KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

Implementation and Evaluation of Zero-Knowledge Proofs of Knowledge

Boran Car

Thesis submitted for the degree of
Master of Science in
Electrical Engineering

Thesis supervisors:
Prof. dr. ir. Bart Preneel
Prof. dr. ir. Ingrid Verbauwhede

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

Mentors:
Ir. Josep Balasch
Ir. Alfredo Rial

Academic year 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 and Tables

List of Figures

List of Tables

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 is abstracted away. All these exchanges involve even more exchanges and sub-protocols at a lower level. The protocol referred to is the Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL).

TLS allows establishing a secure channel and communicating 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, revealing the user's identity while the modern society is based around anonymity. 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. Traditional approaches solved this using paper but advantages of computer processing and the need to be fast and profitable are in direct contrast with this 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 are the basic building block for such anonymous credentials as they allow one to prove knowledge of a secret without actually revealing it. This field is gaining momentum as of recent and new 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. For example, 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). Another example is the recent advancement of introducing self-driving cars. Such embedded devices were a niche when SSL was first designed (not to mention DES and RSA) while today these devices are mainstream (smart cards, tablets, smart-phones...), 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 while designing such small devices. These trade-offs give rise to a plethora of unique system and/or processor architectures and it is up to the designer of such embedded devices to implement the industry standard protocols allowing 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] released every 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 and automated tools are definitely 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 boundaries even at later stages of the design work-flow, dubbed “The Softening of Hardware” [28]. This allows for more granular trade-offs which can lead to more efficient designs and the automated framework should also allow this exploration.

Cryptographers themselves could also benefit from such automated tools as 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 ZKCrypt. These tools were built for general purpose computers, using higher level tools (such as GMP[5], a multi-precision library), and are not very well suited for small, embedded devices. While these tools could in theory be ported to embedded 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) co-design spectrum.

[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://gmplib.org/

1.3 Original Contribution

A custom framework is needed to address the specific needs of HW-SW codesign and our approach is to extend the CACE Project Zero Knowledge Compiler (CACE ZKC). Designing a complete framework from scratch is deemed impractical and the CACE ZKC is already a sufficiently good compiler for Σ protocols (a zero knowledge proof of knowledge protocol involving 3 rounds: commitment, challenge and response). A higher level language (PSL) is compiled into a lower level language (PIL) and we have extended PIL so that another work-flow can be taken for features not supported by CACE ZKC. Such features are multi-party protocols, constant expressions and type inference (see Sub-section 4.2.3).

We have designed a parser that reads this PIL input (see Section 4.4) and by using LLVM (a compiler infrastructure framework) we transform this low level language into an even lower level representation (LLVM IR) which is of the Static Single Assignment (SSA) form (see Sub-section 3.3.2). We find this SSA form expressive and suitable for easy Data Flow Graph (DFG) extraction on which we base out HW-SW co-design spectrum exploration. As LLVM IR does not support modular arithmetic, we have added type tracking and type inference to our framework (see Section 4.4). The tracked types have a defined mapping to existing 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.

We chose to provide two back-ends (see Section 4.6) targeting the two edges of the HW-SW co-design spectrum. A GEZEL back-end targets a combinatorial hardware implementation (see Sub-section 4.6.2) while the C+GMP back-end targets a software implementation (see Sub-section 4.6.1). GEZEL and CACE were also extended (see Sections 4.2 and 4.3, respectively) to allow terminal communication with the "outside" world (outside referring here to the other side of the communication terminal).

As a use case for our framework, we chose to implement Schnorr's Identification Protocol (see Sub-subsection 2.3.1) on a custom system developed within a course here at KU Leuven (see Appendix A). We needed to add special optimization passes that translate modular operations into Montgomery operations (see Section 5.3). With these passes added, we were able to generate the target implementation using our framework in an automated way and achieve satisfiable results (see Section 5.4).

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 formal definition for Zero Knowledge Proofs of Knowledge (ZKPK). The chapter ends with discussing Data Flow Graphs and Control Flow Graphs, basic constructs in HW-SW codesign.

Chapter 3 gives a basic overview of the CACE Project Zero Knowledge Compiler

(CACE ZKC) which will be the base of our framework. CACE ZKC defines a higher-level and a lower-level language and both will be covered in some detail. This is followed by a comparison of related frameworks with CACE ZKC itself. Next are the tools which were used to create the custom framework (ANTLR, LLVM, GMP and GEZEL).

Chapter 4 presents our custom framework. It starts by presenting the motivation for a custom framework from two separate sides: the need for such frameworks in general and the need for embedded realization of ZKPK protocols. Next we present our extensions to CACE ZKC and GEZEL and we conclude the chapter by detailing the realization of each step within our work-flow.

Chapter 5 gives a use case of our framework, Schnorr's Identification Protocol implemented on an existing system. We give a general overview of the system and detail the small modifications made to our framework to allow an automated implementation.

Chapter 6 gives the conclusions and future work ideas that were not done during the course of this thesis, but our framework nevertheless allows those ideas to be explored.

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 which will later on be used to create a custom compiler framework.

Zero knowledge proofs of knowledge are the basis of this thesis so they need to be introduced here. First we start with group theory and modular arithmetic, then we proceed with defining zero knowledge proofs of knowledge and give a basic example of a protocol.

We conclude the chapter with introducing control flow graphs and data flow (data dependency) graphs which are needed to reason about the intermediate form the compiler generates.

2.1 Formal Languages

The theory of formal languages is needed to introduce a parser which is the basic building block of a compiler. Here we give a short overview of formal languages based on [14] and for a more detailed approach refer the reader to [13].

We start with basic definitions of a formal language:

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 $x = a_0a_1 \cdots a_n$ 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)**.** 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 $L_1L_2 = \{xy | x \in L_1, y \in L_2\}$ is the concatenation of L_1 and L_2.

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

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

$$L^+ = LL^*$$

We now have a formal basis to define a more powerful tool called regular expressions. These regular expressions will allow us to formally, yet compactly, specify the words of a language.

2.1.1 Regular Expression

When writing lexer rules for our compiler, we will need a language that defines words of the language of our interest. Such a language is called regular expressions and we briefly define it here:

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^+ □

Now that we have the tool to specify the words of our language, we need to specify the interactions between these words (e.g. which words can follow a certain word, what the word ordering is). Such rules are specified by grammars.

2.1.2 Grammars

When writing parser rules, we will need a language to specify the interactions between the words of our language. Such a language is called a grammar and we briefly define it here:

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

$$\alpha \rightarrow \beta$$

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

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

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

We now have all the tools to specify the input language of our compiler. The parser of our compiler will transform an input file, according to these rules, to a form suitable for further processing. The forms we deal at this step are concrete syntax trees and abstract syntax trees.

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:

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

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

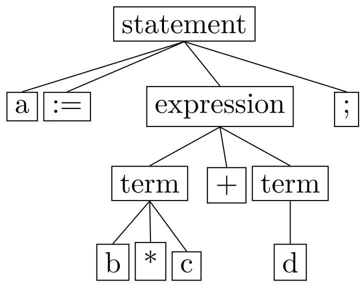

Figure 2.1: Concrete Syntax tree for a simple input and a simple grammar

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.

