KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

Implementation and Evaluation of Zero-Knowledge Proofs of Knowledge

Boran Car

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

Promotoren:
Bart Preneel
Ingrid Verbauwhede

Assessoren:
Claudia Diaz
Francky Catthoor

Begeleiders:
Josep Balasch
Alfredo Rial

Academiejaar 2011 – 2012

Preface

Boran Car

Contents

Preface **i**

Abstract **iii**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

1.1 Zero Knowledge Proofs of Knowledge 1
1.2 HW-SW codesign 2
1.3 Domain specific languages 2

2 Technical preliminaries **5**

2.1 Formal Languages [4] 5
2.2 Montgomery Product 7
2.3 Turing Machine 9
2.4 Zero Knowledge Proofs of Knowledge 9
2.5 Data Flow Graph and Control Flow Graph 10

3 Existing frameworks and tools **11**

3.1 CACE Project 11
3.2 ZKPDL 14
3.3 Comparison of CACE and ZKPDL 14
3.4 ANTLR [11, 10] 14
3.5 LLVM 15
3.6 GEZEL 16

4 Custom framework **19**

4.1 Motivation 19
4.2 Extensions to GEZEL 20
4.3 Extensions to CACE 20
4.4 PIL frontend 21
4.5 Coprocessor design 22

5 Use case: Direct Anonymous Attestation **25**

6 Conclusion **27**

Bibliography **29**

Abstract

List of Figures

1.1 Gajski-Kuhn Y-chart 3

2.1 Montgomery product flow 8
2.2 Turing machine 9

3.1 CACE typical workflow [1] 12
3.2 ZKPDL typical flow [8] 14
3.3 LLVM typical workflow 15

4.1 Custom framework (extensions to CACE highlighted) 20
4.2 LLVM custom workflow (changes highlighted) 21
4.3 PIL frontend flow 21
4.4 Linker 22

List of Tables

3.1 Detailed comparison of ZKPoK compilers [2] 17

Chapter 1

Introduction

1.1 Zero Knowledge Proofs of Knowledge

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

- Discrete Logarithm Problem
- Factorization Problem

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

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

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

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

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

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

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

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

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

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

1.2 HW-SW codesign

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

Today's stringent measures require a bigger inter-dependency. Instead of searching for a best algorithm tailored for specific hardware or searching for the best hardware tailored to a specific algorithm/task, why not design the hardware and software at the same time. This way bigger decision and trade-offs can be made more granular [14].

1.3 Domain specific languages

C and C++ are currently the dominant languages for embedded system programming. It is therefore natural to expect that real-world embedded cryptography applications be implemented in either of those two languages. This can lead to hidden and difficult problems to debug. A lot of cases were left unspecified or undefined in the C or C++ standard. Hence, compiler implementations have a choice on how to implement it. The usual approach is to define it as what is most efficient in terms of the platform. This sacrifices the safety and portability to ensure efficiency.

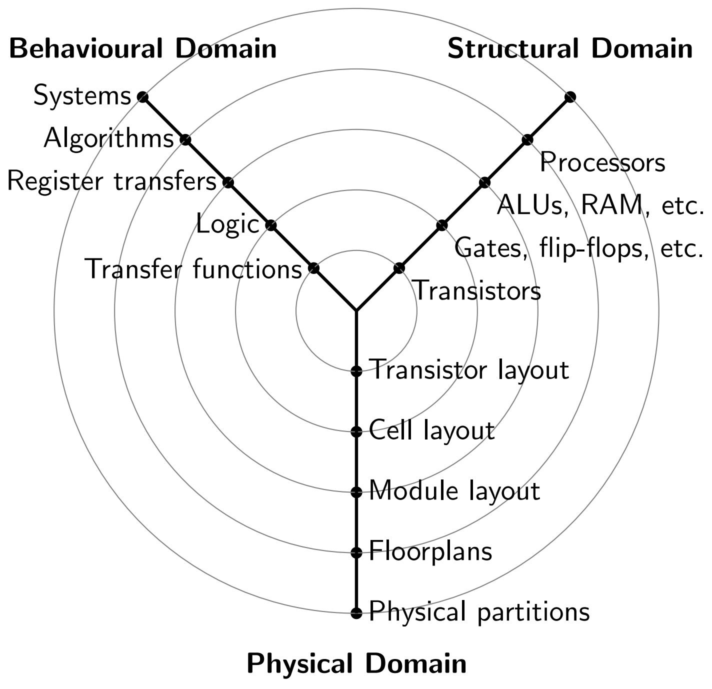


FIGURE 1.1: Gajski-Kuhn Y-chart

```
#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);
}
```

