

# Implementation and Evaluation of Zero-Knowledge Proofs of Knowledge

Boran Car

Thesis voorgedragen tot het  
behalen van de graad van Master  
of Engineering: Electrical  
Engineering

**Promotoren:**

Bart Preneel  
Ingrid Verbauwhede

**Assessoren:**

Claudia Diaz  
Francky Catthoor

**Begeleiders:**

Josep Balasch  
Alfredo Rial

© Copyright K.U.Leuven

Without written permission of the promotor and the authors it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to Departement Elektrotechniek, Kasteelpark Arenberg 10 postbus 2440, B-3001 Heverlee, +32-16-321130 or via e-mail [info@esat.kuleuven.be](mailto:info@esat.kuleuven.be).

A written permission of the promotor is also required to use the methods, products, schematics and programs described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

# Preface

*Boran Car*

# Contents

<b>Preface</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Zero Knowledge Proofs of Knowledge . . . . .	1
1.2 HW-SW codesign . . . . .	2
<b>2 Technical preliminaries</b>	<b>5</b>
2.1 Formal Languages [6] . . . . .	5
2.2 Group arithmetic . . . . .	8
2.3 Turing Machine . . . . .	11
2.4 Zero Knowledge Proofs of Knowledge . . . . .	11
2.5 Data Flow Graph and Control Flow Graph . . . . .	12
<b>3 Existing frameworks and tools</b>	<b>13</b>
3.1 CACE Project . . . . .	13
3.2 ZKPD L . . . . .	20
3.3 Comparison of CACE and ZKPD L . . . . .	20
3.4 Other tools . . . . .	20
<b>4 Custom framework</b>	<b>25</b>
4.1 Motivation . . . . .	25
4.2 Extensions to GEZEL . . . . .	27
4.3 Extensions to CACE Zero Knowledge Compiler . . . . .	27
4.4 PIL front-end . . . . .	29
4.5 GEZEL back-end . . . . .	32
<b>5 Use case: Direct Anonymous Attestation</b>	<b>33</b>
<b>6 Conclusion</b>	<b>35</b>
6.1 Future work . . . . .	35
<b>Bibliography</b>	<b>53</b>

# Abstract

# List of Figures

1.1	Gajski-Kuhn Y-chart . . . . .	3
2.1	Concrete Syntax tree for a simple input and a simple grammar . . . . .	7
2.2	An example AST for a simple expression . . . . .	8
2.3	Parser flow . . . . .	8
2.4	Montgomery product flow . . . . .	10
2.5	Turing machine . . . . .	11
3.1	CACE typical workflow [1] . . . . .	15
3.2	ZKPD L typical flow [12] . . . . .	20
3.3	ANTLR Parser/Lexer generation . . . . .	21
3.4	ANTLR Parser/Lexer generation . . . . .	21
3.5	LLVM typical workflow [8] . . . . .	22
3.6	GEZEL workflow [5] . . . . .	24
4.1	Custom framework (extensions to CACE Zero Knowledge Compiler highlighted) . . . . .	25
4.2	GEZEL and ipblock . . . . .	27
4.3	LLVM custom workflow (changes highlighted) . . . . .	28
4.4	ANTLR workflow . . . . .	30
4.5	PIL frontend flow . . . . .	30
4.6	Linker . . . . .	30

# List of Tables





# Chapter 1

## Introduction

### 1.1 Zero Knowledge Proofs of Knowledge

Today's state-of-the-art cryptosystems are based on the following hard problems:

- Discrete Logarithm Problem
- Factorization Problem

Such problems are not solvable using deterministic Turing machines within polynomial time. As long as computers are based on deterministic Turing machines this does not pose as problem. Simply scaling the solution space has provided an effective countermeasure so far. This is all about to change as the world sees the emergence of quantum computers.

In a post-quantum computer world we would like to seek other problems which are again not solvable within polynomial time using the machines of the present.

Another motivation is that in our modern world, privacy is of the uttermost importance. Let's look at a couple of everyday examples:

- Buying a public-transport ticket
- Voting
- Filling out anonymous questionnaires

So far, using traditional methods, privacy was not a big issue here. For each of the cases:

- you paid for a piece of paper that nobody (unless he hires forensic) could connect to you
- you circled your vote and put it in a box that nobody (again, unless he uses sophisticated methods) could connect to you
- you filled in your questionnaire that nobody could connect to you

The real problem comes from trying to put these things into electronic form. Using smart cards for public transport poses a big problem w.r.t. privacy as nothing assures the user that the system does not track him/her. The advantage of putting these things into electronic form is multiple:

- less paper is wasted (smaller footprint)
- less clutter if multiple smart cards are merged into a single one
- less chance of the user losing his smart card if he only has one

However, one disadvantage strikes more than the others. The possibility of the system to track the user and the user being incapable of protecting his privacy. Traditional authentication schemes required the user identifying himself so this tracking step was an inherent property.

The desired property here is not disseminating any knowledge while still proving to the system that certain properties hold (e.g. the user has privileges and is allowed to proceed with an action).

### 1.2 HW-SW codesign

Traditional embedded device programming was separate from hardware design. The separation went to such extents as having two separate entities within a company working on each of those fields. The software part is generic while the hardware part is faster and more energy efficient.

Today's stringent measures require a bigger inter-dependency. Instead of searching for a best algorithm tailored for specific hardware or searching for the best hardware tailored to a specific algorithm/task, why not design the hardware and software at the same time. This way bigger decision and trade-offs can be made more granular [19].

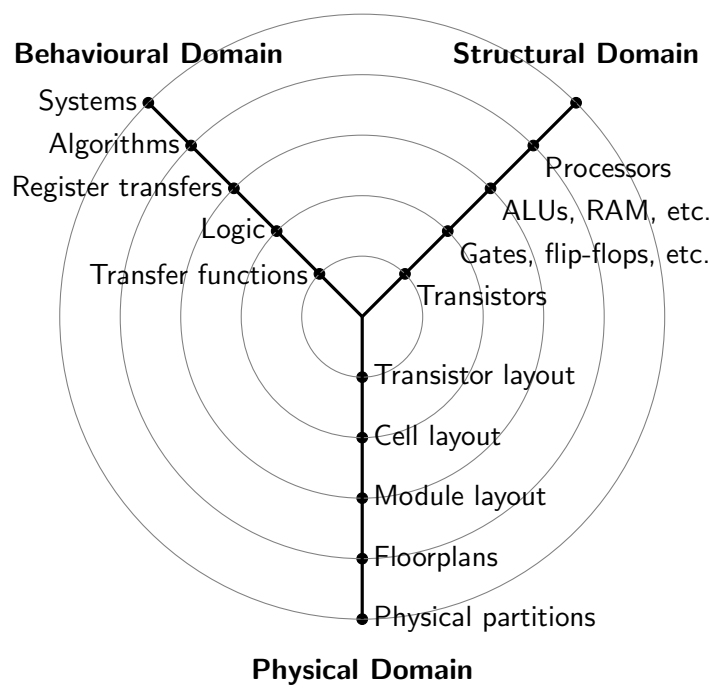


FIGURE 1.1: Gajski-Kuhn Y-chart



## Chapter 2

# Technical preliminaries

The chapter has the goal of introducing the math and theory behind formal languages, parsers, algebra, (zero knowledge) proofs of knowledge.

### 2.1 Formal Languages [6]

**Definition 1** (Alphabet). An alphabet  $\Sigma$  is a set of symbols.

**Definition 2** (String). A string over an alphabet  $\Sigma$  is a sequence of symbols of  $\Sigma$ .

**Definition 3** (Concatenation). Let  $x = a_0a_1 \cdots a_n$  and  $y = b_0b_1 \cdots b_n$ , then the string  $xy = a_0a_1 \cdots a_nb_0b_1 \cdots b_n$  is the concatenation of the strings  $x$  and  $y$ .

**Definition 4** (Sets). The  $\Sigma^*$  denotes the set of all the strings over the alphabet  $\Sigma$ , likewise  $\Sigma^+$  denotes the set of all the non-empty strings over the alphabet  $\Sigma$ .

$$\Sigma^+ \subseteq \Sigma^* \setminus \{\epsilon\}$$

The empty set of strings is denoted  $\emptyset$ .

**Definition 5** (Language). A language over the alphabet  $\Sigma$  is a set of strings over  $\Sigma$ . Members of the language are called words of the language.

**Definition 6** (Concatenation). Let  $L_1$  and  $L_2$  be two languages over alphabet  $\Sigma$ , the language  $L_1L_2 = \{xy|x \in L_1, y \in L_2\}$  is the concatenation of  $L_1$  and  $L_2$ .