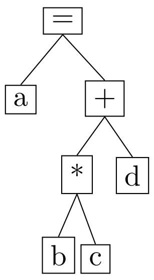

Figure 2.2: An example AST for a simple expression

After all the definitions of the input and the output have been made, it remains to define the actual tool transforming from the input to the output. This tool is referred to as the parser.

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 later manipulation. Figure 2.3 shows the typical parser flow.

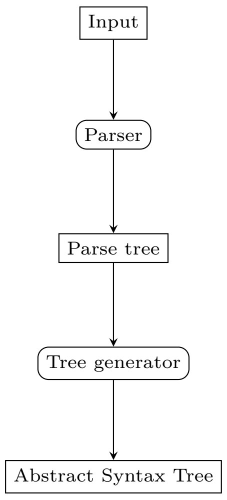

FIGURE 2.3: Parser flow

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

Before we go into defining proofs of knowledge we need to define the space we are operating on. We base our definitions on [20], but the approach we take is a bit different. We choose to start with the general and deduce the specific from it rather than taking an inductive approach.

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.

After defining the space we are operating on, it remains to define the basis of this thesis, zero knowledge proofs of knowledge.

2.3 Zero Knowledge Proofs of Knowledge

Zero knowledge proofs of knowledge give us a powerful tool that allows us to prove knowledge of a secret without actually revealing the secret. Here we need a more formal definition which we take from [4] and present as a brief overview. Some minor additions were added 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 [4].

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.

To make a more practical definition we restrict to a subset of zero knowledge proofs of knowledge called Σ-protocols.

2.3.1 Σ-protocols

A Σ-protocol is a three round honest verifier zero-knowledge proof of knowledge [27]. 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

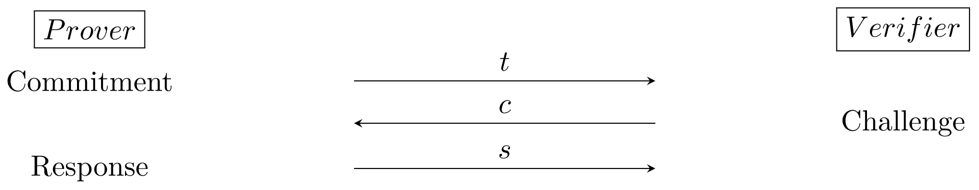

FIGURE 2.4: Sigma protocol flow

Schnorr's Identification Protocol

A simple example of a Σ protocol is Schnorr's Identification Protocol [26, 27]. 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 gives an overview of the protocol execution.

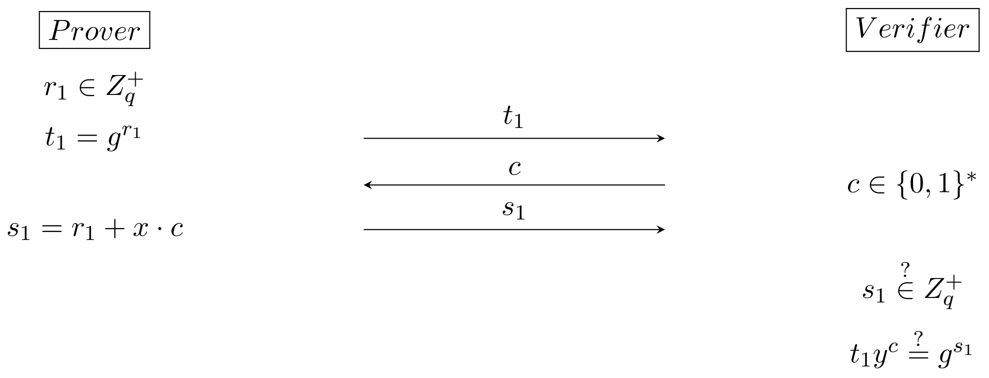

FIGURE 2.5: Schnorr's Identification Protocol

Camenisch-Stadler Notation

A simple and popular notation for specifying Sigma protocols is the Camenisch-Stadler notation [5]. 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

Our compiler will generate an intermediate form used for detailed processing. This form is supposed to represent all the operations of the protocol and can be represented as a directed graph detailing the flow of the program.

The protocol is input via a text-file and is inherently sequential. This means that the operations are laid out one after the other. This inherently imposed ordering need not be the only one and to make a better ordering w.r.t. to some goal we need constructs that will allow us to extract all the constraints on the ordering. We keep the nodes representing the operations while we classify each of the edges as either a Data Edge or a Control Edge:

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.

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.

The resulting graph can be separated into two independent graphs called a Data Flow Graph and a Control Flow Graph.

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.

A Data Flow Graph (DFG) completely and uniquely specifies the algorithm, whereas the Control Flow Graph (CFG) gives the actual implementation of the algorithm. After extracting the DFG, a different CFG can be constructed satisfying the constraints of the DFG [25].

Chapter 3

Existing Frameworks and Tools

The goal of this chapter is to review existing frameworks for implementing Zero Knowledge Proofs of Knowledge as well as tools that will be used to design our custom framework.

The chapter starts by analyzing CACE Project Zero Knowledge Compiler, which will be the basis of our custom framework. A brief overview of the framework is given followed by the Protocol Specification Language (PSL) and the Protocol Implementation Language (PIL) overview. More attention is given to PIL since it will be the language of choice of our custom framework.

A brief description of related frameworks follows, namely ZKPDL and ZKCrypt. Only a short overview is given and a high-level comparison with our custom framework. The chapter then continues on describing tools that will be used to build our custom framework:

- ANTLR is a parser generator tool producing LL(*) parsers, we will use it to construct a PIL parser which will be the front-end of our framework.

- LLVM is a compiler infrastructure framework that has recently seen wide use in many fields relating to computer science. We will use it as a middle layer along with its intermediate language LLVM IR.

- GMP is a popular multi-precision library used within the CACE ZKC. The changes that will be introduced to PIL later on will make it incompatible with the CACE ZKC, therefore we need to provide this implementation as well.

- GEZEL is a cycle-true cosimulation environment and a hardware description language. We will target GEZEL language in our framework, allowing us to perform HW-SW co-design.

3.1 CACE Project Zero Knowledge Compiler

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 by setting the following goals:

- 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

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 came to cryptography 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). Figure 3.1 depicts the typical flow of the CACE ZKPK Compiler:

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

1. The ZKPK to be implemented is specified in Protocol Specification Language (PSL). PSL is based on Camenisch-Stadler notation (see Sub-subsection 2.3.1) and allows the specification of complex Σ protocols in an easy way.
2. The Protocol Compiler transforms the specification in PSL into an implementation in Protocol Implementation Language (PIL). PIL is a Turing-complete language on its own and it also allows for automated verification.
3. The framework transforms PIL code into an implementation in C or Java that can be compiled and executed on a target architecture. Two modules are generated along with an additional support library providing math and communication. The two modules are a prover and a verifier, using network sockets for communication.
4. The PIL code is verified using the Protocol Verification Toolbox (PVT). On input the specification in PSL and the implementation in PIL, PVT employs the theorem prover Isabelle/HOL [21] to create a proof of the soundness guarantees of the generated ZKPK implementation.
5. Human readable documentation in LaTeX is also generated from PIL code.