The evaluation order is unspecified so depending on the compiler, the output may be 3, 4, 7 or 4, 3, 7. The compiler usually chooses the optimal way to calculate it given a platform.

The security world does not handle undefined and unspecified values well. It makes sense to build protections against unsafe, undefined and unspecified operations into the language. It also makes sense to make the language easy to learn and understand. This is the common requirement of a domain specific language, a language tailored for a specific domain.

Chapter 2

Technical preliminaries

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

2.1 Formal Languages [4]

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)**.** The 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

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

2.1.1 Regular expression

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

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

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

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

2.1.2 Grammars

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

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

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

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

Definition 11 (LL grammar)**.** A context free grammar G is an LL grammar if $\beta \in \Sigma(\Sigma \cup V)^*$

2.2 Montgomery Product

Modular multiplications would involve trial division after the multiplication step were it not for the algorithm know as Montgomery product invented by Peter L. Montgomery [9].

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

Definition 12. Define an radix R such that $RR^{-1} - NN' = 1$. The residue $\overline{a}$ of a is $\overline{a} = aR$

Theorem 2. The algorithm for Montgomery reduction is

Algorithm 1 Montgomery reduction

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

Proof. Assume $T < R$, then

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

□

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

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

```
end function
```

Koç goes a step further and inlines the reduction to see possible optimizations when implementing the algorithm on multiple architectures [5]:

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

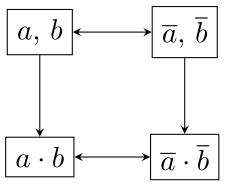
If such a primitive is provided on an architecture, the reduction then becomes:

```
function MontgomeryReduce($T$)
    return MontgomeryProduct($T$, $R^2$)
end function
```

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

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

This is also depicted in Figure 2.1 where both paths are shown when starting from the integer domain.



$$\begin{tikzcd}
\boxed{a,\ b} \arrow[r, leftrightarrow] \arrow[d] & \boxed{\overline{a},\ \overline{b}} \arrow[d] \\
\boxed{a \cdot b} \arrow[r, leftrightarrow] & \boxed{\overline{a} \cdot \overline{b}}
\end{tikzcd}$$

Figure 2.1: Montgomery product flow

2.3 Turing Machine

A Turing machine is a mathematical model of computation [13]. Informally, it consists of:

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

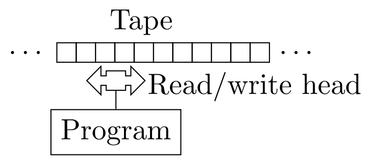


Figure 2.2: Turing machine

A more formal definition comes from Hopcroft [4]. A Turing machine is defined as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, q_0, F, \delta \rangle$

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

2.4 Zero Knowledge Proofs of Knowledge

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

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

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

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

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

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

Informally, the previous two properties state that

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

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

- perfectly indistinguishable
- statistically indistinguishable
- computationally indistinguishable

2.5 Data Flow Graph and Control Flow Graph

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

The usual step in converting an algorithm in software to an algorithm in hardware is to extract the DFG [12].

Chapter 3

Existing frameworks and tools

This chapter starts with an overview of existing frameworks for implementing Zero Knowledge Proofs of Knowledge. Currently available are:

- CACE Project compiler
- ZKPDL, a library
- Idemix

A detailed comparison of the frameworks follow. The chapter then continues on describing ANTLR, a parser generator tool producing LL(*) parsers. Next on is LLVM, a compiler infrastructure framework that has seen wide use recently in many fields relating to compilers and computers in general. ANTLR and LLVM will be used to make a custom compiler later on so some knowledge about them is necessary. Next is GEZEL, a cycle-true cosimulation environment which can also generate VHDL or Verilog.

