KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

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:

Prof. dr. ir. Bart Preneel
Prof. dr. ir. Ingrid Verbauwhede

Assessoren:

Prof. dr. ir. Claudia Diaz
Prof. dr. ir. Francky Catthoor

Begeleiders:

Ir. Josep Balasch
Ir. Alfredo Rial

Academiejaar 2011 – 2012

Preface

Boran Car

Contents

Abstract

Zero-Knowledge Proofs of Knowledge are protocols that allow one to prove possession or knowing of a secret without actually revealing it. Goals of current Zero-Knowledge Proofs of Knowledge compiler frameworks are to make the implementation of these protocols easy for those outside the field of cryptography while at the same time making it easy for those inside the field to prototype such protocols. We acknowledge that such frameworks have made it easy to develop the protocols but we note that they have not made it easy or general to those implementing the protocols on the end devices. Such frameworks either target C/C++ code generation or an interpreted language. They also use high-level multi-precision arithmetic libraries which are too resource demanding for small embedded devices. These limitations as well as hidden pitfalls of C/C++ have motivated us to extend one of those frameworks and build our own framework from it. This framework is targeted specifically at HW-SW co-design exploration while not losing any generality of the framework it extends. We hope that this will ease the work for those outside the field and allow for a more widespread usage of these protocols.

List of Figures

Chapter 1

Introduction

1.1 Motivation

Cryptography involves complex building blocks. Yet, it is so widespread today that the user cannot even browse the web without it. Whenever the user browses a secure web-site, a plethora of cryptographic exchanges that happen underneath is abstracted away. The protocol referred to is the Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL). All these exchanges involve even more exchanges and sub-protocols at a lower level.

TLS allows to establish a secure channel and communicate securely with another party. This is the base then for log-in procedures where the user enters his credentials and the system performs checks for each of the operations the user wants to execute. Such checks involve at their heart knowing which user is requesting. The modern society is based around anonymity, however. The users do not want to be tracked when performing everyday tasks like parking a car at a parking lot, buying a public transportation ticket or proving that they are of age. Such things need to be secure while at the same time anonymous. The traditional approach was to use paper and this is still used for voting. The advantages of computer processing and the need to be fast and profitable is in direct contrast with traditional approaches and providers are slowly introducing electronic systems. Anonymous credentials are needed to ensure both the processing efficiency for the providers and the anonymity for users.. Zero Knowledge Proofs of Knowledge allow one to prove knowledge of a secret without actually revealing it. As such, they are a basic building block for such anonymous credential proving and the field is seeing more and more advancements. New crypto protocols are being proposed such as e-voting, e-petitions, e-cash, group signatures and others.

The notion of the user need not be specifically linked to a human user. Direct Anonymous Attestation (DAA) specifically targets Trusted Platform Computing (TPM) devices that can be found in, but not limited to general purpose computers. The user can also be a smart sensor network or a smart accessory (both of which are common today). A recent advancement is the introduction of self-driving cars. Such embedded devices were a niche when SSL was first designed (not to mention

DES and RSA). Today these devices are mainstream (smart cards, tablets, smartphones...), taking more and more of the field from general purpose computers. The advantages are mostly small-size and low-power but other trade-offs can be made when designing such small devices. These trade-offs give rise to a plethora of unique system and/or processor architectures. It is up to the designer of such embedded devices to implement the industry standard protocols to allow intercommunication with other existing devices. Most programmers are not cryptographers and are prone to many errors if implementing cryptography protocols in a general purpose programming language. Even cryptographers themselves are prone to errors which is easily observable by the amount of iterations of OpenSSL[1] and GnuTLS[2] that are released per year. The protocols that currently dominate the web (protocols up to TLS 1.2) were specified a long time ago, yet the errors introduced by implementing them in a general purpose language could still not be avoided. Worse for that matter is that the need for secure embedded devices rises well beyond the available cryptographers. Automated tools are thus needed for implementing cryptography protocols.

When it comes to embedded devices, the barrier between the hardware and the software is getting thinner and is no longer uncommon for implementations crossing from one side to the other even at later stages of the design work-flow. This has been dubbed "The Softening of Hardware" [23]. This allows for more granular trade-offs and can lead to more efficient designs. The automated framework should also allow this exploration.

Cryptographers themselves could also benefit from such automated tools. Current general purpose languages and tools for the cryptography field involve a lot of common, shared code that has to be rewritten over and over. A domain specific language that cryptographers can easily understand and read fluently without ambiguities can only lead to better tested and proven protocols. The ideal split of this word would be for cryptographers to test, specify and write protocols in such a language and the users to use that language for their applications. Lastly, to quote: "Computers are incredibly fast, accurate, and stupid. Human beings are incredibly slow, inaccurate, and brilliant. Together they are powerful beyond imagination."[3]

1.2 Related Work

The need for automated tools for Zero Knowledge Proofs of Knowledge has led to the creation of the CACE[4] Project Framework, the ZKPDL Library and IBM's Idemix[5] project. These tools were built for general purpose computers. They use higher level tools (such as GMP[6], a multi-precision library) and are not very well suited for small, embedded devices. While these tools could in theory be ported to embedded

[1] http://www.openssl.org
[2] http://www.gnu.org/software/gnutls/
[3] Unknown author, falsely attributed to Albert Einstein
[4] http://www.cace-project.eu
[5] http://www.zurich.ibm.com/idemix/
[6] http://gmplib.org/

devices, this is usually not practical as the generality sacrifices some efficiency. Even in the best case, these tools do not allow exploring the hardware-software (HW-SW) codesign.

1.3 Original Contribution

A custom framework is needed to address the specific needs of HW-SW codesign. The approach taken is to extend the CACE Project Zero Knowledge Compiler as designing a complete framework from scratch is deemed impractical. The CACE Project Zero Knowledge Compiler already provides a sufficiently good Zero Knowledge Proof of Knowledge compiler for Σ protocols (a zero knowledge proof of knowledge protocol involving 3 rounds: commitment, challenge and response). This compiler compiles a higher level language (PSL) into a lower level language (PIL). This lower level language was extended so that another work-flow can be taken for non-supported features in the CACE Project Zero Knowledge compiler. Such features are multi-party protocols and constant expressions. These both allow for specifying other protocols that are not Σ transformable. Such an example is the Direct Anonymous Attestation Join protocol.

