ASSIGNMENT OF MASTER’S THESIS

Title: Implementation and Effectiveness Evaluation of the VeraGreg Scheme on a Low-Cost Microcontroller

Student: Bc. Jan Říha
Supervisor: Ing. Jakub Klemsa
Study Programme: Informatics
Study Branch: Design and Programming of Embedded Systems
Department: Department of Digital Design
Validity: Until the end of summer semester 2019/20

Instructions

Implement the VeraGreg scheme, including, but not limited to, Paillier cryptosystem, big number arithmetics, etc., for chosen well-secured low-cost microchip. The implementation will maximally employ its available security features while each part of the implementation will be carefully analyzed, in particular with respect to side-channel attacks.

Besides the implementation of VeraGreg, implement a naive scheme using symmetric encryption and compare these approaches, in particular from performance and effectiveness point of view.

References

Will be provided by the supervisor.
Master’s thesis

Implementation and Effectiveness Evaluation of the VeraGreg Scheme on a Low-Cost Microcontroller

Bc. Jan Říha

Department of Digital Design
Supervisor: Ing. Jakub Klemsa

May 6, 2019
Acknowledgements

Thanks to all. Thanks a lot.
Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 6, 2019
Czech Technical University in Prague
Faculty of Information Technology
© 2019 Jan Říha. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author’s permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis
Abstrakt

Homomorfní šifrování je efektivním způsobem jak zajistit soukromí a zároveň zachovat možnost zpracování dat. Framework VeraGreg, na rozdíl od jiných existujících homomorfních kryptosystémů, umožňuje verifikaci operací, které byly s šifrovými texty provedeny.

Tato práce se zabývá implementací frameworku VeraGreg a zhodnocením jeho efektivity v porovnání s naivním schématem založeným na symetrické šifře. Pro implementaci byl zvolen zabezpečený mikrokontrolér CEC1302, v rámci práce byla vytvořena nová knihovna pro aritmetiku velkých čísel a také dosud nepublikovaná implementace Paillierova kryptosystému využívající hardwarový RSA akcelerátor.

Framework VeraGreg je v porovnání s naivním schématem 200krát pomalejší a zabírá o třetinu více místa v paměti programu, není tedy vhodnou alternativou k symetrickým kryptosystémům. Na druhou stranu zachovává soukromí uživatele a zároveň umožňuje provádět výpočty se zašifrovanými daty včetně ověření, zda během výpočtu nedošlo k jejich změně.

Kláčová slova VeraGreg, homomorfní šifrování, mikrokontrolér, Paillierův kryptosystém
Homomorphic encryption is an effective way of securing data privacy while maintaining the possibility to process the data. The VeraGreg framework, unlike other existing homomorphic cryptosystem allows for verification of computation that was done with the encrypted data.

This work deals with an implementation of the VeraGreg framework and its effectiveness comparison with a naïve scheme based on symmetric encryption. Secure microcontroller CE1302 was chosen as the implementation platform. A new library for multiprecision integer arithmetic was created as well as the first published implementation of Paillier cryptosystem using hardware RSA accelerator.

The VeraGreg framework is 200 times slower compared to the naive scheme and occupies one third more space in the program memory, so it is not a suitable alternative to symmetric cryptosystems. On the other hand, it provides privacy to the user while allowing computations with the encrypted data, and verifying that is has not been manipulated during the computation.

**Keywords** VeraGreg, homomorphic encryption, microcontroller, Paillier cryptosystem
Contents

Citation of this thesis ........................................ vi

Introduction ............................................. 1

1 State-Of-The-Art ........................................ 3
1.1 Homomorphic Encryption .............................. 3
  1.1.1 Goldwasser–Micali Cryptosystem ................. 4
  1.1.2 Benaloh Cryptosystem ............................ 4
  1.1.3 Unpadded RSA .................................... 5
  1.1.4 Paillier Cryptosystem ............................ 6
1.2 Fully Homomorphic Encryption ......................... 7
1.3 The VeraGreg Framework ............................. 8
  1.3.1 Initialization Algorithm – Init ................... 9
  1.3.2 Identifier Granting Algorithm - Grant .......... 10
  1.3.3 Encryption Algorithm - E ......................... 10
  1.3.4 Addition Algorithm – Add ....................... 10
  1.3.5 Decryption Algorithm – D ....................... 10
1.4 Microcontrollers with Cryptographic Accelerators ... 11
1.5 Side-Channel Attacks and Countermeasures .......... 11
  1.5.1 Timing Attacks .................................. 12
  1.5.2 Power Analysis Attacks ......................... 12
  1.5.3 Countermeasures ................................ 12
    1.5.3.1 Hiding .................................. 12
    1.5.3.2 Masking .................................. 13
1.6 Communication Protocols ............................. 14
  1.6.1 Modbus ........................................ 14
  1.6.2 Abstract Syntax Notation One .................. 15
1.7 Multiprecision Arithmetic ............................ 15
  1.7.1 Algorithms for Multiprecision Integer Arithmetic ... 15
    1.7.1.1 Integer Multiplication ...................... 15

ix
3.2.2.1 Random Number Generation .............................................. 40
3.2.3 Decryption ........................................................................... 40
3.3 VeraGreg scheme ........................................................................ 41
  3.3.1 Init ................................................................................. 41
  3.3.2 Grant .............................................................................. 41
  3.3.3 Encrypt ............................................................................ 42
  3.3.4 Add ................................................................................ 43
  3.3.5 Decrypt ............................................................................ 43
3.4 Communication ........................................................................... 43
3.5 Naive symmetric scheme ................................................................. 43
3.6 PC demo app .............................................................................. 44

4 Testing .......................................................................................... 45
  4.1 Arithmetic library .................................................................... 46
  4.2 Paillier cryptosystem ................................................................. 46
  4.3 Communication ....................................................................... 46
  4.4 System level ............................................................................ 46

5 Experimental Evaluation ................................................................. 47
  5.1 Measured Metrics ................................................................... 47
  5.2 Experiment setup .................................................................... 47
  5.3 Paillier on RSA ....................................................................... 48
  5.4 VeraGreg Framework vs. Naive Scheme ................................. 48
  5.5 Side channel attack countermeasures ..................................... 48
  5.6 Results ................................................................................ 49

6 Future work .................................................................................. 53
  6.1 Side channel attacks ................................................................. 53
  6.2 Usable implementation ............................................................. 53
  6.3 VeraGreg framework ............................................................... 53

Conclusion ....................................................................................... 55

Bibliography ...................................................................................... 57

A Acronyms ...................................................................................... 63

B Running the VeraGreg Demo ............................................................ 65

C Contents of the Enclosed Electronic Media ................................... 67
List of Figures

1.1 MODBUS frame .................................................. 14
2.1 Diagram of the VeraGreg framework implementation .......... 22
2.2 Development board Clicker 2 for CEC1302 ....................... 27
2.3 Paillier decryption data flow ..................................... 30
3.1 Number representation in bigi library .......................... 34
1.1 Test environment structure ...................................... 45
5.1 Capacitor attachment to the board ............................... 50
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Secure microcontrollers comparison</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>VeraGreg secret key knowledge</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of execution times of Paillier cryptosystem</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of memory requirements of Paillier cryptosystem</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Comparison of execution times of the naïve scheme and VeraGreg</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of memory requirements of the naïve scheme and VeraGreg framework</td>
<td>49</td>
</tr>
</tbody>
</table>
Introduction

In the age of massive data collection, and growing capabilities of systems for data processing, privacy protection becomes essential for both individuals and companies. Homomorphic encryption is an effective tool which can be employed to secure data privacy while keeping the possibility of processing encrypted data by third-party providers.

This work describes, to the best of my knowledge, the first implementation of the VeraGreg framework. The novelty of VeraGreg framework lies in the fact that unlike other existing homomorphic encryption schemes, VeraGreg can verify calculations that were performed with encrypted data.

The first chapter recalls homomorphic encryption, VeraGreg framework and libraries and algorithms for multiprecision integer arithmetic. Furthermore, it describes communication protocols suitable for the Internet of Things (IoT), secure microcontrollers and side channel attacks.

The second chapter deals with the selection of suitable algorithms and platform for implementation of the VeraGreg framework. Chapter three contains a description of an implementation of the VeraGreg framework, including a novel library for multiprecision integer arithmetic and a naïve framework based on symmetric encryption. Chapter four briefly describes testing of all implemented libraries and also system-level testing. The last chapter deals with a comparison of the VeraGreg framework and a naïve solution, as well as with the comparison of different implementations of Paillier cryptosystem.
This section gives an introduction and the state of current research. First, I describe homomorphic encryption and its classification, then describe the VeraGreg framework and available microcontrollers with cryptographic accelerators. Next, I deal with communication protocols suitable for the IoT and multiprecision arithmetic.

Homomorphic encryption, especially fully homomorphic, is currently at the forefront, mostly in relation to privacy issues.