3.1 CACE Project

The CACE Project compiler has the following typical flow (as depicted in Figure 3.1):

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

The PIL itself gives all the details of a protocol and is easily understandable and exportable to LaTeX. The ease of understanding the steps will allow better cryptographic analysis. The tool must now provide an accurate and representable translation which does not lose any properties.

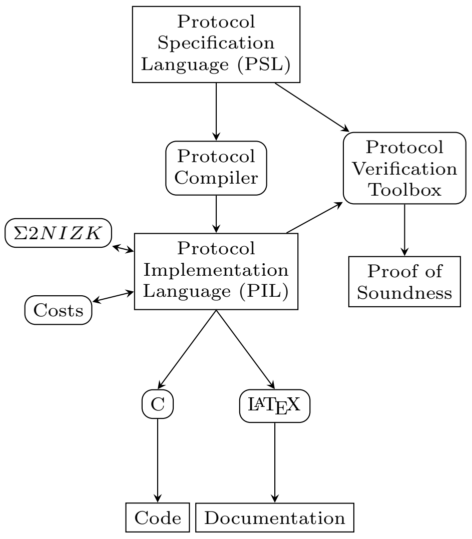

Protocol Specification Language (PSL)
Protocol Compiler
Protocol Verification Toolbox
Σ2NIZK
Protocol Implementation Language (PIL)
Costs
Proof of Soundness
C
LaTeX
Code
Documentation

Figure 3.1: CACE typical workflow [1]

3.1.1 PIL

PIL, the Protocol Implementation Language is a language for specifying protocols. As a language on its own it has support for the following features:

- global shared constants (parameters)
- global constants (parameters)
- global variables
- local variables
- conditionals
- loops
- functions
- predicates

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

```
/*
 * 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 ZKPDL

ZKPDL is a description language for describing Zero Knowledge Proofs of Knowledge [8].

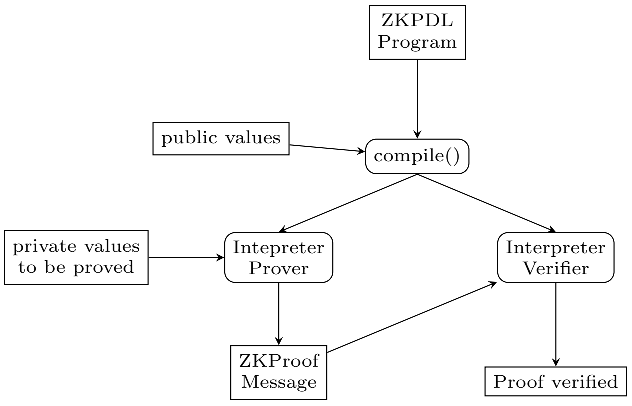


Figure 3.2: ZKPDL typical flow [8]

3.3 Comparison of CACE and ZKPDL

A comparison between CACE and ZKPDL is given in table 3.1.

3.4 ANTLR [11, 10]

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

The input to ANTLR is a context-free grammar augmented with syntactic and semantic predicates. Syntactic predicates allow arbitrary look-ahead while semantic predicates allow the state constructed up to the point of the predicate to direct the parse [11].

The lexer and parser grammars can be combined into a single file. In such cases, parser rules are written in lowercase, while the lexer rules are written in uppercase. ANTLR workflow allows for tree generators and tree walkers.

3.5 LLVM

LLVM (Low Level Virtual Machine) is a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs [7].

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 3.3.

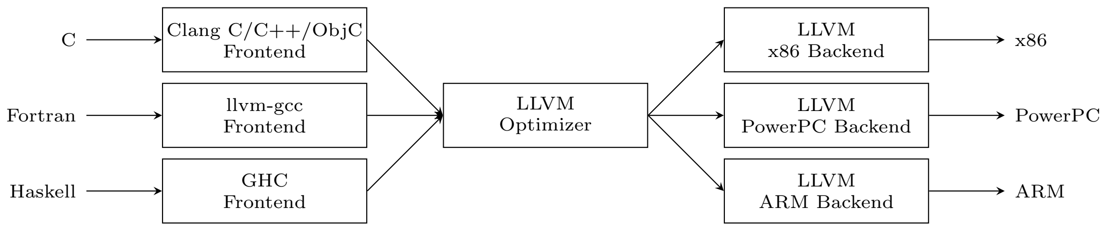


FIGURE 3.3: LLVM typical workflow

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)

3.5.1 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 [6]. It resembles the assembly language of a infinitely many registers Load-Store based RISC processor.

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

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

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

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

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

done:
  ret i32 %b
}
```

