure model allows defining what structures of ASTs should be allowed in the language.

The most important basic building block of the AST is called a *concept*. A concept defines the structure of a node in the AST by specifying its properties, children and references. Properties are single named values with a type that is either a primitive (e.g. boolean, string or integer), an enumeration type or a constrained data type. The latter two will be explained further down.

Children are other nodes hooked to a node to form a tree. Each specified child represents a group of children defined by a name, a concept as the type for all children and a cardinality. The cardinality specifies how many nodes can be in the group and is written in UML style. The available cardinalities are [1], [0..1], [1..n] or [0..n], meaning exactly one, zero or one, one or more, or zero or more respectively.

References are used to connect nodes that are not directly in a parent-children relationship. They are defined by a name, a concept as the type and a cardinality. The cardinality for references can only be either [1] or [0..1].

A concept can extend another concept, much in the same way as subtyping works in languages like Java. There are *concept interfaces*, of which a concept can implement many, again working similarly as to any high-level language like Java. A root concept is one that can be used as an AST root node. A concept can be defined as a root concept, by setting „instance can be root“ to true.

An example concept defining a computation can be seen in Listing 9.1. It extends a concept, implements two interfaces and contains one boolean property, three children and no references.

---

```

concept Computation extends      GenericDeclaration
                        implements  IMethodLike
                        ScopeProvider

instance can be root: false
alias: <no alias>
short description: <no short description>

properties:
isCryptoComp : boolean

children:
computationParties : PartyReference[0..n]
resultParties      : SinglePartyReference[0..n]
body               : StatementList[1]

references:
<< ... >>

```

---

Listing 9.1: Example Concept

Instead of a new concept, one can also define a new *enumeration*, *constrained data type* or *primitive data type*. An enumeration contains several members, of which each is defined by a name and presentation string. It is used to define a new enumerated data type for which we know all possible values. Constrained data types are new string types constrained with a regular expression to limit the range of acceptable values. New primitive data types can be defined to extend the three existing ones - *int*, *boolean* and *string*- just by specifying a name for it. This is not recommended though, because it requires a lot of hardcoding.

### 9.3.2. Predefined Concepts

MPS defines a concept *BaseConcept* that all concepts extend, much like *Object* in Java. It also defines the three primitive types boolean, integer and string. Some widely usable interfaces are also already defined, for example the *INamedConcept* that defines a string property called „name“.

Besides these basic structures, MPS also comes with a complete implementation of the Java language as an MPS language, referred to as the *BaseLanguage*. This language defines everything from *ClassConcept* over *StaticMethodCall* to *LocalVariableDeclarationStatement*. *BaseLanguage* is very well suited to extend from, because it provides every building block of a high-level object-oriented language that can then be built upon.

### 9.3.3. Editor

It would be unintuitive and cumbersome to always manipulate the AST directly. Thus, one can define one or more editors for each concept of the language to display it more naturally. Each editor consists of editor cells, which render a node into the MPS editor window. A cell is a distinct part of the display and there are many different editor cells

available, from constant cells containing a string, over property, child and reference cells to collection cells containing a list of editor cells themselves. To describe an editor, one uses a dedicated language called *editor language*.

The most basic editor is the *concept editor*. It is responsible for rendering and editing one node for one concept. One can also define multiple concept editors for the same concept to offer different views. The choice which editor is shown is done by *context hints* that are either pushed by the end-user or specified to be active in certain places in the code by the language designer. A user could for example switch between a textual and tabular representation of some data, or the designer could want a different representation of concept A depending on whether it is a child of concept B or concept C. It is also possible to create *editor components* which are responsible for rendering and editing parts of a node. Such an editor component is then used like any editor cell. This is useful if one has several concept editors with parts that need to be rendered similarly. An example editor to the concept in Listing 9.1 can be seen in Listing 9.2.

---

```
<default> editor for concept Computation
node cell layout:
  [–
    {name} ?^ crypto {
      Executing Parties = (> % computationParties % /empty cell:<default> < )
      Result Parties = (> % resultParties % /empty cell:<default> < )
      % body %
    }
  –]

inspected cell layout:
  <choose cell model>
```

---

Listing 9.2: Example Editor

In the editor model, one can also add actions for specific cells. One way to do this is by defining a *cell action map*, where one can specify what happens if a certain action is executed on the cell, for example if it is clicked on or deleted. For example, one could implement that deleting keyword „static“ also toggles property *isStatic* from true to false. Another way is with a *cell key map*, which is similar to the cell action map but reacts to keystrokes. For example, typing „[“ after a type transforms it into the corresponding array type.

*Transformation menus* are another aspect defined in the editor module. They help the user interact with the AST, by transforming a node when a specific text is typed beside the node. For example, typing „final“ after the „static“ keyword transforms „final“ into a keyword and marks the associated definition as *isFinal*.

Displaying different language elements like keywords, constants and comments in different colours and fonts helps with the readability of the code. The presentation of each editor cell can be set independently or in so-called stylesheets to be used in many cells.

### 9.3.4. Actions

The action aspect allows to define actions that modify the AST, beyond the simple editor actions. Paste wrappers make it possible for the end user to paste code into the AST by modifying the AST according to the type of the pasted fragment. If a user for example wanted to copy a *LocalVariableDeclaration* directly into a *ClassConcept*, it should be transformed into a *FieldDeclaration* automatically. Copy-paste handlers give you the possibility to customise what exactly is being copied or pasted. For example, if an expression is copied from one class to another, all references should be re-resolved to be valid in the new context.

### 9.3.5. Constraints

The constraints aspect can be used to define additional constraints on the language structure that are not expressible in the structure model. One can indicate whether the current concept can be a child, parent or ancestor node of another node in the AST. This can be useful if for example you want to restrict the types of expressions that can be used in a concept. Additionally, one can define constraints on properties and references, for example restricting the scope of a reference.

Finally, one can define the default scope of the concept. This default scope is used if any link points to an instance of this concept and does not define a scope itself. When MPS needs to set a reference on a node, it looks up the associated scope constraint. Scopes in MPS are a way to locate suitable targets for a specific reference. For that, a scope defines a set of node instances that are allowed as targets for this reference. If no scope is found, MPS considers all compatible nodes as allowed. If a scope is found, it is used to restrict the set of nodes that are allowed. Two different kinds of scopes can be defined: inherited scopes and reference scopes.

To use an *inherited scope*, one has to have another concept which implements the *ScopeProvider* interface and with it certain resolution mechanisms. The inherited scope then refers to that concept for scope declaration. To implement the *ScopeProvider* interface, one has to implement a *getScope* method, which returns a scope constructed from nodes that are in the AST. *Inherited scopes* are for example useful for single references to delegate the scope to their declaration.

A *reference scope* directly provides a function that calculates a scope from the current concept. This is faster but less scalable. An example constraint that restricts the scope of a reference can be seen in Listing 9.3.

### 9.3.6. Behaviour

The behaviour aspect makes it possible to add *BaseLanguage* code onto concepts to define constructors and methods on nodes. The code in the constructor will be executed whenever a new node of this concept is created. This is useful to initialise some properties, children or references of the node. Methods are used to add often used utility, in order to simplify the code.

---

```

concepts constraints SinglePartyReference {
  can be child <none>

  can be parent <none>

  can be ancestor <none>

  <<property constraints>>

  link {ref}
    referent set handler <none>
    scope (referenceNode, contextNode, containmentLink, position, linkTarget,
      operationContext)→Scope {
      node<Protocol> protocol = contextNode.ancestor<concept = Protocol, +>;
      return ListScope.forNamedElements(protocol.parties);
    }
    <no presentation (deprecated)>

    default scope <no default scope>
  }
}

```

---

Listing 9.3: Example Constraints

### 9.3.7. Type System

The type system aspect makes it possible to calculate types of AST fragments, as well as perform static checks. This is done through different kinds of rules. In an *inference rule*, one can define the type of a concept by specifying equations and inequations. An example inference rule that defines one equation can be seen in Listing 9.4.

---

```

inference rule typeof_SinglePartyReference {
  applicable for concept = SinglePartyReference as singlePartyReference
  applicable always
  overrides false

  do {
    typeof(singlePartyReference) ::= typeof(singlePartyReference.ref);
  }
}

```

---

Listing 9.4: Example Inference Rule

MPS' type system contains two different versions of subtyping: weak and strong, and one can use the *subtyping rules* to define both kinds of sub-typing relations between types. *Comparison rules* are used to define whether and how two types are comparable. Other rules define how overloading operators work, provide a way to efficiently solve inequations and make it possible to substitute a type for another one. Because our language did not introduce any types and we thus did not use any of these rules, we will not go into further detail here.

The type system aspect is also used to define and report semantic errors to the user of the language by using the *checking rules*. These rules can be applied as errors, asserts, infos and warnings and can reference e.g. styling, performance or correctness. This is a good way to make the user aware of any problems as early as possible. One can also define *quick fixes* for these semantic errors so that the end-user can automatically and quickly fix the error in a proposed manner. An example checking rule that tests if a class is static, and an applicable quick fix that makes it static if needed can be seen in Listing 9.5 and Listing 9.6.

---

```

checking rule checkStatic_ClassConcept {
  applicable for concept = ClassConcept as classConcept
  overrides <none>

  do {
    if ((classConcept as PartyDeclaration == null) && (classConcept.parent as
      PartyDeclaration != null)) {
      if(!classConcept.isStatic){
        error "Top level fields have to be static" -> fieldDeclaration;
      }
    }
  }
}

```

---

Listing 9.5: Example Checking Rule

---

```

quick fix makeClassStatic

arguments:
node<ClassConcept> classConcept

fields:
<< ... >>

description(node)->string {
  "Make class static";
}

execute(node)->void {
  classConcept.isStatic.set(true);
}

```

---

Listing 9.6: Example Quick Fix



### 9.3.8. Intentions

Intentions provide fast access to some operations that change the syntax of the language. Examples include inverting an if condition, creating a variable that is referenced or adding a modifier. The intentions aspect allows one to define intentions for one's own language. There are two different kinds of intentions in MPS: regular intentions and surround-with intentions.

An example intention that adds or removes a modifier can be seen in Listing 9.7.

---

```
intention AddEncryptedCompModifier for concept Computation {
    error intention : false
    available in child nodes : false

    description(node, editorContext)->string {
        node.isCryptoComp ? "Remove 'crypto' modifier" : "Add 'crypto' modifier";
    }

    <isApplicable = true>

    execute(node, editorContext)->void {
        node.isCryptoComp.set(!node.isCryptoComp);
    }
}
```

---

Listing 9.7: Example Intention

### 9.3.9. Others

Some aspects we will not go into further detail here are the data flow aspect, the text generation aspect, the refactoring aspect, the migrations aspect, the scripts aspect, the test aspect, the feedback aspect, the find usages aspect and the plugin aspect. There also exists the possibility to define custom language aspects.

# Chapter 10

## Chisel Implementation

In this chapter, we describe how we implemented Chisel as a domain-specific language in MPS. We first describe the basic structure of the language and then go into details of the implementation of parties and computations. For reference, a diagram showing the concept structure of our final language design, and how our concepts interact with the existing *BaseLanguage* concepts, is shown in Figure 10.1.

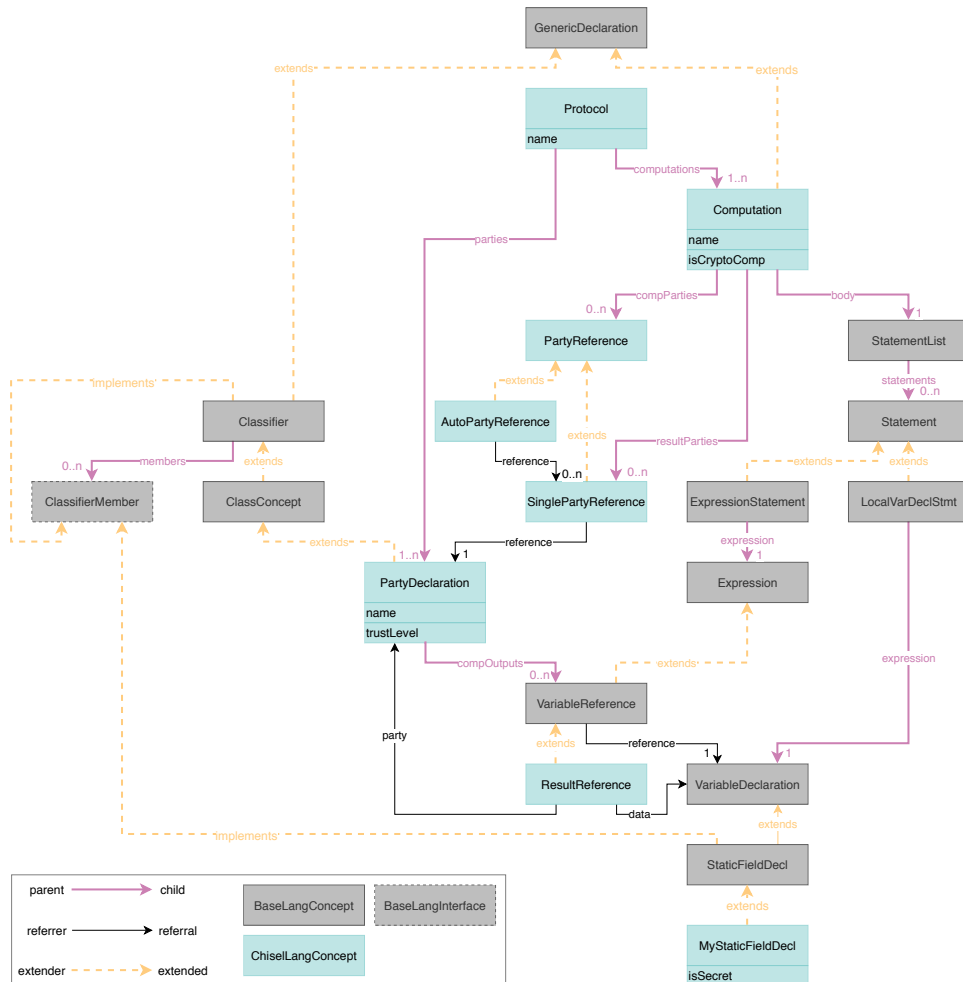


Figure 10.1.: Chisel Class Structure

## 10.1. Basic Structure

The top-most concept in Chisel is a protocol, as seen on the top of the class structure diagram. Starting a new project always begins by creating a new protocol. This makes sure that the same structure is kept for all programs and thus improves consistency and readability. The protocol has the same name as the file name and is mostly there to help the user distinguish their protocols. As per our design decision to separate parties and computations, the protocol has two children „parties“ and „computations“. Because it is sensible that a protocol needs one of each and to serve as a starting point for the user, both children have cardinality [1..n]. We give more details on the implementation of the *Party* and *Computation* concepts in the two following sections.