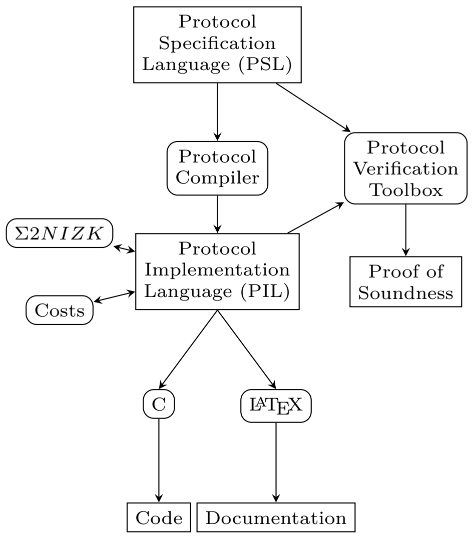
Protocol Specification Language (PSL)
Protocol Compiler
Protocol Verification Toolbox
Σ2NIZK
Protocol Implementation Language (PIL)
Costs
Proof of Soundness
C
LaTeX
Code
Documentation

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

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, allowing specification of complex Σ protocols [1, 3]. PSL is best explained following an example of a simple protocol, Schnorr's Identification Protocol (see Sub-subsection

2.3.1). We start from the Camenisch-Stadler notation for Schnorr's Identification Protocol:

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

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

```
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 and 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 $\phi(a) : Z_q^+ \to Z_p^* := g^a$. 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 Protocol Implementation Language (PIL) is the low level/intermediate language of CACE ZKPK Compiler. The language itself gives all the rounds and computations of a protocol and is meant to be easy to understand and learn, aiding verification of correctness from a user's point of view. PIL is also Turing-complete with completely specified semantics allowing automatic verification. The following features are supported in PIL:

- 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 declarations and definitions visible to all other blocks. Each block can define multiple functions, each having inputs and outputs and consisting of assignments, 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 with inputs of the current function matching the output of the previous function. For example, the outputs of Round0 from Prover 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

Although the CACE Project Zero Knowledge Compiler (CACE ZKC) will be the framework of choice for extension in this thesis, ZKPDL and ZKCrypt will be presented as well for the sake of completeness. It is mostly a brief overview along with a brief comparison with CACE ZKC.

3.2.1 ZKPDL/Cashlib

ZKPDL is a ZKPK description language used by the Cashlib framework 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 and only supports non-interactive proofs of knowledge [19]. ZKPDL does allow generation of parameters which PIL lacks [3] but this can be mitigated by defining and using constant expressions.

3.2.2 ZKCrypt

ZKCrypt is a framework built atop the CACE ZKC framework [2]. It leverages CertiCrypt[2] to produce a formal assurance of the correctness. Where CACE only allows proof of soundness using Isabelle/HOL, ZKCrypt allows formal assurance of the implementation satisfying completeness, proof of knowledge and zero knowledge properties.

ZKCrypt generates formal evidence of each of the CACE ZKC compilation steps except code generation and, as such, is complementary to the framework developed within the course of this thesis.

3.3 Other Tools

3.3.1 ANTLR

ANTLR[3] (ANother Tool For Language Recognition) is a parser generator tool that generates LL(*) parsers. The tool accepts grammar definitions as input files and

[2]http://www.msr-inria.inria.fr/projects/sec/certicrypt/index.html
[3]http://www.antlr.org

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 [23, 22].

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).

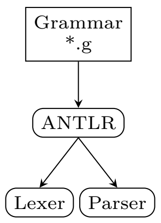

FIGURE 3.2: ANTLR Parser/Lexer generation

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.

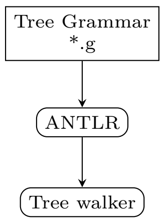

FIGURE 3.3: ANTLR Tree walker generation

3.3.2 LLVM

LLVM[4] 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 [17].

[4]http://llvm.org

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.

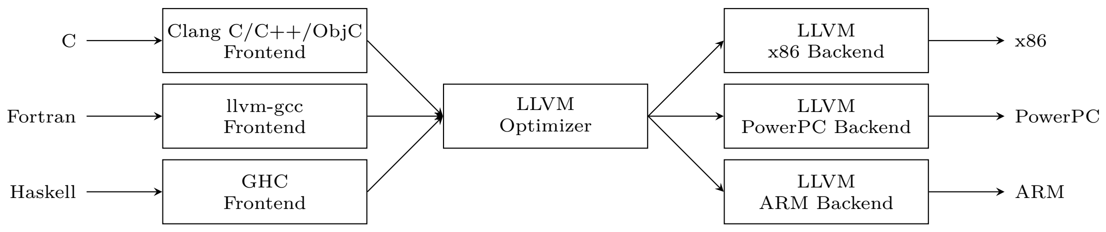

Figure 3.4: LLVM typical workflow [15]

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 [15, 16]. 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 an infinitely many registers Load-Store based RISC processor. The following snippet, taken from [15], demonstrates how two different functions implemented in C (one without recursion and one with recursion) are compiled to LLVM IR:

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}
```

```
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
}
```

A central concept while constructing LLVM IR is the *Module* [18], corresponding loosely to a source file. A *Module* consists of functions, global variables and symbol table entries.

3.3.3 GMP

GMP[5] is a multi-precision library supporting signed integers, rational numbers and floating-point numbers with precision limited only the available memory. The library provides a consistent calling interface across the different supporting types. Main target applications of the library are cryptography and computer algebra systems. The library itself aims to be highly efficient by using different algorithms for different operand sizes as well as extensive use of inline assembly and additional processor instruction sets.

3.3.4 GEZEL

GEZEL is a cycle-accurate hardware description language (HDL) using the Finite-State-Machine + Datapath (FSMD) model [9].