**Definition 7** (Kleene closure). Let  $L$  be a language over  $\Sigma$ . Define

$$L^0 = \{\epsilon\}$$

$$L^i = LL^{i-1} \text{ for } i \geq 1$$

the *Kleene closure* of  $L$ , denoted by  $L^*$  is the language:

$$L^* = \bigcup_{i \geq 0} L^i$$

the *positive closure* is

$$L^+ = \bigcup_{i \geq 1} L^i$$

It can be observed that

$$\begin{aligned} L^* &= L^+ \cup \{\epsilon\} \\ L^+ &= LL^* \end{aligned}$$

### 2.1.1 Regular expression

**Definition 8** (Regular expression). A regular expression over  $\Sigma$  is defined inductively as follows:

1.  $\emptyset$  is a regular expression and represents the empty language
2.  $\epsilon$  is a regular expression and represents the language  $L = \{\epsilon\}$
3. For each  $c \in \Sigma$ ,  $c$  is a regular expression and represents the language  $L = \{c\}$
4. For any regular expressions  $r$  of language  $R$  and  $s$  of language  $S$ :
  - $r + s$  is a regular expression representing language  $R \cup S$
  - $rs$  is a regular expression representing language  $RS$
  - $r^*$  is a regular expression representing language  $R^*$
  - $r^+$  is a regular expression representing language  $R^+$

**Theorem 1.**  $rr^*$  can be represented as  $r^+$

*Proof.*  $rr^*$  represents the language  $RR^*$  which is  $R^+$  □

### 2.1.2 Grammars

**Definition 9** (Grammar). A grammar is a 4-tuple  $G = \langle \Sigma, V, S, P \rangle$ :

1.  $\Sigma$  is a finite non-empty set called the *terminal alphabet*. The elements of  $\Sigma$  are called *terminals*.
2.  $V$  is a finite non-empty set disjoint from  $\Sigma$ . The elements of  $V$  are called *non-terminals* or *variables*.
3.  $S \in V$  is a distinguished symbol called the *start symbol*
4.  $P$  is a finite set of *productions (rules)* of the form

$$\begin{aligned} \alpha &\rightarrow \beta \\ \alpha &\in (\Sigma \cup V)^* V (\Sigma \cup V)^* \\ \beta &\in (\Sigma \cup V)^* \end{aligned}$$

**Definition 10** (Context-free grammar). The grammar  $G$  is a context free grammar iff  $|\alpha| = 1$  ( $\alpha = V$ ).

### 2.1.3 Concrete Syntax Tree

A Concrete Syntax Tree, also called a Parse Tree is an ordered tree representation of the input according to a given formal grammar. For example, given the following input:

```
a := b * c + d;
```

and the following grammar:

```
statement → ID := expression;
expression → term + term
term → ID * ID
term → ID
```

the tree depicted in Figure 2.1 is a Concrete Syntax Tree.

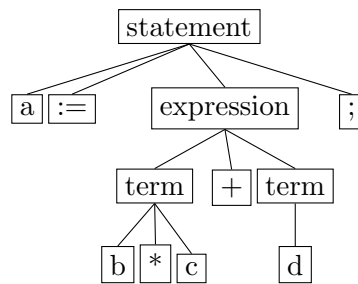


FIGURE 2.1: Concrete Syntax tree for a simple input and a simple grammar

### 2.1.4 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntax of the input. Given the same example

```
a := b * c + d;
```

one possible variant of the tree is illustrated in Figure 2.2. Comparing it with the Parse Tree from Figure 2.1 one can see that the choice of abstraction is arbitrary.

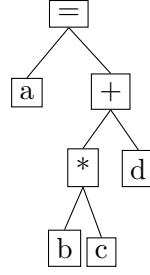


FIGURE 2.2: An example AST for a simple expression

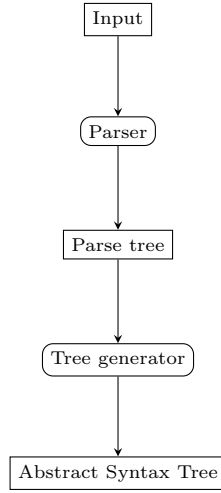


FIGURE 2.3: Parser flow

## 2.2 Group arithmetic

### 2.2.1 Montgomery Product

Modular multiplications would involve trial division after the multiplication step were it not for the algorithm known as Montgomery product invented by Peter L. Montgomery [13].

The basic step and the property of the algorithm is computing the residues modulo  $N$ .

**Definition 11.** Define an radix  $R$  such that  $RR^{-1} - NN' = 1$ . The residue  $\bar{a}$  of  $a$  is  $\bar{a} = aR$

**Theorem 2.** The algorithm for Montgomery reduction is



---

**Algorithm 1** Montgomery reduction

---

```

function REDUCE( $T$ )
   $m \leftarrow (T \bmod R)N' \bmod R$ 
   $t \leftarrow (T + mN)/R$ 
  if  $t \geq N$  then return  $t - N$ 
  else return  $t$ 
  end if
end function

```

---

*Proof.* Assume  $T < R$ , then

$$\begin{aligned}
 m &= TN' \\
 mN &= TN'N \\
 &= TRR^{-1} - T \\
 &= -T \bmod R \\
 tR &= T + mN \\
 &= T \bmod N
 \end{aligned}$$

□

To compute the Montgomery product, one can now apply the reduction to the product of the reduced  $\bar{a}$  and  $\bar{b}$ :

```

function MONTGOMERYPRODUCT( $\bar{a}, \bar{b}$ )
   $t \leftarrow \bar{a} \cdot \bar{b}$ 
  return REDUCE( $t$ )
end function

```

Koç goes a step further and inlines the reduction to see possible optimizations when implementing the algorithm on multiple architectures [7]:

---

```

function MONTGOMERYPRODUCT( $\bar{a}, \bar{b}$ )
   $m \leftarrow (\bar{a} \cdot \bar{b} \bmod R)N' \bmod R$ 
   $t \leftarrow (\bar{a} \cdot \bar{b} + mN)/R$ 
  if  $t \geq N$  then return  $t - N$ 
  else return  $t$ 
  end if
end function

```

---

If such a primitive is provided on an architecture, the reduction then becomes:

---

```

function MONTGOMERYREDUCE( $T$ )
  return MONTGOMERYPRODUCT( $T, R^2$ )
end function

```

---

Montgomery product eliminates the trial division. It does so at the expense of computing the residues. However, these residues can be precomputed in the beginning. Then, for a sufficient number of Montgomery multiplications, the amortized cost is negligible. The problem of computing the modular product is then given by:

1. Transform into the Montgomery domain
2. Compute the product in the Montgomery domain
3. Transform the result back into the integer domain

This is also depicted in Figure 2.4 where both paths are shown when starting from the integer domain.

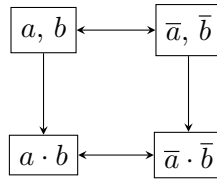


FIGURE 2.4: Montgomery product flow

## 2.3 Turing Machine

A Turing machine is a mathematical model of computation [18]. Informally, it consists of:

- A tape divided into cells
- A head that reads and writes
- A finite table (action table or transition function)
- A state

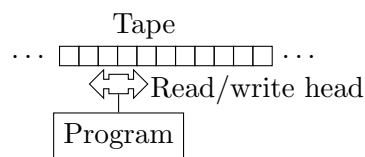


FIGURE 2.5: Turing machine

A more formal definition comes from Hopcroft [6]. A Turing machine is defined as a 7-tuple  $M = \langle Q, \Gamma, b, \Sigma, q_0, F, \delta \rangle$

- $Q$  is a finite, non-empty set of states
- $\Gamma$  is a finite, non-empty set of the tape alphabet/symbols
- $b \in \Gamma$  is the blank symbol
- $\Sigma \subseteq \Gamma \setminus \{b\}$  the set of input symbols
- $q_0 \in Q$  is the initial state
- $F \subseteq Q$  is the set of final or accepting states
- $\delta : Q \setminus F \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  is the transition relation

## 2.4 Zero Knowledge Proofs of Knowledge

Sometimes one party wants to prove knowing a secret to the other party but without actually revealing that secret. Let's take a corporate espionage example. Suppose that we wish to buy a secret but are not convinced of the seller's honesty (he may leave us without money and run away with our secret). We then devise a protocol by which we can be convinced that the seller knows the secret.

To define a proof of knowledge it is convenient to use a Turing machine. In this case let's define an interactive Turing machine (ITM) with 5 tapes:

- Input tape (read-only)
- Receiving tape (read-only)
- Sending tape (write-only)
- Output tape (write-only)
- Working tape (read-write)

An interactive protocol is an ordered pair of ITMs that share the same input tape, and have receiving and sending tapes cross-connected (sending to receiving, receiving to sending).

An interactive proof system can be represented by the following two properties:

- Completeness  $(x, w) \in \mathcal{R} \Rightarrow P(\text{accept}) > \frac{2}{3}$
- Soundness  $(x, w) \notin \mathcal{R} \Rightarrow P(\text{accept}) < \frac{1}{3}$

Informally, the previous two properties state that

- if a statement is true, a honest prover will be able to convince the verifier of the validity with a large probability
- if a statement is false, no dishonest prover will be able to convince the verifier of the validity with a probability larger than the threshold

To make it a zero-knowledge an additional property is needed. We can state that the prover should not release any knowledge on the secret it possesses. To make it more formal, we state that a third party should not be able to distinguish between a successful communication and an unsuccessful one. Depending on how indistinguishable it is, we can differentiate three cases:

- perfectly indistinguishable
- statistically indistinguishable
- computationally indistinguishable

### 2.5 Data Flow Graph and Control Flow Graph

An algorithm is completely and uniquely defined by its Data Flow Graph (DFG), whereas the implementation of an algorithm is what gives it its Control Flow Graph (CFG).

The usual step in converting an algorithm in software to an algorithm in hardware is to extract the DFG [17].