Our implementation thus started with these three concepts. At this point, we could already define a protocol like the one seen in Listing 10.2. This code sample defines three parties each with one input, as well as four simple addition computations.

---

**Protocol:** CounselorTest

**Parties:**

```
Counselor { input: [ 0, 1, 1, 0, 0, 0 ], trusts: Government, MedicalCenter }
MedicalCenter { input: [ 10, 20, 10, 50, 1, 90 ], trusts: Government }
Government { input: [ 1000, 1000, 2000, 5000, 1000, 2000 ], trusts: <<...>> }
```

**Computations:**

```
Counselor + MedicalCenter done by [ Government ]
Counselor + MedicalCenter done by [ MedicalCenter ]
Counselor + MedicalCenter done by [ MedicalCenter, Government ]
Counselor + MedicalCenter done by [ Counselor ]
```

---

Listing 10.2: Language Draft

This design was extremely restrictive in terms of inputs and computations that can be specified. Each party was synonymous with its single fixed input, which had to be a vector of numbers. Therefore, even if we allowed more complex arithmetic expressions in the computations, it would not be easy to express realistic scenarios where parties can specify different (secret) inputs and/or have public information.

We therefore decided to turn parties into a structure type that can contain a variety of different member variables with different secret/public settings. We implemented this by using a Java class to represent each party, and only allowing static members. Being static, each class element has to be accessed through the class directly. The party class itself can not be instantiated and thus only one of each party exists.

To allow more complex computations, we turned computations into a series of statements, similar to a Java function. This way we can use all available Java statements to define more complex computations, e.g. definitions, comparisons, loops and return statements. The big difference to a normal function is that a computation does not take any parameters. Instead it can access all variables defined in all parties to use as input, in order to serve as the connection between parties and computations. According to our design, we also wanted to make it clear what the executing parties and result parties of each computation are and thus added these as children to the computation.

To help us determine how complete our design was, we did a check for feature completeness, as will be described in Chapter 11. This led us to discover some missing features that could be traced back to two main limitations: parties can not define new types/classes and the result of a computation is not available as an input on the parties receiving it. In the current design, we addressed all these problems to arrive at a point where a meaningful computation could be specified without errors.

**Matrices** Matrices are extremely common in secure computation, so we wanted to support them in our tool as well. The MPS *BaseLanguage* does not have a way to specify matrices natively, so we had to write our own separate language *Matrix*. We made this a separate language to allow more modularity and to not overload the Chisel language with non-cryptographic concepts. Users can specify matrices in an easy, inline way and have basic operations like addition, subtraction and multiplication available. We only added the basic operations that are necessary for specifying most algorithms, but it would be easy to extend *Matrix* by more advanced matrix operations like inversion or determinant computation.

**Keywords and Styling** Using a projectional editor adds a lot of uneditable keywords into the code, e.g. because sections of the code are separated by headers. To make it easier for the user to distinguish these keywords from editable code, we implemented a colour scheme in the form of a stylesheet. We differentiate between three kinds of keywords. The first kind is a custom editable keyword, for example the result parties or the *secret* annotation. These keywords are displayed in **turquoise text**. The second kind is a custom uneditable keyword, for example the aforementioned section headers. These keywords are displayed in **purple text**. The third kind is uneditable text, like autogenerated information and is written in *grey italics*. Parts that are „normal“ code use the Java syntax highlighting scheme. We confirmed these choices with a small survey, where 75% preferred this colour scheme over other ones.

**Deployment Runtime Semantics** Our goal is to implement a benchmarking tool, that is why we keep all parties with all inputs in the same file. If we want to add a backend to create a deployable solution, we need to make several adjustments to our language, because it needs slightly more information. These changes are important but are all trivial to implement. All parties together with their inputs would still be defined in the same file, but without instantiating any runtime inputs. This means that especially all secret inputs would not be instantiated. Each party would then have to create a singleton instance of their corresponding party class and provide the needed inputs. We would thus also stop declaring all inputs as static and instead only define those as static that can be instantiated at compile time, i.e. stay the same for all runs of the protocol.

## 10.2. Parties

To define a party, we declared concepts *PartyDeclaration* and *PartyReference*.

**PartyDeclaration** *PartyDeclaration* extends *ClassConcept*, meaning it is the equivalent of a Java class. This way, we can reuse lots of functionality. *PartyDeclaration* can be found on the left in the class structure diagram as a child of *Protocol*. Each *PartyDeclaration* has an enumeration property *CorrectnessTrustLevel*, which can be either trusted, semi-honest or malicious. These values mark how trustworthy each party is, and could affect if computations on these parties have to be done securely or not. A *PartyDeclaration* also has as children [0..n] *VariableReferences* that are the computation results the party has access to. This makes it easy for the user to see and reference those. By using a checking rule, we inform the user when this list gets outdated, and the user can update it via a quick fix. In the future, we would like to automatically apply these changes. At the moment, such rules lead to crashes because they are constantly active. This could be fixed if they could be activated like an event listener, but this is currently not possible in MPS.

The concept *ClassConcept* extends the concept *Classifier*, which has so-called *ClassifierMembers* as children. *ClassifierMember* is an interface and is implemented by most other concepts that can be members of a class, for example other classes, fields and methods. We used these members as a way to specify the input data of a party because we could reuse a lot of *BaseLanguage* functionality by doing this.

To make sure the inputs could be accessed via the party name, we restricted *PartyDeclaration* to only allow static fields or static classes as direct members by utilising a checking rule. In a runtime scenario, encrypted inputs would have to be sent between the parties before any computation is done. We can thus assume that each party also has access to all public variables at the beginning of the computations, because these unencrypted values are a magnitude smaller than the encrypted ones and can be sent basically for free. For this reason we do not have to support „real static“ values that do not change between runs. Because the protocol is specified in one file, using the access modifiers (private, protected and public) does not add functionality. For this reasons and to improve readability we set every newly defined member to protected and do not show the user this information.

As per our design, we wanted the possibility to define each input as either secret or not. We cannot change any *BaseLanguage* concepts, so to achieve this functionality, we added a new *MyStaticFieldDeclaration* that extends *FieldDeclaration* and added a boolean property indicating whether it is secret. To make sure only our new declarations will be used, we automatically transform each declaration that is written in the body of the party into a non-secret *MyStaticFieldDeclaration* by using a checking rule and automatically applied quick fix. In contrast to the quick fix above, this one can be done automatically, because it only has to be activated once when a new *FieldDeclaration* is created.

Our editor for *PartyDeclaration* is simple, it first prints the trust level and the name, then in brackets follow the inputs and finally the list with computation results. As mentioned above, we did not use the access modifier and thus hid it by adding custom editors to the affected *ClassifierMembers* to remove the visibility annotation. This editor was selected by adding a context hint to the editor of *PartyDeclaration*.

**PartyReference** *PartyReference* is an abstract concept and is extended by two concrete concepts *SinglePartyReference* and *AutoPartyReference*. *PartyReference* as well as its extensions can be found in the middle of the class structure diagram as children of *Computation*. *SinglePartyReference* simply has one reference to a *PartyDeclaration* and is thus used everytime a party has to be referenced, for example when accessing an input. *AutoPartyReference* has [0..n] *SinglePartyReferences* as children and is used for representing the automatically computed set of executing parties, as explained in Section 8.2.1. An example of some party definitions can be seen in Listing 10.3.

---

```
Protocol = MyProtocol
```

```
Parties:
```

```
semi-honest FirstParty {
    static int input = 1;
    static secret double secretInput = 2.0;
Variables from Computations: << ... >>
}
semi-honest SecondParty {
    static class myClass {
        static int classInt;
        myClass(int myClassInt){
            myClass.classInt = myClassInt;
        }
    }
    static secret myClass[] array = {new myClass(1), new myClass(2), new myClass(3)};
Variables from Computations: << ... >>
}
```

---

Listing 10.3: Example Parties

## 10.3. Computations

Each computation is represented by a single *Computation* concept, found on the right side of the class structure diagram as a child of *Protocol*. A computation has a name, a boolean property indicating if it is cryptographic or not and three children: The first child is a list of [0..n] *PartyReferences* as the executing parties, the second child a list of [0..n] *SinglePartyReferences* as the result parties and the third child is a single *StatementList* as the body of the computation. The last child makes a computation essentially behave the same as a pure Java function without any arguments and without explicitly saving the return value.

The first list of executing parties is the only place where a *AutoPartyReferences* can be used to display an automatically determined suitable list of parties. For the user's convenience, such a concept is automatically inserted if the field is left empty. If the computation has at least one return party, there has to be a return statement in the body of the computation. Otherwise, the parties would not get a result from the computation and thus would not make sense to be specified. The value of the return statement marks the result of the computation, which is made accessible to all return parties. This access happens via the parties' and values' name. We thus have to make sure the return value

has a name. To achieve that, the return statement has the additional condition that it cannot return a value directly but instead needs a named value. This value can then be accessed in another computation via the party's name. Because these variables are not defined in the class of the party and are thus out of scope, we could not access them using the existing static member access with a dot operator.

We instead wrote a custom built-in function *ResultReference* that takes a party and a variable from that party's "Variables from Computations" list and returns a reference to that value. See Listing 10.4 for an example of how *ResultReference* is used. One could also specify a custom dot operator to access these variables, but for clarity reasons we did not want to do this kind of overloading in our language.

In summary, there are two ways that a computation gets input: static members of parties or results of previous computations. We assume that all parties come together to execute all computations in order, i.e. we assume that there exists a global main function that calls the computations from top to bottom. If we allowed each party to have their own local main function, we could get problems if different parties specified different orders.

In order to access cryptographic functionality, we have to connect our language to some cryptography library. This library compiles the user-specified computations into cryptographic protocols and thus these computations have to be available at compile time. Computations thus cannot be defined as secret. If a party needs to specify secret computations, they can achieve this by multiplying with a secret matrix.

Our editor for *Computation* first prints the computations name and the crypto modifier if the computation is marked as such. Then there are two custom editable keywords „Executing Parties = “ and „Result Parties = “. Finally, the body of the computation is printed.

An example of some computation definitions can be seen in Listing 10.4.

---

**Computations:**

```
FirstComputation {
    Executing Parties = FirstParty
    Result Parties = FirstParty
    double square = FirstParty.secretInput * FirstParty.secretInput;
    return square;
}
SecondComputation crypto {
    Executing Parties = auto // auto: FirstParty, SecondParty
    Result Parties = FirstParty, SecondParty
    double result = SecondParty.array[0].classInt * Result Reference ( FirstParty ,
        square );
    return result;
}
```

---

Listing 10.4: Example Computations

## Part IV

# Evaluation

*The truth is, most of us discover where we are heading when we arrive.*  
— Bill Watterson

In this part, we first describe the two different evaluations we did on Chisel. The first one tests the expressiveness of Chisel, while the second one tests the usability of Chisel. Finally, we end this thesis with a conclusion including possible future work.





# Chapter 11

## Evaluation of Expressiveness

To test our language for feature completeness, we defined a medical task, complicated enough to use all functionality of Chisel, while still being a realistic application. We also needed this task to be solvable using either FHE or MPC, so that we can compare the solutions to both FHE and MPC tools. The implementation of this medical task in Chisel can be found in Section 10.2 and Section 10.3.

### 11.1. Task Specification

We define three parties in this task: patient  $A$ , pharmaceutical company  $B$  and hospital  $C$ .  $A$  is taking one of  $B$ 's drugs and is showing some symptoms, potentially as a reaction to the drug.  $A$  wants to find out if those symptoms are related to the drug, and whether or not they are at risk of developing severe side effects.  $B$  has a database from a drug trial that maps genomes to symptoms during an illness.  $C$  has a database which maps symptoms of ill people to their possible illnesses.  $A$ 's symptoms are early symptoms, and thus do not match the symptoms in  $C$ 's database, which were recorded when the patient was already ill. Thus,  $B$ 's data is needed to connect them. See also the following graph Figure 11.1.

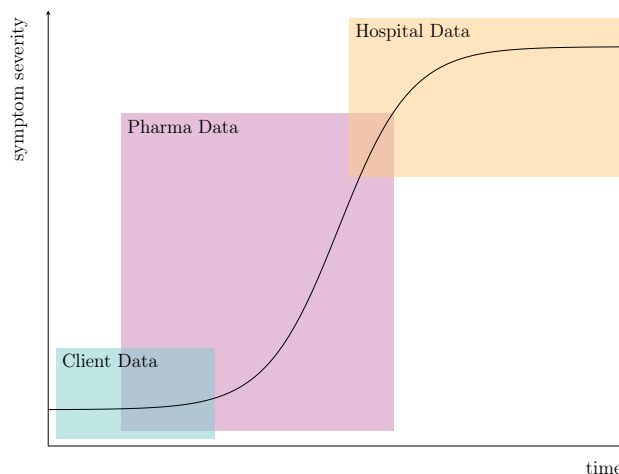


Figure 11.1.: Time versus severity of symptoms and which party owns what data.