The LLVM (a compiler infrastructure framework targeting tailored for aggressive optimizations) is used to transform this low level language into a even more lower level representation (LLVM IR) that can be compiled on most of the general purpose processors LLVM supports (x86, ARM, PowerPC, MIPS). The LLVM IR does not allow for modular arithmetic so the framework is extended with type tracking and inference. The tracked types have a defined mapping to the LLVM IR types that back them in the lower level. This choice presented itself easier than extending the LLVM IR, which is a very cumbersome process. A GEZEL (a cycle-accurate co-simulation environment) back-end was written for LLVM such that it is possible to synthesize the protocol directly onto hardware. This also allows to explore HW-SW co-design by means of Data Flow Graph - Control Flow Graph (DFG-CFG) balancing. GEZEL and CACE were also extended to allow terminal communication with the "outside" world (outside referring here to the other side of the communication terminal).

1.4 Thesis Structure

Chapter 2 covers the needed preliminaries for understanding the background behind this thesis. Formal languages, grammars and parsers are covered as the custom framework will include a textual front-end. Next we explain the mathematical background in group theory and modular arithmetic, followed by a definition for Zero Knowledge Proofs of Knowledge. The chapter ends with discussing Data Flow Graphs and Control Flow Graphs, basic tool in HW-SW codesign.

Chapter 3 gives a basic overview of the CACE Project Zero Knowledge Compiler. The compiler defines a higher-level and a lower level language and both will be covered in some detail. Next are the tools which were to create the custom frame-

work. ANTLR[7] is a parser generator tool that was used to create the parser. The code generated from the parsed input is fed to LLVM[8], a compiler infrastructure framework. GEZEL, a co-simulation environment, is the end-target in which the designs can be verified and validated.

Chapter 4 gives details of the custom framework. It shows how the custom framework was extended from the CACE Project Zero Knowledge Compiler. Next it details the needed extensions to allow more protocols to be implemented.

Chapter 5 gives a use case of the custom framework. It shows how to specify and compile a protocol (the DAA protocol in this case).

Chapter 6 concludes and gives future work ideas that were not done during the course of this thesis, but the custom framework nevertheless allows for such implementations.

[7] http://www.antlr.org
[8] http://llvm.org

Chapter 2

Technical Preliminaries

This chapter has the goal of introducing the math and theory behind formal languages, parsers, algebra, (zero knowledge) proofs of knowledge, control and data flow in a program. Knowledge behind formal languages is needed to design a parser. A parser will later be used to create a custom compiler. Since the thesis deals with cryptography applications, knowledge of group theory and modular arithmetic is needed as well. The definition of (zero knowledge) proofs of knowledge follows. They are a basic building block for this thesis. Last is the definition of control and data flow in a program. This will be used to analyze and transform the protocols into hardware.

2.1 Formal Languages

The theory outlined here attempts to give a short overview of formal languages and is based on [10]. A more detailed approach can be found in [9].

Definition 1 (Alphabet)**.** An alphabet Σ is a set of symbols.

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

Definition 3 (Concatenation)**.** Let $x = a_0 a_1 \cdots a_n$ and $y = b_0 b_1 \cdots b_n$, then the string $xy = a_0 a_1 \cdots a_n b_0 b_1 \cdots b_n$ is the concatenation of the strings x and y.

Definition 4 (Sets)**.** Σ^* denotes the set of all the strings over the alphabet Σ. Likewise, Σ^+ denotes the set of all the non-empty strings over the alphabet Σ.

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

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

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

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

Definition 7 (Kleene closure)**.** Let L be a language over Σ. 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

$$L^* = L^+ \cup \{\epsilon\}$$

$$L^+ = LL^*$$

2.1.1 Regular Expression

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

1. $\emptyset$ is a regular expression and represents the empty language
2. ϵ 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. Σ is a finite non-empty set called the *terminal alphabet*. The elements of Σ are called *terminals*.
2. V is a finite non-empty set disjoint from Σ. 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

$$\alpha \to \beta$$

$$\alpha \in (\Sigma \cup V)^* V (\Sigma \cup V)^*$$

$$\beta \in (\Sigma \cup V)^*$$

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:

$$\begin{aligned} \text{statement} &\to \text{ID} := \text{expression}; \\ \text{expression} &\to \text{term} + \text{term} \\ \text{term} &\to \text{ID} * \text{ID} \\ \text{term} &\to \text{ID} \end{aligned}$$

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

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.

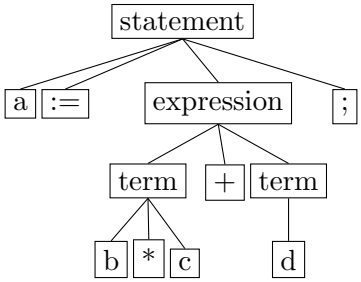

Figure 2.1: Concrete Syntax tree for a simple input and a simple grammar

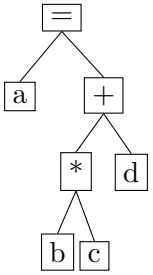

Figure 2.2: An example AST for a simple expression

2.1.5 Parser

Given a string L, satisfying grammar G, a parser tries to find the derivation of L from S. This derivation is the Concrete Syntax Tree (Parse Tree). This derivation may further be abstracted into an Abstract Syntax Tree that is easier for manipulation later. Figure 2.3 shows the typical parser flow.

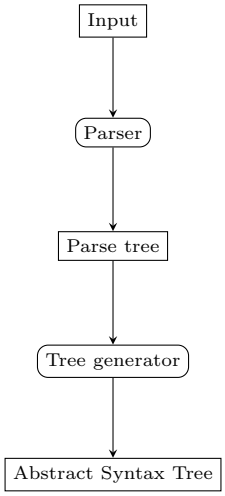

Figure 2.3: Parser flow

There are two approaches to parsing:

Top-down parsing the derivation starts from the top (the root) of the parse tree and proceeds downwards.

Bottom-up parsing the derivation starts from the bottom (the leaves) of the parse tree and proceeds upwards.

The most popular top-down parser is an LL (left to right left-most derivation) parser. The parser reads the input left to right and at each step produces the left-most derivation of the input. Sub-types of this parser include the symbol look-ahead which is the amount of next, unseen symbols the parser can "look at". For example, an LL(1) parser can only look 1 symbol ahead while an LL(*) parser is a parser with arbitrary look-ahead.

2.2 Group Theory and Modular Arithmetic

The theory here is based on [16]. The approach is to start with the general and build the specific from it.