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

```
sfg increment {
```

[5]http://gmplib.org/

```
  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 [9]. 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

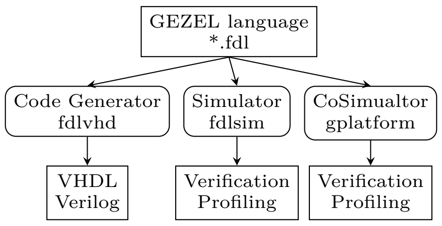

Figure 3.5: GEZEL workflow [9]

The co-simulation tool allows to cosimulate GEZEL designs with instruction-set simulations [9]. 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. This framework is based upon CACE ZKC framework with our custom additions added. Figure 4.1 gives an overview of our extensions which will be presented in Section 4.2 with the implementation details coming at the end of the chapter.

The chapter starts with presenting our motivation for a custom framework from two separate sides:

- The very need for such frameworks where we defend domain specific languages as viable alternatives to general purpose programming languages
- The need for cheap, embedded hardware realizations of such complex protocols. It is here that we have observed limitations in the CACE ZKC framework that we will try to alleviate using our extensions

We will target two separated extremes of the HW-SW co-design spectrum and as such provide two back-ends (a purely software one in C using GMP and a purely hardware one in GEZEL). To make the GEZEL Back-end functional we needed to extend the GEZEL language (see Sub-section 3.3.4) to allow new operators and interactive communication with the host the simulation is running on. These GEZEL extensions are covered in Section 4.3.

We conclude the chapter by presenting the flow of the custom framework from the front-end to the back-end in a sequential manner, the way it processes the input files given to it.

4.1 Motivation

4.1.1 Domain Specific Languages

A domain specific language is a language specifically tailored for a target application. It hides all the operations that are deemed not important and allows an abstract overview of the operations. As such, it is directly contrasted with general purpose programming languages like C.

Also, C is not a very safe language for cryptography applications as can be seen from the designer's goal of C to design a "portable" assembler. The design choice was favoring speed and efficiency over strict definitions and specifications so some things are left undefined or unspecified. It is left to the end compilers to implement it as they choose with the usual choice just implementing it in the most efficient way for the target platform. For example, the following code demonstrates 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 cryptography world should not have neither undefined nor unspecified behaviors and for this a domain specific language is a key point of this thesis.

Similar logic can be applied to other general purpose programming languages as well. Either they add complexity or make trade-offs not suitable for the field of

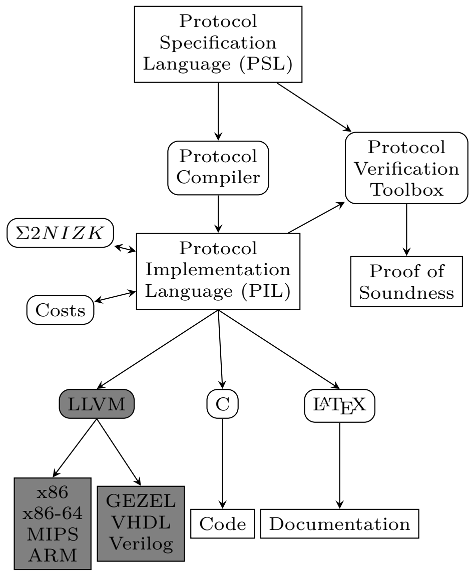

Figure 4.1: Custom framework (extensions to CACE Zero Knowledge Compiler highlighted)

cryptography. As such, the approach of current Zero Knowledge Proof of Knowledge frameworks is to define their own domain-specific language and we will follow suit by basing ours on PIL (see Sub-section 3.1.3).

4.1.2 CACE Zero Knowledge Compiler Limitations

CACE ZKC supports only C as an implementation target, using GMP (see Sub-section 3.3.3) as its multi-precision library. We perceive limitations with this approach w.r.t to small, embedded devices.

The generality coming from C and GMP incurs a penalty. For example, GMP requires two steps to perform a modular multiplication, namely multiplication and a reduction step. The multiplication step will incur a performance loss since 2 n-bit operands produce a 2n-bit result when multiplied, and the additional memory needs to be allocated and de-allocated. Also, by transforming a modular multiplication into 2 steps, the information that it was a modular multiplication is lost and has to be re-generated. This complicates optimization as well as formal verification.

The tight coupling of the main code and the supporting library limit HW-SW co-design. The CACE generated code expects a supporting library to conform to a specific interface. This limits the co-design to only coarsely-grained since it does not allow exposing of the internals of the operations supported.

4.1.3 LLVM IR

The LLVM intermediate form, LLVM IR, is of the SSA (Static Single Assignment) form (see Sub-section 3.3.2), allowing easier DFG extraction. More aggressive optimizations are possible this way as it is possible to see the usage of all the intermediate results. Also, the extracted DFG allows us to generate a corresponding CFG for a hardware realization. We can also support multiple execution modes as 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

4.2 Extensions to CACE Zero Knowledge Compiler

4.2.1 Terminal Functionality

Although we extended CACE from outside, leaving its internals intact, it was decided to add terminal functionality to the core. This allowed us to add serial communication support and to cross-check the implementations generated by our framework with the CACE implementations. Serial communication is simpler as it can be implemented over an RS232 communication protocol and it will be the default communication of our framework.

4.2.2 Lower Level

An overview of the extensions to the CACE ZKC framework was already presented in Figure 4.1 and is detailed in Figure 4.2. The input file is processed by a PIL front-end which generates LLVM IR code. Optimization passes are performed on this IR before finally transforming it to desired targets via suitable back-ends.

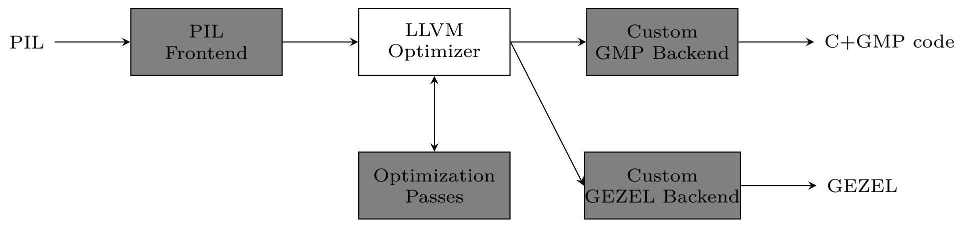

Figure 4.2: LLVM custom workflow (changes highlighted)

Optimization Passes

The compiler translates the input operation by operation and as such necessarily generates sub-optimal code or code that requires a non-trivial CFG. An example is when reading from and writing to global variables because the compiler cannot know the general context of use while translating a single store/load operation. Every access to a global variable will produce a store when writing or a load when reading and this clearly requires a non-trivial CFG since memory operation impose an inherent ordering due to only one operation allowed withing a single clock cycle.

Store/Load Pass The job of the Store/Load optimizer is to eliminate unnecessary reading from the memory. In our model of computation we assume that only the current round of the protocol is allowed to access the memory. As such, all the values stored to the memory need not be read from the memory again during the execution of the same round.

Back-ends

Two back-ends are provided targeting the two extremes of the HW-SW co-design spectrum. A purely software implementation in C using GMP is generated by the C+GMP back-end while a purely hardware implementation is generated by the GEZEL back-end.

We believe that this is a good starting point as it shows the two disjoint modes of operation, namely a non-trivial CFG generated in the case of a software implementation and a trivial CFG in the case of a hardware implementation. It is not difficult to make further enhancements that will allow to explore the co-design spectrum in the “middle”.

Formal Verification

CACE ZKC already offers a formal proof of correctness covering the transformation from PSL to PIL, and ZKCrypt extends it with further proofs (see Sub-section 3.2.2. Our framework is complementary and we need to analyze the security of the transformation from PIL to LLVM IR and LLVM IR to the back-ends whenever possible.

LLVM IR allows us to verify the correctness of the types since LLVM IR uses code of the static single assignment form, where every variable is assigned a value exactly once. This allows security assertions to be taken to a lower level than the one allowed by CACE. To assure the correctness of types, first it is necessary to lay out the DFG of the protocol round. For each node of the DFG preconditions and postconditions cab be established, such as the ones in Table 4.1. Hoare logic [12] can then be used to prove the correctness of the transformation from PIL to the LLVM IR.

As for the transformation of LLVM IR into target implementations, it might not always be possible to prove correctness, especially if the target implementation is a physical hardware realization. Since it was impossible to make this assurance it was deemed unnecessary to go this low and the blocks are assumed to be correct (if the preconditions are satisfied, the postconditions will be assured by the block) which is as low as we could go. For the software case, such properties are assumed to be assured by the library used.

Operation	Preconditions	Postconditions
Modular addition	$x \in Z_q^+, y \in Z_q^+$	$z \in Z_q^+$
Modular multiplication	$x \in Z_p^*, y \in Z_p^*$	$z \in Z_p^*$
Modular exponentiation	$x \in Z_p^*, y \in Z_p$	$z \in Z_p^*$
Zero extension	$x \in Z_p$	$z \in Z_p$

TABLE 4.1: Operation pre-conditions and post-conditions

4.2.3 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]. The grammar was re-written from scratch to allow this change to be a simple relaxation of the grammar rules