1.1 Homomorphic Encryption

Homomorphic encryption is a type of encryption that allows anyone with the public key to perform particular arithmetic operations over encrypted data in such a way that the result of an operation over encrypted data decrypts into a result of another operation over unencrypted data.

Homomorphic cryptosystems can be used for various purposes in the field of electronic electoral systems [1] that guarantee anonymity. The following definition mathematical description of homomorphic encryption [2]:

Definition 1 ([2]) Let \((P; C; K; E; D)\) be an encryption scheme, where \(P; C\) are the plaintext and ciphertext spaces, \(K\) is the key space, and \(E; D\) are the encryption and decryption algorithms. Assume that the plaintexts form a group \((P; \cdot)\) and the ciphertexts forms a group \((C; \circ)\), then the encryption algorithm \(E\) is a map from the group \(P\) to the group \(C\), i.e., \(E_k : P \rightarrow C\), where \(k \in K\) is either a secret key (in a secret key cryptosystem) or a public key (in a public-key cryptosystem). For all \(a\) and \(b\) in \(P\) and \(k\) in \(K\), if

\[ E_k(a) \circ E_k(b) = E_k(a \cdot b) \]  

(1.1)

the encryption scheme is homomorphic.
A definition similar to the Definition 1 is given also in [1]. Cryptosystems meeting this definition are also referred to as partially homomorphic encryption (PHE).

1.1.1 Goldwasser–Micali Cryptosystem

Goldwasser–Micali cryptosystem is the first published homomorphic probabilistic cryptosystem [3]. It is based on the quadratic residuosity problem, which asks, given integers \(a\) and \(n\), whether there exists such integer \(b\), for which holds the following equation:

\[ b^2 \equiv a \mod n. \]

Key generation. The public key consists of \(n = pq\) where \(p, q\) are large primes, and \(y\) such that for Legendre symbols it holds \((\frac{y}{p}) = -1\) and \((\frac{y}{q}) = -1\). Large primes \(p\) and \(q\) form the private key.

Encryption. Plaintext \(m\) is converted into sequence of bits \((m_0, m_1, \ldots, m_j)\). Ciphertext \(c\) is a sequence of integers \((c_0, c_1, \ldots, c_j)\), such that \(\forall i \in \{0, \ldots, j\} : c_i = y_i^2 \cdot x^m_i \mod n\) and \(y_i\) is random integer from \(\mathbb{Z}_n^*\).

Decryption. For each \(c_i = (c_0, c_1, \ldots, c_j)\) it is determined, whether it is a quadratic residue. If \(c_i\) is a quadratic residue, then \(m_i = 0\), otherwise \(m_i = 1\).

As can be seen from the description of encryption, each bit is encrypted with a number from \(\mathbb{Z}_n^*\), which must be several thousands bits long to ensure security. This disproportionate increase in the size of the ciphertext is the reason why this cryptosystem is not used in practice and serves more like a proof-of-concept.

Homomorphic property. If bits \(m_1\) and \(m_0\) are encrypted into ciphertexts \(c_0\) and \(c_1\), respectively, then the product \(c_1 \cdot c_0\) is equal to \(m_0 \oplus m_1\), where \(\oplus\) stands for exclusive or operation.

1.1.2 Benaloh Cryptosystem

The Benaloh cryptosystem [4] is an extension of the Goldwasser–Michali cryptosystem [2]. The main advantage of this cryptosystem is a reduction of extreme size of ciphertext compared to plaintext. Again, it is a probabilistic asymmetric cryptosystem.

Key generation. Let \(r\) be an integer representing block length, all plaintexts \(m \in \mathbb{Z}_r\). Let \(n = pq\), where \(p\) is a prime for which it holds \(r | (p - 1)\); \(q\) is also a prime, for which it holds \(gcd(q - 1, r) = 1\). Further, \(y \in \mathbb{Z}_n^*\) such that \(y^{(p-1)(q-1)/r} \mod n \neq 1\) is chosen. The public key is a pair \((n, y)\), private key consists of the primes \((p, q)\).
1.1. Homomorphic Encryption

Encryption. Ciphertext \( c = y^m \cdot u^r \mod n \), where \( u \) is a random integer from \( \mathbb{Z}_n^* \) and \( m \in \mathbb{Z}_r \).

Decryption. Calculate \( m_i = (y^{-i} \cdot c)^{(p-1)(q-1)} \mod n \) for all \( i \in \mathbb{Z}_r \) until \( m_i = 1 \), then \( m = i \).

This decryption algorithm can only be used for small block sizes. It is possible to precompute results for bigger blocks, or combine both approaches (baby step–giant step algorithm). This algorithm has time and memory complexity of \( O(\sqrt{r}) \).

The decryption process is computationally demanding, as implied in the description.

Homomorphic property. The homomorphic property of the Benaloh cryptosystem allows addition and subtraction of ciphertexts according to following equations:

\[
c_1 \cdot c_2 \mod n = (y^{m_1} \cdot u_1^r)(y^{m_2} \cdot u_2^r) \mod n = y^{m_1+m_2} \cdot (u_1 \cdot u_2^r) \mod n \quad (1.2)
\]

\[
\implies Dec(Enc(m_1) \cdot Enc(m_2) \mod n) = Dec(Enc(m_1 + m_2) \mod n) \quad (1.3)
\]

1.1.3 Unpadded RSA

The simplest example of homomorphic cipher is one of the most famous asymmetric ciphers—RSA \([5]\) in its basic form (without padding). RSA security is based on the problem of integer factorization.

Key generation. The basis of RSA cryptosystem is a parameter \( n = p \cdot q \), where \( p \) and \( q \) are large primes. RSA public key is the pair \((e,n)\), where \( e \) satisfies: \( 1 < e < \lambda(n) \) and \( \text{gcd}(e,\lambda(n)) = 1 \). RAS private key is a pair \((d,n)\), where \( d \equiv e^{-1} \mod \lambda(n) \).

Encryption. The encryption is done by modular exponentiation. Message \( m \) is encrypted as \( c \) according to the following equation: \( c \equiv m^e \mod n \).

Decryption. Message \( m \) is obtained from a ciphertext \( c \) according to the following equation: \( m \equiv c^d \mod n \).

Homomorphic property. The above described cryptosystem satisfies definition \([1]\). The plaintext group is the multiplicative group \( \mathbb{Z}_n^* \), the same holds for ciphertexts. Furthermore, it holds that a product of two ciphertexts is decrypted into product of corresponding plaintexts, indeed \( E_{pk}(m_1) \cdot E_{pk}(m_2) = m_1^e \cdot m_2^e \mod n = (m_1 \cdot m_2)^e \mod n = E_{pk}(m_1 \cdot m_2) \).

Considering the property mentioned above, unpadded RSA is homomorphic encryption. Unfortunately, RSA without padding is not semantically secure (IND–CPA) \([2]\).
1. State-Of-The-Art

1.1.4 Paillier Cryptosystem

In 1999 Paillier published a design of an asymmetric cryptosystem based on composite residuosity class problem [6]. This problem has computational complexity comparable to RSA [6].

In the original paper [6], the author suggested three variants of the cryptosystem, currently the designation Paillier cryptosystem is used for the probabilistic "Scheme 1". This cryptosystem is probabilistic and IND-CCA1 compliant [7].

Key generation. \( n = p \cdot q \) where \( p, q \) are primes, such that \( \gcd(p \cdot q, \phi(p \cdot q)) = 1 \) \( g \in \mathbb{Z}_{n^2}^* \) such that order of element \( g \) in group \( \mathbb{Z}_{n^2}^* \) is nonzero multiple of \( n \). Further, \( \lambda = \text{lcm}(p-1, q-1) \). Public key is pair \((n, g)\), private key is the parameter \( \lambda \).

Encryption. Given plaintext \( m < n \) and a random integer \( r \) from \( \mathbb{Z}_{n^2}^* \) such that \( r < n \), the ciphertext \( c \) is computed according to the following equation

\[
c = g^m \cdot \mod n^2.
\] (1.4)

Decryption. Ciphertext \( c \) satisfies \( c < n^2 \). Plaintext \( m \) is computed as follows: \( m = \frac{L(c^\lambda \mod n^2)}{L(g^\lambda \mod n^2)} \mod n \), where \( L(x) = \frac{x - 1}{n} \).

Homomorphic property. Homomorphic properties of Paillier cryptosystem are following:

\[
D(E(m_1) \cdot E(m_2) \mod n^2) = m_1 + m_2 \mod n,
\] (1.5)
\[
D(E(m)^k \mod n^2) = km \mod n,
\] (1.6)
\[
D(E(m_1)^{m_2} \mod n^2) = m_1 \cdot m_2 \mod n,
\] (1.7)

where \( \forall m \in \mathbb{Z}_n, k \in \mathbb{N} \).