## 11.2. Party Inputs

**Patient A**  $A$  has a list of their early symptoms  $symptoms_A$ , as well as their sequenced genome  $g_A$ .

**Pharmaceutical company B**  $B$  has a private list of patients from the drug trial with their sequenced genomes and the symptoms that were recorded throughout the trial.

$$patients_B = \begin{bmatrix} p_0^B \\ p_1^B \\ \vdots \\ p_q^B \end{bmatrix} = \begin{bmatrix} genome_0^B & s_{00}^B & s_{01}^B & \dots & s_{0n}^B \\ genome_1^B & s_{10}^B & s_{11}^B & \dots & s_{1n}^B \\ \vdots & \vdots & \vdots & & \vdots \\ genome_q^B & s_{q0}^B & s_{q1}^B & \dots & s_{qn}^B \end{bmatrix} \quad (11.1)$$

In this list,  $s_{jy}^B$  is a vector of symptoms that patient  $j$  had at time  $y$ .  $B$  also has a public function  $f_B$  that takes a list of symptoms as well as  $patients_B$  and returns a list of likely future symptoms.

**Hospital C**  $C$  has a private list of patients with their sequenced genomes and the symptoms that were recorded when the patient came in for an illness, as well as associated illnesses.

$$patients_C = \begin{bmatrix} p_0^C \\ p_1^C \\ \dots \\ p_r^C \end{bmatrix} = \begin{bmatrix} genome_0^C & s_{00}^C & s_{01}^C & \dots & s_{0m}^C & k_{00}^C & k_{01}^C & \dots & k_{0m}^C \\ genome_1^C & s_{10}^C & s_{11}^C & \dots & s_{1m}^C & k_{10}^C & k_{11}^C & \dots & k_{1m}^C \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ genome_r^C & s_{r0}^C & s_{r1}^C & \dots & s_{rm}^C & k_{r0}^C & k_{r1}^C & \dots & k_{rm}^C \end{bmatrix} \quad (11.2)$$

In this list,  $s_{jy}^C$  is a vector of symptoms that patient  $j$  had at time  $y$ , which was later determined to be illness  $k_{jy}^C$ .  $C$  also has a public function  $f_C$  that takes a list of symptoms as well as  $patients_C$  and returns a list of possible illnesses.

**Implementation** The following listing Listing 11.2 presents how these parties can be written in Chisel. Notice that we were able to specify the input exactly as defined above, with the only difference being that we added concrete dummy input to show how inputs and especially matrices are defined.

---

**Protocol** = MegaTask

**Parties:**

```

semi-honest A_Patient {
  static Matrix<Integer> symptoms_A = |1|
                                     |0|
                                     |1|;

  static secret double genome_A = 1346;
Variables from Computations: << ... >>
}

semi-honest B_Pharma {
  static class patientB {
    static double genome;
    static Matrix<Integer> symptoms;
    patientB(double myGenome, Matrix<Integer> mySymptoms){
      patientB.genome = myGenome;
      patientB.symptoms = mySymptoms;
    }
  }

  static secret Matrix<patientB> patients_B = |new patientB(1234, |1,1,0|)|
                                              |new patientB(4532, |0,1,1|)|
                                              |new patientB(7853, |1,0,0|)|;

  static Matrix<Double> f_B = |1,4,2,4|
                             |4,1,6,7|
                             |5,1,6,1|;
Variables from Computations: << ... >>
}

semi-honest C_Hospital {
  static class patientC {
    static double genome;
    static Matrix<Integer> symptoms;
    static string illness;
    patientC(double myGenome, Matrix<Integer> mySymptoms, string myIllness) {
      patientC.genome = myGenome;
      patientC.symptoms = mySymptoms;
      patientC.illness = myIllness;
    }
  }

  static secret Matrix<patientC> patients_C = |new patientC(4324, |0,0,1|,"illX")|
                                              |new patientC(5785, |1,1,1|,"illY")|
                                              |new patientC(9876, |0,1,0|,"illZ")|;

  secret static Matrix<Double> f_C = |5,5,3,1|
                                    |6,7,9,3|
                                    |7,9,0,1|;
Variables from Computations: << ... >>

```

---

Listing 11.2: Medical Task Parties

### 11.3. FHE Protocol

We now detail how this task can be solved by using FHE. We use notation  $Enc_X(y)$  to describe a value  $y$  that is encrypted with  $X$ 's secret key.

1.  $A$  encrypts  $g_A$ , then sends  $symptoms_A$  and  $Enc_A(g_A)$  to  $B$ .
2.  $B$  computes

$$symptoms_B = f_B(symptoms_A, patients_B) \quad (11.3)$$

which returns a list of symptoms people develop when they have  $symptoms_A$  as early symptoms.

3.  $B$  encrypts  $symptoms_B$ , then sends  $Enc_B(symptoms_B)$  to  $C$ .
4.  $C$  computes

$$Enc_B(risks_C) = f_C(Enc_B(symptoms_B), patients_C) \quad (11.4)$$

which is an encrypted function that takes a genome and a list of symptoms and returns a risk per illness.

5.  $C$  sends  $Enc_B(risks_C)$  to  $B$ .
6.  $B$  decrypts  $Enc_B(risks_C)$  to  $risks_C$ , then computes

$$Enc_A(risk_A) = risks_C(symptoms_B, Enc_A(g_A)) \quad (11.5)$$

which is an encrypted list of risks per illness for  $A$ .

7.  $B$  sends  $Enc_A(risk_A)$  to  $A$ .
8.  $A$  decrypts  $Enc_A(risk_A)$  to  $risk_A$ .

**Implementation** The following listing presents how the FHE protocol can be written in Chisel. We were able to implement this complex computation exactly how we specified it above by using multiple computations with different executing and result parties.

**Computations:**


---

```

First {
    Executing Parties = B_Pharma
    Result Parties = B_Pharma
    Matrix<Double> symptoms_B = B_Pharma.f_B.mul(A_Patient.symptoms_A.mul(B_Pharma.
        patients_B));
    return symptoms_B;
}
Second crypto {
    Executing Parties = C_Hospital
    Result Parties = C_Hospital
    Matrix<Double> risks = C_Hospital.f_C.mul(Result Reference( B_Pharma, symptoms_B
        ).mul(C_Hospital.patients_C));
    return risks;
}
Third crypto {
    Executing Parties = B_Pharma
    Result Parties = A_Patient
    Matrix<Double> risk_A = Result Reference( C_Hospital, risks ).mul(Result
        Reference( B_Pharma, symptoms_B ).mul(A_Patient.genome_A));
    return risk_A;
}

```

---

Listing 11.3: Medical Task FHE Protocol

## 11.4. MPC Protocol

We now detail how this task can be solved by using MPC. This first MPC protocol is the simplest one that achieves this.

1.  $A$  inputs  $symptoms_A$  and  $g_A$ .
2.  $B$  inputs  $patients_B$  and  $f_B$ .
3.  $C$  inputs  $patients_C$  and  $f_C$ .
4.  $A$ ,  $B$  and  $C$  compute

$$\begin{aligned}
 symptoms_B &= f_B(symptoms_A, patients_B) \\
 risks_C &= f_C(symptoms_B, patients_C) \\
 risk_A &= risks_C(symptoms_B, g_A)
 \end{aligned} \tag{11.6}$$

5.  $A$  receives the output.

**Implementation** The following listing presents how this simple MPC protocol can be written in Chisel. Notice that we can utilise the automatic executing parties for this protocol.

**Computations:**


---

```

MPC crypto{
  Executing Parties = auto // auto: A_Patient, B_Pharma, C_Hospital
  Result Parties = A_Patient
  Matrix<Double> symptoms_B = B_Pharma.f_B.mul(A_Patient.symptoms_A.mul(B_Pharma.
    patients_B));
  Matrix<Double> risks = C_Hospital.f_C.mul(symptoms_B.mul(C_Hospital.patients_C));
  Matrix<Double> risk_A_MPC = risks.mul(symptoms_B.mul(A_Patient.genome_A));
  return risk_A;
}

```

---

Listing 11.4: Medical Task Simple MPC Protocol

In contrast to the previous protocol, this next protocol does each of the three computations with only those parties that need to participate. This improves performance and privacy.

**Precomputation**

1.  $A$  reveals  $symptoms_A$ .  $B$  computes

$$symptoms_B = f_B(symptoms_A, patients_B) \quad (11.7)$$

**First MPC computation**

1.  $B$  inputs  $symptoms_B$ ,  $C$  inputs  $patients_C$
2.  $B$  and  $C$  compute

$$risks_C = f_C(symptoms_B, patients_C) \quad (11.8)$$

and  $B$  gets the result.

**Second MPC computation**

1.  $A$  inputs  $g_A$ ,  $B$  inputs  $symptoms_B$  and  $risks_C$
2.  $A$  and  $B$  compute

$$risk_A = risks_C(symptoms_B, g_A) \quad (11.9)$$

and  $A$  gets the result.

The following listing presents how this more complex MPC protocol can be written in Chisel. Again we were able to specify the MPC protocol exactly to our specifications by utilising three computations, the first of them local and thus not cryptographic.

---

**Computations:**

```

First {
    Executing Parties = B_Pharma
    Result Parties = B_Pharma
    Matrix<Double> symptoms_B = B_Pharma.f_B.mul(A_Patient.symptoms_A.mul(B_Pharma.
        patients_B));
    return symptoms_B;
}
Second crypto {
    Executing Parties = B_Pharma, C_Hospital
    Result Parties = B_Pharma
    Matrix<Double> risks = C_Hospital.f_C.mul(Result Reference ( B_Pharma ,
        symptoms_B ).mul(C_Hospital.patients_C));
    return risks;
}
Third crypto {
    Executing Parties = A_Patient, B_Pharma
    Result Parties = A_Patient
    Matrix<Double> risk_A = Result Reference ( B_Pharma , risks ).mul(Result
        Reference ( B_Pharma , symptoms_B ).mul(A_Patient.genome_A));
    return risk_A;
}

```

---

Listing 11.5: Medical Task Complex MPC Protocol





# Chapter 12

## Evaluation of Usability

To measure how well we achieved the goals we set for ourselves, we designed a study in which we ask people to write code in our tool as well as in existing tools. After using each tool, participants also rate it in four categories. Based on their code and the survey we can then analyze if our tool is easier to learn, less error-prone and more adaptive to changes in the setting.

### 12.1. Research Questions

Our study is developed based on our goal to answer these research questions.

- (Q1) Does Chisel help users avoid introducing security issues into their code?
- (Q2) Do users feel that Chisel is intuitive and transparent?
- (Q3) Are users faster when developing with Chisel than with existing tools?

### 12.2. Study Design

Our design for this validation study was roughly based on Tiefenau et al. [58]. We put together a series of tasks with increasing complexity to test how well a user was able to work with a specific tool. The study then consisted of two parts, one for Chisel and one for the existing tools. In each part, participants were given an introduction to the current toolchain and be asked to implement the series of tasks. Finishing a part early was possible after at least one hour had passed, and each part lasted at most two hours. After each part, users were asked five questions about how well they liked working with the tool, which together with information we gained from recording their screen and studying their programs was the basis for our evaluation.

The introduction to each tool included how to open the program and create a new file, as well as some tips to help participants write code. We also included links to the documentation of each tool. The participants were provided with an Ubuntu virtual machine that had all tools already installed. Each tool also included a tutorial file, where we documented the most important features for each tool. Users were told to read through this tutorial before attempting to solve the tasks.

Every participant then tried to solve all tasks in Chisel. Because Chisel supports FHE and MPC, but there are no other tools like this out there, we needed to give the user an

FHE and an MPC tool for the comparison. We chose to let the user decide which tool to use for each task, to highlight that this decision might sometimes be difficult to make. We decided to use Cingulata again because it proved to provide useful feedback in the first user study we did. For an MPC tool, we decided to not use HyCC again, because of the difficulty people had with the makefile. Instead, we opted for MP-SPDZ, which is a Python-based tool with very intuitive input declaration.

We switched the order of the two toolchains for half the participants so that we can rule out any dependencies. For example, reading the tasks for the first time probably takes more time than reading them the second time. The entire study document can be found in Appendix C.

## 12.3. Task Design

Based on the findings of another study [58], we decided to not focus on the tasks' realism, but instead try to use a wide range of features. We also focused on showcasing the pros and cons of all three tools

### 12.3.1. Task 1

Our goal with the first task was to let the participant familiarise themselves with the tools. The first subtask did not include any secret variables and simply tasked the user to return a constant integer. This task was mainly included to make sure the participant knew how to create new protocols.

The second subtask was a classic FHE protocol on two parties, with the first party computing the sum of the other's secret variables. The third subtask was a classic MPC protocol on three parties, with all three parties computing the sum of their inputs. It also introduced the concept of creating new secret classes. One party first had to do an aggregation, whose result was then the input into the multi-party computation. With the second and third subtask, we wanted to highlight the difficulty in specifying two different protocols in a single tool. With the aggregation in the third subtask we tested whether the participants noticed that it could be done locally and thus be much faster.