Definition 11 (Group)**.** A group $(G, \circ)$ is a set G with a binary operation $\circ$ defined on it that satisfies the following properties:

1. (Closure) $a, b \in G \implies a \circ b \in G$
2. (Associativity) $(a \circ b) \circ c = a \circ (b \circ c)$ $\forall a, b, c \in G$
3. (Identity element) $\exists I \in G$ such that $a \circ I = I \circ a = a$
4. (Inverse element) $(\forall a \in G)(\exists a^{-1} \in G)$ such that $a \circ a^{-1} = a^{-1} \circ a = I$

Definition 12 (Abelian Group)**.** A group G is abelian iff $a \circ b = b \circ a$ $\forall a, b \in G$

Definition 13 (Finiteness and Order)**.** A group G is finite iff $|G|$ is finite. The number of elements in G is called the *order* of the group.

Definition 14 (Subgroup)**.** A non-empty subset $H \subseteq G$ is a subgroup iff H itself is a group w.r.t. to the binary operation of G. H is a proper subgroup iff $H \neq G$.

Definition 15 (Cyclic Group)**.** A group G is cyclic iff $(\exists a \in G)(\forall b \in G)(iisInteger|b = a^i)$. Such an a is called a generator of G. The group generated by a is denoted as $\langle a \rangle$.

Definition 16 (Order)**.** Let G be a group and $a \in G$. The order of a is the least positive integer t such that $a^t = I$, provided that such an integer exists. If such an integer does not exist, the order is defined to be ∞.

Theorem 2 (Lagrange's Theorem)**.** If G is a finite group and H is a subgroup of G then $|H|$ divides $|G|$.

Corollary 1. The order of $a \in G$ divides $|G|$.

Definition 17 (Congruency)**.** An integer a is said to be congruent to integer b modulo integer n, written $a \equiv b \pmod{n}$, iff n divides $a - b$ (denoted as $n|a - b$). n is called the *modulus* of the congruency.

Definition 18 (Greatest Common Divisor)**.** Given two integers a and b, the greatest common divisor $d = \gcd(a, b)$ is the largest integer d such that $d|a$ and $d|b$.

Definition 19. Two integers, a and b, are relatively prime to each other iff $\gcd(a, b) = 1$.

Definition 20. The integers modulo n, denoted Z_n, is the set of integers $\{0, 1, 2, \ldots, n - 1\}$.

Theorem 3. Let $d = \gcd(a, n)$. Then the congruence equation $ax = b \pmod{n}$ has d solutions x iff d divides b.

Corollary 2. Let $a \in Z_n$. a has a multiplicative inverse, denoted a^{-1}, iff $\gcd(a, n) = 1$.

Definition 21. The multiplicative group of Z_n is $Z_n^* = \{a| \gcd(a, n) = 1\}$. Specifically, if n is prime, $Z_n^* = \{a|1 \leq a \leq n-1\}$. The identity element of the multiplicative group is 1.

Definition 22. The additive group of Z_n is $Z_n^+ = \{a|0 \leq a \leq n - 1\}$.

Definition 23 (Euler Phi Function)**.** The Euler phi function, $\phi(n)$, also called Euler totient function, gives the number of integers from the interval $[1, n]$ which are relatively prime to n.

Corollary 3. The order of the group Z_n^* is $\phi(n)$.

Corollary 4. For a prime p, $\phi(p) = p - 1$.

Corollary 5. If $\gcd(m, n) = 1$, $\phi(mn) = \phi(m)\phi(n)$.

Theorem 4 (Euler's Theorem)**.** If $a \in Z_n^*$, then $a^{\phi(n)} = 1 \pmod{n}$.

Corollary 6 (Fermat's Theorem)**.** If $a \in Z_p^*$, where p is prime, then $a^{p-1} = 1 \pmod{p}$.

Corollary 7. $a \in Z_n^*$ is a generator of the group Z_n^* iff the order of a is $\phi(n)$. a is also called a primitive element of Z_n^* then.

2.3 Zero Knowledge Proofs of Knowledge

The definitions are taken from [3] and presented as a brief overview. Some minor additions were made for clarity.

Definition 24 (Interactive Proof of Knowledge)**.** Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a polinomially bounded binary relation and let L_R be the language defined by R. An interactive proof of knowledge is a protocol (P, V) that has the following properties:

1. (Completeness) $(x, w) \in R \implies [V, P(w)](x) = T$
2. (Validity) There exists a probabilistic expected polynomial-time machine K (Knowledge extractor) such that for every $\tilde{P}$, for all polynomials $f(\cdot)$ and all sufficiently large $x \in L_R$:

$$p((x, K^{\tilde{P}(x)}) \in R) \geq p([V, \tilde{P}](x) = T) - \frac{1}{f(|x|)}$$

The probabilities are taken over all random choices of V, P, $\tilde{P}$, K. The notation $[V, P(w)](x)$ denotes the protocol execution with the secret x and the prover output w. $\tilde{P}$ includes malicious provers. $K^{\tilde{P}(x)}$ denotes oracle access to $\tilde{P}(x)$. $f(\cdot)$ denotes the knowledge error.

Definition 25 (Soundness)**.** $(\forall \tilde{P})(\forall x \notin L_R)(p([V, \tilde{P}](x) = T) < \frac{1}{2})$

The additional soundness property is associated with $x \notin L_R$. By repeating the protocol many times, it can be made arbitrarily small [3].

Definition 26 (Indistinguishability)**.** Let $L \in \{0,1\}^*$ be a language and let $A = \{A(x)\}_{x \in L}$ and $B = \{B(x)\}_{x \in L}$ be two ensembles of random variables indexed by L. Ensembles A and B are:

- perfectly indistinguishable if for all $x \in L$ the variables $A(x)$ and $B(x)$ are identically distributed
- statistically indistinguishable if for every polynomial $f()$ and for all sufficiently long $x \in L$:

$$\sum_{\alpha \in \{0,1\}^*} |p(A(x) = \alpha) - p(B(x) = \alpha)| < \frac{1}{f(|x|)}$$

- computationally indistinguishable if for every probabilistic polynomial-time algorithm D, for every polynomial $f()$ and for all sufficiently long $x \in L$:

$$|p(D(x, A(x) = 1) - p(D(x, B(x) = 1)| < \frac{1}{f(|x|)}$$

Definition 27 (Zero-Knowledge)**.** An interactive protocol (P, V) is said to be perfectly/statistically/computationally zero-knowledge if for every probabilistic polynomial-time verifier $\tilde{V}$ there exists a probabilistic expected polynomial time simulator $S_{\tilde{V}}$ so that the two ensembles

$$\{[\tilde{V}, P(w)](x)\}_{x \in L} \text{ and } \{S_{\tilde{V}}(x)\}$$

are perfectly/statistically/computationally indistinguishable.