The central concept in constructing LLVM IR is the Module. The Module represents the top level container of the LLVM IR. There are multiple ways on how to generate the IR. The one described here is by building a graph and creating objects along that graph.

3.6 GEZEL

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

	CACE compiler	ZKPDL compiler
Basic proof goals	knowledge of preimages under arbitrary group homomorphisms, including RSA or Paillier type homomorphisms	exponentiation homomorphisms only
Basic protocols	SigmaPhi, Damgard/Fujisaki, SigmaExp	SigmaPhi, Damgard/Fujisaki
Composition techniques	Boolean AND, OR, and providing knowledge of k out of n secret values	Boolean AND only
Input language	inspired by the standard notation for ZK-PoK	inspired by the English language
Output language	C for implementation, LaTeXfor documentation	specifically design language with interpreter and API to C++
Type of output	complete source code, which can directly be compiled to machine code	meta code or libraries, which partly have to be instantiated by the calling procedure
Optimizations	reduction of redundant terms in the proof goal	1. reduction of redundant terms in the proof goal 2. possibility of caching precomputations to reduce runtime
Additional features	1. existence of a formal verification toolbox, which proves the correctness of the compilation process 2. modular design with interfaces to other cryptographic compilers developed within the CACE project	possibility to specify the generation of protocol inputs

TABLE 3.1: Detailed comparison of ZKPoK compilers [2]

Chapter 4

Custom framework

4.1 Motivation

The C code generated by CACE uses GNU GMP which is a multi-precision arithmetic library. This library is tailored for desktop computers and is not well suited for small embedded devices. The ZKPDL library is an interpreter library which is also not well suited for small embedded devices.

The custom framework build withing the course of this thesis will attempt to combine the advantages from both projects. As the CACE project is deemed more suited for the task, the custom framework will extend CACE. The custom framework will aim to be a compiler that can produce:

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

LLVM was chosen as the compiler infrastructure. Its intermediate form (LLVM IR) is of SSA (Static Single Assignment) form which allows easy extraction of DFG (Data Flow Graph). This allows for more aggressive optimization as well as conversion directly to HDL (Hardware Description Language).

4.2 Extensions to GEZEL

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

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

4.3 Extensions to CACE

Extensions to the CACE frameworks are presented in Figure 4.1. The custom part is further explained in Figure 4.2.

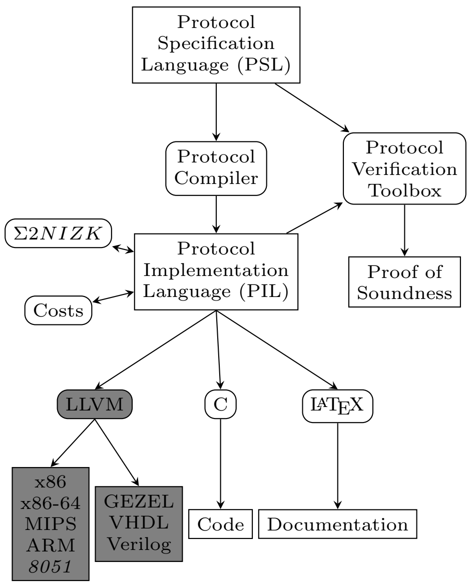


Figure 4.1: Custom framework (extensions to CACE highlighted)