The use of random value during encryption provides the Paillier cryptosystem with a self–blinding property, which means that the ciphertext can be changed to another only with the knowledge of the public key and without altering the decryption result. If "Scheme 1" is used for encryption, the property is:

\[
\forall m \in \mathbb{Z}_n \ a \ r \in \mathbb{N}:
\]
\[
D(E(m) r^n) \mod n^2 = m
\] (1.8)
1.2. Fully Homomorphic Encryption

**Damgård–Jurik cryptosystem** In 2001 Damgård and Jurik published a generalization of Paillier cryptosystem [8], called Damgård–Jurik cryptosystem. In this cryptosystem, encryption and decryption operations are performed $\mod n^{s+1}$, where $n$ is an RSA modulus (a product of two large primes), and $s$ is an integer greater or equal to 1. This choice allows encryption of larger plaintexts, a plaintext $m$ must satisfy $m < n^s$. The main disadvantage of this cryptosystem is decreased security if the size of modulus is fixed, e.g., when using a hardware accelerator.

1.2 Fully Homomorphic Encryption

For a cryptosystem to be considered fully homomorphic, its ciphertexts and plaintexts must form at least a ring. A ring is an algebraic structure that allows to perform two different arithmetic operations over its elements. Therefore it is possible to implement an arbitrary function over encrypted data, e.g., exponentiation, division.

Considering the property mentioned above, it is possible to use fully homomorphic encryption for securing cloud computations; applications are emerging even for neural networks [9], which allow classification of encrypted data without the knowledge of the private key.

Yi et al. define the homomorphic encryption in the following way [2]:

**Definition 2** ([2]) Let $(P; C; K; E; D)$ be an encryption scheme, where $P; C$ are the plaintext and ciphertext spaces, $K$ is the key space, and $E; D$ are the encryption and decryption algorithms. Assume that the plaintexts form a ring $(P, \oplus_P, \otimes_P)$ and the ciphertexts form a ring $(C, \oplus_C, \otimes_C)$; then the encryption algorithm $E$ is a map from the ring $P$ to the ring $C$, $E_k : P \to C$, where $k \in K$ is either a secret key (in the secret key cryptosystem) or a public key (in the public-key cryptosystem). For all $a, b \in P$ and $k \in K$, if

$$E_k(a) \oplus_C E_k(b) = E_k(a \oplus_P b), \quad (1.9)$$

$$E_k(a) \otimes_C E_k(b) = E_k(a \otimes_P b), \quad (1.10)$$

the encryption scheme is fully homomorphic.

A cryptosystem with such properties was first mentioned in 1978 by Rivest [10], however, the question of existence of such cryptosystem remained unanswered for a long time. In 2009, Craig Gentry published the first proposal of a fully homomorphic cryptosystem [11].

Gentry’s cryptosystem is based on a somewhat homomorphic cryptosystem which is able to perform only limited number of ciphertext multiplications, because each ciphertext includes a certain level of noise. The level of noise increases with every multiplication, up to the point where the ciphertext cannot be decrypted to the original value.
1. State-Of-The-Art

Gentry came up with an idea of so called bootstrapping operation. This operation lowers the noise level in a noisy ciphertext, thus allows to perform unlimited number of arithmetic operations with ciphertexts.

Gentry’s cryptosystem is computationally demanding and due to this fact, it is not used in practice. However, other authors followed his ideas and created fully homomorphic cryptosystems over integers [12], BFV cryptosystem [13] based on ring learning with errors or BGV [14] cryptosystem without the need of bootstrapping.

The aforementioned cryptosystems are computationally less demanding than Gentry’s cryptosystem and several of them were implemented in libraries for fully homomorphic encryption. HElib library [15] implements BGV cryptosystem [14], Microsoft SEAL implements BFV [13] and CKKS [16] cryptosystems.

In 2017, the Homomorphic Encryption Standardization Initiative [17] was launched to standardize the implementations of fully homomorphic cryptosystems.

1.3 The VeraGreg Framework

The VeraGreg framework is a novel cryptosystem with homomorphic properties, which allows verification of the arithmetic operations performed with the ciphertexts.

All homomorphic cryptosystems are inherently malleable, they allow anyone with the public key to change the ciphertext in such a way that this ciphertext decrypts to different plaintext [18]. This feature can be undesirable in some applications. One possible way of removing this feature is verification of operations that were done with the ciphertext.

VeraGreg is defined as framework, consisting of five algorithms (Init, Grant, E, Add, D). Formal definition according to [18] follows:

**Definition 3 ([18])** Let $D$ denote an additive Abelian group—the data space, $\lambda \in \mathbb{N}$ the security parameter, $B$ the set of ID’s, $C$ the ciphertext space, $K_P$, $K_S$ the public and secret key space, respectively. VeraGreg Framework is a 5-tuple of probabilistic polynomial time algorithms (Init, Grant, E, Add, D),

- **Init**: $\{1\}^* \rightarrow K_P \times K_S$,
- **Grant**: $\{1\}^* \times B^* \times D \rightarrow B^* \times B \cup \{\perp\}$,
- **E**: $K_S \times B \times K_D \rightarrow C$,
- **Add**: $K_P \times \mathbb{Z}^{\lfloor B\rfloor} \times C^* \rightarrow C$,
- **D**: $K_S \times \mathbb{Z}^{\lfloor B\rfloor} \times C \rightarrow D \cup \{\perp\}$,
for which it holds: \( \forall n \in \mathbb{N}, \forall (d_i)_{i=1}^n \in D^n \) with corresponding valid \((b_i)_{i=1}^n\) granted by \(\text{Grant}_\lambda\), \(\forall B \in \mathbb{Z}^{|B|}, B[b_i] = n_i, B[b] = 0 \) for \(b \notin \{b_i\}_{i=1}^n\), and a keypair \((pk, sk) \leftarrow \text{Init}_\lambda\),

1. if \(B\) is policy-compliant,

\[
\operatorname{Pr}\left[Dsk\left(B, \text{Add}_{pk}(B, E_{sk}(b_i, d_i)_{i=1}^n)\right) = \sum_{i=1}^n n_i \cdot d_i \right] \in \text{OW}_\lambda,
\]

i.e., the encryption is additively homomorphic,

2. if \(B\) not policy-compliant,

\[
\operatorname{Pr}\left[Dsk\left(B, \text{Add}_{pk}(B, E_{sk}(b_i, d_i)_{i=1}^n)\right) = \bot \right] \in \text{OW}_\lambda,
\]

(1.11)
i.e., the policy-incompliant list is discarded,

3. \(\forall B' \in \mathbb{Z}^{|B'|}, B' \neq B\)

\[
\operatorname{Pr}\left[Dsk\left(B', \text{Add}_{pk}(B', E_{sk}(b_i, d_i)_{i=1}^n)\right) = \bot \right] \in \text{OW}_\lambda,
\]

(1.12)
i.e., the framework detects any list forgery,

4. otherwise

\[
\operatorname{Pr}[Dsk(\cdot, \cdot) = \bot] \in \text{OW}_\lambda,
\]

(1.13)
i.e., any invalid ciphertext is detected.

In [18], authors propose concrete algorithms for \(\text{Init}, \text{Grant}, E, \text{Add}\) and \(D\) forming an instance of the VeraGreg framework.

### 1.3.1 Initialization Algorithm – Init

Given a security parameter \(\lambda\), this algorithm creates keys for underlying cryptographic primitives according to \(\lambda\) and computes cryptosystem parameters. Plaintext must be at least \(\delta\) bits long and it must be possible to add \(2^\nu\) ciphertexts before overflow.

The algorithm sets \(\mu_1 = \max(\lambda, \delta + \nu)\) and \(\mu_2 = \lambda\). Then initializes the homomorphic cryptosystem (AHE) in such way, that the cryptosystem is able to encrypt data \(\nu + \lambda + \mu_1 + \mu_2\) bits long. In the next step, Init algorithm initializes the symmetric encryption algorithm (SE) so that it is able to encrypt \(\lambda\) bits long data.

Finally, this algorithm generates two random integers, \(m_1\) and \(m_2\), that are \(\mu_1\) and \(\mu_2\) bits long.

The public key is the public key of the AHE. The private key consists of the AHE private key, the SE key and integers \(m_1\) and \(m_2\).
1. **State-Of-The-Art**

1.3.2 **Identifier Granting Algorithm - Grant**

This algorithm first checks whether the input data is in compliance with a policy $\mathcal{P}$. According to proposition in [18], this could be length of the data, i.e., whether the data is $\delta$ bits long. For valid data, Grant algorithm returns a unique identifier, otherwise it returns $\perp$.