Definition 28 (Honest Verifier Zero-Knowledge)**.** An interactive protocol (P, V) is said to be perfectly/statistically/computationally zero-knowledge if there exists a probabilistic expected polynomial time simulator $S_{\tilde{V}}$ so that the two ensembles

$$\{[V, P(w)](x)\}_{x \in L} \text{ and } \{S_V(x)\}$$

are perfectly/statistically/computationally indistinguishable.

2.3.1 Σ-protocols

A Σ-protocol is a three round honest verifier zero-knowledge proof of knowledge [22]. The name comes from the protocol's flow resemblance to the Greek letter Σ as shown in Figure 2.4. The rounds of the protocol are:

1. Commitment (t) - the prover commits to a value and sends that value to the verifier

2. Challenge (c) - the verifier computes a random challenge and asks the prover to output a value for that challenge

3. Response (s) - the prover responds with a new computed value

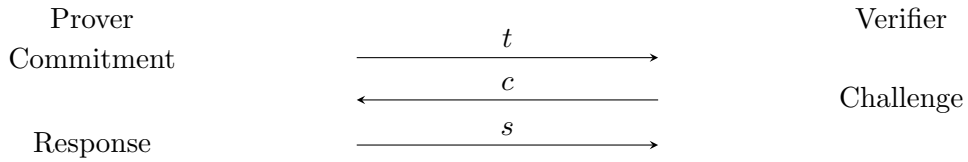

Figure 2.4: Sigma protocol flow

Schnorr's Identification Protocol

A simple example of a Σ protocol is Schnorr's Identification Protocol [21, 22]. The secret is a discrete logarithm x of y with respect to g in some finite group $G = \langle g \rangle$ with prime order q, subgroup of Z_p^*. The prover must prove knowledge of this secret to a verifier under the homomorphism $f(a) : Z_q^+ \to G \subset Z_p^* := g^a$.

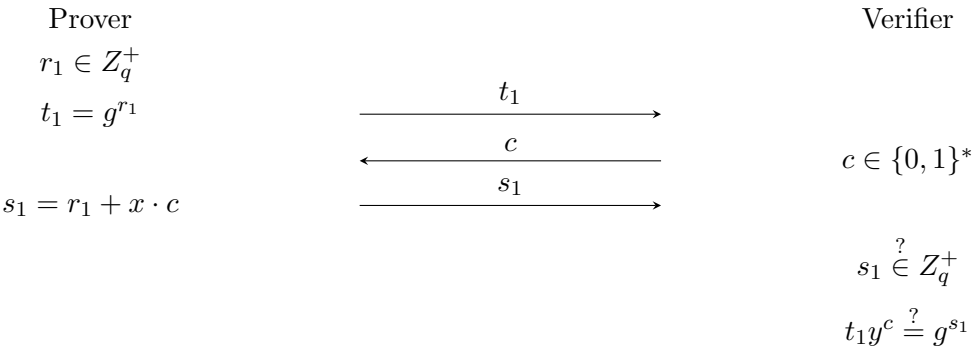

Figure 2.5: Schnorr's Identification Protocol

Camenisch-Stadler Notation

A simple and popular notation for specifying Sigma protocols is the Camenisch-Stadler notation [1, 4]. The Schnorr Identification Protocol can be represented by the following:

$$\text{ZPK}\left[(x) : y = g^x\right]$$

meaning prove knowledge of x, using a zero knowledge proof of knowledge, such that $y = g^x$.

2.4 Data Flow Graph and Control Flow Graph

Definition 29 (Data Edge)**.** A data edge is an ordered relation between two operations such that the output/result of one operation is the input to the other operation.

Definition 30 (Control Edge)**.** A control edge is an ordered relation between two operations such that the second operation has to be executed after the first finishes executing.

Definition 31 (Data Flow Graph)**.** A data flow graph is a graph of the nodes representing the operations connected by data edges.

Definition 32. A control flow graph is a graph of the nodes representing the operations connected by control edges.

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 [20].

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 will be covered in more detail while ZKPDL will be briefly mentioned. 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[1] 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 user's 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

[1] http://www.cace-project.eu

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) - the user specifies the protocol using Camenisch-Stadler notation (see Sub-subsection 2.3.1)
2. Generate PIL (Protocol Interface Language) from PSL - the framework transform it into a lower-level language for specifying operations
3. Generate C or Java code from PIL - the framework generates a code that is possible to compile and execute on a target architecture
4. (Optional) Verify PIL code using PVT - the framework applies the formal verification to the PIL code
5. (Optional) Generate LaTeX from PIL - the framework generates LaTeXcode with the protocol flow

3.1.2 PSL

The Protocol Specification Language (PSL) is a high level language of CACE Project WP3 for specifying Proofs of Knowledge based on Camenisch-Stadler notation. It allows specification of complex Σ protocols [1, 2]. The explanation is best given following an example of a simple protocol, Schnorr's Identification Protocol (see Sub-subsection 2.3.1). The Camenisch-Stadler notation is:

$$\text{ZPK}\left[(x) : y = g^x\right]$$

The PSL language is structured into blocks, and for the Schnorr example the blocks are as follows:

- Declarations - specifying all the variables used within the protocol

```
Declarations {
  Prime(1024) p;
  Prime(160) q;
  Zmod+(q) x;
  Zmod*(p) g, y;
}
```

This example shows how to declare prime numbers as well as elements of a residue group. Here, p is declared as a prime of 1024 bits and q is declared as a prime of 160 bits, $x \in Z_q^+$ and $g, y \in Z_p^*$.

- 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.

- Properties - specifying the properties of the protocol

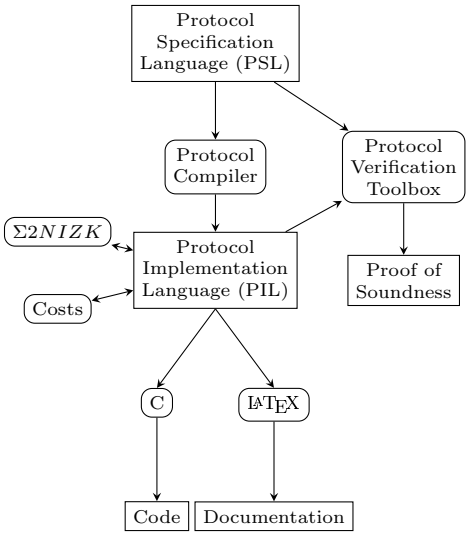

FIGURE 3.1: CACE Project Zero Knowledge Compiler typical workflow [1]

```
Properties {
  KnowledgeError := 80;
  SZKParameter := 80;
  ProtocolComposition := P_1;
}
```

This example shows how to specify the knowledge error, which is 2^{-80} in this case. The tightness is specified at 2^{-80} as well. The protocol composition allows to specify multiple protocols via AND, OR, XOR. Since this is a simple case, only one protocol is used.