## Chapter 3

# Existing frameworks and tools

This chapter starts with an overview of existing frameworks for implementing Zero Knowledge Proofs of Knowledge. Current examples include CACE Project Zero Knowledge Compiler, ZKPDL and IBM Idemix. CACE project and ZKPDL will be covered. The chapter then continues on describing other tools that will be used for this thesis: ANTLR, a parser generator tool producing LL(\*) parsers and LLVM, a compiler infrastructure framework that has seen wide use recently in many fields relating to compilers and computers in general. ANTLR and LLVM will be used to make a custom compiler so some knowledge about them is necessary. Next is GEZEL, a cycle-true cosimulation environment which can also generate VHDL or Verilog.

### 3.1 CACE Project

#### 3.1.1 Framework overview

The CACE (Computer Aided Cryptography Engineering) Project was an European project aiming at developing a toolbox for security software. It attempted to ease the creation of cryptographic software for those outside the domain. The goals were:

- Automatic translation from natural specification - the term natural is taken from the users perspective, meaning something “natural” for the user, not dwelling too much into the specific niches of cryptography, giving an abstract overview
- Automatic security awareness, analysis and corrections - to be able to detect side channels that are unintentionally introduced, warn the user and offer corrective actions
- Automatic optimization for diverse platforms - different platforms are suited for different operations, assume different usage patterns etc..., the toolbox should be as most insensitive as it can to the platform it is implemented on

### 3. EXISTING FRAMEWORKS AND TOOLS

---

Apart from these goals that deal with the end-user, the project had strategic goals of opening a new field of research and promoting automatic tools when it comes to crypto software. The project itself was split into multiple working groups:

- WP1 Automating Cryptographic Implementation - dealing with the low level crypto operations, searching and identifying side channel attacks, providing a domain specific language
- WP2 Accelerating Secure Network - dealing with basic operations for module intercommunication
- WP3 Bringing Proofs of Knowledge to Practice - dealing with implementing a compiler for Proofs of Knowledge
- WP4 Securing Distributed Management of Information - dealing with higher operations for module intercommunication
- WP5 Formal Verification and Validation - dealing with analysis of the correctness, assuring the user of the protocol validity

The WP3 working group is of the importance for this thesis as it deals directly with proofs of knowledge. The end result of the working group was a compiler along with a specification language (PSL) and an intermediate language (PIL). The compiler has the following typical flow (as depicted in Figure 3.1):

1. Write PSL (Protocol Specification Language)
2. Generate PIL (Protocol Interface Language) from PSL
3. Generate C or Java code from PIL
4. (Optional) Generate LaTeX from PIL

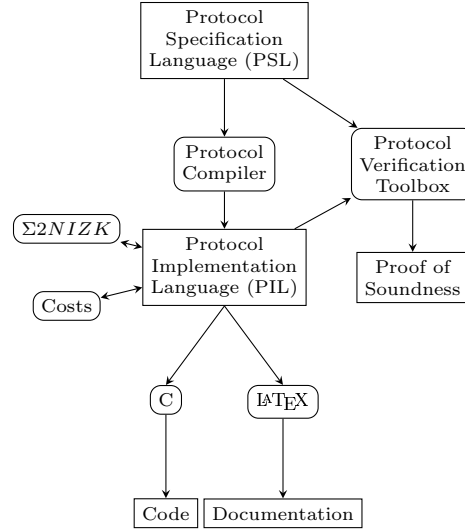


FIGURE 3.1: CACE typical workflow [1]

### 3.1.2 PSL

The Protocol Specification Language is a high level language of CACE Project WP3 for specifying Proofs of Knowledge based on Camenisch-Stadler notation. It allows specification of complex  $\Sigma$  protocols [1, 2].

The language is structured into blocks:

- Declarations - specifying all the variables used within the protocol

```

Declarations {
  Prime(1024) p; //specifies a prime number of 1024 bits
  Prime(160) q;
  Zmod+(q) x; //specifies an element of the additive residue class
               group modulo q
  Zmod*(p) g, y; //specifies elements of the multiplicative
                  residue class group modulo p
}
  
```

This example shows how to declare prime numbers as well as elements of a residue group.

- Input - specifying which of the variables are public and which are private (to the Verifier or the Prover)

```

Inputs {
  Public := y, p, q, g;
  ProverPrivate := x;
}
  
```

This example shows how to specify which are public known variables and which are known only to the prover. It is also possible to specify a variable known only to the verifier.

### 3. EXISTING FRAMEWORKS AND TOOLS

---

- Properties - specifying the properties of the protocol

```
Properties {  
  KnowledgeError := 80; //the required knowledge error, in this  
    example 2(-80)  
  SZKParameter := 80; //the Statistical Zero-Knowledge Parameter  
    specifies the tightness of the protocol  
  ProtocolComposition := P_1; //how the protocols are composed (  
    AND, OR, XOR, ...)  
}
```

- Specifying the protocol itself (which homomorphism to use and which relation needs to be proven)

```
SigmaPhi P_1 {  
  Homomorphism (phi : Zmod+(q) -> Zmod*(p) : (a) |-> (g^a)); //  
    specifies the homomorphism under which knowledge of the  
    preimage has to be proven  
  ChallengeLength := 80; //the bit length of the challenge the  
    verifier generates  
  Relation ((y) = phi(x)); //the relation which needs to be proven  
}
```

The previous steps combined give a specification of the Schnorr protocol:

```
Declarations {  
  Prime(1024) p;  
  Prime(160) q;  
  Zmod+(q) x;  
  Zmod*(p) g, y;  
}  
  
Inputs {  
  Public := y, p, q, g;  
  ProverPrivate := x;  
}  
  
Properties {  
  KnowledgeError := 80;  
  SZKParameter := 80;  
  ProtocolComposition := P_1;  
}  
  
SigmaPhi P_1 {  
  Homomorphism (phi : Zmod+(q) -> Zmod*(p) : (a) |-> (g^a));  
  ChallengeLength := 80;  
  Relation ((y) = phi(x));  
}
```



### 3.1.3 PIL

The low level/intermediate language of the CACE Project WP3 is PIL (Protocol Implementation Language). The language itself gives all the details of a protocol and is meant to be easy to understand and easy to learn. This can aid in verifying the correctness from a users point of view. The constructs of the language are completely specified and allow an automatic verification of correctness and checking of side-channels.

As a language on its own it has support for the following features:

- global shared constants (parameters)

```
Common (
  Z l_e = 1024;
  Z SZKParameter = 80;
  Prime(1024) n
) {
  ...
}
```

- global constants (parameters)

```
Prover (
  Zmod+(q) x;
  Zmod+(q) v
) {
  ...
}
```

- global variables

```
Prover (
  ...
) {
  Zmod+(q) s, r;
  ...
}
```

- local arguments
- conditionals
- loops
- functions

```
Def (Zmod*(p) _t_1): Round1(Void) {
  _r_1 := Random(Zmod+(q));
  _t_1 := (g^_r_1);
}
```

- predicates

### 3. EXISTING FRAMEWORKS AND TOOLS

---

```
x := Random(Zmod+(q));  
CheckMembership(x, Zmod+(q));  
Verify(x == x);
```

- type alias

```
_C = Int(80) _c;
```

Proof entities are specified as blocks and there is always a Common block with all declarations and definitions visible to all other blocks. Each block can define multiple functions that have inputs and outputs defined. A function consists of assignments or loops or conditional flow. The execution order is specified via block function pairs:

```
ExecutionOrder := (Prover.Round0, Verifier.Round0, Prover.Round1,  
    Verifier.Round1, Prover.Round2, Verifier.Round2);
```

The communication itself is specified via these functions. The inputs of the current function must match the output of the previous function. For example, The outputs of Round0 from Prover must match the inputs of Round0 from Verifier.

Again, the Schnorr protocol is used as an example, automatically generated from the PSL that was given in 3.1.2.

```

/*
 * This is PIL code automatically produced from /var/www/cace/tmp/cace
 * -861bd493b372e0d63fa9aafbcfa5613d/input.psl
 * Date: Tuesday, December 06, 2011
 * Time: 23:41:32
 *
 * The proof goal was potentially simplified to the following form:
 * P_1
 *
 */

ExecutionOrder := (Prover.Round0, Verifier.Round0, Prover.Round1,
  Verifier.Round1, Prover.Round2, Verifier.Round2);
Common (
  Z SZKParameter = 80;
  Prime(1024) p = 17;
  Prime(160) q = 1;
  Zmod*(p) y = 1, g=3
) {
}
Prover(Zmod+(q) x) {
  Zmod+(q) _s_1=1, _r_1=4;

  Def (Void): Round0(Void) {
  }

  Def (Zmod*(p) _t_1): Round1(Void) {
    _r_1 := Random(Zmod+(q));
    _t_1 := (g^_r_1);
  }

  Def (_s_1): Round2(_C=Int(80) _c) {
    _s_1 := (_r_1+(x*_c));
  }
}

Verifier() {
  Zmod*(p) _t_1;
  _C=Int(80) _c;

  Def (Void): Round0(Void) {
  }

  Def (_c): Round1(_t_1) {
    _c := Random(_C);
  }

  Def (Void): Round2(Zmod+(q) _s_1) {
    CheckMembership(_s_1, Zmod+(q));
    Verify((_t_1*(y^_c)) == (g^_s_1));
    // @ Verification equations for P_1

```

```

    }
}

```

## 3.2 ZKPD L

ZKPD L is a description language for describing Zero Knowledge Proofs of Knowledge [12].