```
proof     :          execution_order (block)*
          ;
```

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

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;
  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.

4.3 Extensions to GEZEL

We decided to modify GEZEL to allow communication with the simulation host. This allows the simulated hardware within GEZEL to communicate with other programs running on the host computer. Such a change allowed us to cross-check the target implementations generated by our framework with the implementations generated by the CACE ZKC.

Another change we have introduced is adding a modular exponentiation operator. We acknowledge that GEZEL's expressiveness allows for such a construct within the language itself, however, such a step would require a lot of time and clearly was not a central point in this thesis. We have, however, wrapped the operation execution within a modular exponentiator block to allow such a block to be constructed later.

4.3.1 Terminal Communication ipblock

Serial communication with the simulation host was realized by creating an ipblock. This ipblock allows connecting to the host via pseudo-terminals or to other devices via serial ports as shown in Figure 4.3.

$$\begin{tikzcd}\text{ipblock} \arrow[r, leftrightarrow] & \text{GEZEL}\end{tikzcd}$$

FIGURE 4.3: GEZEL and ipblock

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 parsed numbers making it easier to use within GEZEL.

```
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.

4.3.2 Modular exponentiation

Although GEZEL supports modular operations (addition, subtraction and multiplication), it does so via two binary operations. First the normal operation is performed and then the modular reduction. This may be possible for addition, subtraction and multiplication where the largest intermediate result is twice the size, but it is definitely not possible for modular exponentiation where the intermediate size grows exponentially. This, along with complex hardware realizations were probably the main reasons for not having this already in GEZEL.

The modular exponentiation is not a central point in the thesis and, 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 operation is:

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

The GMP library, for reasons of intermediate results growing exponentially, supports only a modular exponentiation. This is a ternary operation requiring a modulus as well. This required a significant change in the GEZEL core but the advantages of a having a well tested and working implementation of modular exponentiation outweighed the disadvantages. To allow the modular exponentiation to be later implemented purely in GEZEL, we have wrapped the operation in a block:

```
dp modexp (in inx, iny  : ns(1024);
           in modulus   : ns(1024);
           out output   : ns(1024)) {
  always {
    output = inx ** iny % modulus;
  }
}
```

4.4 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).

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 need for no transformation, only a translation. This allows for a code that is more verifiable and more secure as the properties

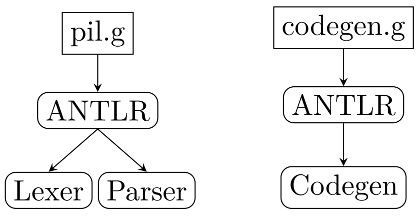

Figure 4.4: Lexer, parser and tree walker generation

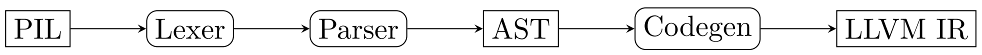

Figure 4.5: PIL frontend flow

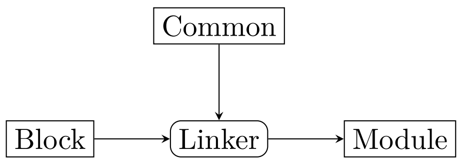

Figure 4.6: Linker

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. 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.

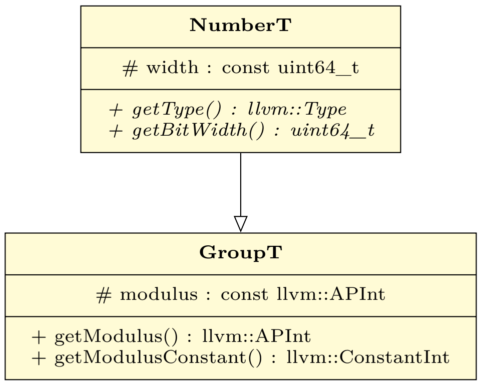
NumberT
# width : const uint64_t
+ getType() : llvm::Type
+ getBitWidth() : uint64_t
GroupT
# modulus : const llvm::APInt
+ getModulus() : llvm::APInt
+ getModulusConstant() : llvm::ConstantInt

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 [24].

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 within 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 functions act as adapters to the appropriate LLVM IR instructions 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);
                }
        |       ...
```

Since the **type** possesses all the information about the type (modulus, interval...) only the operands need to be passed in.

4.4.4 Variable Scoping

A hierarchical based approach to scoping is used meaning that each level of scoping allows access to all the variables from the previous scope. The current scope thus includes variables from the previous scope as well as local variables.

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

static stack< map<const char*, Variable, cmp_str>* > Vars;
```

Entering a block definition or a function definition generates a new level of scope and a copy of all the variables from the previous scope is pushed on the top of the stack. Likewise, leaving a block definition or a function definition removes all the variables from the previous scope by popping the top of the stack. That way, local scope is always accessible by peeking the top of the stack and local variables are added and found via this indirection.

4.5 Optimization Passes

Optimization passes search the generated LLVM IR for specific use patterns and modify them accordingly with the optimization they are supposed to perform. LLVM, as a compiler infrastructure framework, already provides the common ones such as dead code elimination and we merely complement them with other optimization passes that are valid for our specific model. For example, Store/Load will search for a Load instruction following a Store instruction with the same memory location and remove the redundant Load instruction.

4.5.1 Store/Load

The instructions of the block are iterated in order searching for Store instructions. When a Store instruction is found, another search is started from that point searching for a Load instruction loading from the same memory address. For each such Load instruction found, all the instructions that the Load instruction dominates (that use the loaded value) are modified to use the value the Store instruction was supposed to store.

```
for(BasicBlock::iterator I = BB.begin(); I != BB.end(); I++) {
  if(StoreInst *SI = dyn_cast<StoreInst>(I)) {
    Value *destination = SI->getPointerOperand();

    BasicBlock::iterator J(I);

    for(J++; J != BB.end(); J++) {
      if(LoadInst *LI = dyn_cast<LoadInst>(J)) {
        if(LI->getPointerOperand() == destination) {
          LI->replaceAllUsesWith(SI->getOperand(0));
          J--;
          LI->removeFromParent();
          changed = true;
        }
      }
    }
  }
}
```

This optimization, aside from the obvious performance optimization, has the goal of ensuring a trivial CFG per round. As discussed before, memory operations impose an inherent ordering due to only one operation allowed within a single clock cycle. The optimization presented here removes the need to load from the same memory location more than once and as such there is no imposed ordered but the DFG constrained one.