### 12.3.2. Task 2

Our goal with the second task was to let the participant implement protocols that highlight some of the more difficult parts in secure computation. The first subtask was a joint computation of the average of two secret inputs, one from each of the two parties. By using the average, we introduced a division into the protocol, which is one of the trickiest operations with secret computation. For example, division is not available in Cingulata; and in MP-SPDZ one has to use a specific function.

The second subtask extended the first one, but now with each party giving  $n$  inputs. This was done so that a loop had to be used, another tricky subject in most existing tools. While one can use a normal loop in Cingulata, in MP-SPDZ a special construct has to be used.

### 12.3.3. Task 3

Our goal with the third task was to let the participant implement a more complex protocol, using matrices. The task included three parties, each having a matrix and a vector. Together these define a linear system of equations (LSE). Each party aimed to get the solution to this LSE without revealing their inputs to each other. Because our goal was not to test the participants on their LSE solving skills, we provided a pseudocode algorithm for forward elimination and backward substitution.

## 12.4. Result evaluation

In order to compare the results we get from this study and to answer research question (Q1), we categorise the errors that users make while solving the tasks.

- **Plaintext errors** These errors are tool-independent and also happen in a non-cryptographic environment. For example a user typing `int a = 3` instead of `int a = 5` would be categorised as a plaintext error.
- **Cryptography errors** These errors are tool-dependent and affect the output of the protocol in some way. We differentiate between three different kinds of cryptographic errors.
  - **Security errors** These errors directly affect the secure computation aspect of the protocol. For example, a leaked secret input or unwanted access to a computation result would be categorised as a security error.
  - **Correctness errors** These errors do not diminish the security of the protocol but lead to a wrong result. Many tools provide their own alternatives for if-else-constructs and for-loops. Not using this alternative often leads to wrong computation outputs and would thus be an example of a correctness error.
  - **Minor errors** These errors include all cryptographic errors that do not directly affect security or correctness. Examples include errors that lead to a decline of performance, for example declaring a public variable as secret or adding an unused computation.

A second evaluation criterion will be the small survey that users fill out after having worked with a tool. They are asked to rate the tool on a scale of one to five in terms of the overall experience, ease of use, transparency and time consumption. These questions will help us compare the different tools and answer research question (Q2), but also give us additional insight. For example, an interesting question is if the users' experience with a tool correlates to the correctness of the programs they write in it.

The third evaluation criterion is the time spent on each task. We do not ask users to track this themselves, but we can get approximate times by looking at the screen recording. This will allow us to answer research question (Q3).

## 12.5. Demographic

Our goal was to test the usefulness of the tools themselves. To make sure that people would not be struggling with writing code in general, we required participants to have programming experience in Java, C++ and Python comparable to a student in the Computer Science master. We did not require users to have experience with crypto techniques or any of the tools used in the study, because the target demographic of these tools is laypeople.

All participants, provided they attempted to seriously solve the tasks, were compensated with 50 CHF. They had the right to withdraw from the study at any time without specifying reasons.

## 12.6. Support

In a longer study like this, participants must not get stuck on small problems or technicalities that do not have to do with the main task we want to evaluate. Thus, we set up a support chat with the participants where they could ask any questions they had. We used Slack, an easy to use instant messaging platform. When asking a participant for feedback at the end of a multi-hour study like this, they might not remember all the problems they had. The chat client also helped us solve this problem, and we were able to collect a lot of feedback already during the study.

Based on [58], we used a playbook as well as a predefined procedure for answering questions:

1. If we could answer the question by pointing the participant to the task description, the tutorial or the online documentation, this was done first.
2. Otherwise, if the question had nothing to do with cryptography or a specific tool, we would answer right away.
3. Finally, if the question was about a specific tool we tried to first give only a hint. If their problem persisted, we would give them a solution, e.g. a specific line of code. If this still did not resolve their problem, we would ask them to skip this subtask and continue with the next.

## 12.7. Ethics

Participants received a document with the goals of the study and the general procedure of the study. They were also informed that their screens will be recorded and data will be collected and signed a consent form to agree with these conditions. We made sure to mention that we are only interested in the relative usability of the tools and not in their performance, to make sure they did not feel stressed. Since the study is designed to be done on-site, it required Ethics Commission approval even though it does not collect personal data. The whole study was accepted without adjustments by the Ethics Commission of ETH Zürich.

## 12.8. Results

Due to social distancing regulations as a reaction to the outbreak of the 2019 Coronavirus epidemic, we could conduct the study with only two participants before it was postponed. From these two trial runs, we did gain enough information though to affirm that the design of the study is solid and will lead to meaningful results. We found that the biggest struggle with Chisel was MPS and its projectional editor, which goes against normal coding practices. Such issues could, in theory, be avoided by implementing the final language design in a traditional parser-based approach. This is a good first insight, as it means that the design of our language is solid.



# Chapter 13

## Conclusion

### 13.1. Summary

During this project, we designed, implemented and evaluated an accessible high-level language for advanced cryptography. Users can write their applications in an easy to use, intuitive and transparent way and get encouraged to write secure, safe protocols. We designed two studies to help us design the language and evaluate it and learned a lot about study design in the process. While we did confirm that our study works in its current form, we could not conduct it and thus did not get significant results from it. Our research questions could thus not be fully answered, but we are positive that this can be rectified by conducting the survey.

### 13.2. Future Work

During our thesis, we revealed several ideas to extend and improve the work we did. While the limited time frame did not allow us to implement these ideas, we think that they would make for very interesting future work.

#### 13.2.1. Extensions to Chisel

**Different return values** Adding the possibility to return several values from a computation would make Chisel even more expressive, especially if different values could be returned to different parties. This is not easy to implement, because we currently employ the Java-like defined return statement to return result values. The return statement includes additional functionality, like defining the return type of a method or marking statements that come after it as unreachable. Changing the return statement implies completely rewriting how these things work as well.

**Executing and result parties as variables** Allowing access to the executing and result parties in the computation would allow the user to cycle through these parties, which is often a desired functionality in cryptographic protocols. To implement this, we would need to automatically generate two variables for each computation and then automatically keep their values on track. Generating variables and automatic upkeep are two difficult subjects in MPS, thus this task is not easily implemented.



**Copy-paste handlers** Copying and pasting code in MPS means reorganising nodes in the AST, which often introduces the need for rewiring references or rewriting code. This automatic reorganising can be achieved by writing copy-paste handlers.

**Different trust levels** At the moment, each party has one trust level, but it would be interesting if each party had a trust level for each of the other parties. For example, A could trust B, while C does not trust B. This would allow for a more detailed and realistic description of protocols.

**Generator** Our language is currently not connected to any advanced cryptography libraries. To achieve that, we would need to write a generator that transforms Chisel code into Java or C++ code, so that it can then be given as input to a library. Writing a generator is a gargantuan task and will thus be left to be tackled by another thesis.

**Variables from Computation** As mentioned already, these references currently have to be updated manually by activating a quick fix. It would be a lot more intuitive if this change would happen automatically. Additionally, in the current state, all these variables can be referenced from any computation. Restricting a computation's access to results of computations that happen before it would improve the correctness of the program.

**Automatic executing parties** The computation of this group of parties is at the moment very simple but could be extended to be smarter and take into account more variables. Additionally, our validation study trial run suggests that automatically filling in the executing parties might not lead to the best results.

**Order of computations** At the moment, we consider the computations to happen from top to bottom. It would be interesting to think about how to make this more versatile and allow more complex protocols.

### 13.2.2. Improvements to the Studies

**First study** The easiest way to improve the meaningfulness of the results of our first study would be to repeat it with more participants. This way, statistically significant results could be achieved and clear validation of our hypotheses could be done.

**Second study** We unfortunately did not have the time to carry out our validation study beyond a proof of concept. Again, executing this study with a high number of participants would get us statistically significant answers to our research questions and thus show if we fully reached the goals we had with Chisel.

From our trial runs, we also gained some insight on how we could improve the study. It was sometimes hard to estimate what the user is currently doing only from the screen recording because we often do not know if they are currently looking at the code on screen or at the printed out task description. Only distributing the task description as a document on the computer would be helpful to better distinguish these two activities.

# Bibliography

- [1] Andrew C. Yao. “Protocols for secure computations”. In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. ISSN: 0272-5428. Nov. 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.38.
- [2] David Chaum, Claude Crépeau, and Ivan Damgard. “Multiparty Unconditionally Secure Protocols”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. Chicago, Illinois, USA. New York, NY, USA: ACM, 1988, pp. 11–19. ISBN: 978-0-89791-264-8. DOI: 10.1145/62212.62214.
- [3] Thomas R.G. Green. “Cognitive dimensions of notations”. In: *People and Computers V*. University Press, 1989, pp. 443–460.
- [4] Alan F. Blackwell and Thomas R.G. Green. “A Cognitive Dimensions questionnaire optimised for users”. In: *PPIG*. 2000.
- [5] Jakob Nielsen. *Ten Usability Heuristics*. 2002. URL: <https://tfa.stanford.edu/download/TenUsabilityHeuristics.pdf> (visited on 11/05/2019).
- [6] J. Johnston, Jan H.P. Eloff, and Les Labuschagne. “Security and human computer interfaces”. In: *Computers & Security* 22.8 (Dec. 2003), pp. 675–684. ISSN: 0167-4048. DOI: 10.1016/S0167-4048(03)00006-3.
- [7] Ka-Ping Yee. “Aligning security and usability”. In: *IEEE Security Privacy* 2.5 (Sept. 2004), pp. 48–55. ISSN: 1540-7993, 1558-4046. DOI: 10.1109/MSP.2004.64.
- [8] Steven Clarke. “Describing and Measuring API Usability with the Cognitive Dimensions”. In: Jan. 2006.
- [9] Almut Herzog and Nahid Shahmehri. “User help techniques for usable security”. In: *Proceedings of the 2007 symposium on Computer human interaction for the management of information technology - CHIMIT '07*. Cambridge, Massachusetts: ACM Press, 2007, p. 11. DOI: 10.1145/1234772.1234787.
- [10] Craig Gentry. “A fully homomorphic encryption scheme”. PhD thesis. Stanford University, 2009.
- [11] Wilko Henecka et al. “TASTY: Tool for Automating Secure Two-party Computations”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA. New York, NY, USA: ACM, 2010, pp. 451–462. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866358.
- [12] Ronald Kainda, Ivan Fléchaïs, and A.W. Roscoe. “Security and Usability: Analysis and Evaluation”. In: *2010 International Conference on Availability, Reliability and Security*. Feb. 2010, pp. 275–282. DOI: 10.1109/ARES.2010.77.

## Bibliography

- [13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption without Bootstrapping”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts. New York, NY, USA: Association for Computing Machinery, 2012, pp. 309–325. ISBN: 978-1-4503-1115-1. DOI: 10.1145/2090236.2090262.
- [14] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Tech. rep. 144. 2012.
- [15] Sascha Fahl et al. “Rethinking SSL Development in an Appified World”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany. New York, NY, USA: ACM, 2013, pp. 49–60. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516655.
- [16] Shai Halevi and Victor Shoup. *HElib: Design and implementation of a homomorphic-encryption library*. 2013. URL: <https://github.com/shaih/HElib/blob/master/%20doc/designDocument/he-%20library.pdf>.
- [17] Christopher Meyer and Jörg Schwenk. *Lessons Learned From Previous SSL/TLS Attacks - A Brief Chronology Of Attacks And Weaknesses*. Tech. rep. 049. 2013.
- [18] Sascha Fahl et al. “Why Eve and Mallory (Also) Love Webmasters: A Study on the Root Causes of SSL Misconfigurations”. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. Kyoto, Japan. New York, NY, USA: ACM, 2014, pp. 507–512. ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590341.
- [19] Matthew Fredrikson and Benjamin Livshits. “ZØ: An Optimizing Distributing Zero-Knowledge Compiler”. In: 2014, pp. 909–924. ISBN: 978-1-931971-15-7.
- [20] A. 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*. May 2014, pp. 655–670. DOI: 10.1109/SP.2014.48.
- [21] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. “Armadillo: A Compilation Chain for Privacy Preserving Applications”. In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. SCC '15. Singapore, Republic of Singapore. New York, NY, USA: ACM, 2015, pp. 13–19. ISBN: 978-1-4503-3447-1. DOI: 10.1145/2732516.2732520.
- [22] Craig Costello et al. “Geppetto: Versatile Verifiable Computation”. In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 253–270. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.23.
- [23] Daniel Demmler, Thomas Schneider, and Michael Zohner. “ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation”. In: *NDSS*. 2015. DOI: 10.14722/ndss.2015.23113.

- [24] Léo Ducas and Daniele Micciancio. “FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second”. In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 617–640. ISBN: 978-3-662-46800-5. DOI: 10.1007/978-3-662-46800-5\_24.
- [25] Matthew Green and Matthew Smith. “Developers Are Users Too: Designing Crypto and Security APIs That Busy Engineers and Sysadmins Can Use Securely”. In: Washington, D.C.: USENIX Association, Aug. 2015.
- [26] Riad S. Wahby et al. “Efficient RAM and control flow in verifiable outsourced computation”. In: *Network & Distributed System Security Symposium (NDSS)*. Edition: Network & Distributed System Security Symposium (NDSS). Feb. 2015.
- [27] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. “You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users”. In: *2016 IEEE Cybersecurity Development (SecDev)*. Nov. 2016, pp. 3–8. DOI: 10.1109/SecDev.2016.013.
- [28] Peter Leo Gorski and Luigi Lo Iacono. “Towards the Usability Evaluation of Security APIs”. In: *HAISA*. 2016.
- [29] Marcel Keller. *MP-SPDZ: Versatile framework for multi-party computation*. 2016. URL: <https://github.com/data61/MP-SPDZ>.
- [30] Bryan Parno et al. “Pinocchio: Nearly Practical Verifiable Computation”. In: *Communications of the ACM* (Feb. 2016). Edition: Communications of the ACM Publisher: ACM - Association for Computing Machinery.
- [31] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. “A Survey of Attacks on Ethereum Smart Contracts (SoK)”. In: *Principles of Security and Trust*. Ed. by Matteo Maffei and Mark Ryan. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2017, pp. 164–186. ISBN: 978-3-662-54455-6. DOI: 10.1007/978-3-662-54455-6\_8.
- [32] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70694-8. DOI: 10.1007/978-3-319-70694-8\_15.
- [33] Clayton Lewis. “Methods in user oriented design of programming languages”. In: *PPIG*. 2017.
- [34] Andreas Stefik and Stefan Hanenberg. “Methodological Irregularities in Programming-Language Research”. In: *Computer* 50.8 (2017), pp. 60–63. ISSN: 0018-9162. DOI: 10.1109/MC.2017.3001257.
- [35] Federico Tonasseti. *Domain Specific Languages - The what, why and how of DSLs*. Feb. 2017. URL: <https://tomassetti.me/domain-specific-languages/> (visited on 02/27/2020).