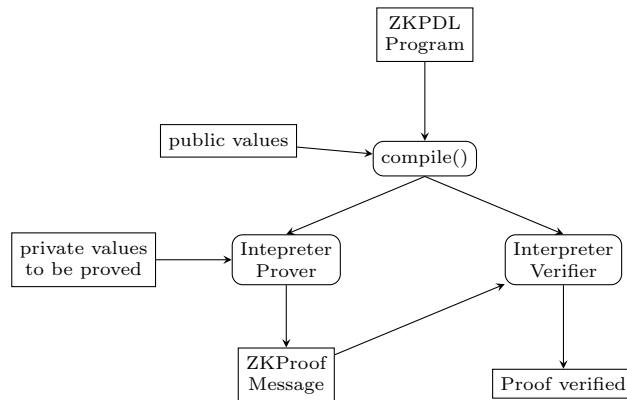


FIGURE 3.2: ZKPD L typical flow [12]

## 3.3 Comparison of CACE and ZKPD L

### 3.4 Other tools

#### 3.4.1 ANTLR

ANTLR (ANother Tool For Language Recognition) is a parser generator tool that generates LL(\*) parsers. The tool accepts grammar definitions as input files and produces output in the target language which can be specified between C, Java or Python.

The input to ANTLR is a context-free grammar that must be of the LL form. This means that there should be no left recursion or ambiguities when encountering the first element on the left. Left factoring is usually used to solve this, but this can sometimes lead to rules which have counter intuitive representation. The grammar can be augmented with syntactic and semantic predicates to cope with this [15, 14].

ANTLR can generate both lexers and parser. The two grammars can be combined in a single file. In such cases, parser rules are written in lowercase, while the lexer rules are written in uppercase. Upon generation, two separate entities will be created, a parser source and a lexer source in the target language (as illustrated in Figure 3.3).

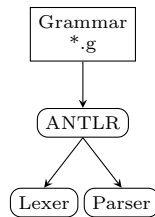


FIGURE 3.3: ANTLR Parser/Lexer generation

The output of the parser generated by ANTLR is a Parse Tree. ANTLR also allows specifying a tree transformation to apply to this Parse Tree. This can be used to automatically generate an Abstract Syntax Tree (AST).

ANTLR can also generate a tree parser/walker that visit each node of the AST and applies a certain operation or produces a certain output. The generation of such a walker is illustrated in Figure 3.4.

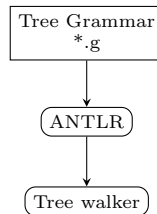


FIGURE 3.4: ANTLR Parser/Lexer generation

### 3.4.2 LLVM

LLVM (Low Level Virtual Machine) is a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs [10].

Traditional compilers were tailored for only a few languages (with the exception of GCC). However, all traditional compilers suffer from the large inter-dependency of the basic blocks (Front-end, Optimizer, Back-end). LLVM tries to solve this by providing an intermediate form called the LLVM IR. A typical flow involving the basic blocks is depicted in Figure 3.5.

Due to its modularity, LLVM has recently seen increased usage in a number of independent fields:

- implementing a C/C++ compiler (Clang)
- implementing C-to-HDL translation
- implementing a Haskell compiler (GHC)

### 3. EXISTING FRAMEWORKS AND TOOLS

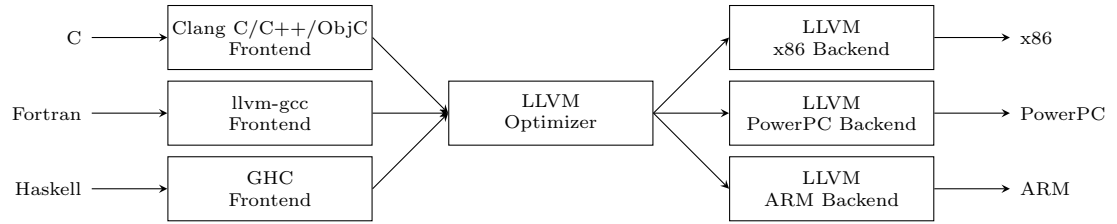


FIGURE 3.5: LLVM typical workflow [8]

- implementing Secure Virtual Architectures (SVA)
- implementing dynamic translation
- implementing OpenGL drivers (Mac OS X)
- implementing OpenCL drivers (AMD)

#### LLVM IR

The LLVM IR is the intermediate representation language of the LLVM project. On its own it is a first-class language with well defined semantics [8, 9]. Variables are in the SSA (Static Single Assignment) form meaning that they can be only assigned once and they keep that value for their entire lifetime. All the values residing in memory need to be loaded to a variable first and stored back to memory if they wish to be saved. Instructions operate solely upon variables. In this respect, the LLVM IR resembles the assembly language of a infinitely many registers Load-Store based RISC processor.

```
unsigned add1(unsigned a, unsigned b) {  
    return a+b;  
}  
  
// Perhaps not the most efficient way to add two numbers.  
unsigned add2(unsigned a, unsigned b) {  
    if (a == 0) return b;  
    return add2(a-1, b+1);  
}
```

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}  
  
define i32 @add2(i32 %a, i32 %b) {  
entry:  
    %tmp1 = icmp eq i32 %a, 0  
    br i1 %tmp1, label %done, label %recurse
```

```
recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

The central concept in constructing LLVM IR is the Module. Each module consists of functions, global variables and symbol table entries. Modules can be combined using the LLVM linker [11].

### 3.4.3 GEZEL

GEZEL is a cycle-accurate hardware description language (HDL) using the Finite-State-Machine + Datapath (FSMD) model [5].

The basic element is a Signal Flow Graph. It groups operations that are to be executed concurrently in the same clock cycle.

```
sfg increment {
    a = a + 1;
}
```

One or more of these SFGs are used to form a datapath which is the main building block. It is the smallest GEZEL unit that can stand on its own and be simulated [5]. A datapath can be thought of as a *module* in Verilog or an *entity* in VHDL. Here is a full contained example of a counter in GEZEL:

```
dp counter(out value : ns(2)) {
    reg c : ns(2);
    always {
        value = c;
        c = c + 1;
        $display("Cycle ", $cycle, ": counter = ", value);
    }
}

system S {
    counter;
}
```

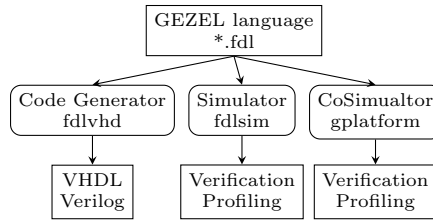


FIGURE 3.6: GEZEL workflow [5]

Figure 3.6 shows how the GEZEL language can be used as an input to:

- fdlvhd - a generator that can generate synthesizable VHDL or Verilog
- fdlsim - a cycle accurate simulator used to verify and validate the design
- gplatform - a co-simulation tool used for HW/SW co-design purposes

The co-simulation tool allows to cosimulate GEZEL designs with instruction-set simulations [5]. Supported processors are ARM, AVR, 8051, MicroBlaze and PicoBlaze. The cosimulation tool allows for designing a processor-coprocessor pair for a general purpose processor and a custom dedicated coprocessor.



## Chapter 4

# Custom framework

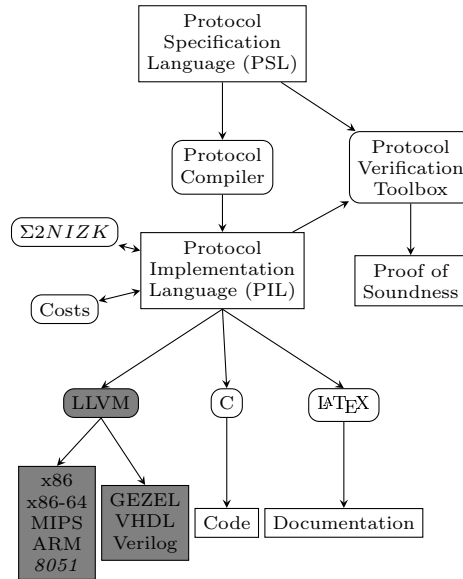


FIGURE 4.1: Custom framework (extensions to CACE Zero Knowledge Compiler highlighted)

## 4.1 Motivation

### 4.1.1 Domain specific language

C is not a very safe language for cryptography applications. This can be seen already from the goal of C which was to design a “portable” assembler. The design choice was not to sacrifice efficiency so some things are intentionally left undefined or unspecified in the standard. It is left to the end compilers to implement it as they chose. The usual choice is just to implement in the most efficient way for the target platform. For example, the following code uses this unspecified behavior:

```
#include <stdio.h>

int b(void) { puts("3"); return 3; }
int c(void) { puts("4"); return 4; }

int main(void)
{
    int a = b() + c();
    printf("%d\n", a);

    return 0;
}
```

The evaluation order is unspecified so depending on the compiler and the platform, the output may be 3, 4, 7 or 4, 3, 7. The crypto world should not have undefined nor unspecified behaviors so this is why a domain specific language is a key point of this thesis.