- Specifying the protocols itself (the homomorphism to use and the relation to be proven)

```
SigmaPhi P_1 {
  Homomorphism (phi : Zmod+(q) -> Zmod*(p) : (a) |-> (g^a));
  ChallengeLength := 80;
  Relation ((y) = phi(x));
}
```

The relation to be proven is $y = g^x$ which is specified as the homomorphism $\phi(a) : Z_q^+ \rightarrow Z_p^* := g^a$. The ChallengeLength allows to customize the protocol for certain devices. The generated protocol will be repeated until the required knowledge error is met. For example, a challenge length of 1 will repeat the protocol 80 times to satisfy the knowledge error of 2^{-80}.

The previous blocks combined give a complete PSL specification of Schnorr's Identification 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 learn. This can aid in verifying the correctness from a user's point of view. The constructs of the language are completely specified and allow an automated verification.

As a language on its own, PIL 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;
...
}
```

- conditionals

```
IfKnown(...) {

} Else {

}
```

- loops

```
For i In [1,2] {
   ...
}
```

- functions

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

- predicates

```
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 Sub-section 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 Related Frameworks

3.2.1 ZKPDL/Cashlib

ZKPDL is a description language for describing Zero Knowledge Proofs of Knowledge. It is used by the framework Cashlib to implement e-cash. The framework uses an interpreter based approach and applies result caching to speed up computations. Unlike PIL, ZKDPL is not Turing complete as it does not allow branching or conditionals. Also, ZKPDL only supports non-interactive proofs of knowledge [15]. ZKPDL does allow generation of parameters which PIL lacks [2] but this can be mitigated by defining and using constant expressions.

Because of its generality, a standard notation of specifying proofs of knowledge as well as the ability to formally verify code and generate documentation, the CACE Project Zero Knowledge Compiler will be the framework of choice for extension in this thesis.

3.3 Other Tools

3.3.1 ANTLR

ANTLR[2] (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 chosen among 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 [18, 17].

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 3.2).

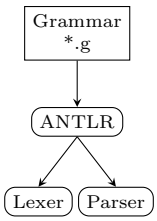

Figure 3.2: ANTLR Parser/Lexer generation

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 3.3.

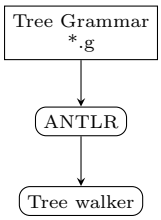

Figure 3.3: ANTLR Tree walker generation

[2] http://www.antlr.org

3.3.2 LLVM

LLVM[3] 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 [13].

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 3.4.

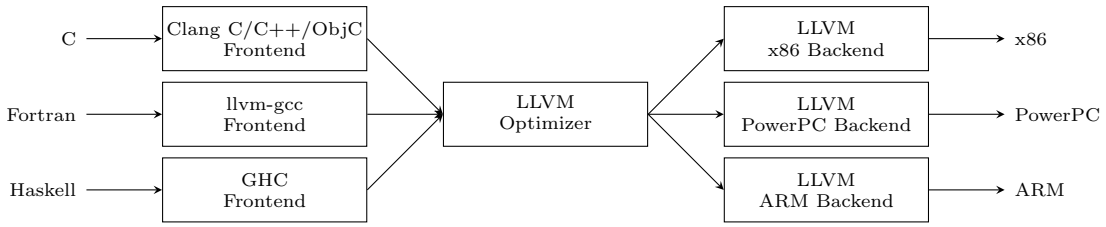

FIGURE 3.4: LLVM typical workflow [11]

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)
- 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 [11, 12]. 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

[3] http://llvm.org

to be saved. Instructions operate solely upon variables. In this respect, the LLVM IR resembles the assembly language of an 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 [14].

3.3.3 GEZEL

GEZEL is a cycle-accurate hardware description language (HDL) using the Finite-State-Machine + Datapath (FSMD) model [8].

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

[8]. 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 3.5 shows how the GEZEL language can be used as an input to:

- fdlvhd - a generator that can generate synthesizeable 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

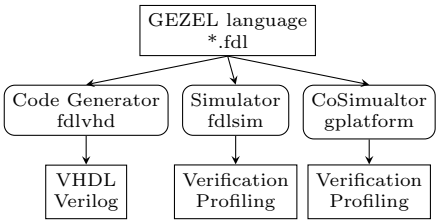

FIGURE 3.5: GEZEL workflow [8]

The co-simulation tool allows to cosimulate GEZEL designs with instruction-set simulations [8]. 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

This chapter aims to present the custom framework that was developed within the course of this thesis. It starts with the motivation of domain specific languages then gives the reasoning to extend the CACE Project Zero Knowledge Compiler. Figure 4.1 gives an overview of the custom extensions. They are presented in detail in Section 4.3. All of targets except for GEZEL come from LLVM so the GEZEL target will also be covered.

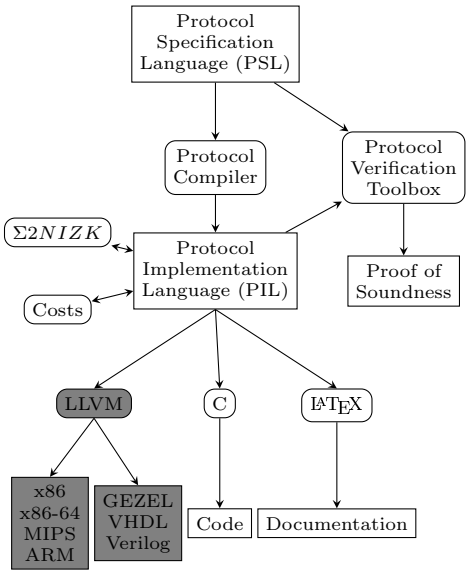

FIGURE 4.1: Custom framework (extensions to CACE Zero Knowledge Compiler highlighted)

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 choose. 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.

A domain specific languages hides all the operations that are deemed not important and allows an abstract overview of the operations. This

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. Separation of the supporting library allows plugging in of a custom multi-precision library but this is impractical. There are various devices available and writing a multi-precision library for each of them is error prone and inefficient.

The fact that the CACE Project Zero Knowledge 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. However, the approach of 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. LLVM already allows for

- Interpreted code - by using its interpreter
- JIT (Just In Time) compiled code - by compiling the code at runtime

- Compiled code - by allowing to code to be compiled later on

Also, since the intermediate form of LLVM 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.