1.3.3 **Encryption Algorithm - $E$**

Encryption algorithm inputs the data $d$ and its identifier $b$ (the output of the Grant algorithm). With the secret parameters $m_1$ and $m_2$ computes $p$, according to

$$p = (SE(b) \cdot m_1 + d) \cdot m_2. \quad (1.14)$$

The intermediate value $p$ is then encrypted with the AHE. Thus, the whole encryption algorithm can be described by Equation (1.15), where $c$ is the output of algorithm $E$,

$$c = E_{sk}(d, b) = AHE((SE(b) \cdot m_1 + d) \cdot m_2). \quad (1.15)$$

1.3.4 **Addition Algorithm – Add**

Given a list of identifiers $\mathcal{B}$, for which it holds $\mathcal{B}[b_i] = n_i, \mathcal{B}[b] = 0$ for $b \notin \{b_i\}_{i=1}^n$ and $\sum_{i=1}^n n_i \leq 2^\nu$ and corresponding set of ciphertexts, Add algorithm creates an aggregate according to

$$Add_{pk}(\mathcal{B}, (c_i)_{i=1}^n) = \bigoplus_{i=1}^n n_i \cdot c_i = \bigoplus_{i=1}^n n_i \cdot AHE(p_i). \quad (1.16)$$

The symbol $\bigoplus$ in the equation (1.16) represents addition of ciphertexts in the selected homomorphic cryptosystem.

Furthermore, the Add algorithm aims to preserve the following property: $\sum n_i \cdot d_i < m_1$, where $n_i$ is the number of occurrences of each ciphertext in the list of ciphertexts.

1.3.5 **Decryption Algorithm – D**

The decryption algorithm first checks whether the list of identifiers $\mathcal{B}$ complies with the policy $\mathcal{P}$. If not, it outputs $\perp$. Otherwise the algorithm deciphers $c$ with the AHE’s private key into an intermediate value $\tilde{p}$. If $\tilde{p} \mod m_2 \neq 0$, the output is $\perp$. As the next step, $\tilde{b}_{SE}$ is computed according to Equation (1.17) and D checks whether Equation (1.18) holds,

$$\tilde{b}_{SE} = \tilde{p}/(m_1 m_2), \quad (1.17)$$
1.4. Microcontrollers with Cryptographic Accelerators

\[ \tilde{b}_{SE} = \sum_{i=1}^{n} n_i \cdot SE(b_i), \]  \hspace{1cm} (1.18)

while it outputs ⊥ if Equation (1.18) does not hold. Otherwise the data in
the aggregate corresponds with identifiers in the provided list \( B \) and the final
step of decryption is computed according to

\[ d = (\tilde{p} \div m_2) \mod m_1. \]  \hspace{1cm} (1.19)

1.4 Microcontrollers with Cryptographic Accelerators

Microcontrollers with accelerators of cryptographic operations are currently
used in many applications, ranging from set-top-boxes, through the Internet
of Things (IoT) devices to payment terminals. Except for accelerators, these
microcontrollers include other features improving their security. These are
usually side-channel attacks countermeasures, reverse engineering defenses and
secured non-volatile memories. All big manufacturers of microcontrollers have
secure microcontrollers in their product portfolio. STMicroelectronics N.V.
has ST31 and ST33 product lines [19], Microchip Technology Inc. manufactures
CEC microcontrollers [20], Texas Instruments Inc. has MSP430 and
C2000 lines [21] and Maxim Integrated Inc. offers DeepCover [22].

However, it is difficult to access the documentation, since some manu-
facturers only give access to commercial subjects after signing a non-disclosure
agreement (NDA). Another problem is the availability of development kits
with these microcontrollers, their cost, and availability of development envi-
rónments and compilers.

1.5 Side-Channel Attacks and Countermeasures

Side channel attack is a cryptographic algorithm attack that does not use
mathematical vulnerabilities, but vulnerabilities arising from a particular al-
gorithm implementation. In [23] Paar defines a side channel attack as a passive
attack. This means that during the attack the device is not manipulated and
only its standard interface is used.

These attacks focus on side channels that are most often the power con-
sumption during the execution of the cryptographic algorithm or the execution
time.

Countermeasures against these attacks depend on particular implementation
and device. Some types of protection are implemented on hardware level, others on software level.
1. State-Of-The-Art

1.5.1 Timing Attacks

Timing attacks exploit the fact that the execution time of an operation depends on its input data.

According to [23], timing attacks are often used against cryptographic algorithms that perform modular exponentiation with the secret, such as RSA decryption or message signing. The assumption, as for all side channel attacks, is that the attacker has access to the input and output of the algorithm, and is able to measure the execution time.

To be able to mount the attack, the attacker must identify the exact moment of the operation depending on the secret key. This is done by measuring the power consumption and identifying power peaks that might correspond for example with modular exponentiation.

1.5.2 Power Analysis Attacks

The first published power analysis attack is the simple power analysis [24]. This type of side channel attack uses the dependency of the processed data and the power consumption of the device.

The basis of power analysis attack is a measurement of the device power consumption during the execution of the cryptographic algorithm and the knowledge of open and encrypted text. On the basis of known ciphertext or plaintext, the power consumption hypothesis is created for each part of the key. Then, the correct power consumption hypothesis, and thus the key, is selected by correlation with the measured power traces.

This variant is also called the correlation power analysis (CPA) and was published in 2004 [25]. This kind of attack was used for example to break the KeeLoq cryptosystem [26].

1.5.3 Countermeasures

Countermeasures against side channel attacks consist of masking or hiding the information present in these channels. This section describes the protection against power analysis attacks.

Side channel protection can be implemented at hardware or software level. Since this work deals with an implementation on a microcontroller, only software protection methods are described below.

1.5.3.1 Hiding

The aim of hiding is to hide the information in the side channels, that is, to make sure that the consumption of the device does not depend on the data being processed (intermediate results). Two approaches are used to achieve this goal.
1.5. Side-Channel Attacks and Countermeasures

The first approach to hiding is to randomize the consumption of the device in each clock cycle, the second approach is to unify consumption in each clock cycle so that it does not change during the execution of the cryptographic algorithm.

Achieving truly random or constant consumption is practically impossible, hence consumption-masking approaches are used directly on the power consumption or alter its course over time.

The DPA attack requires power consumption traces measured during encryption to be aligned with each other. If this condition is not met, the attack becomes more complex. This enables to use randomized consumption over time as a countermeasure against DPA. This is typically achieved by inserting dummy rounds, that is, empty operations in a random number of cryptographic algorithm downs. The disadvantage of this countermeasure is the increase of execution time.

Another way to hide information about intermediate results is to change the order of operations randomly; the advantage is that they do not affect the execution time. However, this method cannot be applied to all cryptographic algorithms and their parts.

Furthermore, it is possible to hide the consumption of the device. The total device consumption can be described as

\[ P_{\text{device}} = P_{\text{stat}} + P_{\text{noise}} + P_{\text{data}}. \] (1.20)

The \( P_{\text{data}} \) component is important for the attack, but the attacker only has access to \( P_{\text{device}} \). Therefore, if the \( P_{\text{noise}} \) (random switching activity) component is sufficiently large and random, it will prevent recognizing the \( P_{\text{device}} \) component in the overall power consumption \( P_{\text{data}} \). The \( P_{\text{stat}} \) component represents the static power consumption of the device.

The component \( P_{\text{noise}} \) can be increased by turning on peripheral circuits (AD converter, communication interface, timers, ...), which will, on the other hand increase the total microcontroller consumption.

1.5.3.2 Masking

Unlike hiding which tries to hide the information present in the side channel, masking tries to change this information. This is accomplished by changing intermediate values.

When using masking, each intermediate value \( i \) is masked by a randomly generated mask \( m \) so that \( i_m = i \times m \). The mask \( m \) is generated repeatedly for each run of the algorithm, so it is impossible for the attacker to recognize it. The \( \times \) operation is defined by the operations used in the cryptographic algorithm. For example, it can be XOR, modular multiplication, or modular addition.
Two ways of securing RSA against power analysis attack are described in \cite{27}. The first one is message blinding, when the message $m$ is masked by $v$ so that $m_v = m \cdot v \mod n$. This value is then decrypted to $v^d \cdot m \mod n$.

The second way how to use masking for RSA protection is exponent blinding \cite{28}. The private exponent $d$ is masked as follows. $d_m = d + m \cdot \phi(n) \mod n$ where $m$ is a random mask. The result is unmasked automatically because $v^{d_m} \mod n = v^d \mod n$.

1.6 Communication Protocols

Due to the anticipated focus of the VeraGreg framework on applications in IoT \cite{18}, both Modbus and own Abstract Syntax Notation One (ASN.1) based protocol are considered. Modbus is used in industry and home automation, ASN.1 is used in telecommunication (fourth generation mobile networks \cite{29}) and cryptographic applications (SSL certificates). This section describes both of these variants in detail.

1.6.1 Modbus

Modbus is a serial communication protocol developed by Modicon in 1979 for industrial programmable logic controllers. It has become a standard for communication between industrial devices a is also used for smart home devices. The standard is managed by The Modbus Organization, which consists of manufacturers of industrial automation devices \cite{30}.

The standard \cite{30} specifies application layer for client-server type of communication on different types of networks or buses, for example RS232, Ethernet, RS485, wireless networks or optical networks.

![MODBUS frame.](image)

Figure 1.1: MODBUS frame.

Figure 1.1 depicts a diagram of a Modbus frame, which is the basic unit for communication. The frame consists of a target device address, a function code, data and a checksum. The whole frame is called the Application Data Unit (ADU), the part of the frame with function code and data is called the Protocol Data Unit (PDU).