#### 4.1.2 Extending CACE Zero Knowledge Compiler

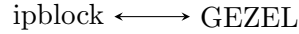
The C code generated by CACE uses GNU GMP which is a multi-precision arithmetic library. This library is tailored for desktop computers and is not well suited for small embedded devices. Granted, it is possible to plug in a custom multi-precision library but it is impossible, error prone and inefficient to design a library for every imaginable device. The fact that this compiler already uses its own domain specific language makes it a useful candidate for extension. It is possible to use LLVM as a low level language and assure portability and type strictness.

ZKPDL also has its own domain specific language. The approach taken, with creating an interpreter is considered too inefficient for embedded devices.

As the CACE project is deemed more suited for the task, the custom framework will extend CACE. The custom framework will aim to be a compiler that can produce:

- Interpreted code
- JIT (Just In Time) compiled code
- Compiled code

Also, since its intermediate form is of the SSA (Static Single Assignment) form, it is believed that more aggressive optimizations are possible. Since the DFG (Data Flow Graph) can be easily extracted from the SSA form, hardware realizations are also made possible.



ipblock  $\longleftrightarrow$  GEZEL

FIGURE 4.2: GEZEL and ipblock

## 4.2 Extensions to GEZEL

A modification to GEZEL was needed to allow interactivity. A terminal ipblock was made during the course of this thesis. This ipblock allows connecting to the host via pseudo-terminals or to other devices via serial ports as shown in Figure 4.2.

The ipblock terminal is based on POSIX termios functionality so this limits the simulator availability to POSIX systems.

## 4.3 Extensions to CACE Zero Knowledge Compiler

### 4.3.1 Terminal functionality

The C support library has been extended with terminal functionality. Terminal communication is simpler as it can be implemented over an RS232 communication protocol. The file adding this functionality is `comm-term.c`. Changes to the adapter `comm.c` were also needed to forward the requests. After implementing the terminal functionality, a test was carried between a generated prover and a verifier and later on between the generated entities and dummy entities written in GEZEL. This allowed for cross-testing the terminal implementation of GEZEL as well.

### 4.3.2 Lower end

Extensions to the CACE frameworks are presented in Figure 4.1. LLVM has already proven itself in many areas so the choice was made to transform the lower-level PIL into LLVM IR. LLVM IR can then be used as a starting point for multiple targets.

The custom part is further explained in Figure 4.3. The input file is processed by a PIL fronted which generates LLVM IR code which can be fed to either existing optimizers or a custom one can be written. The resulting IR can then be fed to a backend for an existing architecture or a new back-end can be made (such as 8051 and GEZEL in this case).

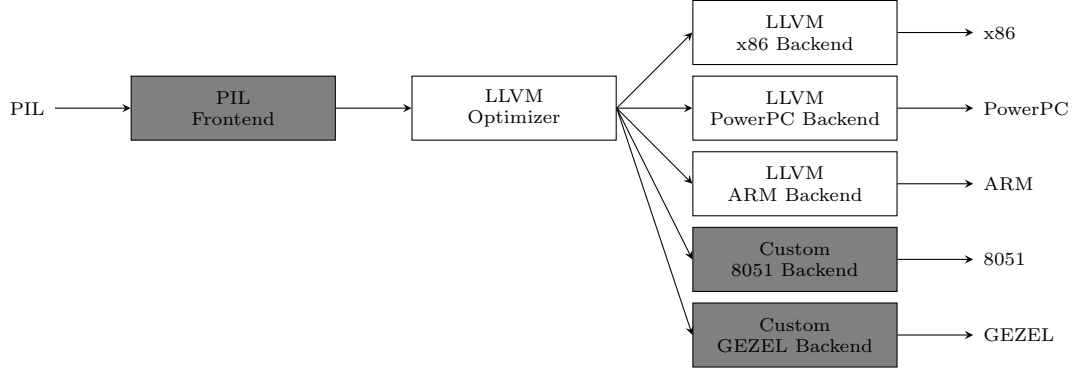


FIGURE 4.3: LLVM custom workflow (changes highlighted)

### 4.3.3 Extensions to PIL

#### Multiple blocks

The base PIL language supported only a Prover and a Verifier block besides the Common block. This makes it impossible to implement a more than 2 parties protocol. As this is simply a relaxation of the rules, the change to support 3 and more parties was trivial.

#### Compile time/constant expressions

Compile time/constant expressions were needed to allow more advanced specifications of protocols. The PIL language does not specify constant expressions as parameters of variables. When designing more complex protocol where parameters of variables have to be adjusted, one needs constant expressions. The benefit of constant expressions can be seen from the following example:

```

Common (
  Z l_n = 1024;
  Z l_f = 160;
  Z l_e = 410;
  Z l_e_1 = 120;
  Z l_v = 1512;
  Z l_phi = 80;
  Z l_H = 160;
  Prime(l_n) n = 17
) {
}

Smartcard (
  Zmod*(n) p, q;
  Int(l_f + l_phi + l_H) f
) {
  Zmod*(n) S, _Z, R;

```

```

Int(l_n + l_phi) v_1;
Int(l_v) v;
...

```

Without constant expressions, one would need to recompute the values manually and enter them every time a change was desired. This re-computation and reentering is prone to errors and as such undesirable when designing a crypto framework. The grammar change to allow constant expressions is very simple

```

group      :      ( 'Zmod+' | 'Zmod*' ) ' ( ' expr ' ) '
               |      'Prime' ' ( ' expr ' ) '
               |      'Int' ' ( ' expr ' ) '
               |      'Z'
               ;

```

This change allows it only syntactically so some semantic processing will be needed to ensure that they are constant expressions which can be evaluated at compile time by the compiler.

### Type inference

To be able to check for correctness, one needs to determine the resulting type of a certain expression. This can be also used to allow one to omit a type declaration. The following example illustrates this:

```

Zmod*(p) b;
x := Random(Int(80));
a := b^x;

```

For the case of variable x, it's type can be inferred as `Int(80)` since the `Random` function can only return a random value of the provided type. For the case of variable a, the situation is a bit more complex. However, if the operation of exponentiation is defined as applying the multiplication operation many times, then the type of a can only be `Zmod*(p)` by the definition of the multiplicative modular residue group. A similar case can be made when multiplying an element of the additive modular residue group with an integer. This means that the type inference is well defined for any acceptable operation in PIL.

## 4.4 PIL front-end

PIL frontend flow is depicted in Figure 4.5. An input PIL is read by the Lexer producing input for the Parser. The Parser reads these and generates an abstract syntax tree (AST) which is fed to the Codegen tree-walker that generates the code (in the form of LLVM IR). Both the lexer grammar and the parser grammar are specified in the file `pil.g`. The tree-walker and the code generator is specified in the file `codegen.g`.

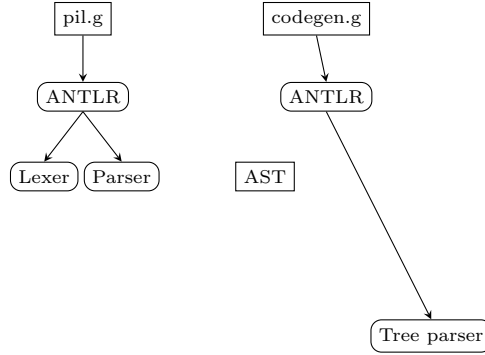


FIGURE 4.4: ANTLR workflow

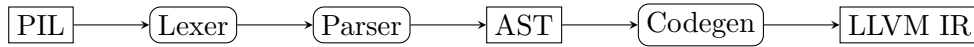


FIGURE 4.5: PIL frontend flow

The code generation process generates one module per block. The Common block module is augmented with the functions provided by the VM. Every other block except the Common block gets a Common block linked in. This process is depicted in Figure 4.6.

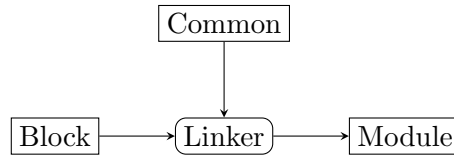


FIGURE 4.6: Linker

The private parameters as well as global variables are transformed as LLVM IR global variables, the private parameters being constant in this case. Each function of a block gets transformed as an LLVM IR function with its input and output arguments transformed as such. LLVM allows for multiple return values so this is used as well when there are multiple return values.

LLVM's constant folder is used to evaluate constant time expressions to simplify them into a constant value.

When dealing with group arithmetic which LLVM does not support, there are two possibilities:

1. extend the LLVM's type system to include group types - this involves editing the core LLVM files, adding a new `DerivedType`, changing the bit-code format and adding changes to the LLVM parser (both binary and textual). Also, binary operations have to be redefined to support these new types [4].
2. use a simpler, existing LLVM type and make the compiler do the extra work - the `IntegerType` of LLVM can be used for arbitrarily sized integers.

The first option allows for preserving the information about the modular residue groups to the lowest level. The transformation to a primitive type supported by the target architecture happens only at a later stage. Or, if the target architecture supports modular residue groups, there is no need for a transformation, only a translation. This allows for a code that is more verifiable and more secure as the properties are kept to the lowest possible levels (no exploits can be made under secure starting conditions). The disadvantage of this method is the changes that need to be introduced at the heart of such a complex framework as LLVM is. This is both time consuming and error prone. Also, it becomes more difficult to track newer versions of LLVM (with possibly better optimizations and more features) if the changes are not integrated back into the main project. This method also breaks compatibility with existing LLVM applications so a specially patched version of LLVM needs to be distributed.