4.2 Extensions to GEZEL

4.2.1 Terminal communication ipblocks

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.

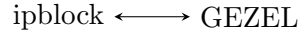

FIGURE 4.2: GEZEL and ipblock

The ipblock terminal is based on POSIX termios functionality so this limits the simulator availability to POSIX systems. This ipblock was originally aimed to be connected to gplatform's 8051 simulator but the simulator core itself does not allow easy connections of an external serial port. The decision was made to make a transceiver ipblock that receives and transmits arbitrary numbers. Such a transceiver allows easier designs inside GEZEL as there is no need to parse the stream input.

```
ipblock my_term(in tx     : ns(1024);
                out rx    : ns(1024);
                in sr     : ns(2);
                out done : ns(1)) {
  iptype "transceiver";
}
```

Output to the "outside" world is given to the *tx* pin (marked here as in because the directions are relative to the ipblock). Input from the "outside" world is taken from the *rx* pin. The "outside" world denotes the process running outside of the simulation that is connected via a terminal. The *sr* specifies the operation: 0 for no operation, 1 for sending, 2 for receiving. After the operation is done, the ipblock sets the *done* pin to 1.

4.2.2 Modular exponentiation

GEZEL supports modular operations (addition, subtraction and multiplication) via two binary operations, first the normal operation and then taking the modulus. GEZEZ, however, did not support modular exponentiation. It was either necessary to implement it in GEZEL code or to extend GEZEL with the exponentiation operation. As the exponentiation is much easier to realize in C++ code the decision

was made to extend GEZEL. This required rewriting the grammar, adding a new operation and adding run-time emitting of the code for the exponentiation.

The syntax of the modular exponentiation is:

```
y = g ** x % p;
```

GEZEL supports unary, binary and ternary operations. Since GEZEL uses the GMP library, the modular exponentiation needed to be a ternary operation. The GMP library supports only modular exponentiation when using arbitrary precision integers. This required a significant change in the core but the advantages of a having a well tested and working implementation of modular exponentiation outweighed the disadvantages.

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 either be fed to existing optimizers or a custom one can be written. The resulting IR can then either 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).

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 multiparty protocol[1]. As this is simply a relaxation of the rules, the change to the grammar and the code generator 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 protocols 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;
```

[1] In the cryptographic world, this means three or more parties

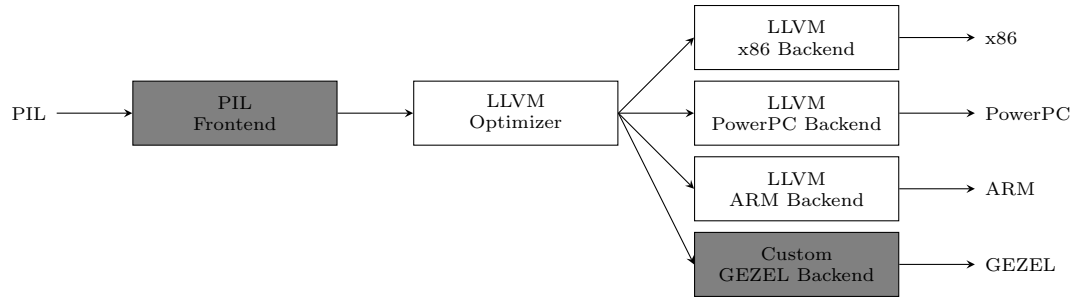

FIGURE 4.3: LLVM custom workflow (changes highlighted)

```
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, its 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.

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.

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 [7].

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

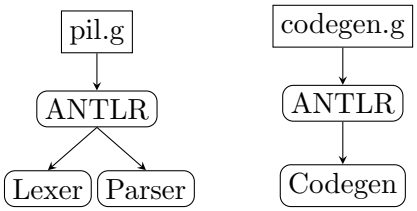

FIGURE 4.4: Lexer, parser and tree walker generation

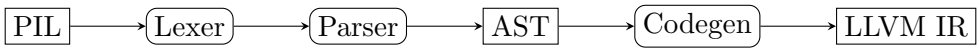

FIGURE 4.5: PIL frontend flow

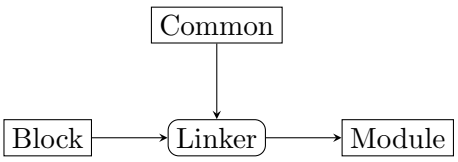

FIGURE 4.6: Linker

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. The types are backed by an IntegerType of the appropriate length in the LLVM IR. 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 third 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. The rule includes a syntactic predicate to test if it is a declaration of an alias or an alias is referenced.

```
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] }
      ;
```

If the syntactic predicate matches, the alias is stored in a hashmap, otherwise the hashmap is searched for the alias. This syntactic predicate effectively implements a 1 symbol look-ahead.

4.4.2 Type system

The compiler has to keep track of the types. A simple type system was designed that allows both numbers and modular residues. The UML of this type system is given in Figure 4.7.

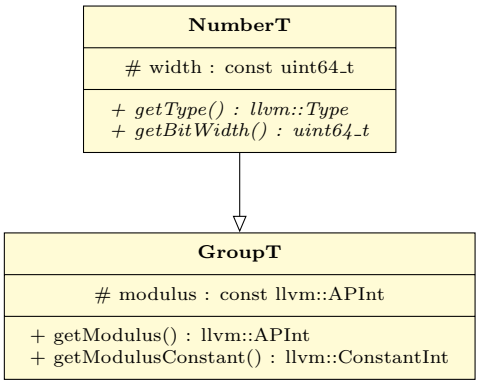

FIGURE 4.7: Type system UML

NumberT represents any general number type that can occur (types Int and Z), while GroupT represents group types (types Zmod+ and Zmod*). The getType() method returns the backing type in the LLVM IR (IntegerType of the appropriate length). The modulus of the GroupT is stored as an LLVM arbitrary precision integer (llvm::APInt) and it can be fetched by using the getter getModulus(). getModulusConstant() is a helper method that wraps it in a ConstantInt which LLVM can then use for constant folding.

4.4.3 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 [19].