Functions and their codes are defined in the MODBUS Application Protocol Specification \cite{30}. Basic functions are Read Discrete Inputs, Read Coils, Write Single Coil, Write Multiple Coils.
1.6.2 Abstract Syntax Notation One

Abstract Syntax Notation One (ASN.1) is not a protocol. It is a framework for designing and creating protocols from serializable structures.

Due to the serialization of data structures, it is possible to use these structures in serial communication.

The standard defined data encoding supports many formats, among them XML, Json, BER, DER and CER, which are multiplatform. The advantage of ASN.1 is that the data structures are human readable.

Despite being published in 1984, the ASN.1 standard is becoming widespread lately, mainly due to spread of the https protocol and usage of ASN.1 based protocols in fourth generation mobile networks.

1.7 Multiprecision Arithmetic

Most cryptographic algorithms or scientific computations require operations with numbers that are larger than the machine word of the implementation platform. This type of arithmetic operations is known as multiprecision, or arbitrary precision arithmetic.

1.7.1 Algorithms for Multiprecision Integer Arithmetic

Addition, subtraction, bit shifts, and logical operations can be easily implemented using arithmetic operations that use machine word sized operands. Multiplication, division, and modular arithmetic operations are more computationally demanding and there are several ways of implementing these operations.

1.7.1.1 Integer Multiplication

When multiplying two numbers, \(m\) and \(n\) bits long, the result may have a length up to \(m + n\) bits.

Knuth describes the basic algorithm for multiplication in [31], implementation of this algorithm in C is given in [32]. This algorithm has complexity \(O(n^2)\).

In 1975, Karatsuba published the first algorithm with complexity \(O(n^\log_2 3)\) [33]. This algorithm is recursive and suitable for operands that are hundreds of bits long. Detailed description is given by [32].

Toom–Cook algorithm is a generalization of Karatsuba’s algorithm, its basic idea is to divide operands \(a\) and \(b\) into smaller numbers, each of them \(l\) bits long. These numbers are then recursively processed. Detailed description, including the analysis of complexity is given in [31].

Schönhage–Strassen [34] is the fastest known algorithm for multiplication of large integers. It has complexity \(O(n \cdot \log n \cdot \log \log n)\) and it is used for...
large numbers, e.g., in the GMP library [35] is used for numbers longer than 1700 machine words. The basic idea is converting numbers to polynomials and performing recursive fast Fourier transform and linear convolution.

1.7.1.2 Integer Division

Division is the most computationally demanding integer arithmetic operation. The output of integer division is floor of a quotient and a remainder.

The basic algorithm for division of multiprecision integers is ”Algorithm D” described in [31]. According to the author, this algorithm is a variation of schoolbook division.

Another option when computing quotient $A/B$ is to compute $\frac{1}{B}$ first and then multiply $A \cdot \frac{1}{B}$. This way, division can be replaced by multiplication which can be done by algorithm from the previous Section 1.7.1.1.

A comprehensive overview of multiprecision integer division algorithms is given by [36].

1.7.1.3 Modular Reduction

Basic operation in modular arithmetic is reduction. Given integer $x$, it returns remainder $b$ after division by modulus $n$, according to

\[ x = a \cdot n + b \equiv b \mod n. \quad (1.21) \]

The na"ive way of implementing modular reduction is division, which outputs the quotient and the remainder. However, computational complexity of the multiprecision integer division makes this approach ineffective.

Barret’s reduction algorithm [37] is faster than division, it uses only multiplication, addition, subtraction and left bit shift. Its detailed description is given in [38] as the algorithm 14.42, its transcription is shown below as Algorithm 1.

16
Algorithm 1 Barrett’s Modular Reduction

**INPUT:** $A, N$ (each max $n$ words of $k$ bits)

**OUTPUT:** $A \mod N$

1: $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor$
2: $q_2 \leftarrow q_1 \cdot \mu$
3: $q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$
4: $r_1 \leftarrow x \mod b^{k+1}$
5: $r_2 \leftarrow q_3 \cdot m \mod b^{k+1}$
6: $r \leftarrow r_1 - r_2$
7: if $r < 0$ then
8: $r \leftarrow r + b^{k+1}$
9: while $r \geq m$ do
10: $r \leftarrow r - m$
11: return $g \cdot b$

As it can be seen from Algorithm 1, it is the constant $\mu$ which is precomputed according as

$$\mu = \lfloor b^{2k}/m \rfloor,$$

(1.22)

Where $m$ is the modulus, $b$ is the radix and $k$ is the length of the modulus $m$ in bits. This precomputation requires division, however the constant $\mu$ can be used until the modulus changes.

1.7.1.4 Modular Multiplication

There are several ways of implementing modular multiplication. The na"ïve approach is to follow classical multiplication by modular reduction, i.e., in the first step, standard multiprecision multiplication is performed, and in the second step, the intermediate value is reduced by the modulus.

Another way of implementing modular multiplication is using the double-and-add algorithm, which only employs left shift and addition [39]. After each shift or addition, a trial division (subtraction of modulus) is performed. This approach is less memory demanding than the previous one.

In 1985, Peter Montgomery published a method for fast modular multiplication of multiprecision integers [40]. The main idea behind Montgomery’s approach is to convert numbers into a different representation (aka. the Montgomery domain) which eliminates the need for reduction by the modulus $n$ and trial division, and also decreases memory requirements of modular multiplication as there is no need to store the intermediate result.

**Montgomery Domain Arithmetic**

Conversion to the Montgomery domain is performed according to $\tilde{A} \equiv A \cdot R \mod N$ where $\tilde{A} \in \mathbb{Z}_n$ represents integer $A \in \mathbb{Z}_N$ in Montgomery domain. For
odd modulus $N$ holds that $w^{n-1} \leq N < w^n$ where $w = 2^k$, $k$ is the length of the machine word and $\gcd(N, r) = 1$. $R$ is an integer, for which holds $R = w^n$.

Multiplication in the Montgomery domain is defined as follows

$$ \tilde{A} \odot \tilde{B} \mod N \equiv \tilde{A}\tilde{B}R^{-1} \mod N. \quad (1.23) $$

Multiplication in the Montgomery domain can be also used for number conversion from and to Montgomery domain. For conversion to Montgomery domain, one operand is the number being converted, the second operand is $R^2$, then the result of Montgomery multiplication is $\tilde{A}$, according to

$$ A \odot R^2 \mod N \equiv \tilde{A} \equiv A \cdot R \mod N. \quad (1.24) $$

Conversion from Montgomery domain can be done as follows

$$ \tilde{A} \odot 1 \mod N \equiv A \cdot R \cdot R^{-1} \equiv A \mod N. \quad (1.25) $$

For multiprecision integer Montgomery multiplication gives [41] the following algorithm 2:

**Algorithm 2** Modular Multiplication in the Montgomery Domain

**INPUT:** $A, B, N$ (each max $n$ word of $k$ bits, $M'_0$ for which it holds $-N \cdot M'_0 \equiv 1 \mod 2^k$)

**OUTPUT:** $ABR^{-1} \mod N$

1: $C \leftarrow 0$
2: for $i = 0; i < k; i + +$ do
3: \quad $C \leftarrow C + a_i \cdot b$
4: \quad $q \leftarrow M'_0 \cdot C \mod 2^k$
5: \quad $C \leftarrow (C + Nq)/2^k$
6: if $C \geq N$ then
7: \quad return $C - N$
8: else
9: \quad return $C$

Modular reduction in the step 4 of Algorithm 2 can be done by reading the last machine word in number representation. Division in the step 5 can be implemented as left bit shift, or better, as array shift by one element.

Because of the necessity of precomputation of parameter $M'_0$, Algorithm 2 is suitable for computations where more multiplications are done with the same modulus, typically modular exponentiation.

1.7.1.5 Exponentiation

(Modular) exponentiation is the core operation of many asymmetric cryptosystems such as Diffie–Hellman [42], RSA [5], Paillier cryptosystem [6], Goldwasser–Micali cryptosystem [9], etc.
As a basic approach, Menezes et al. [38] describe right-to-left binary exponentiation, see Algorithm 3.

**Algorithm 3** Right-to-left exponentiation

**INPUT:** $g, e$ where $g$ is an element of a group and $e \geq 1$

**OUTPUT:** $g^e$

1: $A \leftarrow 1$
2: $S \leftarrow g$
3: **while** $e \neq 0$ **do**
4:  **if** $e$ is odd **then**
5:  $A \leftarrow A \cdot S$
6:  $e \leftarrow \lfloor e/2 \rfloor$
7:  **if** $e \neq 0$ **then**
8:  $S \leftarrow S \cdot S$
9: **return** $A$

Division in step 6 in Algorithm 3 can be implemented by a left shift. Multiplication in steps 5 and 8 can be implemented by one of algorithms mentioned in Section 1.7.1.1.