## Bibliography

- [36] Chamila Wijayarathna, Nalin A. G. Arachchilage, and Jill Slay. “A Generic Cognitive Dimensions Questionnaire to Evaluate the Usability of Security APIs”. In: *Human Aspects of Information Security, Privacy and Trust*. Ed. by Theo Tryfonas. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 160–173. ISBN: 978-3-319-58460-7. DOI: 10.1007/978-3-319-58460-7\_11.
- [37] Niklas Büscher et al. “HyCC: Compilation of Hybrid Protocols for Practical Secure Computation”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada. New York, NY, USA: ACM, 2018, pp. 847–861. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243786.
- [38] Eduardo Chielle et al. *E3: A Framework for Compiling C++ Programs with Encrypted Operands*. Tech. rep. 1013. 2018.
- [39] Eric Crockett, Chris Peikert, and Chad Sharp. “ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easY”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS ’18*. Toronto, Canada: ACM Press, 2018, pp. 1020–1037. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243828.
- [40] Shai Halevi. “Advanced Cryptography: Promise and Challenges”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada. New York, NY, USA: ACM, 2018, pp. 647–647. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3268995.
- [41] Sukrit Kalra et al. “ZEUS: Analyzing Safety of Smart Contracts”. In: *NDSS*. 2018. DOI: 10.14722/ndss.2018.23082.
- [42] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. “xJsnark: A Framework for Efficient Verifiable Computation”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. May 2018, pp. 944–961. DOI: 10.1109/SP.2018.00018.
- [43] Reza M. Parizi, Amritraj, and Ali Dehghantanha. “Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security”. In: *Blockchain – ICBC 2018*. Ed. by Shiping Chen, Harry Wang, and Liang-Jie Zhang. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 75–91. ISBN: 978-3-319-94478-4. DOI: 10.1007/978-3-319-94478-4\_6.
- [44] Sergei Tikhomirov et al. “SmartCheck: Static Analysis of Ethereum Smart Contracts”. In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. May 2018, pp. 9–16.
- [45] Alexander Viand and Hossein Shafagh. “Marble: Making Fully Homomorphic Encryption Accessible to All”. In: *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC ’18. Toronto, Canada. New York, NY, USA: ACM, 2018, pp. 49–60. ISBN: 978-1-4503-5987-0. DOI: 10.1145/3267973.3267978.

- [46] Josh Aas et al. “Let’s Encrypt: An Automated Certificate Authority to Encrypt the Entire Web”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. London, United Kingdom: Association for Computing Machinery, Nov. 2019, pp. 2473–2487. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363192.
- [47] David W. Archer et al. “RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications”. In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC’19. London, United Kingdom. New York, NY, USA: Association for Computing Machinery, 2019, pp. 57–68. ISBN: 978-1-4503-6829-2. DOI: 10.1145/3338469.3358945.
- [48] Fabian Boemer et al. “nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data”. In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC’19. London, United Kingdom: Association for Computing Machinery, Nov. 2019, pp. 45–56. ISBN: 978-1-4503-6829-2. DOI: 10.1145/3338469.3358944.
- [49] Hao Chen, Ilaria Chillotti, and Yongsoo Song. *Multi-Key Homomorphic Encryption from TFHE*. Tech. rep. 116. 2019.
- [50] Roshan Dathathri et al. “CHET: An Optimizing Compiler for Fully-homomorphic Neural-network Inferencing”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA. New York, NY, USA: ACM, 2019, pp. 142–156. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314628.
- [51] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. “SEALion: a Framework for Neural Network Inference on Encrypted Data”. In: *arXiv:1904.12840 [cs, stat]* (Apr. 2019). arXiv: 1904.12840.
- [52] Marcella Hastings et al. “SoK: General Purpose Compilers for Secure Multi-Party Computation”. In: *2019 IEEE Symposium on Security and Privacy (SP)* (2019), pp. 1220–1237. DOI: 10.1109/sp.2019.00028.
- [53] Eunice Jun et al. “Tea: A High-level Language and Runtime System for Automating Statistical Analysis”. In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology - UIST ’19*. New Orleans, LA, USA: ACM Press, 2019, pp. 591–603. ISBN: 978-1-4503-6816-2. DOI: 10.1145/3332165.3347940.
- [54] Yi Li and Wei Xu. “PrivPy: General and Scalable Privacy-Preserving Data Mining”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’19. Anchorage, AK, USA. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1299–1307. ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330920.
- [55] *Microsoft SEAL (release 3.4)*. Oct. 2019. URL: <https://github.com/Microsoft/SEAL>.
- [56] Mariana Raykova. *Advanced Cryptography On The Way To Practice*. English. Lausanne, June 2019.

## Bibliography

- [57] Kurt Rohloff et al. *Palisade*. 2019. URL: <https://gitlab.com/palisade/palisade-release/-/wikis/home> (visited on 01/08/2020).
- [58] Christian Tiefenau et al. “A Usability Evaluation of Let’s Encrypt and Certbot: Usable Security Done Right”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS ’19*. London, United Kingdom: ACM Press, 2019, pp. 1971–1988. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363220.
- [59] Jess Whittlestone et al. “Ethical and societal implications of algorithms, data, and artificial intelligence: a roadmap for research”. In: (2019), p. 59.
- [60] Chuan Zhao et al. “Secure Multi-Party Computation: Theory, practice and applications”. In: *Information Sciences* 476 (Feb. 2019), pp. 357–372. ISSN: 0020-0255. DOI: 10.1016/j.ins.2018.10.024.
- [61] Ilaria Chillotti et al. “TFHE: Fast Fully Homomorphic Encryption Over the Torus”. In: *Journal of Cryptology* 33.1 (Jan. 2020), pp. 34–91. ISSN: 1432-1378. DOI: 10.1007/s00145-019-09319-x.
- [62] *Use fall detection with Apple Watch*. URL: <https://support.apple.com/en-us/HT208944> (visited on 03/17/2020).

## Part V

# Appendix

*I am glad you are here with me. Here at the end of all things, Sam.*  
— John Ronald Reuel Tolkien

This part contains the complete version of the taxonomy table as well as both studies.





# Chapter A

## Taxonomy

### A.1. Legend

- this tool fully supports this feature.
- this tool does not fully support this feature.
- this feature is not applicable for this tool.
- ? it is not determinable if this tool supports this feature.

## Appendix A. Taxonomy

	Technique				Architecture		Documentation				Automation					Target Personas									
	FHE	MPC GC	MPC SS	MPC Hybrid	Verifiable Computation		Library	Extension	Paper	Language docs	Example code	Example docs	Open source	Technique choice	Library choice	Key generation	Noise checks	Ciphertext parameters	Circuit generation	Total novice	Some programming	Advanced programming	Some crypto	Advanced crypto	Expert crypto
Alchemy	●					Haskell			●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
CHET	●					C++		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Cingulata	●					C++		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
E3	●					C++		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Marble	●						●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
NGraph-HE	●					nGraph		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Ramparts	●					Julia		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
SEALion	●					TensorFlow		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
TASTY	●	●				Python		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
ABY		●	●	●		C++	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Frigate			●			C		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
HyCC				●		C			●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
MP-SPDZ		●		●		Python		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
OblivM		●				SCVM		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
PICCO						C		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
PrivPy			●			Python		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Wysteria			●	●				●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Buffet					●	C		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
xJSnark					●	Java		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Z0					●	C#		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
zkay					●	Solidity		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Table A.1.: Taxonomy of advanced cryptography tools.

## Chapter B

# Exploratory User Survey

This is the user survey we conducted before designing our language. It is represented here in a simplified format. The order in which the four tools were shown was randomised.

## Secure Computation Accessibility Survey

The goal of this study is to evaluate the usability of novice-oriented tools for secure computation. No prior knowledge beyond common programming skills is required. There will be 10 pages in this survey and it should take you about 20 minutes to complete it. You have a chance to win a 50 CHF Amazon gift card at the end of the survey.

### Please carefully read the following paragraphs:

Currently, virtually all communication on the Internet is encrypted. However, the data is decrypted when it reaches the endpoint, e.g. a cloud server. This is currently necessary to allow the cloud to work with the data, for example to give you machine learning based recommendations. Secure computation makes it possible to do the same work, but on encrypted data. Thus the cloud never sees the underlying data. There are two main types of secure computation that this survey will discuss:

**Fully Homomorphic Encryption (FHE):** Let us assume that Alice has stored a database of encrypted information in the Cloud. FHE allows the Cloud to compute on this data and give her an encrypted result which she can decrypt to the correct result. The cloud never sees the data or the result.

**Secure Multi-Party Computation (SMPC):** Let us assume that in a group of  $n$  people each person  $i$  has a secret  $s_i$ . SMPC allows them to compute a function  $f(s_1, \dots, s_n)$  without anyone having to reveal their secret to the other parties. This can work even if there are malicious participants in the group that share wrong information or do not follow the protocol.

## Cingulata code snippet

These two code snippets together define a secure computation. Assume that the first snippet has access to the files generated by the second one. You do not have to understand the code completely.

```
1 #include <bit_exec/tracker.h>
2 #include <ci_context.h>
3 #include <ci_int.h>
4 #include <int_op_gen/mult_depth.h>
5 #include <numeric>
6
7 using namespace std;
8 using namespace cingulata;
9
10 CiInt average(vector<CiInt> x){
11     CiInt sum(CiInt::u32v(0));
12     for (CiInt c : x){
13         sum += c;
14     }
15     return sum / CiInt::u32v(x.size());
16 }
17
18 int main() {
19     // Set bit tracker as executor
20     // and empty Boolean circuits of optimized multiplicative depth.
21     CiContext::set_config(make_shared<BitTracker>(), make_shared<IntOpGenDepth>());
22
23     // define input variables
24     vector<CiInt> firstSecretInput(4, CiInt::u32);
25     CiInt secondSecretInput(CiInt::u8);
26
27     // read in variables from secret input
28     for (int i = 0; i < 4; i++){
29         cin >> firstSecretInput[i];
30     }
31     secondSecretInput.read("b");
32
33     // define output variable
34     CiInt result(CiInt::u8);
35
36     // compute result
37     result = average(firstSecretInput) + secondSecretInput;
38
39     // write out result to standard output
40     result.write("result");
41
42     // specify that Boolean circuit will be saved in file named "addition"
43     CiContext::get_bit_exec_t<BitTracker>()->export_blif(blif_name, "addition");
44 }
```

```
1 CIRCUIT=@CIRCUIT@
2 APPS_DIR=@APPS_DIR@
3
4 # generate input and output folders
5 mkdir -p input
6 rm -f input/*.*
7
8 mkdir -p output
9 rm -f output/*.*
10
11 # generate keys
12 $APPS_DIR/generate_keys
13
14 # do encryption of inputs
15 NR_THREADS=1
16
17 $APPS_DIR/encrypt -v --threads $NR_THREADS $APPS_DIR/helper --bit-cnt 32 2 2 4 4
18
19 TMP=$APPS_DIR/helper --bit-cnt 32 --prefix input/iib_5
20 $APPS_DIR/encrypt -v --threads $NR_THREADS $TMP
21
22 # execute the function
23 time $APPS_DIR/dyn_omp $CIRCUIT --threads $NR_THREADS # -v
24
25 # decrypt the output
26 OUT_FILES="ls -v output/*"
27 $APPS_DIR/helper --from-bin --bit-cnt 32 $APPS_DIR/decrypt $OUT_FILES
```

1. Describe in one sentence what you think the code does. Feel free to guess.
2. How certain are you about your previous answer? (1=Just a guess to 5=Very confident)
3. How important were the comments for you? (1=Not necessary to 5=Essential)
4. Which part of the code did you find the least clear? Please specify line number(s).
5. Please check all the statements that you think are true.
  - "secondSecretInput" contains a string at some point in the code.
  - "firstSecretInput" and "secondSecretInput" come from the same party.
  - The final output is 8.
  - At some point, this code computes the average of two numbers.
  - There are 4 parties involved in this computation.