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 is done with GroupT and NumberT:

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

class GroupT : public NumberT {
  ...
```

```
};
```

Here the addWithSubFrom method is the accept method. The same was done for each of the operators. When the compiler encouters 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 withing the addWithSubFrom **this** points to the accurate type for the second operand [6]. 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

The GEZEL Back-end extracts the Data Flow Graph (DFG) and implements one GEZEL datapath per round (Figure 4.8). The Control Flow Graph (CFG) is maximally unrolled such that the DFG within a round is purely combinatorial. Each invocation of addition, multiplication, exponentiation requires a new instantiation of an adder, multiplier, exponentiator, respectively.

The choice of maximally unrolling the CFG was made as it already allows architecture exploration at one end of the spectrum while the other is provided by the software implementation. The implementations in the middle of the spectrum should be derived from the closer side by rolling the CFG or adding more to the software. Such techniques require more complex signaling and scheduling and, as time was limited, were not taken.

The memory accesses are converted to register accesses and each datapath gets access to each of the registers. To prevent name collisions register inputs and outputs are decorated as shown below:

```
dp round1(in  r_r_1_i : ns(160); out r_r_1_o : ns(160);
          in  r_s_1_i : ns(160); out r_s_1_o : ns(160);
          out _t_1 : ns(1024)) {

  sig s_r_1 : ns(1024);
  sig s_t_1 : ns(1024);

  use random1(11, s_r_1);
  use modexp1(3, s_r_1, 23, s_t_1);

  always {
    r_r_1_o = s_r_1;
    r_s_1_o = r_s_1_i;

    _t_1 = s_t_1;
  }
}
```

The example is generated from the Schnorr's Identification Protocol PIL (see example of 3.1.3). Each register is prefixed with r and suffixed with i if referring to the input from the register or o if referring to the output from the register. The internal signals within a datapath are prefixed with s.

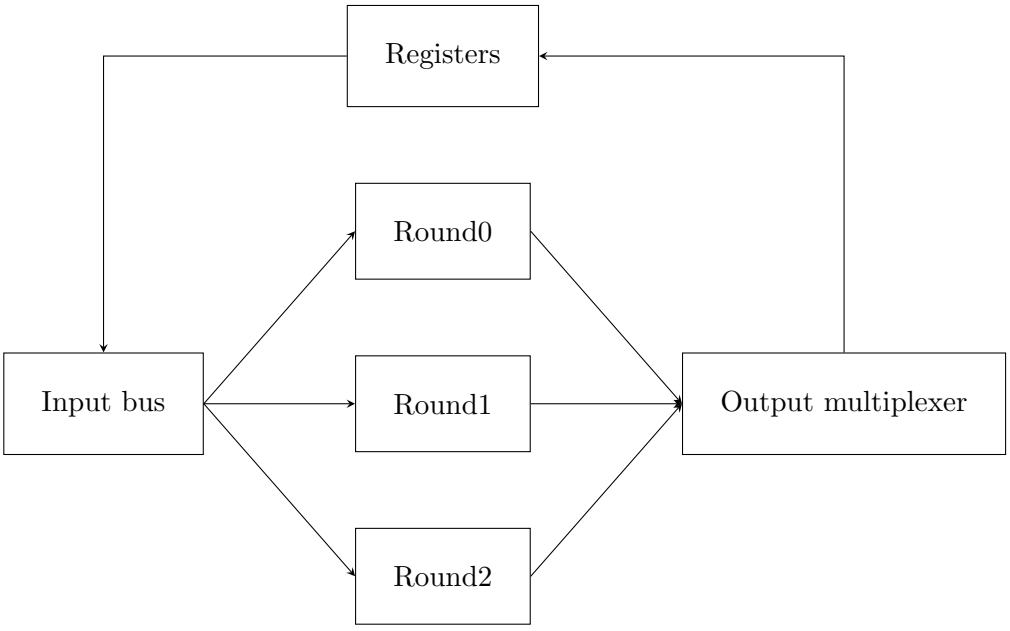

FIGURE 4.8: Generated GEZEL model of Schnorr's protocol

Chapter 5

Use Case: Direct Anonymous Attestation

This chapter will attempt to show the advantages of the custom framework by implementing a simplified Direct Anonymous Attestation (DAA) Join protocol taken from [5]. The reason Join was chosen is that it is not a Σ protocol and cannot be implemented using the CACE Project Zero Knowledge Compiler. Join protocol also serves as an example of a more complex and specific protocol as it is a multiparty protocol. Again, the benefit of the custom framework is that it allows the input of basic PIL as well. The Sign protocol is not of interest here.

5.1 Join protocol

The Join protocol is the first phase in the DAA protocol. Its goal is to establish the certificate. The protocol is shown in figure 5.1. The protocol is not a Sigma protocol and cannot be specified using the PSL language. Also, the protocol uses 3 parties which the CACE Project Zero Compiler does not support. This, and similar multiparty protocols were the reason for extensions to the PIL language (see Subsection 4.3.3) in the first place. It is observable how the specification in the PIL language closely follows the protocol flow diagram.

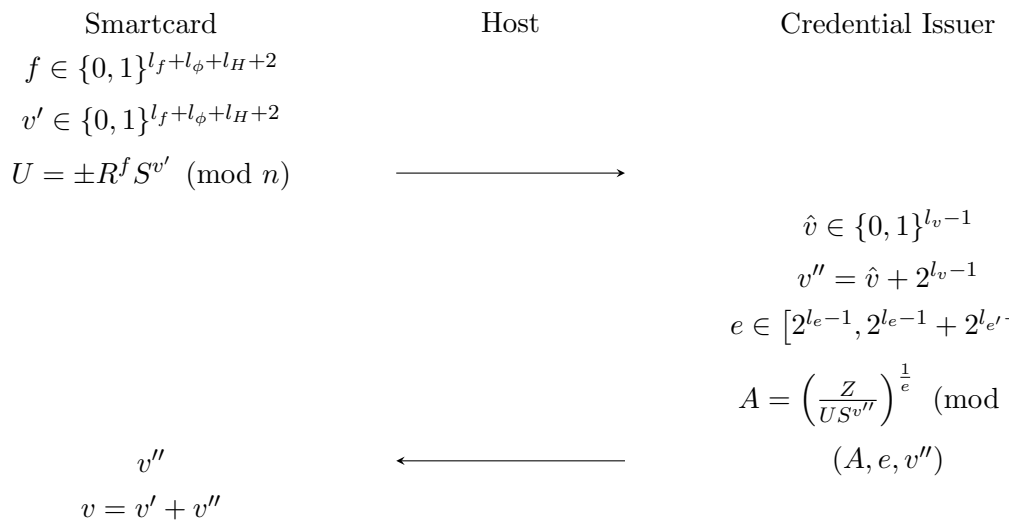

Figure 5.1: DAA Join protocol

It is also visible how the usage of constant expressions aids the writer in specifying the protocol. It suffices to only change the parameter and the constant expressions will recompute the sizes according to the expression.