The second option is easier as the changes to the compiler are only local and do not break compatibility with existing LLVM applications. It is also easier as there is no hard work associated with changing the core of LLVM. More work has to be assigned to the compiler because it needs to track types and apply the appropriate operations in the case of modular residue groups. The burden is also on the back-end if an architecture supports modular residue groups since it now has to regenerate the lost information. An example of such an architecture can be an automatic verifier whose job is now made more complex. This re-generation of information might also not be always completely accurate.

The first approach was attempted first but was deemed too time consuming and error prone. Also it would require some standardization with LLVM upstream to allow future tracking. The second approach was chosen as the one to base the work on. Type inference was used to deduce the resulting type of an operation. The operation was then sent to appropriate 3 argument operations with the first 2 being the operands as integers and the 3rd operand being the modulus.

#### 4.4.1 Type alias

Type aliases are evaluated and substituted by the parser. The reasoning behind them can go both ways. Although they are more semantic than syntactic since they convey information (as a creation of another type), they are simple enough to be evaluated and substituted by the parser. This can be paralleled with a pre-processor in languages which support it.

#### 4.4.2 Type inference

The easiest approach for determining a resulting type from two operands is to use double dispatch and code each of the resulting functions. There were some attempts at bringing multiple dispatch to C++ but this has still not been realized [16].

A common way to simulate it in C++ is to use a visitor pattern. The base interface or the base class defines accept methods. The visitor then calls each of

the classes dispatching itself as a parameter. Here is how it done with GroupT and NumberT:

```
class NumberT {
...
virtual NumberT *addWithSubFrom(const NumberT *first);
...
virtual NumberT *operator+(const NumberT &other) const {
    return other.addWithSubFrom(this);
}
}
```

Here the addWithSubFrom method is the accept method. The same was done for each of the operators. When the compiler encounters a + b where a and b have NumberT as a base class, it will call the operator+. This call happens through a vtable so the this pointer always points to the accurate type for the first operand. By calling the addWithSubFrom method, another vtable dispatch happens and now this points to the accurate type for the second operand [3]. Since both of the operands are now correctly resolved it remains to return the resulting type of the operation. Extending the type inference simply involves writing a function for each combination of the operand types. This is somewhat easier than using Run Time Type Information and writing an if-then-else or similar for each of the cases.

Since GroupT derives from NumberT the operator calls will remain this way unless overridden. The accept methods will need to be overridden as they provide the custom logic for each of the combinations.

The classes NumberT and GroupT also define methods for creating an LLVM instruction for addition, subtraction, multiplication and exponentiation. These function as adapters and make the process of code-generation uniform. Here is an excerpt from the code generation:

```
expr[const char *id] returns [Value *value, NumberT *type]
:
|      ^('+' lhs=expr[$id] rhs=expr[$id]) {
|          $type = *$lhs.type + *$rhs.type;
|          $value = $type->createAdd($lhs.value, $rhs.value);
|      }
```

## 4.5 GEZEL back-end



## Chapter 5

### Use case: Direct Anonymous Attestation



## Chapter 6

# Conclusion

### 6.1 Future work

LLVM should be extended with the modular residue group types and this work should be submitted upstream. The gains were already discussed in section 4.4. This will require some standardization before changes are applied and will take some time. This is one of the main reasons it was not done within the course of this thesis.

While extending the CACE Project Zero Knowledge Compiler, it was noticed that each of the tools have their own lower-level language. In the long run, this only makes it difficult to contribute new features/extend, optimize and target new platforms. It is possible for LLVM to become a common language for all the tools from the CACE Project. This not only makes it portable to a wider range of platforms but it also moves the burden of optimizations to an already proven framework.

The framework developed during the course of this thesis could and should serve as a starting point for HW-SW codesign automation tools. Due to lack of time, and lack of support in the simulators, limited exploration was done in this context, targeting only GEZEL code. There is no reason not to include C-GEZEL code co-generation and employ an 8051. Such a micro-controller is usually found on modern-day smart-cards.



# Appendices



```

grammar pil;

options {
    output=AST;
    //ASTLabelType=CommonTree;
    ASTLabelType=pANTLR3_BASE_TREE;
    language=C;
}

tokens {
    INTERVAL; PARAM; PROOF; ORDER; BLOCK; INP; OUTPUT; BODY;
    VARIABLE; FUNCTION;
}

@includes {
    #include <map>
    using namespace std;
}

@postinclude {
    struct cmp_str
    {
        bool operator()(const char *a, const char *b)
        {
            return strcmp(a, b) < 0;
        }
    };

    map<const char *, pANTLR3_BASE_TREE, cmp_str> aliases;
}

proof      :      execution_order (block)* -> ^(PROOF ^(ORDER
    execution_order) ^(BLOCK block)*)
    ;

execution_order
    :      'ExecutionOrder' ':= ' '(' step (',' step)* ')' ';' -> (
        step)*
    ;

step      :      ID '.' ID -> ^(ID ID)
    ;

block      :      ID '(' (params+=variable_declaration)? (',' params+=
    variable_declaration)* ')'
        '{' (vars+=variable_declaration ';' )* (
            function_declaration)* '}'
        -> ^(ID ^(PARAM $params*)? ^(VARIABLE $vars*)?
            ^(FUNCTION function_declaration)*)?
    ;

function_declaration
    :      'Def' '(' function_params ')' ':' ID '('
        function_params ')' '{' body '}'

```

## 6. CONCLUSION

---

```

                                -> ^ (ID ^ (OUTPUT function_params) ^ (INP
                                function_params) ^ (BODY body)?)
                                ;
function_params
:      'Void'
|      argument_declaration ( ';' '! argument_declaration)*
;

argument_declaration
:      type! ? variable_init [$type.tree] ( ',' '! variable_init [
$type.tree] ) *
;

body
:      (statement)*
;

statement
:      ID ':=' expr ';' -> ^ (':=' ID expr)
|      function_call ';' '!
|      ifknown
;

ifknown :      'IfKnown' '(' topExpr ')' '{'
              body
              '}' ( 'Else' '{'
              body
              '}' ) ?
;

topExpr :      expr (( '==' '^' '!=' '^' ) expr) ?
;

expr :      ( '+' '^' | '-' ) ? priExpr (( '+' '^' | '-' '^' ) priExpr)*
;

priExpr :      (( '(' '! expr ')' '!' ) | terminal) (( '^' '^' | '*' '^' ) priExpr) ?
;

terminal
:      NUMBER
|      function_call
|      ID
;

function_call
:      'Random' ^ '(' '! type ')' '!'
|      'Verify' ^ '(' '! topExpr ')' '!'
|      'CheckMembership' ^ '(' '! ID ', '! type ')' '!'
;

variable_declaration
:      type! variable_init [$type.tree] ( ',' '! variable_init [
$type.tree] ) *

```



```

;

variable_init [pANTLR3_BASE_TREE t]
:      ID ('=' NUMBER)? -> ^(ID { $t } NUMBER?)
;

type   :      alias
        |      group
        |      interval
;

alias   :      (ID '=' )=> ID! '='! (group { aliases [(const char*)$ID.
        text->chars] = $group.tree; } | interval { aliases [(const char*)$ID.
        text->chars] = $interval.tree; })
        |      ID -> { aliases [(const char*)$ID.text->chars] }
;

group   :      ('Zmod+'|'Zmod*')^ ('(! expr ') '! )
        |      'Prime'^ ('(! expr ') '! )
        |      'Int'^ ('(! expr ') '! )
        |      'Z'
;

interval:      '[' from=expr ',' to=expr ']' -> ^(INTERVAL $from $to)
;

ID      :      ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
;

NUMBER  :      ('0'..'9'|'a'..'z'|'A'..'Z')+;

COMMENT
:      '//' ^('\'n'|\'r')* \'r'? \'n' { $channel=HIDDEN; }
|      '/*' ( options { greedy=false; } : . )* '*/' { $channel=HIDDEN; }
;

NEWLINE :      \'r'? \'n' { $channel=HIDDEN; } ;

WS      :      (' |\t')+ { $channel=HIDDEN; } ;

```

```

tree grammar codegen;

options {
    tokenVocab=pil;
    ASTLabelType=pANTLR3_BASE_TREE;
    language=C;
}

@includes {
    #include "llvm/DerivedTypes.h"
    #include "llvm/ExecutionEngine/ExecutionEngine.h"
    #include "llvm/ExecutionEngine/JIT.h"
    #include "llvm/LLVMContext.h"
    #include "llvm/Linker.h"