4.6 Back-ends

Two back-ends have been developed targeting the two corners of the HW-SW co-design spectrum. A purely software implementation is provided via a C back-end that uses GMP as its multi-precision arithmetic library and a purely hardware implementation is provided via GEZEL. Little attention was given to automated implementations in the middle of the spectrum as they require solving complex optimization problems such as scheduling.

4.6.1 C+GMP Back-end

The C+GMP Back-end implements the operations sequentially one after the other. Each round is implemented as a separate function with the output and input parameters being the arguments of the function. The intermediate results are kept in local variables of the function with the allocation and de-allocation simulating stack allocation and de-allocation.

Below is an example of the commitment round within the Schnorr protocol:

```
void Round1(mpz_t Out0) {
  mpz_t _r_1, _t_1;
  mpz_init(_r_1);
  mpz_init(_t_1);

  mpz_urandomm(_r_1, rstate, _mpz_11);
  mpz_set(g_r_1, _r_1);
  mpz_powm(_t_1, _mpz_3, _r_1, _mpz_23);
  mpz_set(Out0, _t_1);

  mpz_clear(_r_1);
  mpz_clear(_t_1);
}
```

Variables are named the same as they are in the input PIL file with global variables being prefixed with g. As the GMP library does not allow to input constants at operation invocation, they need to be prepared beforehand. They are declared as global variables with the prefix _mpz_ and the respective constant. The special **init** function initializes these constants along with global variables and the random number generator:

```
void init() {
  mpz_init(g_r_1);
  mpz_init(g_s_1);

  mpz_init_set_str(_mpz_3, "3", 10);
  mpz_init_set_str(_mpz_4, "4", 10);
  mpz_init_set_str(_mpz_11, "11", 10);
  mpz_init_set_str(_mpz_23, "23", 10);

  gmp_randinit_default(rstate);
}
```

And the special **clear** function clears them:

```
void clear() {
  mpz_clear(g_r_1);
  mpz_clear(g_s_1);

  mpz_clear(_mpz_3);
  mpz_clear(_mpz_4);
  mpz_clear(_mpz_11);
  mpz_clear(_mpz_23);
}
```

4.6.2 GEZEL Back-end

The GEZEL Back-end implements one GEZEL datapath per round (Figure 4.8). The CFG is trivial and there is no imposed operation ordering but the DFG constrained one. The hardware implementation is thus purely combinatorial and each invocation of addition, multiplication, exponentiation requires a new instantiation of an adder, multiplier, exponentiator, respectively.

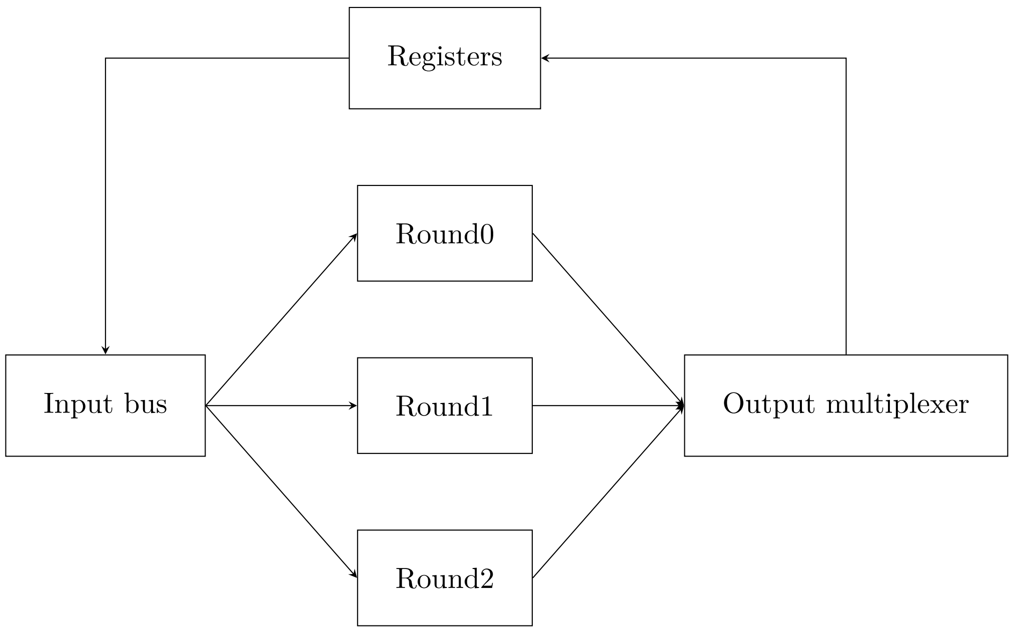

FIGURE 4.8: Generated GEZEL model of Schnorr's protocol

Below is an example from the commitment round of the Schnorr Protocol:

```
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 memory accesses are converted to register accesses and each datapath gets access to each of the registers. 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.

Auxiliary signals represent intermediate values and register accesses and are declared first followed by the block instances. Constant values are passed as such and represent a programmable interconnect connecting either to ground or supply. Lastly, the **always** block manages the wire cross-connections. Values to be stored in registers are either connected to a controlling signal or in a loop from the current register output. Register controlling outputs are propagated to the main data path instantiating the party:

```
dp prover {
  reg r_r_1 : ns(160);
  reg r_s_1 : ns(160);

  ...

  sig r_r_1_o_0, r_r_1_o_1, r_r_1_o_2 : ns(160);
  sig r_s_1_o_0, r_s_1_o_1, r_s_1_o_2 : ns(160);

  ...

  use Round0(r_r_1, r_r_1_o_0, r_s_1, r_s_1_o_0);
  use Round1(r_r_1, r_r_1_o_1, r_s_1, r_s_1_o_1, _t_1);
  use Round2(r_r_1, r_r_1_o_2, r_s_1, r_s_1_o_2, _c, _s_1);