In case of modular exponentiation, multiplication in steps 5 and 8 is done by modular multiplication. If modular multiplication is done in Montgomery domain, variable $A$ in step 1 must be initialized to $R$ and input $g$ must be in Montgomery domain.

Menezes et al. [38] also describe a variant of the above-mentioned algorithm, left-to-right binary exponentiation algorithm. The difference is in the way the main cycle goes through the exponent.

Montgomery powering ladder [43] is a variant of the left to right exponentiation algorithm which provides protection against timing attacks and simple power analysis. This protection results from the fact that both square operation and multiply operation are executed in every step of the algorithm, not depending on the bit in the exponent. Hence the execution time of Montgomery powering ladder is independent of the Hamming weight of the exponent.

All variants of (modular) multiplication can be used for multiplication operations in the Algorithm 4.
1. State-Of-The-Art

Algorithm 4 Montgomery powering ladder

**INPUT:** \(e, g\) where \(g\) is an element of a group and \(e = (e_{n-1}, ..., e_0) \geq 1\)

**OUTPUT:** \(g^e\)

1. \(R_0 \leftarrow 1\)
2. \(R_1 \leftarrow g\)
3. for \(i = n - 1\) downto 0 do
4. \hspace{1em} if \(e[i] = 1\) then
5. \hspace{2em} \(R_0 \leftarrow R_0 \cdot R_1\)
6. \hspace{2em} \(R_1 \leftarrow R_1 \cdot R_1\)
7. \hspace{1em} else
8. \hspace{2em} \(R_1 \leftarrow R_0 \cdot R_1\)
9. \hspace{2em} \(R_0 \leftarrow R_0 \cdot R_0\)
10. return \(R_0\)

1.7.2 Libraries

There are various implementations of multiprecision arithmetic in many programming languages. They can be divided into two groups based on the implemented operations and the intended use.

The first group consist of libraries for general mathematical computations, such as GMP [35], gmpy2 [44], Java BigInteger [45], and many others. These libraries are used in Wolfram Mathematica and Maple computing systems. The GMP library is also used in the source code of the GCC compiler collection.

The second group consists of libraries that implement cryptographic operations that require computations with numbers bigger than machine word size. The best known are OpenSSL [46], and WolfSSL [47]. There are also libraries focused and optimized for microcontrollers, for example ARM–Crypto–Lib [48] and AVR–Crypto–Lib [49]. These two libraries are unmaintained since 2015.

The library that fits into both groups is bigdigits [50]. It implements all basic arithmetic operations and offers optimized versions for operations used in cryptographic algorithms (for example modular exponentiation). It is primarily focused on 64-bit processors.
Analysis and Design

This section is focused on the implementation of the VeraGreg framework at the system level and which parts the implementation consists of. Then it discusses individual parts of the implementation, cryptographic primitives and parameters for implementation of the VeraGreg framework (key length, input data size) and communication interface. The last part of this section describes the choice of the platform, and analysis and design of a library implementing multiprecision integer arithmetic.

2.1 System Design

The implementation of the VeraGreg framework consists of two main parts, communication interface and libraries that implement VeraGreg and underlying cryptographic primitives. Blocks in the Figure 2.1 represent libraries. The figure does not show the library bigi.h, which provides multiprecision arithmetic operations for paillier.h and veragreg.h. The library bigi_io.h is a part of communication interface. The library aes.h is a wrapper around cryptographic accelerator for AES cipher. policy.h implements simple policy.

2.2 VeraGreg Design

This section discusses the choice of specific algorithms for instance of the VeraGreg framework described in [IS]. In particular, the selection of cryptographic primitives. The main component is veragreg.h, the policy is implemented in its own library policy.h so that it can be easily changed.

2.2.1 Underlying Cryptographic Algorithms

The definition of the VeraGreg framework is very general, the cryptographic algorithms must only meet the criteria mentioned in the framework definition, i. e., the group of plaintexts of the homomorphic cipher must be isomorphic
2. Analysis and Design

Figure 2.1: Diagram of the VeraGreg framework implementation

with modular addition and its elements must be numbers at least $\nu+\lambda+\mu_1+\mu_2$ bits long.

However, the selected parameters for the underlying cryptographic algorithms should also meet security requirements, namely the NIST recommendations for key lengths [51].

2.2.1.1 Additively Homomorphic Cipher

Paillier cryptosystem [6] was chosen for the implementation of the VeraGreg scheme. The main reason for this decision is the possibility of acceleration of the encryption and decryption algorithms using existing hardware, an RSA accelerator. The decryption process in other cryptosystems (e.g., the Benaloh cryptosystem [4]) is significantly more computationally demanding than the encryption process.

Damgård–Jurik cryptosystem [8] has similar properties as the Paillier cryptosystem, however there is a drawback if the number of bits for the computation is fixed, e.g., when hardware accelerator is used. Damgård–Jurik
cryptosystem works with numbers mod \( n^{s+1} \) where \( n = p \cdot q \). If the number of
bits for the number \( n^{s+1} \) is fixed, with increasing \( s \), the length of the primes
\( p \) and \( q \) decreases which lowers the security of the cryptosystem.

For these reasons, Paillier cryptosystem was chosen as the additively ho-
momorphic cipher for the VeraGreg scheme. It has the desired homomorphic
properties, it can be accelerated by using existing hardware and both encryp-
tion and decryption can be implemented efficiently.

Paillier cryptosystem is based on a similar problem as RSA \([6]\), thus the
length of the primes that form the modulus \( n \) must follow the same rec-
ommendations as RSA. Since the computations in Paillier cryptosystem are
performed in group \( \mathbb{Z}_{n^2}^* \), it requires double the number of bits for number
representation in order to achieve the same level of security as RSA.

NIST recommends 2048 bits for RSA modulus \([51]\) and that is why secure
microcontrollers are usually equipped with 2048-bit RSA accelerator. Since
Paillier cryptosystem relies on very similar problem as RSA, the parameter
\( n \) of Paillier cryptosystem should be also at least 2048 bits long. Paillier
cryptosystem uses modular exponentiation \( \mod n^2 \), which requires 4096 bits
long numbers. Computations with these numbers can not be accelerated with
the 2048 bit RSA hardware module. For this reason, the implementation of
the Paillier cryptosystem uses 1024-bit \( n \).

Since this work is only a proof–of–concept and is not supposed to be used in
practice, the fact that the key size does not comply with the NIST recommen-
dation \([51]\) can be neglected. However, for a real world implementation, the
key size should comply with the NIST recommendation \([51]\), which requires
at least 2048 bits for RSA, thus for the Paillier cryptosystem, microcontroller
with 4096 bit RSA accelerator is needed.

More microcontrollers with 4096-bit RSA accelerators can be expected to
appear on the market, since with the growing accessibility of huge compu-
tational power, the length of the keys used in cryptography increases. This
can be illustrated by the development of the NIST recommendation for key
length, SP 800-57. In revision 3 \([52]\) published in 2012, the recommended key
length for RSA was 1024 bits. In the current revision 4 \([51]\) published in 2016,
1024-bits long RSA key is considered equal to 80 bit symmetric cipher and
thus not secure for general use hence the recommended RSA modulus length
is at least 2048 bits.

2.2.1.2 Symmetric Cipher

The VeraGreg symmetric cipher should be also accelerated and must be able
to encrypt the range of identifiers defined in VeraGreg’s Policy (block length
must be at least \( \lambda \)). Secure microcontrollers are usually equipped with 128-bit
AES \([53]\) or triple DES \([54]\) accelerators. The PRESENT cipher \([55]\) could
also be used for the implementation of the VeraGreg scheme, however, there
is currently no microcontroller with hardware acceleration of the PRESENT.
2. Analysis and Design

Considering the fact that AES is widely used and the availability of secure microcontrollers with AES accelerators, AES was chosen as the symmetric cipher for the implementation of the VeraGreg framework. AES is also mentioned as suitable cipher in the original paper that defines VeraGreg framework [18].

2.2.2 System parameters

The parameters of the implemented VeraGreg scheme result from the decisions mentioned previously in this section.

- $\lambda = 128$ bits (AES block size),
- Grant must return 128-bit value (AES block size),
- length of the data - 32 bits,
- $\mu_1 = \mu_2 = 128$ bits,
- $\nu = 8$, it is possible to create an aggregate of 255 values.

For all of the values mentioned above, the inequalities in Equation 2.1 must hold, since they are necessary for encryption and decryption of the data by the Paillier cryptosystem with modulus $n$ which is 1024 bits long.

\[
m \leq n \\
\lambda + \mu_1 + \mu_2 + \nu \leq n \\
128 + 128 + 128 + 8 \leq 1024 \\
392 \leq 1024
\]  

2.2.3 Policy

Policy checks whether the list of data identifiers complies with defined rules. For the experimental implementation, following simple rules were chosen:

- the number of values in the aggregate is smaller than 256,
- the number of values in the aggregate is greater than 3,
- one value cannot repeat.