```

```
#include "llvm/Module.h"
#include "llvm/PassManager.h"
#include "llvm/Analysis/Verifier.h"
#include "llvm/Analysis/Passes.h"
#include "llvm/Target/TargetData.h"
#include "llvm/Transforms/Scalar.h"
#include "llvm/Support/IRBuilder.h"
#include "llvm/Support/TargetSelect.h"
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <map>
#include "GroupType.h"
using namespace std;
using namespace llvm;

struct Variable {
    bool variable;
    NumberT *type;
    Value *value;
};

struct Arg {
    const char *id;
    NumberT *type;
    Value *value;
    bool global_reference;
};
}

@postinclude {
    struct cmp_str
    {
        bool operator()(const char *a, const char *b)
        {
            return strcmp(a, b) < 0;
        }
    };

    static Module *Common = new Module("Common", getGlobalContext());
    static Module *TheModule;
    static IRBuilder<> Builder(getGlobalContext());
    static map<const char*, Variable, cmp_str> Vars;
    static map<const char*, NumberT*, cmp_str> Types;

    Value* operator_call(const char *name, const char *fname, Value
        *a, Value *b, Value *mod)
    {
        Function *function = TheModule->getFunction(fname);

        vector<Value*> args;

        args.push_back(a);
        args.push_back(b);
    }
}
```

```

        args.push_back(mod);

        return Builder.CreateCall(function, args, name);
    }

    void init_common()
    {
        Function::Create(
            FunctionType::get(
                Type::getIntNTy(getGlobalContext(),
                               1024),
                vector<Type*>(0, Type::getIntNTy(
                    getGlobalContext(), 1024)),
                false),
            Function::ExternalLinkage, "Random", Common);
        Function::Create(
            FunctionType::get(
                Type::getInt1Ty(getGlobalContext()),
                vector<Type*>(1, Type::getIntNTy(
                    getGlobalContext(), 1024)),
                false),
            Function::ExternalLinkage, "CheckMembership",
            Common);

        Function::Create(
            FunctionType::get(
                Type::getIntNTy(getGlobalContext(),
                               1024),
                vector<Type*>(3, Type::getIntNTy(
                    getGlobalContext(), 1024)),
                false),
            Function::ExternalLinkage, "modadd1024", Common
        );

        Function::Create(
            FunctionType::get(
                Type::getIntNTy(getGlobalContext(),
                               1024),
                vector<Type*>(3, Type::getIntNTy(
                    getGlobalContext(), 1024)),
                false),
            Function::ExternalLinkage, "modsub1024", Common
        );

        Function::Create(
            FunctionType::get(
                Type::getIntNTy(getGlobalContext(),
                               1024),
                vector<Type*>(3, Type::getIntNTy(
                    getGlobalContext(), 1024)),
                false),
            Function::ExternalLinkage, "modmul1024", Common
        );

        Function::Create(

```

```
FunctionType::get(
    Type::getIntNTy(getGlobalContext(),
        1024),
    vector<Type*>(3, Type::getIntNTy(
        getGlobalContext(), 1024)),
    false),
Function::ExternalLinkage, "modexp1024", Common
);

Function::Create(
    FunctionType::get(
        Type::getInt1Ty(getGlobalContext()),
        vector<Type*>(1, Type::getInt1Ty(
            getGlobalContext()))),
    false),
    Function::ExternalLinkage, "Verify", Common);
}

NumberT *NumberT::addWithSubFrom(const NumberT *first) const {
    return const_cast<NumberT*>(first); }
NumberT *NumberT::addWithSubFrom(const GroupT *first) const {
    return const_cast<GroupT*>(first); }

NumberT *NumberT::mulWithExpOn(const NumberT *first) const {
    return const_cast<NumberT*>(first); }
NumberT *NumberT::mulWithExpOn(const GroupT *first) const {
    return const_cast<GroupT*>(first); }

Value * NumberT::createNeg(Value *a) const {
    return Builder.CreateNeg(a);
}

Value * NumberT::createAdd(Value *a, Value *b) const {
    return Builder.CreateAdd(a, b);
}

Value * NumberT::createSub(Value *a, Value *b) const {
    return Builder.CreateSub(a, b);
}

Value * NumberT::createMul(Value *a, Value *b) const {
    return Builder.CreateMul(a, b);
}

Value * NumberT::createExp(Value *a, Value *b) const {
    return Builder.CreateMul(a, b);
}

Value * GroupT::createNeg(Value *a) const {
    return operator_call("modsub1024", ConstantInt::get(
        getGlobalContext(), APInt(1024, 0)), a, Builder.
        CreateIntCast(this->getModulusConstant(),
            IntegerType::get(getGlobalContext(), 1024), false));
}
```

```

    Value * GroupT::createAdd(Value *a, Value *b) const {
        return operator_call("", "modadd1024", a, b, Builder.
            CreateIntCast(this->getModulusConstant(),
                IntegerType::get(getGlobalContext(), 1024), false));
    }

    Value * GroupT::createSub(Value *a, Value *b) const {
        return operator_call("", "modsub1024", a, b, Builder.
            CreateIntCast(this->getModulusConstant(),
                IntegerType::get(getGlobalContext(), 1024), false));
    }

    Value * GroupT::createMul(Value *a, Value *b) const {
        return operator_call("", "modmul1024", a, b, Builder.
            CreateIntCast(this->getModulusConstant(),
                IntegerType::get(getGlobalContext(), 1024), false));
    }

    Value * GroupT::createExp(Value *a, Value *b) const {
        return operator_call("", "modexp1024", a, b, Builder.
            CreateIntCast(this->getModulusConstant(),
                IntegerType::get(getGlobalContext(), 1024), false));
    }
}

proof[const char *part]
    : { init_common(); }^(PROOF execution_order common (block[$part
        ]))*
    ;

execution_order
    :      ^(ORDER (step))*
    ;

step      :      ^(ID ID)
    ;

common    :      ^(BLOCK ^(ID { TheModule = Common; } param? global?
        function?))
    ;

block[const char *part]
    :      ^(BLOCK ^(ID {
        Linker *linker = new Linker((const char*)$ID.
            text->chars, (const char*)$ID.text->chars,
            getGlobalContext());
        linker->LinkInModule(Common);
        TheModule = linker->getModule();
        } param? global? function?)) { TheModule->dump
            (); }
    ;

param     :      ^(PARAM (param_declaration))*
    ;

```

## 6. CONCLUSION

---

```
global      :      ^(VARIABLE (global_declaration)*)
              ;

function:      ^(FUNCTION (function_declaration)*)
              ;

param_declaration returns [Value *value]
      :      ^(par=ID type=type_declaration
              (NUMBER
               { $value = ConstantInt::get(getGlobalContext(),
               APInt($type.type->getBitWidth(), (const char
               *)$NUMBER.text->chars, 10)); }
               | { $value = ConstantInt::get(getGlobalContext(),
               , APInt($type.type->getBitWidth(), 0)); } ))
      {
          Variable parx;
          parx.value = $value;
          parx.type = $type_declaration.type;
          parx.variable = false;

          Vars[(const char*) $par.text->chars] = parx;
      }
      ;

global_declaration returns [Value *value]
      :      ^(var=ID type_declaration
      {
          $value = new GlobalVariable(*TheModule,
          $type_declaration.type->getType(), false, GlobalVariable::
          ExternalLinkage, 0, (const char *) $svar.text->chars);
      } (NUMBER)?)
      {
          Variable varx;
          varx.value = $value;
          varx.type = $type_declaration.type;
          varx.variable = true;

          Vars[(const char*) $var.text->chars] = varx;
      }
      ;

argument_declaration returns [Arg *arg]
      :      (^ (ID type_declaration)) => ^(var=ID type_declaration {
          arg = new Arg();
          $arg->id = (const char*)$ID.text->chars;
          $arg->type = $type_declaration.type;
          $arg->global_reference = false;
          })
      | (var=ID {
          arg = new Arg();
          $arg->id = (const char*)$ID.text->chars;
          $arg->type = Vars[$arg->id].type;
          $arg->value = Vars[$arg->id].value;
          $arg->global_reference = true;
          })
      ;
```

```

type_declaration returns [NumberT *type]
:
    group { $type = $group.type; }
|
    interval { $type = $interval.type; }
;

function_declaration returns [Function *func, vector<Arg *> outs,
    vector<Arg *> inps]
:
    ^ (ID ^ (OUTPUT ( 'Void' | ( out=argument_declaration { $outs
        .push_back($out.arg); } ) * ) )
        ^ (INP ( 'Void' | ( inp=argument_declaration { $inps
            .push_back($inp.arg); } ) * ) )
        {
            vector<Type *> Inps;

            for (vector<Arg *>::iterator i = $inps.begin();
                i != $inps.end(); i++) {
                Inps.push_back((*i)->type->getType());
            }

            FunctionType *FT;

            if ($outs.size() == 0) {
                FT = FunctionType::get (Type::getVoidTy(
                    getGlobalContext()), Inps, false);
            } else {
                FT = FunctionType::get ($outs[0]->type->
                    getType(), Inps, false);
            }

            $func = Function::Create(FT, Function::
                ExternalLinkage, (const char*) $ID.text->
                chars, TheModule);

            BasicBlock *entry_block = BasicBlock::Create(
                getGlobalContext(), "entry", $func);

            Builder.SetInsertPoint(entry_block);

            for (vector<Arg *>::iterator def = $outs.begin();
                def != $outs.end(); def++) {
                if (!(*def)->global_reference) {
                    Variable varx;
                    varx.type = (*def)->type;
                    varx.variable = false;

                    Vars[(*def)->id] = varx;
                }
            }

            vector<Arg *>::iterator def = $inps.begin();

            for (Function::arg_iterator arg = $func->
                arg_begin(); arg != $func->arg_end(); arg++,
                def++) {