  ...
```

Logic is added that multiplexes these values (r_r_1_o_x, r_s_1_o_x) to the respective register such that only the current executing round can modify the value in the register.

Chapter 5

Use Case: Schnorr's Identification Protocol on a Custom System

This chapter aims to present a use case of our framework and the possibilities for extension. We take Schnorr's Identification Protocol and implement it on a system not unlike today's smart cards. This requires us to write a custom back-end for our framework and here we detail the steps.

5.1 Motivation

So far, two extremes of the HW-SW co-design spectrum have been explored using our framework and we think there is little need to justify an exploration of the spectrum somewhere in the "middle". Instead of generating our own system we take an existing system and demonstrate the flexibility and adaptability of our framework.

The target system is not unlike today's smart cards in the manner that it uses a small and cheap micro-controller coupled with a custom co-processor to offload the more demanding calculations. We have already discussed the advantages of Zero Knowledge Proofs of Knowledge over traditional policy and authentication verification (see Section 1.1) and we deem the Schnorr's Protocol simple, yet concise enough to demonstrate the applicability of such protocols and our framework.

5.2 Target System

The system targeted by the framework is an existing system developed within a course here at KU Leuven. The system includes a general purpose micro-controller, 8051 in this case, coupled with a custom designed cryptographic co-processor (see Appendix A).

The co-processor implements the Montgomery multiplication as the project deemed this the most critical operation that needed a hardware implementation. The operations of modular multiplication and modular exponentiation are handled in software using the Montgomery multiplication as a building block.

Command queuing (see Appendix A) was used as a technique to compensate for the slower execution speed of the main processor. It also allows the main processor to perform other tasks while the co-processor is executing the uploaded instructions.

5.3 Implementation Flow Example

First we start with the PSL implementation of Schnorr's Identification Protocol (already given in Sub-section 3.1.2). This PSL file is given to the CACE ZKC (see Section 3.1) to produce the resulting PIL and C files. The C files are compiled to produce an actual implementation of a prover and a verifier which are then configured and tested on a general purpose computer. This part is equivalent to the CACE ZKC flow.

Next, our extension takes the resulting Schnorr PIL file and compiles it into an intermediate form capturing the operations within it. As the target does not support modular multiplication and modular exponentiation, intrinsic lowering is performed, adding the necessary transfer to and from the Montgomery domain. Since this step would generate a lot of redundant intermediate results, a special optimization step is run to eliminate these intermediate results.

5.4 Results and Possible Improvements

5.4.1 Results

The entire protocol run requires around 2 mil. cycles, which is roughly 0.5 s running at the proposed frequency of 4 MHz.

5.4.2 Further Improvements

The approach taken was to reuse the software library provided. We note that the ideal goal step would be to emit the required instructions directly and such a step is not difficult as we are dealing with LLVM, a compiler infrastructure framework. Emitting the instructions directly instead of using library calls can yield a further optimization as the results are transferred less from the co-processor to the main processor. Here we omitted this step as the gains were too little and we deemed the final compilation not to be a key point of this thesis.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Previous compiler frameworks for Zero Knowledge Proofs of Knowledge (ZKPK) targeted general purpose processors and had no HW-SW co-design exploration capabilities. We have shown that HW-SW co-design is possible by extracting the Data Flow Graph (DFG) and have proven the PIL language expressive, yet simple enough for easy DFG extraction. By implementing the two extremes of the co-design spectrum, we have opened the path for automated exploration of the spectrum in the "middle". The changes we have introduced do not interfere with the standard CACE ZKC work-flow as we allow previous CACE examples to work with our framework as well. We have merely added extensions to PIL that could later prove useful outside of the CACE ZKC work-flow.

The extensions to the PIL Language have allowed more complex, multiparty protocols as well as parameter specification inside of PIL files. Protocols can be implemented simply by writing each of the steps of the protocol flow in the PIL language. The transformation is also reversible thanks to the CACE ZKC LaTeX output. The advantages of a domain specific language have clearly been shown throughout this thesis.

Our extensions to the CACE ZKC support library have allowed communication over serial ports and thus multiple combinations of general purpose computer and embedded device interconnectivity can be made. Serial port communication is easier to implement and it can also serve for cross-checking and cross-validating of custom implementations.

Finally, our framework is easily extendable to support new points in the HW-SW co-design spectrum as we have clearly shown with our use case. The custom system targeted by the use case is not unlike the commodity hardware available for these purposes and the protocol chosen, albeit simple, is still of practical value.

6.2 Future Work

We believe that this work should eventually give rise to an automated HW-SW co-design framework where the user will be offered fine-grained choice of trade-offs. Automated HW-SW balancing could be made that allows trading speed for smaller device size and less power consumption.

We also believe that 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 and mostly deal with verifiability and security. The process requires some standardization before changes are applied and as this takes time, it was not pursued within the course of this thesis.

Eventually, we think that our framework should support automatic parameter generation as leaving these choices to the end user might be problematic from a usage perspective as well as a security perspective. The user simply cannot and should not know all the needed constraints of the protocol and the only constraints he should be concerned about are implementation constraints. The framework developed within the course of this thesis already allows for compile time/constant expression and these expressions can be used as a starting point for defining a generator macro/function that should automatically generate the required parameters.

Finally, while extending the CACE ZKC, 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. We find it an interesting possibility for the LLVM IR to become a common language for the entire CACE Project.

Appendices

Appendix A

HW - SW codesign project

The goal of this project was to design a cryptographic system that would support 1024-bit encryption RSA and ElGamal. The processor given was an 8051-based. The choice is similar to what many smart-card manufactures offer [10, 11]. Basically, the 8051 is an industry proven, well tested and widespread micro-controller. This it owes to having been around for more than 20 years.

Since the processor was not fast or powerful enough, HW-SW co-design was employed to offload some operations onto a separate coprocessor.

A.1 System design

The 8051 has 4 8-bit ports available for IO. Due to port sharing, not all of these ports can be used in specific configurations (e.g. using an external memory takes away the ports P0 and P2; using interrupts, UART or external ROM takes away port P3). This was the reason why only port P1 was chosen as a means of port IO signalization with the coprocessor. Only 2 pins were used for the signalization so the rest can still be used for other purposes.

Shared memory was used as a means of communicating the data to the coprocessor. Addresses 0x600–0x801 are mapped to the coprocessor's addresses 0x000–0x201. Addresses 0x000–0x600 are left for the main processor.

The operations chosen for offloading to the coprocessor were the Montgomery multiplication and Montgomery inversion for speed and flexibility reasons.

The fastest the Montgomery multiplication could be made in SW was 1.3 s@12 MHz which is not fast enough. HW realization on the other hand took 2300 cycles, which goes roughly to 0.2 ms@12 MHz.

For the inversion case it took 16 s@12 MHz, while in the HW realization 1.3 mil.cycles, roughly 100 ms@12 MHz.

For the modular exponentiation case it was decided to keep it in SW using the Montgomery multiplication of the coprocessor. Since the main processor is 12 times slower some techniques have been applied to eliminate the communication overhead.

The frequency of the system was lowered to 4 MHz to match most of the manufacturers.

A.2 Coprocessor design

The coprocessor has:

- 6 1024-bit registers (u, v, p, R, S, Q)
- 1 1024-bit datapath (adder and a shifter)
- 10-bit datapaths and registers for loop and address counters

The shared memory is used in the following way:

- 0x600–0x680 1024-bit number
- 0x680–0x700 1024-bit number
- 0x700–0x780 1024-bit number
- 0x780–0x800 command queue (up to 128 commands)
- 0x800 state signaling from the coprocessor

The instructions are the following:

- Halt - stops the execution of the coprocessor and signals the done to the main processor
- Init - initializes the coprocessor (just sets the modulus for now)
- Montgomery multiplication - executes the Montgomery multiplication on the registers u and v and stores the result back to u
- Montgomery squaring - executes the Montgomery squaring of the register u and stores the result back to u
- Montgomery inversion - executes the Montgomery inversion of the register u and stores the result back to u (destroys the previous values of u and v)
- Load u from the shared memory - loads the register u from the shared memory location 0x00 or 0x80 or 0x100
- Load v from the shared memory - loads the register v from the shared memory location 0x00 or 0x80 or 0x100
- Store result to shared memory - stores the result of the operation to memory location 0x00 or 0x80 or 0x100
- Store quotient to memory (Montgomery only) - stores the quotient of the Montgomery operation to memory location 0x00 or 0x80 or 0x100

Special instructions were provided for loading from shared memory because the coprocessor is 24 times faster with loading from memory (12 clock cycles for 1 instruction cycle, 2 instruction cycles for loading on the processor). This way, a third parameter can be stored in the shared memory for faster fetching.

A.2.1 Command queueing

The idea is to allocate special storage in the shared memory for the commands the coprocessor should execute. The main processor then simply fills this memory with the commands the coprocessor needs to execute. The main processor then starts the coprocessor which will execute these instructions.

This is also useful if the main processor should be able to perform other operations while the coprocessor executes the commands provided. The method has also been tested with the main processor being slower 144 (additional 12x slowdown) times than the coprocessor and it gave satisfying results (only 5x increase in cycle numbers for RSA).

A.2.2 Datapath

The datapath design revolves around a 1024-bit adder and a shifter in series. The shifter can shift one position to the left, one position to the right or just pass the input.

The inputs to the adder are multiplexed between 0, 1, u, v, p, R, S, power for the x operand. For the v operand, the multiplication step is multiplexed instead of the power.

The result always gets assigned back to u. This was to allow chaining of Montgomery multiplications when exponentiation is performed. This way, no redundant copying is necessary from/to the shared memory. What is also possible this way is to update the shared memory while the coprocessor is executing (this helps the elimination of the communication overhead). The technique was not tested within this project.

A.2.3 Modifications to the Montgomery product

The Montgomery product computation algorithm has been modified to include computing the quotient (as is called in [29]). This allows for doubling the bit-length ([29, 8]) of the crypto-coprocessor in software. This technique was not tested within this project.

A.3 Implementation

A.3.1 Software in C

A library is provided exposing primitives which are executed on the coprocessor:

- montpro - Montgomery product
- montinv - Montgomery inversion
- modexp - Modular exponentiation

Also provided are the following software methods:

- add1024 - adding 1024-bit numbers
- subtract1024 - subtracting 1024-bit numbers
- multiply1024 - multiplies 1024-bit numbers to produce a 2048-bit
- larger_or_equal - checks if the number is larger or equal than a number

These operations were left in software as they were already fast enough there. Multiplication was left in case someone would want to use the bit doubling method. These functions are documented in the lib.h header file.

A.3.2 Hardware in VHDL

During the co-design phase, separate datapaths were made for the adder and the bit shifter. This was to allow a different implementation in VHDL as the GEZEL-to-VHDL tool (fdlvhd) generates one .vhd file per datapath. The target devices usually have sophisticated CLA logic implemented so that efficient adders in terms of space and speed can be generated.

The same reasoning goes for ASIC design. Usually the libraries provided from the foundries will include efficient designs of adders which can be combined to produce the required adder. Even if they don't, having a separate block allows the designer to concentrate more on where optimization really counts (critical data path, major effect on area).

In our specific case, the GEZEL-to-VHDL tool generated 2 1024-bit adders, one with carry-in and one without it. Manual intervention was required to change it to a single 1024-bit adder with carry-in. Later on 2 513-bit adders were used to reduce the slice count even further. Putting 2 smaller adders helped the synthesis tool spread out and perform better interconnect. Custom add_sub.customxx.vhd are provided in the vhdl subfolder.

A.4 Results

The RSA encryption and decryption with provided exponents (9-bit for encryption and 1024-bit for decryption) take around 5 mil.cycles to complete in total. On the 4 MHz clock frequency this amounts to around 1.25 s (average for encryption/decryption 600 ms).

The ElGamal encryption and decryption with provided exponents (128-bit for encryption and decryption) take around 4.5 mil.cycles to complete in total. On the 4 MHz clock frequency this amounts to around 1.125 s (average for encryption/decryption 560 ms).

These times put the design along with most of the systems presented in [10, 11].

Eventually the implemented design did not meet the area constraint. It used 101% of the device (Table A.1). The maximal frequency is 20.449 MHz which is well above the required 4 MHz.

	Used	Available	%
Slices	13951	13696	101%
Flip-flops	6267	27392	22%
4-input LUTs	26033	27392	95%

TABLE A.1: Implementation results

Following is an excerpt from the synthesis tools report (available as the file sys. syr):