The maximal number of values in the aggregate is limited by the parameter $\nu$, which sets the maximal number of ciphertext additions before an overflow occurs. From the practical point of view, the limit for the number of values in an aggregate is the amount of RAM available on the microcontroller and the size of the identifier.

The limit for the number of values in the aggregate was selected with respect to memory constraints of the microcontroller. The amount of memory needed for the VeraGreg implementation should leave enough space for the main application.
2.2.4 Communication

The communication protocol is based on the ASN.1 standard. For the implementation, the library tiny-asn1 was used. The protocol consists of four types of messages as depicted below.

\[
\text{VeraGregProtocol DEFINITIONS ::= BEGIN}
\]

\[
\text{VeraGregEncrypt ::= SEQUENCE}
\]
\[
\text{\hspace{1cm} \{ dataEnc INTEGER(3..255) \}}
\]

\[
\text{VeraGregEncDataOut ::= SEQUENCE}
\]
\[
\text{\hspace{1cm} \{ encData IA5String, idData IA5String \}}
\]

\[
\text{VeraGregDecrypt ::= SEQUENCE}
\]
\[
\text{\hspace{1cm} \{ data IA5String, length INTEGER(3..255), idList SEQUENCE(SIZE(1..255)) OF IA5String \}}
\]

\[
\text{VeraGregDecDataOut ::= SEQUENCE}
\]
\[
\text{\hspace{1cm} \{ data IA5String \}}
\]

\[
\text{END}
\]

\text{VeraGregEncrypt} represents third party request for data. After receiving this message, the device send requested number of \text{VeraGregEncDataOut} messages.

The message type \text{VeraGregEncDataOut} represents the output of the VeraGreg scheme. It consists of a ciphertext and an identifier of the encrypted data. Both of these numbers are represented by hexadecimal strings.

\text{VeraGregDecrypt} represents the decryption request from the third party. It consists of an aggregate of ciphertexts and a list of identifiers corresponding with the ciphertexts.

The messages are serialized using BER and sent or received via serial interface.

2.3 Naïve Scheme

A naïve scheme was implemented for an effectiveness evaluation of the VeraGreg framework. This scheme is based on AES symmetric cipher. It assumes
2. Analysis and Design

Table 2.1: Secure microcontrollers comparison

<table>
<thead>
<tr>
<th>Name</th>
<th>Accelerators</th>
<th>Price</th>
<th>Secure NVM</th>
<th>Chip</th>
<th>Dev Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEC1302</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>1.93 $</td>
<td>39 $</td>
</tr>
<tr>
<td>CEC1702</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>1.93 $</td>
<td>39 $</td>
</tr>
<tr>
<td>MAX32510</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>NA</td>
<td>561 $</td>
</tr>
</tbody>
</table>

that the service provider knows the key and thus does not guarantee any privacy.

The naïve scheme consists of a library for random number generation and a wrapper library for the AES accelerator. Because this scheme is implemented for a purpose of comparison with the VeraGreg framework, the communication interface is one-way. After a defined number of encryptions and decryptions, the measured data is sent via serial interface of the microcontroller.

2.4 Platform

According to the thesis’ task, the implementation platform should be low cost and secure. These two requirements go against each other, since the target applications of secure microcontrollers are not low–cost solutions. Manufacturers assume that if some information must be secured, it is valuable and hence the higher cost of the secure microcontroller will not change the price of the final product.

2.4.1 Platform Requirements

Based on the selection of the algorithms in the previous section and from the thesis’ task following requirements arise:

- RSA accelerator with at least 2048 bits,
- AES accelerator,
- secure non–volatile memory (optional),
- low–cost (low price per piece and cheap development kit).

There are many secure microcontrollers available, however, only a few of them meet the aforementioned requirements. Find an overview of suitable secure microcontrollers in Table 2.1. The price of the MAX32510 microcontroller is not publicly available, it is only available on request upon ordering.
2.4. Platform

2.4.2 Selected Platform

MAX32510 manufactured by Maxim Integrated appears to be the best candidate. In addition to all the platform requirements it also contains a tamper protection which deletes contents of the memory if the temperature or voltage is out of defined range. Furthermore, it contains an anti-decapsulation protection.

Unfortunately, the microcontroller, its documentation and development kit are not publicly available. In order to access the documentation and development tools it is necessary to sign a non-disclosure agreement (NDA), which Maxim Integrated concludes only with commercial subjects and not universities, according to the company sales department.

As the implementation platform we have chosen CEC1302 from Microchip. It contains 32-bit ARM M4 core, RSA, AES and SHA hardware accelerators and true random number generator. The only development board available today is Clicker 2 for CEC1032, depicted in Picture 2.2, manufactured and sold by MikroElektronika [57].

The choice of CEC1302 is a compromise because it does not contain secure NVM. On the other hand it is lowcost and the development tools are freely available which, in the end, is the most important criteria when choosing an implementation platform.

All of the development tools for the microcontroller CEC1302 are parts of MikroC for ARM [58] integrated development environment offered by MikroElek-
tronika.

Secure non-volatile memory is available as an external module SECURE 3 CLICK [59] for CEC1302.

2.5 Arithmetic library

None of the existing libraries for multiprecision integer arithmetic is suitable for the implementation of the VeraGreg framework on the selected microcontroller. The general library GMP [35] is not optimized for the use with microcontrollers, furthermore, it uses dynamic memory allocation. On the other hand, ARM-Crypto-Lib and AVR-Crypto-Lib are optimized for microcontrollers, but they do not implement the division operation and the greatest common divisor operation. What is more, they have not been maintained since 2014 and they are not ANSI C compatible.

The bigdigits library [50] is ANSI-C compatible, but modular multiplication is implemented by generic integer multiplication and the reduction step is done by division. This approach is ineffective, hence it would be necessary to implement modular multiplication and modular exponentiation. Dynamic memory allocation can be disabled, however doing so would require considerable changes of the library source code.

Due to the aforementioned reasons, a new library for multiprecision integer arithmetic was implemented. Another reason for this decision was the limitation of the choice of development tools for the selected platform.

The MikroC for ARM compiler differs from frequently used compilers (GNU gcc toolchain, LLVM) in integer promotion — during arithmetic operation and assignment to a bigger data type, the arithmetic operation is not done in the size of the target of the assignment, but in the size of the operands.

Porting an existing library would therefore require a lot of effort, comparable to a creation of new library. By creating own library, code homogeneity is ensured, as well as the fact that the implemented algorithms are optimized for the selected platform.

For the implementation of the VeraGreg framework and underlying cryptographic primitives, it was necessary to implement basic arithmetic operations including modular inversion and greatest common divisor.

The implemented library is multiplatform, it can be used on ARM and x86 processors. The size of the machine word and the size of a large integer can be easily set before compilation.

The library consists of two parts: bigi.h and bigi_io.h. bigi.h provides functions for arithmetic operations and defines the following data types.

- **bigint**: an array of machine-word-sized unsigned integers representing a number,
- **bigint_type**: a machine-word-sized unsigned integer,
bigint_type_big: a double-machine-word-sized unsigned integer.

bigi_io.h provides functions for converting numbers to or from a string and for output to terminal or serial interface. I/O functions are separated so that they can be easily replaced according to the platform or user requirements.

2.6 Side-Channel Analysis and Countermeasures

This section discusses possible vulnerabilities of the implementation of the VeraGreg framework, with respect to side channel attacks. Security of cryptographic primitives is discussed first, followed by a discussion about security of the whole VeraGreg scheme.

2.6.1 Possible Weak Points

In order to secure the implementation of the VeraGreg framework, it was necessary to identify possible weak points in the underlying cryptographic algorithms, namely Paillier cryptosystem and AES.

Since the security analysis of AES with respect to side channel attacks is thoroughly described in [27], only the analysis of the Paillier cryptosystem is described.

2.6.1.1 Paillier cryptosystem

In order to compromise the Paillier cryptosystem, the attacker must know either the primes $p$ and $q$, or the private key $d$.

The figure 2.3 depicts the order of function calls during decryption in the Paillier cryptosystem. Green colored functions do not work with any portion of the private key. Functions working directly with the private key are highlighted in red and functions that use intermediate values including the private key are marked by yellow. The white arrows show places, where it could be possible to create a power hypothesis for differential power analysis.

2.6.1.2 VeraGreg framework

The definition of compromise of the VeraGreg framework depends on individual use case — the kind of data secured by VeraGreg and the sensitivity of the result. In the most paranoid case, the VeraGreg framework is compromised if the attacker can create ciphertext. The variants of comprising the VeraGreg framework are shown in Table 2.2.

In order to compromise the VeraGreg framework, the attacker must compromise the underlying cryptographic algorithms, in this case AES and Paillier cryptosystem. Furthermore, she must know the secret numbers $m_1$ and $m_2$. 