## AcTool code snippet

This code snippet defines a secure computation. You do not have to understand the code completely.

```
1 #include <acTool.h>
2
3 // define parties
4 bank = new InputParty();
5 cloud = new CompParty();
6
7 void main(){
8     // define input
9     bank.secretInput1 = new SecretInput{[2, 2, 4, 4]};
10    bank.secretInput2 = new SecretInput{[4, 8, 8, 4]};
11
12    // do computation
13    cloud.compute(average(sum(bank.secretInput1), sum(bank.secretInput2)));
14    // cloud party reveals the result to the bank
15    cloud.revealTo(bank);
16 }
17
18 // function to be computed
19 int average(int[] x) {
20     return sum(x)/count(x);
21 }
```

1. Describe in one sentence what you think the code does. Feel free to guess.
2. How certain are you about your previous answer? (1=Just a guess to 5=Very confident)
3. How important were the comments for you? (1=Not necessary to 5=Essential)
4. Which part of the code did you find the least clear? Please specify line number(s).
5. Please check all the statements that you think are true.
  - There are 2 parties in the code snippet.
  - At some point, this code computes the average of two numbers.
  - The final output is 8.
  - "bank" has access to the decrypted result of the computation.
  - "cloud" has access to the decrypted result of the computation.

## HyCC code snippet

These two code snippets together define a secure computation. Assume that the first snippet has access to the file of the second one and vice versa. You do not have to understand the code completely.

```
1 //define vector-like structure
2 typedef struct
3 {
4     int x_1, x_2, x_3, x_4;
5 } Vec4;
6
7 //function to compute
8 int average(Vec4 x)
9 {
10     int sum = x.x_1 + x.x_2 + x.x_3 + x.x_4;
11     return sum / 4;
12 }
13
14 int mpc_main(Vec4 INPUT_A_firstSecretInput, int INPUT_B_secondSecretInput)
15 {
16     //compute result
17     return average(INPUT_A_firstSecretInput) + INPUT_B_secondSecretInput;
18 }
```

```
1 # define locations of references
2 CBMC_GC = ../../bin/cbmc-gc
3 CIRCUIT_SIM = ../../bin/circuit-sim
4 ABY_CBMC_GC = ../../ABY/build/bin/aby-hycc
5
6 # define location of function
7 mpc_main.circ: main.c
8 | $(CBMC_GC) $^
9
10 # what to run
11 .PHONY: clean run-sim
12
13 # clean output files
14 clean:
15 | rm -f mpc_main.circ mpc_main.stats
16
17 # define inputs for simulation and run it
18 run-sim:
19 | @$(CIRCUIT_SIM) mpc_main.circ --spec "
20 |     INPUT_A_firstSecretInput := {x_1: 2; x_2: 2; x_3: 4; x_4: 4};
21 |     INPUT_B_secondSecretInput := 5;
22 |     print;"
```

1. Describe in one sentence what you think the code does. Feel free to guess.
2. How certain are you about your previous answer? (1=Just a guess to 5=Very confident)
3. How important were the comments for you? (1=Not necessary to 5=Essential)
4. Which part of the code did you find the least clear? Please specify line number(s).
5. Please check all the statements that you think are true.
  - "mpc\_main" returns 3.
  - There are 2 parties in this computation.
  - "INPUT\_A\_firstSecretInput" and "INPUT\_B\_secondSecretInput" come from two different parties.
  - At some point, this code computes the average of two numbers.
  - The call to "average" returns 3.



## MpcM code snippet

These three code snippets together define a secure computation. You do not have to understand the code completely.

```
1 // This code is on all parties
2 #include <mpc_m.h>
3
4 //technically IP & port instead of string
5 parties = {new Party("A"), new Party("B"), new Party("C")};
6
7 // defining the computation
8 Computation f_a = average();
9 // give input to function as secret
10 giveSecretInput(4, f_a);
11
12 //polling until everyone is ready
13 readyToStartComp(f_a);
14 // waiting for computation to finish
15 int y = waitForResult(f_a);
16
17 // function to be computed
18 int average(int[] x){ return sum(x)/count(x); }
```

```
1 // This code is on all parties
2 #include <mpc_m.h>
3
4 //technically IP & port instead of string
5 parties = {new Party("A"), new Party("B"), new Party("C")};
6
7 // defining the computation
8 Computation f_b = average();
9 // give input to function as secret
10 giveSecretInput(2, f_b);
11
12 //polling until everyone is ready
13 readyToStartComp(f_b);
14 // waiting for computation to finish
15 int y = waitForResult(f_b);
16
17 // function to be computed
18 int average(int[] x){ return sum(x)/count(x); }
```

```
1 // This code is on all parties
2 #include <mpc_m.h>
3
4 //technically IP & port instead of string
5 parties = {new Party("A"), new Party("B"), new Party("C")};
6
7 // defining the computation
8 Computation f_c = average();
9 // give input to function as secret
10 giveSecretInput(8, f_c);
11
12 //polling until everyone is ready
13 readyToStartComp(f_c);
14 // waiting for computation to finish
15 int y = waitForResult(f_c);
16
17 // function to be computed
18 int average(int[] x){ return sum(x)/count(x); }
```

1. Describe in one sentence what you think the code does. Feel free to guess.
2. How certain are you about your previous answer? (1=Just a guess to 5=Very confident)
3. How important were the comments for you? (1=Not necessary to 5=Essential)
4. Which part of the code did you find the least clear? Please specify line number(s).
5. Please check all the statements that you think are true.
  - There are 4 parties involved in this computation.
  - "y" will never have a value other than 4.
  - At some point, the three "y" contain different values.
  - The computation is done by one party.
  - There is a chance that the computation will not terminate.

## **General Questions**

1. Gender (male, female, other)
2. Age
3. Job / Background
4. Total years programming experience
5. Do you have any security/cryptography knowledge? If yes, was it helpful?



## Chapter C

# ValidationStudy

This is the validation study that we did to evaluate our language. The order in which the two toolchains were shown was randomised.

# Evaluating the Usability of Crypto Compilers

## C.1. Study Procedure

This study consists of two parts. In each part, you will be given an introduction to the current toolchain and will be asked to implement a series of tasks. Note that the tasks are the same in both parts.

The two toolchains are


- Chisel, which features a Java-like language.
- MP-SPDZ & Cingulata, which respectively feature Python and C++ as their high-level language.

The first page of each part's description will tell you which toolchain to use for this task. For each of the sections, you have two hours to solve the tasks. Should you feel that you have completed all tasks in a part already, you can proceed to the next part (assuming that at least one hour has passed). After proceeding to the second part, you are not allowed to go back and make adjustments to the first part.

The next page of this document will give you a short introduction to secure computation, to provide some context and background. It will also introduce some terminology that might appear in the task descriptions. Please **feel free to refer back to these introductory pages at any point during the study**.

## C.2. Study System

We have provided an Ubuntu virtual machine on the computer in front of you. Please do not exit the VM or leave the full-screen mode. Should you accidentally log yourself out, the username is StudyParticipant and the password is 123456.

If you have any questions during the study, **please do not hesitate to contact** us via the  Slack chat app which you should find in the side bar. You also have access to the internet, e.g. to search for documentation.

Please be aware that we will record your screen during the study and record your keystrokes and mouse clicks. In addition, we will save the final state of the VM after you complete both parts. Please do not enter personally identifiable information into the VM and do not log into any online services.

Each tool has been pre-installed and a tutorial with example code is provided for each. This document also includes a quick start section at the beginning of each part, explaining how to open the relevant program and how to write code in the tool.

## C.3. Development Environment

To give a level playing field, you will use similar IDEs for all tools, specifically you will use IDEs from JetBrains. We will be using CLion for C++, PyCharm for Python and MPS for the

Java-like domain specific language. You might already be familiar with some of these IDEs, but if not this is not an issue for this study.

All three share a common UI structure, offer strong syntax highlighting and powerful auto-completion and auto-refactoring tools. In addition, they indicate warnings and errors by underlining the affected statements in the source code. Clicking into underlined statements offers context-sensitive information on the warning and often suggests automatic fixes.

Even if you are familiar with these IDEs, please have a look at this quick tutorial:

<https://github.com/MarbleHE/Chisel/blob/master/IntroToJetBrains.md>

Please turn to the next page.

## C.4. Introduction to Secure Computation

Digitalization and the emergence of affordable cloud-computing have made outsourced computation an essential part of most organisations' operations. When outsourcing computation, special care must be taken by companies to protect their data. During transit and at rest, this is already achieved with widely deployed standard encryption techniques. However, the need for decryption prior to computation undermines this protection. End-to-End encryption, as seen in some messaging applications, avoids this exposure but also prevents server-side computations.

Secure Computation enables us to ensure privacy while maintaining utility by using *advanced crypto*, i.e., techniques beyond conventional symmetric- and public-key encryption. These have become increasingly practical thanks to advances in the underlying theory, general hardware improvements, and more efficient implementations. Fully homomorphic encryption (FHE) and secure multi-party computation (MPC) are two such techniques:

**Fully Homomorphic Encryption** Fully homomorphic encryption allows us to encrypt a secret  $s$  and have someone else compute a function  $f$  on the encryption of  $s$  without ever having access to the underlying secret. However, we can decrypt the result and learn  $f(s)$ . This can be used to implement e.g. cloud-based computation-as-a-service without loss of privacy.

**Secure Multi-Party Computation** Let us assume Alice, Bob and Charlie have secrets  $s_a, s_b, s_c$ , respectively. Using MPC, they can interactively compute a function  $f(s_a, s_b, s_c)$  without having to reveal their secret to the other two. They can also decide which of them will receive the output of the function. This can be used to implement e.g. a secret auction where nobody but the seller and the winner learns the final price.

In this study we evaluate tools that lower the barrier of entry and allow non-cryptographers to realize secure computation solutions for a wide range of applications. Specifically, we are evaluating a set of tools that translate high-level specifications into secure computation solutions. Therefore, you should not need to focus on the cryptographic techniques beyond what is explained in the task descriptions.

## C.5. Terminology

We explain some keywords that are used in the task descriptions:

- **Protocol:** A protocol describes a system with multiple entities that transmit information and perform functions. In the context of this study, a protocol usually describes one specific functionality, e.g. the auction in the example above could be described as a protocol.
- **Party:** One of the entities involved in a protocol, e.g. Alice in the example above.
- **Input:** Any kind of data a party gives as an argument into the protocol, e.g. a bid in an auction.
- **Output:** Any kind of data that results from the protocol, e.g. the winning bid.

## **C.6. Security Model**

Assume that all parties are semi-honest, which means they will do what is specified in the protocol. However, they are still curious and will deduce information when they can. For example, if Alice and Bob each have a secret input and learn the sum of the two values, they can deduce the other's secret value by subtracting their own secret from the sum.



## PART 1: Chisel


Please do not turn over until instructed to do so.

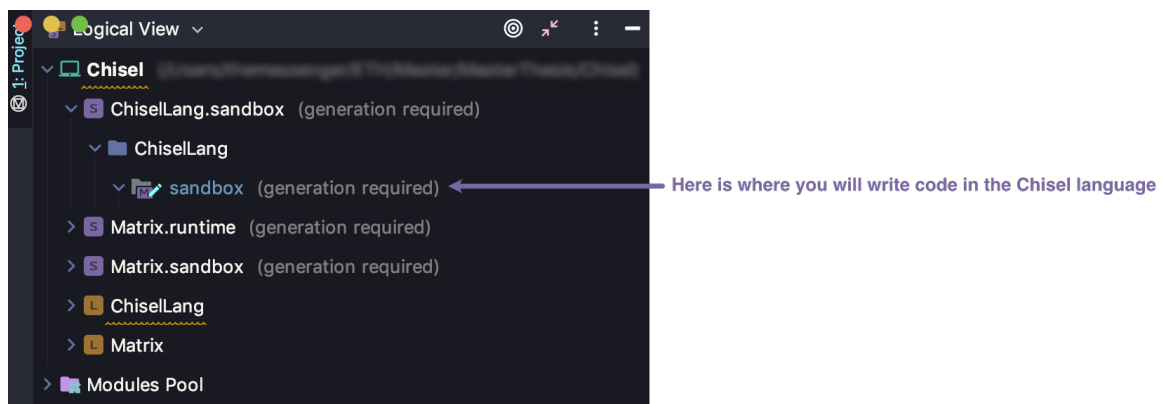
## C.7. Introduction to Chisel

Chisel is a domain-specific language for secure computation. Chisel is based on Java, you will use JetBrains MPS to access it.

You can access the documentation here: <https://github.com/MarbleHE/Chisel>

### C.7.1. Quick Start

1. Open  MPS from the sidebar, Chisel should automatically be opened. If not, you can find it on the Desktop.
2. If it is not already open, press <alt+1> to show the navigation bar. A list of other helpful keyboard shortcuts is provided on a separate sheet.
3. In the navigation bar you should see five directories. The following image shows you where you will be writing code.



4. To create a new protocol, open "ChiselLang.sandbox" → "ChiselLang" → "sandbox", right-click on "sandbox", then select "New" → "ChiselLang" → "Protocol". A template for a protocol will appear, as shown in the following image.

```
Protocol = <no name>

Parties:
  semi-honest <no name> {
    <no statements>
    Variables from Computations: << ... >>
  }

Computations:
  <no name> {
    Executing Parties = auto // auto: <no name>
    Result Parties = << ... >>
    <no statements>
  }
```

### C.7.2. How to write code

You can reference two documents on how to write code in Chisel:

- The online documentation <https://github.com/MarbleHE/Chisel>.
- The tutorial, located in "ChiselLang.sandbox" → "ChiselLang" → "sandbox".