```

```
        if (!(*def)->global_reference)
        {
            arg->setName((*def)->id);

            Variable varx;

            varx.value = arg;
            varx.type = (*def)->type;
            varx.variable = false;

            Vars[(*def)->id] = varx;
        } else {
            arg->setName((*def)->id);

            Builder.CreateStore(arg, (*def)
                               ->value);
        }
    }
    body?
    {
        if ($outs.size() > 0) {
            Variable varx = Vars[$outs[0]->id];

            if (varx.variable) {
                Builder.CreateRet(Builder.
                                CreateLoad(varx.value, $outs
                                             [0]->id));
            } else {
                Builder.CreateRet(varx.value);
            }
        } else {
            Builder.CreateRetVoid();
        }
    }
;

body      :      ^(BODY statement*)
;

statement
:      assignment
|      function_call
;

assignment returns [Value *value]
:      ^(':= ' ID expr[(const char*)$ID.text->chars])
{
    const char *id = (const char*) $ID.text->chars;

    if (Vars.find(id) == Vars.end()) {
        Variable varx;
        varx.variable = false;
        varx.type = $expr.type;
        Vars[id] = varx;
    }
}
```



```

    }

    Variable dest = Vars[id];

    if (!dest.variable) {
        dest.value = $expr.value;

        Vars[id] = dest;
    } else {
        $value = Builder.CreateStore(Builder.
            CreateIntCast($expr.value, dest.type
                ->getType(), false), dest.value);
    }
}

;

function_call returns [Value *value, NumberT *type]
:
    ^('Random' type_declaration)
    {
        Function *CalleeF = TheModule->getFunction("
            Random");
        $value = Builder.CreateCall(CalleeF, vector<
            Value*>(), "calltmp");

        $type = $type_declaration.type;
    }
|
    ^('CheckMembership' argument type_declaration)
    {
        Function *CalleeF = TheModule->getFunction("
            CheckMembership");

        vector<Value *> ArgsV;

        ArgsV.push_back($argument.value);

        $value = Builder.CreateCall(CalleeF, ArgsV, "
            calltmp");

        $type = new GroupT(APInt(1024,0));
    }
|
    ^('Verify' expr[" verify"])
    {
        Function *CalleeF = TheModule->getFunction("
            Verify");

        vector<Value *> ArgsV;

        ArgsV.push_back($expr.value);

        $value = Builder.CreateCall(CalleeF, ArgsV, "
            calltmp");

        $type = new GroupT(APInt(1024,0));
    }
;

```

## 6. CONCLUSION

---

```

argument returns [Value *value]
:      expr["arg"] { $value = $expr.value; }
|      group
;

group returns [NumberT *type]
:      'Z' { $type = new NumberT(32); }
|      ^('Prime' expr["group"]) { $type = new NumberT(
    static_cast<ConstantInt*>($expr.value)->getValue().
    getLimitedValue(1024)); }
|      ^('Int' expr["group"]) { $type = new NumberT(
    static_cast<ConstantInt*>($expr.value)->getValue().
    getLimitedValue(1024)); }
|      ^('Zmod+' expr["group"]) { $type = new GroupT(
    static_cast<ConstantInt*>($expr.value)->getValue()); }
|      ^('Zmod*' expr["group"]) { $type = new GroupT(
    static_cast<ConstantInt*>($expr.value)->getValue()); }
;

alias returns [NumberT *type]
:      ^('= ' ID (group { $type=$group.type; } | interval { $type
    =$interval.type; })))
    {
        Types[(const char*)$ID.text->chars] = $type;
    }
;

interval returns [NumberT *type]
:      ^(INTERVAL expr["lhs"] expr["rhs"]) { $type = new GroupT(APInt
    (32, 0)); }
;

expr[const char *id] returns [Value *value, NumberT *type]
:      ('-' val=expr[$id]) { $type = $val.type; $value = $type
    ->createNeg($val.value); }
|      ^('+ ' lhs=expr[$id] rhs=expr[$id]) { $type = *$lhs.type
    + *$rhs.type; $value = $type->createAdd($lhs.value, $rhs.
    value); }
|      ^('- ' lhs=expr[$id] rhs=expr[$id]) { $type = *$lhs.type
    - *$rhs.type; $value = $type->createSub($lhs.value, $rhs.
    value); }
|      ^('* ' lhs=expr[$id] rhs=expr[$id]) { $type = *$lhs.type
    * *$rhs.type; $value = $type->createMul($lhs.value, $rhs.
    value); }
|      ^('^ ' lhs=expr[$id] rhs=expr[$id]) { $type = *$lhs.type
    ^ *$rhs.type; $value = $type->createExp($lhs.value, $rhs.
    value); }
|      ^('==' lhs=expr[$id] rhs=expr[$id]) { $value = Builder.
    CreateICmpEQ($lhs.value, $rhs.value, $id);}
|      ^('!=' lhs=expr[$id] rhs=expr[$id]) { $value = Builder.
    CreateICmpNE($lhs.value, $rhs.value, $id);}
|      function_call { $type = $function_call.type; $value =
    $function_call.value;}
|      ID

```

```

    {
        Variable varx = Vars[(const char *) $ID.text->
            chars];
        $type = varx.type;
        if (!varx.variable) {
            $value = Builder.CreateIntCast(varx.
                value, IntegerType::get(
                    getGlobalContext(), 1024), false, (
                    const char*)$ID.text->chars);
        } else {
            $value = Builder.CreateIntCast(Builder.
                CreateLoad(varx.value, (const char*)
                    $ID.text->chars), IntegerType::get(
                    getGlobalContext(), 1024), false, (
                    const char*)$ID.text->chars);
        }
    }
    | NUMBER {$type = new NumberT(1024); $value = ConstantInt
        ::get(getGlobalContext(), APInt(1024, (const char*) $NUMBER.
            text->chars, 10));}
    ;

```



# Bibliography

- [1] *A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on  $\Sigma$ -Protocols*. An extended abstract of this work will be presented at ESORICS 2010 stephan.krenn@bfh.ch; thomas.schneider@trust.rub.de 14825 received 10 Jun 2010, last revised 4 Aug 2010. 2010. URL: <http://eprint.iacr.org/2010/339>.
- [2] Endre Bangerter et al. *YACZK: Yet Another Compiler for Zero-Knowledge (Poster Abstract)*.
- [3] Bruce Eckel. *Thinking in C++*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995. ISBN: 0-13-917709-4.
- [4] *Extending LLVM: Adding instructions, intrinsics, types, etc.* URL: <http://llvm.org/docs/ExtendingLLVM.html>.
- [5] *GEZEL Hardware/Software Codesign Environment*. URL: <http://rijndael.ece.vt.edu/gezel2/>.
- [6] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 020102988X.
- [7] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr. “Analyzing and Comparing Montgomery Multiplication Algorithms”. In: *IEEE Micro* 16.3 (1996), pp. 26–33.
- [8] Chris Lattner. “LLVM”. In: *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. Ed. by A. Brown and G. Wilson. CreativeCommons, 2011, pp. 155–171. ISBN: 9781257638017. URL: <http://www.aosabook.org/en/llvm.html>.
- [9] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. See <http://llvm.cs.uiuc.edu>. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.
- [10] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Life-long Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, 2004. URL: <http://llvm.org/pubs/2004-01-30-CGO-LLVM.html>.

- [11] *LLVM Language Reference Manual*. URL: <http://llvm.org/docs/LangRef.html>.
- [12] Sarah Meiklejohn et al. “ZKPDL: A Language-based System for Efficient Zero-Knowledge Proofs and Electronic Cash”. In: *Proceedings of the USENIX Security Symposium*. Washington, D.C., 2010.
- [13] Peter L. Montgomery. “Modular Multiplication Without Trial Division”. In: *Mathematics of Computation* 44.170 (1985), pp. 519–521. URL: <http://www.jstor.org/pss/2007970>.
- [14] T. J. Parr and R. W. Quong. “ANTLR: A predicated-LL(k) parser generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. ISSN: 1097-024X. DOI: [10.1002/spe.4380250705](https://doi.org/10.1002/spe.4380250705). URL: <http://dx.doi.org/10.1002/spe.4380250705>.
- [15] Terence J. Parr and Kathleen S Fisher. “LL(\*): The Foundation of the ANTLR Parser Generator”. In: *PLDI 2011: Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM SIGPLAN. 2011.
- [16] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. *Report on language support for Multi-Methods and Open-Methods for C++*. Tech. rep. N2216. JTC1/SC22/WG21 C++ Standards Committee, 2007. URL: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2216.pdf>.
- [17] Patrick R. Schaumont. *A Practical Introduction to Hardware/Software Code-sign*. Springer, 2010. ISBN: 978-1-4419-5999-7.
- [18] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: [10.1112/plms/s2-42.1.230](https://doi.org/10.1112/plms/s2-42.1.230). eprint: <http://plms.oxfordjournals.org/content/s2-42/1/230.full.pdf+html>. URL: <http://plms.oxfordjournals.org/content/s2-42/1/230.short>.
- [19] Frank Vahid. “The Softening of Hardware”. In: *Computer* 36 (4 2003), pp. 27–34. ISSN: 0018-9162. DOI: <http://dx.doi.org/10.1109/MC.2003.1193225>. URL: <http://dx.doi.org/10.1109/MC.2003.1193225>.