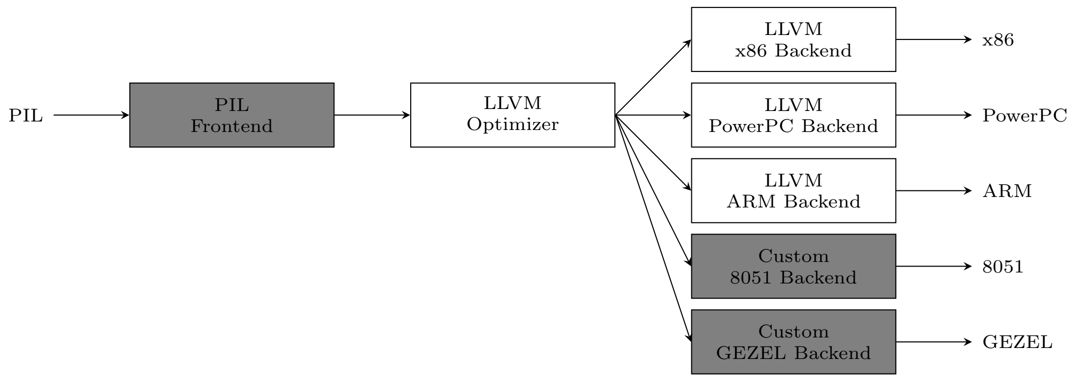


FIGURE 4.2: LLVM custom workflow (changes highlighted)

4.3.1 Extensions to PIL

Compile time constants and constant expressions were needed to allow more advanced specifications of protocols. The CACE PIL parser did not allow constant expression as parameters of variables.

4.4 PIL frontend

PIL frontend flow is depicted in Figure 4.3. An input PIL is read by the Lexer producing input for the Parser. The Parser reads these and generates an abstract syntax tree (AST) which is fed to the Codegen tree-walker that generates the code (in the form of LLVM IR). Both the lexer grammar and the parser grammar are specified in the file pil.g. The tree-walker and the code generator is specified in the file codegen.g.

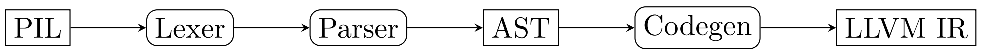


FIGURE 4.3: PIL frontend flow

The code generation process generates one module per block. The Common block module is augmented with the functions provided by the VM. Every other block except the Common block gets a Common block linked in. This process is depicted in Figure 4.4.

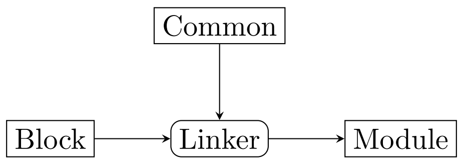


Figure 4.4: Linker

4.5 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.

4.5.1 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).

4.5.2 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.

4.5.3 Modifications to the Montgomery product

The Montgomery product computation algorithm has been modified to include computing the quotient (as is called in [15]). This allows for doubling the bit-length ([15, 3]) of the crypto-coprocessor in software. This technique was not tested within this project.

4.5.4 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.

4.5.5 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.

Chapter 5

Use case: Direct Anonymous Attestation

Chapter 6

Conclusion

Bibliography

[1] *A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on Σ-Protocols*. An extended abstract of this work will be presented at ESORICS 2010 stephan.krenn@bfh.ch; thomas.schneider@trust.rub.de 14825 received 10 Jun 2010, last revised 4 Aug 2010. 2010. URL: http://eprint.iacr.org/2010/339.

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

[3] 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.

[4] 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.

[5] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr. “Analyzing and Comparing Montgomery Multiplication Algorithms”. In: *IEEE Micro* 16.3 (1996), pp. 26–33.

[6] 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.

[7] 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.

[8] 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.

[9] Peter L. Montgomery. “Modular Multiplication Without Trial Division”. In: *Mathematics of Computation* 44.170 (1985), pp. 519–521. URL: http://www.jstor.org/pss/2007970.

[10] 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.

[11] 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.

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

[13] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230. eprint: http://plms.oxfordjournals.org/content/s2-42/1/230.full.pdf+html. URL: http://plms.oxfordjournals.org/content/s2-42/1/230.short.

[14] 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.

[15] 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.