### C.7.3. How to do secure computations

- To mark a variable as secret, write `secret` after the `static` keyword. There also exists an intention to add the `secret` keyword. Note that only top-level variables can be marked as secret.
- To mark a computation as secure, write `crypto` after the computation name. There also exists an intention to add the `crypto` keyword.
- A party that gets the result of a computation can access that result in a later computation. You can see all these references listed under **Variables from Computations** for each party, e.g. "i" in the example below. To access such a variable, you use the **Result Reference** function, e.g. `Result Reference ( MyParty, i )` in the example below.

```

Protocol = MyProtocol

Parties:
  semi-honest MyParty {
    <no statements>
    Variables from Computations: i
  }

Computations:
  MyComputation {
    Executing Parties = auto // auto: MyParty
    Result Parties = MyParty
    int i = 10;
    return i;
  }
  SecondComputation {
    Executing Parties = auto // auto: MyParty
    Result Parties = << ... >>
    int j = Result Reference ( MyParty , i );
  }

```

### C.7.4. Matrices

For a reference on how to define and use matrices, we again refer you to the two main documents:

- The online documentation <https://github.com/MarbleHE/Chisel>.
- The tutorial, located in "ChiselLang.sandbox" → "ChiselLang" → "sandbox".

### C.7.5. How to compile code

- You do not need to manually compile this code, compile errors are shown instantly.
- If you want to get a list of all current errors, right-click on "ChiselLang.sandbox" and select "Make Solution 'ChiselLang.sandbox'".

### C.7.6. Known Bugs

- Please ignore the error "The reference ... is out of search scope" on a party or variable reference.

- The autofill of `auto` in executing parties might show up again after deleting it. Just delete it again, then it should stay deleted.
- Ignore "Generation Required" in the navigation bar.

## C.8. Description of Task 1

The goal of this task is to familiarize yourself with the tools used in this study.

Make sure you have **read through the tutorial** for the tool you are using before starting this task. The subtasks build on each other, you should *not* create a new file for each of them but instead extend or modify your current solution.

Since we are evaluating compilers, **you do not need to actually run your programs**. You can consider a task complete when you have a compiling program that you think implements the protocol from the task description.

If you get stuck on a task remember that you can always use Slack to contact us. However, you should try looking at this document and the tutorials again before contacting us.

### C.8.1.

Write a protocol Task1 with only one party A, that always returns `int result = 15`.

	Parties
	A
Giving secret input	○
Computing	●
Getting result	●

### C.8.2.

Extending the previous one, in this protocol the first party A takes two *secret* inputs `int ageB = 2` and `int ageC = 5` from a second party B and adds them to its own *public* input `int ageA = 3`. The return should now be this sum instead of the constant value. If possible, the `int result` should only be revealed to the second party. Note that no party should ever see the secret input of another party.

	Parties	
	A	B
Giving secret input	○	●
Computing	●	○
Getting result	○	●

Please see Task 14.3 on the next page.

### C.8.3.

Extending on the previous one, in this protocol we use a custom type. A **Person** has attributes `name` (string) and `age` (int). The second party B's input is now a collection of **Person**:

Person 1: "Linda", 40  
 Person 2: "Leone", 45  
 Person 3: "Sindy", 35

First, compute the sum `ageB` of the Persons in this array. If possible, the collection of **Persons** and `ageB` should be secret.

	Parties		
	A	B	C
Giving secret input	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Computing	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Getting result	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>

The first party defines one input `int ageA = 42`, as does a third party C with `int ageC = 38`. Now compute the sum of all the ages and make it available as `int result` to all parties. It is up to you which party/parties are involved in this computation. Note that no party should ever see the secret input of another party.

	Parties		
	A	B	C
Giving secret input	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
Computing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Getting result	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>

## C.9. Description of Task 2

The goal of this task is to define a more realistic protocol.

Make sure you have **read through the tutorial** for the tool you are using before starting this task. The subtasks build on each other, you should *not* create a new file for each of them but instead extend or modify your current solution.

Since we are evaluating compilers, **you do not need to actually run your programs**. You can consider a task complete when you have a compiling program that you think implements the protocol from the task description.

If you get stuck on a task remember that you can always use Slack to contact us. However, you should try looking at this document and the tutorials again before contacting us.

### C.9.1.

Write a protocol with two parties D and E. Each party has one secret input `inputD` and `inputE` respectively. Jointly compute the average of the two inputs and make it available to both parties. Note that no party should ever see the secret input of another party.

	Parties	
	D	E
Giving secret input	●	●
Computing	●	●
Getting result	●	●

### C.9.2.

Extend the previous protocol, now every party has `n` inputs. Again jointly compute the average of all inputs and make it available to both parties. Note that no party should ever see the secret input of another party.

	Parties	
	D	E
Giving secret input	●	●
Computing	●	●
Getting result	●	●

## C.10. Description of Task 3

The goal of this task is to implement a more complex protocol.

Make sure you have **read through the tutorial** for the tool you are using before starting this task. The subtasks build on each other, you should *not* create a new file for each of them but instead extend or modify your current solution.

Since we are evaluating compilers, **you do not need to actually run your programs**. You can consider a task complete when you have a compiling program that you think implements the protocol from the task description.

If you get stuck on a task remember that you can always use Slack to contact us. However, you should try looking at this document and the tutorials again before contacting us.

### C.10.1.

Three parties F, G, H are involved in this task. Each party has a secret matrix and a secret vector:

$$\begin{aligned} MF &= \begin{vmatrix} 1 & 1 & 1 & 3 & -2 & 1 \\ -2 & 2 & 1 & 1 & 2 & -1 \end{vmatrix} & MG &= \begin{vmatrix} 2 & 4 & 4 & 1 & -1 & 2 \\ 1 & -1 & 1 & 3 & 1 & 1 \end{vmatrix} & MH &= \begin{vmatrix} 2 & 1 & 2 & 3 & 2 & 3 \\ -4 & 2 & 5 & 1 & 3 & 1 \end{vmatrix} \\ vF &= \begin{vmatrix} -3 \\ 20 \end{vmatrix} & vG &= \begin{vmatrix} 5 \\ 4 \end{vmatrix} & vH &= \begin{vmatrix} 11 \\ 13 \end{vmatrix} \end{aligned}$$

Together, these inputs define a linear system of equations:

$$M \cdot x = v \Leftrightarrow \begin{vmatrix} MF \\ MG \\ MH \end{vmatrix} \cdot \begin{vmatrix} a \\ b \\ c \\ d \\ e \\ f \end{vmatrix} = \begin{vmatrix} vF \\ vG \\ vH \end{vmatrix}$$

All parties want to know the solution to this system, but none is willing to reveal their secrets.

Write a protocol that solves the system and returns the result to all parties, while keeping all inputs secret.

	Parties		
	F	G	H
Giving secret input	●	●	●
Computing	?	?	?
Getting result	●	●	●

The following two algorithms describe how to solve a linear system of equations:



---

**Algorithm 1** Forward Elimination

---

```

1: Input:  $n \times n$  matrix  $M$  and  $n \times 1$  column-
   vector  $v$ 
2: Output: An equivalent upper-triangular
    $n \times (n + 1)$  matrix in place of  $M$  with corre-
   sponding right-hand side values in the sev-
   enth column.
3: for  $i \leftarrow 0$  to  $n$  do
4:    $M[i, n + 1] \leftarrow v[i]$  //Augment matrix
5: end for
6: for  $i \leftarrow 0$  to  $n - 1$  do
7:   for  $j \leftarrow i + 1$  to  $n$  do
8:     for  $k \leftarrow i$  to  $n + 1$  do
9:        $M[j, k] \leftarrow M[j, k] - M[i, k] \cdot$ 
         $M[j, i]/M[i, i]$ 
10:    end for
11:  end for
12: end for

```

---



---

**Algorithm 2** Backward Substitution

---

```

1: Input:  $n \times (n + 1)$  upper-triangular matrix
    $T$ 
2: Output: A  $n \times 1$  column-vector  $s$  with the
   solutions to the system
3: for  $i \leftarrow n$  to  $0$  do
4:    $s[i] \leftarrow T[i, n + 1]$ 
5:   for  $j \leftarrow i + 1$  to  $n$  do
6:      $s[i] = s[i] - T[i, j] \cdot s[j]$ 
7:   end for
8:    $s[i] = s[i]/T[i, i]$ 
9: end for

```

---

## C.11. Short Survey

1. On a scale of 1 to 5, how was your experience with Chisel?

extremely  
unpleasant      1      2      3      4      5      very enjoyable  
                    ☐      ☐      ☐      ☐      ☐

2. On a scale from 1 to 5, how easy was Chisel to use?

very simple      1      2      3      4      5      extremely com-  
                    ☐      ☐      ☐      ☐      ☐      plex

3. On a scale from 1 to 5, how transparent was the information in Chisel?

very transparent      1      2      3      4      5      too obfuscated  
                            ☐      ☐      ☐      ☐      ☐

4. On a scale from 1 to 5, how time consuming was the implementation of tasks in Chisel?

pleasantly quick      1      2      3      4      5      took too long  
                            ☐      ☐      ☐      ☐      ☐

## PART 2: MP-SPDZ & Cingulata


Please do not turn over until you are finished  
with part 1.

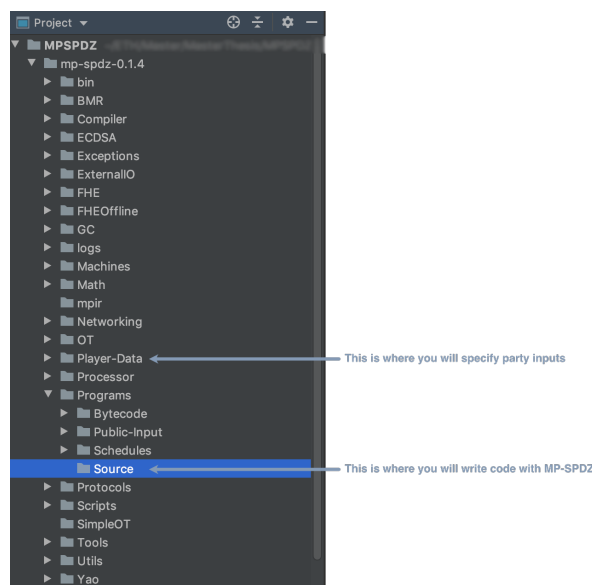
## C.12. Introduction to MP-SPDZ

MP-SPDZ is a tool facilitating the running of secure computation protocols. MP-SPDZ is based on Python, you will use JetBrains PyCharm to access it.

You can access the documentation here: <https://github.com/data61/MP-SPDZ>

### C.12.1. Quick Start

1. Open  PyCharm from the sidebar, MP-SPDZ should automatically be opened. If not, you can find it on the Desktop.
2. If it is not already open, press <alt+1> to show the navigation bar. A list of other helpful keyboard shortcuts is provided on a separate sheet.
3. In the navigation bar you should see several subdirectories. The following image shows you where you will be writing code.



4. To create a new protocol, open "mp-spdz-0.1.4" → "Programs" → "Source", right-click on "Source", then select "New" → "File", provide a name and add ending ".mpc".
5. Inputs are specified in separate files, as described in section C.12.3 below. These files need to be located in the folder "mp-spdz-0.1.4" → "Player-Data".

### C.12.2. How to write code

- To use MP-SPDZ, import the following two things:

```
from Compiler.types import *
from Compiler.library import *
```

You can reference two documents on how to write code in MP-SPDZ:

- The online documentation <https://github.com/data61/MP-SPDZ>.
- The tutorial, located in "mp-spdz-0.1.4" → "Programs" → "Source".

### C.12.3. How to do secure computations

- Use `sint` instead of `int` for secret values.
- As mentioned above, the input from each party is stored in a separate file in folder "mp-spdz-0.1.4" → "Player-Data". The first party's file should be called "Input-P0-0", the second party's "Input-P1-0", and so on. The file should just list all inputs with a space as a separator.
- You can read an input from a file. For example the following command reads the next input from the second party into `a`:  

```
a = sint.get_input_from(1)
```
- You can also define a constant `sint` as follows:  

```
c = sint(15)
```
- Use function `r.reveal()` to provide the result `r` to all parties.

### C.12.4. Matrices

- You can define matrices by typing `<name> = Matrix(<rows>, <cols>, <type>)`.
- You can then initialize it by accessing cells, for example `<name>[0][0] = 0`.
- The following operations are available for Matrices:
  - Matrix Multiplication: `Matrix * Matrix`
  - No Scalar Multiplication
  - Matrix Addition: `Matrix + Matrix`
  - No Matrix Subtraction

For examples look at the tutorial, located in "mp-spdz-0.1.4" → "Programs" → "Source".

### C.12.5. How to compile code

- Run the following command in the Terminal. You can open a Terminal in PyCharm by pressing `<alt+F12>`.  

```
./compile.py <your file name without .mpc>
```

For example, to run the tutorial, enter


```
./compile.py tutorial
```

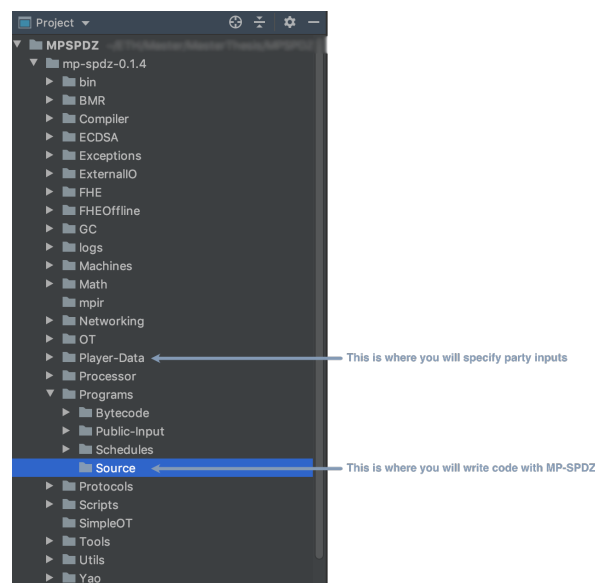
## C.13. Introduction to Cingulata

Cingulata is a compiler toolchain running programs over encrypted data. Cingulata is based on C++, you will use JetBrains CLion to access it.