```
# Adders/Subtractors                                         : 5
 10-bit adder                                                : 2
 11-bit subtractor                                           : 1
 513-bit adder carry in                                      : 2
# Registers                                                  : 6251
 Flip-Flops                                                  : 6251
# Comparators                                                : 1
 1024-bit comparator equal                                   : 1
# Multiplexers                                               : 1
 1026-bit 4-to-1 multiplexer                                 : 1
```

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] José Bacelar Almeida et al. *Full Proof Cryptography: Verifiable Compilation of Efficient Zero-Knowledge Protocols*. Cryptology ePrint Archive, Report 2012/258. http://eprint.iacr.org/. 2012.

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

[4] 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.

[5] 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.

[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] Wieland Fischer and Jean-Pierre Seifert. “Increasing the Bitlength of a Crypto-Coprocessor”. In: *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*. CHES ’02. London, UK, UK: Springer-Verlag, 2003, pp. 71–81. ISBN: 3-540-00409-2. URL: http://portal.acm.org/citation.cfm?id=648255.752711.

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

[10] Helena Handschuh and Pascal Paillier. “Smart Card Crypto-Coprocessors for Public-Key Cryptography”. In: *CARDIS*. 1998, pp. 372–379.

[11] Helena Handschuh and Pascal Paillier. “Smart Card Crypto-Coprocessors for Public-Key Cryptography”. In: *Smart Card Research and Applications*. Ed. by Jean-Jeacques Quisquater and Bruce Schneier. Springer, 2000, pp. 386–394.

[12] C. A. R. Hoare. “An axiomatic basis for computer programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002. ISBN: 3-540-43376-7.

[22] 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.

[23] 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.

[24] 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.

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

[26] C. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology âĂŤ 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.

[27] 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.

[28] 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.

[29] Masayuki Yoshino, Katsuyuki Okeya, and Vuillau Camille. "Unbridle the bit-length of a crypto-coprocessor with montgomery multiplication". In: *Proceedings of the 13th international conference on Selected areas in cryptography*. SAC'06. Montreal, Canada: Springer-Verlag, 2007, pp. 188–202. ISBN: 978-3-540-74461-0. URL: http://portal.acm.org/citation.cfm?id=1756516.1756535.