```
ExecutionOrder := (Smartcard.Round0, Host.Round0, Credential.Round0,
    Host.Round1, Smartcard.Round1);

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;

  Zmod*(n) S, _Z, R
) {

}

Smartcard (

) {
  Int(l_f + l_phi + l_H+2) f;
  Int(l_n + l_phi) v_1;
  Int(l_v) v;

  Def (Zmod*(n) U): Round0(Void) {
    f := Random(Int(l_f + l_phi + l_H + 2));
```

```
    v_1 := Random(Int(l_f + l_phi + l_H + 2));

    U := R^f * S^v_1;
  }

  Def (Void): Round1(Int(l_v-1) v_2) {
    v := v_1 + v_2;
  }
}

Host (

) {
  Zmod*(n) A;
  Int(l_e) e;

  Def (Zmod*(n) U_o): Round0(Zmod*(n) U_i) {
    U_o := U_i;
  }

  Def (Int(l_v-1) v_2_o): Round1(A; e; Int(l_v-1) v_2_i) {
    v_2_o := v_2_i;
  }
}

Credential (

) {
  Def (Zmod*(n) A; Int(l_e) e; Int(l_v-1) v_2): Round0(Zmod*(n) U) {
    v := Random(Int(l_v-1));
    v_2 := v + 2^(l_v-1);
    e := Random([2^(l_e-1), 2^(l_e-1) + 2^(l_e_1-1)]);
    A := (_Z * (U * S^v_2)^(-1))^(e^(-1));
  }
}
```

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Previous compiler frameworks for Zero Knowledge Proofs of Knowledge targeted general purpose processors. It has been shown that the PIL language is expressive, yet abstract enough for Data Flow Graph (DFG) extraction. A direct hardware realization has thus been made possible while also allowing the implementation to target general purpose processors. This opens a path for Control Flow Graph (CFG) generation such that hardware software co-design can be done in an automated way with a high-level control from the user. The changes introduced do not interfere with the standard CACE work-flow as it is still possible to define a PSL and generate a PIL from it. The advantage is that multiple architecture are now supported.

The extensions to the CACE Project Zero Knowledge Compiler support library have allowed communication over serial ports and have thus allowed multiple combinations of general purpose computer and embedded device interconnectivity. Serial port communication is easier to implement and this can also serve for cross-checking and cross-validating of custom implementations.

The extensions to the PIL Language have allowed a complex protocols such as the Direct Anonymous Attestation (DAA) Join protocol to be implemented easily by just translating the protocol flow specification to PIL. The transformation is also reversible thanks to the CACE Project Zero Knowledge Compiler LaTeX output. The advantages of a domain specific language have clearly been shown.

6.2 Future Work

LLVM should be extended with the modular residue group types and this work should be submitted upstream to the development tree of LLVM. 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 from the CACE Project 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 CACE Project Zero Knowledge Compiler does not support automatic generation of parameters. This is something an extension of this framework should also allow. Leaving these choices to the end user might be problematic both from a usage perspective as well as security perspective. The user simply cannot and should not know all the needed constraints. The only constraints the user should set are the bit sizes. The framework developed within the course of this thesis already allows for compile time/constant expression. This expressions can be used as a starting point for defining a generator macro/function that should automatically generate the required parameters.

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. This would require writing an 8051 back-end.

There is no reason not to go beyond writing LLVM back-ends. A pseudo Virtual Machine can be realized by function calls in C. Such a Virtual Machine is then easily deployable to even more architectures as C is an industry standard compiler.

The generated hardware uses a primitive CFG and implements the entire DFG. A DFG uniquely defines an algorithm, while a CFG only defines the execution. Automated CFG-DFG balancing (controlled via a parameter) could be made that allows trading speed for smaller device size and less power consumption.

Bibliography

[1] José Bacelar Almeida et al. "A certifying compiler for zero-knowledge proofs of knowledge based on Σ-protocols". In: *Proceedings of the 15th European conference on Research in computer security*. ESORICS'10. Athens, Greece: Springer-Verlag, 2010, pp. 151–167. ISBN: 3-642-15496-4, 978-3-642-15496-6. URL: http://dl.acm.org/citation.cfm?id=1888881.1888894.

[2] Endre Bangerter et al. *YACZK: Yet Another Compiler for Zero-Knowledge (Poster Abstract)*.

[3] Jan Camenisch. "Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem". Reprint as vol. 2 of *ETH Series in Information Security and Cryptography*, ISBN 3-89649-286-1, Hartung-Gorre Verlag, Konstanz, 1998. PhD thesis. ETH Zurich, 1998.

[4] Jan Camenisch and Markus Stadler. "Efficient Group Signature Schemes for Large Groups (Extended Abstract)". In: *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '97. London, UK, UK: Springer-Verlag, 1997, pp. 410–424. ISBN: 3-540-63384-7. URL: http://dl.acm.org/citation.cfm?id=646762.706305.

[5] Antonio Diaz Castano. "Anonymous ePetitions with secure hardware for smart phones". MA thesis. Universida de Vigo, 2011.

[6] Bruce Eckel. *Thinking in C++*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995. ISBN: 0-13-917709-4.

[7] *Extending LLVM: Adding instructions, intrinsics, types, etc.* URL: http://llvm.org/docs/ExtendingLLVM.html.

[8] *GEZEL Hardware/Software Codesign Environment*. URL: http://rijndael.ece.vt.edu/gezel2/.

[9] 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.

[10] Tao Jiang et al. *Formal Grammars and Languages*. URL: http://www.cs.ucr.edu/~jiang/cs215/tao-new.pdf.

[11] 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.

[12] 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.

[13] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong 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.

[14] *LLVM Language Reference Manual*. URL: http://llvm.org/docs/LangRef.html.

[15] 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.

[16] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237.

[17] 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. URL: http://dx.doi.org/10.1002/spe.4380250705.

[18] 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.

[19] 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.

[20] Patrick R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010. ISBN: 978-1-4419-5999-7.

[21] C. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology CRYPTO 89 Proceedings*. Ed. by Gilles Brassard. Vol. 435. Lecture Notes in Computer Science. 10.1007/0-387-34805-0_22. Springer Berlin / Heidelberg, 1990, pp. 239–252. ISBN: 978-0-387-97317-3. URL: http://dx.doi.org/10.1007/0-387-34805-0_22.

[22] Nigel Smart. *Cryptography: An Introduction*. Mcgraw-Hill College, Dec. 2004. ISBN: 0077099877. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0077099877.

[23] 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.