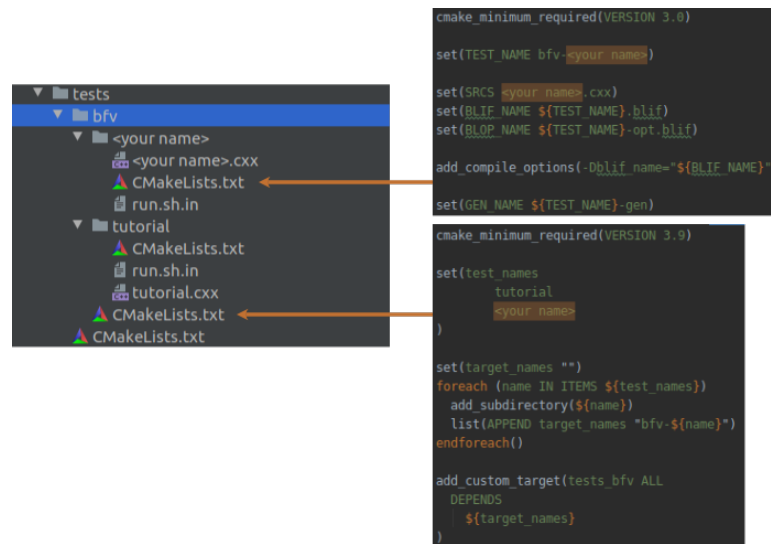
You can access the documentation here: <https://github.com/CEA-LIST/Cingulata/wiki>

### C.13.1. Quick Start

1. Open  CLion from the sidebar, Cingulata should automatically be opened. If not, you can find it on the Desktop.
2. If it is not already open, press <alt+1> to show the navigation bar. A list of other helpful keyboard shortcuts is provided on a separate sheet.
3. In the navigation bar you should see several subdirectories. The following image shows you where you will be writing code.



4. To create a new protocol, open "Cingulata" → "tests" → "bfv", right-click on "bfv", then select "New" → "New Directory" and provide a name. Then right-click on the new directory and select "New" → "New C/C++ Source File" and enter the same name as the directory. This file is where you will write the function.
5. In the "tutorial" directory, you should see two files called "run.sh.in" and "CMakeLists.txt". Copy these files to your new directory. The first file is where you will provide inputs to the function, as described in section C.13.3 below. The second is needed to compile your code, open it and change the two occurrences of "tutorial" to your file name. There is also a "CMakeLists.txt" directly in the "bfv" folder. Add your name below "tutorial".



### C.13.2. How to write code

- To use Cingulata, include the following four files:

```
#include <bit_exec/tracker.hxx>
#include <ci_context.hxx>
#include <ci_int.hxx>
#include <int_op_gen/mult_depth.hxx>
#include <matrix.hxx>
```

- Use the namespace of Cingulata by writing:  
using namespace cingulata;

You can reference two documents on how to write code in Cingulata:

- The online documentation <https://github.com/CEA-LIST/Cingulata/wiki>.
- The tutorial, located in "Cingulata" → "tests" → "bfv" → "tutorial".

### C.13.3. How to do secure computations

- Use `CiInt` instead of `int` for secret values.
- You cannot use division on secret integers.
- Set up the context at the beginning of your main function with  
`CiContext::set_config(make_shared<BitTracker>(), make_shared<IntOpGenDepth>());`
- Export the resulting circuit to a file at the end of your main function with  
`CiContext::get_bit_exec_t<BitTracker>()->export_blif(blif_name, "circuit");`
- The "run.sh.in" file notes where you should write code. There you can define secret variables. For example the following defines secret input "temp" to be a 8-bit integer with value 3:

```
$APPS_DIR/encrypt -v --threads $NR_THREADS '$APPS_DIR/helper --bit-cnt 8 --prefix input/i:ter
```

### C.13.4. Matrices

- You can define matrices of type double by typing `Matrix <name>(<rows>, <cols>)`.
- You can then initialize it by accessing cells, for example `<name>(0, 0) = 0`.
- The following operations are available for Matrices:
  - Matrix Multiplication: `Matrix * Matrix`
  - Scalar Multiplication: `Matrix * scalar`
  - Matrix Addition: `Matrix + Matrix`
  - Matrix Subtraction: `Matrix - Matrix`

For examples look at the tutorial, located in "Cingulata" → "tests" → "bfv" → "tutorial".

### C.13.5. How to compile code

- Run the following command in the Terminal. You can open a Terminal in CLion by pressing `<alt+F12>`.  

```
docker run -it --rm -v $(pwd):/cingu cingulata:bfv
```

### C.13.6. Known Bugs

- It is not possible to assign one `CiInt` to another `CiInt`.
- Using `cout` anywhere garbles the result.
- When you encounter the message "No parameter found with static mode. Automatically switch to interactive mode", just enter "3".



## C.14. Description of Task 1

The goal of this task is to familiarize yourself with the tools used in this study. Remember that in this part you can choose to use either **MP-SPDZ** or **Cingulata**. Think about what tool you are going to use before starting the implementation. You can choose a different tool for each task (but please use the same for all subtasks).

Make sure you have **read through the tutorial** for the tool you are using before starting this task. The subtasks build on each other, you should *not* create a new file for each of them but instead extend or modify your current solution. If you want to change the tool during a task, please start from the beginning in a new file in the other tool.

Since we are evaluating compilers, **you do not need to actually run your programs**. You can consider a task complete when you have a compiling program that you think implements the protocol from the task description.

If you get stuck on a task remember that you can always use Slack to contact us. However, you should try looking at this document and the tutorials again before contacting us.

### C.14.1.

Write a protocol `Task1` with only one party `A`, that always returns `int result = 15`.

	Parties
	A
Giving secret input	○
Computing	●
Getting result	●

### C.14.2.

Extending the previous one, in this protocol the first party `A` takes two *secret* inputs `int ageB = 2` and `int ageC = 5` from a second party `B` and adds them to its own *public* input `int ageA = 3`. The return should now be this sum instead of the constant value. If possible, the `int result` should only be revealed to the second party. Note that no party should ever see the secret input of another party.

	Parties	
	A	B
Giving secret input	○	●
Computing	●	○
Getting result	○	●

Please see Task 14.3 on the next page.

**C.14.3.**

Extending on the previous one, in this protocol we use a custom type. A **Person** has attributes **name** (string) and **age** (int). The second party B's input is now a collection of **Person**:

Person 1: "Linda", 40

Person 2: "Leone", 45

Person 3: "Sindy", 35

First, compute the sum **ageB** of the Persons in this array. If possible, the collection of **Persons** and **ageB** should be secret.

	Parties		
	A	B	C
Giving secret input	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Computing	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Getting result	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>

The first party defines one input **int ageA = 42**, as does a third party **C** with **int ageC = 38**. Now compute the sum of all the ages and make it available as **int result** to all parties. It is up to you which party/parties are involved in this computation. Note that no party should ever see the secret input of another party.

	Parties		
	A	B	C
Giving secret input	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
Computing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Getting result	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>

## C.15. Description of Task 2

The goal of this task is to define a more realistic protocol. Remember that in this part you can choose to use either **MP-SPDZ** or **Cingulata**. Think about what tool you are going to use before starting the implementation. You can choose a different tool for each task (but please use the same for all subtasks).

Make sure you have **read through the tutorial** for the tool you are using before starting this task. The subtasks build on each other, you should *not* create a new file for each of them but instead extend or modify your current solution. If you want to change the tool during a task, please start from the beginning in a new file in the other tool.

Since we are evaluating compilers, **you do not need to actually run your programs**. You can consider a task complete when you have a compiling program that you think implements the protocol from the task description.

If you get stuck on a task remember that you can always use Slack to contact us. However, you should try looking at this document and the tutorials again before contacting us.

### C.15.1.

Write a protocol with two parties D and E. Each party has one secret input `inputD` and `inputE` respectively. Jointly compute the average of the two inputs and make it available to both parties. Note that no party should ever see the secret input of another party.

	Parties	
	D	E
Giving secret input	●	●
Computing	●	●
Getting result	●	●

### C.15.2.

Extend the previous protocol, now every party has `n` inputs. Again jointly compute the average of all inputs and make it available to both parties. Note that no party should ever see the secret input of another party.

	Parties	
	D	E
Giving secret input	●	●
Computing	●	●
Getting result	●	●

## C.16. Description of Task 3

The goal of this task is to implement a more complex protocol. Remember that in this part you can choose to use either **MP-SPDZ** or **Cingulata**. Think about what tool you are going to use before starting the implementation. You can choose a different tool for each task (but please use the same for all subtasks).

Make sure you have **read through the tutorial** for the tool you are using before starting this task. The subtasks build on each other, you should *not* create a new file for each of them but instead extend or modify your current solution. If you want to change the tool during a task, please start from the beginning in a new file in the other tool.

Since we are evaluating compilers, **you do not need to actually run your programs**. You can consider a task complete when you have a compiling program that you think implements the protocol from the task description.

If you get stuck on a task remember that you can always use Slack to contact us. However, you should try looking at this document and the tutorials again before contacting us.

### C.16.1.

Three parties F, G, H are involved in this task. Each party has a secret matrix and a secret vector:

$$MF = \begin{bmatrix} 1 & 1 & 1 & 3 & -2 & 1 \\ -2 & 2 & 1 & 1 & 2 & -1 \end{bmatrix} \quad MG = \begin{bmatrix} 2 & 4 & 4 & 1 & -1 & 2 \\ 1 & -1 & 1 & 3 & 1 & 1 \end{bmatrix} \quad MH = \begin{bmatrix} 2 & 1 & 2 & 3 & 2 & 3 \\ -4 & 2 & 5 & 1 & 3 & 1 \end{bmatrix}$$

$$vF = \begin{bmatrix} -3 \\ 20 \end{bmatrix} \quad vG = \begin{bmatrix} 5 \\ 4 \end{bmatrix} \quad vH = \begin{bmatrix} 11 \\ 13 \end{bmatrix}$$

Together, these inputs define a linear system of equations:

$$M \cdot x = v \Leftrightarrow \begin{bmatrix} MF \\ MG \\ MH \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} vF \\ vG \\ vH \end{bmatrix}$$

All parties want to know the solution to this system, but none is willing to reveal their secrets.

Write a protocol that solves the system and returns the result to all parties, while keeping all inputs secret.

	Parties		
	F	G	H
Giving secret input	●	●	●
Computing	?	?	?
Getting result	●	●	●

The following two algorithms describe how to solve a linear system of equations:

**Algorithm 3** Forward Elimination

---

```

1: Input:  $n \times n$  matrix  $M$  and  $n \times 1$  column-
   vector  $v$ 
2: Output: An equivalent upper-triangular
    $n \times (n + 1)$  matrix in place of  $M$  with corre-
   sponding right-hand side values in the sev-
   enth column.
3: for  $i \leftarrow 0$  to  $n$  do
4:    $M[i, n + 1] \leftarrow v[i]$  //Augment matrix
5: end for
6: for  $i \leftarrow 0$  to  $n - 1$  do
7:   for  $j \leftarrow i + 1$  to  $n$  do
8:     for  $k \leftarrow i$  to  $n + 1$  do
9:        $M[j, k] \leftarrow M[j, k] - M[i, k] \cdot$ 
         $M[j, i]/M[i, i]$ 
10:    end for
11:  end for
12: end for

```

---

**Algorithm 4** Backward Substitution

---

```

1: Input:  $n \times (n + 1)$  upper-triangular matrix
    $T$ 
2: Output: A  $n \times 1$  column-vector  $s$  with the
   solutions to the system
3: for  $i \leftarrow n$  to  $0$  do
4:    $s[i] \leftarrow T[i, n + 1]$ 
5:   for  $j \leftarrow i + 1$  to  $n$  do
6:      $s[i] = s[i] - T[i, j] \cdot s[j]$ 
7:   end for
8:    $s[i] = s[i]/T[i, i]$ 
9: end for

```

---

## C.17. Short Survey

### C.17.1. MP-SPDZ

1. On a scale of 1 to 5, how was your experience with using MP-SPDZ?

extremely unpleasant    1    2    3    4    5    Not used    very enjoyable  
☐    ☐    ☐    ☐    ☐    ☐

2. On a scale from 1 to 5, how easy was MP-SPDZ to use?

very simple    1    2    3    4    5    Not used    extremely complex  
☐    ☐    ☐    ☐    ☐    ☐

3. On a scale from 1 to 5, how transparent was the information in MP-SPDZ?

very transparent    1    2    3    4    5    Not used    too obfuscated  
☐    ☐    ☐    ☐    ☐    ☐

4. On a scale from 1 to 5, how time consuming was the implementation of tasks in MP-SPDZ?

pleasantly quick    1    2    3    4    5    Not used    took too long  
☐    ☐    ☐    ☐    ☐    ☐

### C.17.2. Cingulata

1. On a scale of 1 to 5, how was your experience with using Cingulata?

extremely unpleasant    1    2    3    4    5    Not used    very enjoyable  
☐    ☐    ☐    ☐    ☐    ☐

2. On a scale from 1 to 5, how easy was Cingulata to use?

very simple    1    2    3    4    5    Not used    extremely complex  
☐    ☐    ☐    ☐    ☐    ☐

3. On a scale from 1 to 5, how transparent was the information in Cingulata?

very transparent    1    2    3    4    5    Not used    too obfuscated  
☐    ☐    ☐    ☐    ☐    ☐

4. On a scale from 1 to 5, how time consuming was the implementation of tasks in Cingulata?

pleasantly quick    1    2    3    4    5    Not used    took too long  
☐    ☐    ☐    ☐    ☐    ☐