29
which are a part of the VeraGreg secret key. However, these two numbers can be easily computed with the knowledge of the system parameters, AES and Paillier cryptosystem keys, two ciphertexts \( c_1, c_2 \) and plaintext \( d_1 \) corresponding to \( c_1 \) in a following way: attacker decrypts \( c_1 \) and \( c_2 \) with the AHE key into \( \tilde{c}_1, \tilde{c}_2 \) respectively. Then computes \( m_2 = gcd(\tilde{c}_1, \tilde{c}_2) \). With the knowledge of \( m_2 \), attacker obtains \( m_1 = (\tilde{c}_1/m_2 - d_1)/SE(b_1) \) where \( b_1 \) is identifier corresponding to ciphertext \( c_1 \).
2.6. Side-Channel Analysis and Countermeasures

2.6.1.3 Key Storage
Since the selected platform does not contain secure NVM for storing keys, keys must be stored either in external module with secure NVM or directly in the firmware of the microcontroller.

Provided the fact that CEC1302 contains hardware root of trust and does firmware authentication before boot, keys are stored as constants in the firmware. If we exclude rather extreme types of side channel attacks, like RAM freezing[60] and chip decapsulation, this way of storing keys can be considered secure. Firmware authentication prevents any unauthorized change of the keys.

2.6.2 Implemented Side-Channel Attack Countermeasures
Encryption and decryption operations of the VeraGreg implementation are protected against side channel attacks by hiding in both time and power. For this, peripherals of the CEC1302 microcontroller are used.

During encryption and decryption, a switching activity of peripherals increases power consumption and noise which masks power consumption of these operations.

A timer peripheral is also used for hiding in time. The timer is set to create interrupts in random intervals, thus making the execution time of encryption and decryption random. This countermeasure makes a DPA attack significantly harder [27].

It can be assumed that the hardware accelerators for AES and RSA also implemented some countermeasures against side channel attack. The only weak point can be initialization of the accelerators when the keys are loaded into their memory.

The effectiveness of implemented side channel attacks countermeasures can be evaluated by mounting selected attack on both secured and non-secured variant of the implementation and comparing the number of measured power traces required for key recovery. The aim of the attacks should be complete compromise of the VeraGreg framework, which means recovery of the private key for the homomorphic cryptosystem and the key used by the symmetric cipher.

Another way of checking the implemented side channel attack countermeasure is to use the method described in [61] which evaluates the amount of information present in the power consumption channel without the need for a power model.
CHAPTER 3

Implementation

This chapter describes the implementation of all parts of the VeraGreg framework. Starting with the new multiprecision integer arithmetic library, Paillier cryptosystem and the communication interface description follow. Last part of this chapter is dedicated to the implementation of simple desktop applications, which acts as a server for data processing.

Complete source code of the implementation can be found on attached electronic media in the folder src/veragreg.

In the description of function prototypes, the data type bigint_type * is omitted. If the parameter of the function has a different type, it is mentioned explicitly.

3.1 Arithmetic Library bigi

The library bigi works over a static array of unsigned integer type elements. The size of the array is determined during compilation according to predefined constants, namely size of the machine word on the target platform and required number of bits in the number representation. For Montgomery multiplication, two elements are added in order to prevent overflow [41], as depicted in Figure 3.1 which shows the structure of an array representing a number in our library. Endianness of the number was chosen for easier debugging.

In this case of the VeraGreg framework implementation, the array has \( 2048 / 32 + 2 = 66 \) elements.

The library consists of 4 files, bigi.h, bigi.c, bigi_io.h and bigi_io.c. The file bigi.h defines the prototypes of functions implementing arithmetic operations and following data types:

- bigint an array of machine word–sized integers representing a number,

- bigint_type an unsigned integer of size corresponding with the machine word size,
Figure 3.1: Number representation in bigi library

- **bigint_type_big** a double word.

Each arithmetic operation corresponds with one function. Operands of each arithmetic function are represented by input parameters of the function and passed as pointers to arrays representing numbers.

In order to lower memory requirements of the computations, some operations change the input operands. This fact is mentioned in the function description and a backup of the operands is left to the user. All algorithms were implemented with respect to low memory requirements.

### 3.1.1 Basic arithmetic operations

Basic arithmetic operations are as follows: addition, subtraction, bit shifts, division and multiplication.

#### 3.1.1.1 Addition

The function implementing addition has the following interface:

```c
int bigint_add(A, B, RES)
```

where `A` and `B` are addends, the result is stored into `RES`. This function does not change operands. If overflow occurred during addition, the return value is 1, otherwise the return value is 0.

#### 3.1.1.2 Subtraction

Subtraction is implemented by function very similar to addition, it has the following interface:

```c
int bigint_sub(A, B, RES)
```

where `A` is minuend and `B` is subtrahend, the result is stored into `RES`. The return value signalizes if the result should be negative, in which case the function returns 1, otherwise 0.

#### 3.1.1.3 Bit Shifts

Bit shifts are implemented by functions with following interface:

```c
void bigint_shift_left(A, unsigned int count)
```
3.1. Arithmetic Library bigi

void bigint_shift_right(A, unsigned int count)
The operand A is shifted by number of bits specified by the count parameter.

3.1.1.4 Multiplication

The function void bigint_mult_fit(A, B, RES) implements multiplication. The maximal length of operands A and B is NUM_SIZE/2. This ensures that the length of the result does not not exceed the size of the array used for storing numbers. Shall this assumption not to be true, the behavior of this function is undefined. Multiplication is implemented according to [31].

3.1.1.5 Division

Division is implemented by the function int bigint_div(A, B, REM, RES). It uses Knuth's Algorithm D [31], implemented according to [32]. This algorithm was selected with regard to its straightforward implementation.

Other algorithms described in Section 1.7.1.2 have better asymptotic complexity, but it applies only for numbers that are bigger than this library is intended for.

3.1.2 Modular arithmetic

As mentioned before, modular arithmetic is the core of most asymmetric cryptosystems. For this reason, state-of-the-art algorithms were chosen for its implementation.

3.1.2.1 Modular multiplication

Provided the fact that the target platform of this library is a microcontroller, memory requirements were the most important criteria for choosing modular multiplication algorithm.

For this reason, Montgomery multiplication algorithm (described in Section 1.7.1.4) was chosen for modular multiplication. It requires only $n + 2$ machine words for storing intermediate results where $n$ is the number of machine words in the representation of the modulus.

Multiplication in the Montgomery domain requires precomputation of the parameter $M'_0$, transfer of the operands to the Montgomery domain and transfer of the result from the Montgomery domain.

This represents an overhead that pays off only in a case where there are several modular multiplications in a row (modular exponentiation) or if consequent operations can be done in the Montgomery domain.

Function implementing modular multiplication in the Montgomery domain has the following interface:

bigint_mult_mod_mont(A, B, MOD, bigint_type m_prime, RES). The type
of the parameter m_prime is bigint_type because the parameter is calculated modulo machine word size.

Because of the overhead introduced by modular multiplication in the Montgomery domain, modular multiplication was implemented in another way as well. The second implementation uses the double–and–add algorithm. This algorithm requires the same number of machine words as the modulus for storing intermediate results.

The function interface is similar to the function for multiplication in the Montgomery domain: bigint_mult_mod(A, B, MOD, RES). In each step of the algorithm, the operand B is shifted right, the intermediate value is shifted one bit right and reduced by the modulus MOD, or operand A is added to the intermediate value and the result is reduced by the modulus MOD.

3.1.2.2 Modular exponentiation

Modular exponentiation is implemented in two ways, which differ in the function used for modular multiplication. The first one, naive, uses function bigint_mult_mod, the second one uses bigint_mult_mod_mont function.

Both functions have the same interface, the function using square–and–multiply for modular multiplication has the following interface:
void bigint_pow_mod(A, B, MOD, RES). The function that uses multiplication in the Montgomery domain assumes, that the operand A is a number in the Montgomery domain and has the following interface:
void bigint_pow_mod_mont(A, B, MOD, RES). Parameter A is the base, B is the exponent and MOD represents the modulus. The result is stored in RES. The parameter B is shifted right and if the value is used after the call of one of these functions, it must be copied to another array.

The usage of Montgomery powering ladder [143] secures the implementation against timing attack because in each iteration, both square and multiply operations are executed.

3.1.2.3 Modular reduction

Modular reduction is implemented by Barret’s modular reduction [37]. Function implementing the modular reduction has the following interface:
void bigint_reduce_bar(A, MOD, MU, RES). The parameter MU corresponds with the constant $\mu$ in the algorithm description in Section [1.7.1.3] and must be computed prior to calling this function according to $\mu = \left\lfloor b^{2k}/m \right\rfloor$ where $b$ stands for the radix, $k$ is the length of the modulus MOD in bits and $m$ is the modulus MOD.

Parameter MU can be used until the modulus is changed, or more values of this parameter can be precomputed and then used for the corresponding value according to the modulus. This function can be used for the implementation of modular multiplication.
3.1.2.4 Modular inversion

By modular inversion of a number \( a \in \mathbb{Z}_n^* \) is meant such number \( a^{-1} \) for which it holds \( a \cdot a^{-1} \equiv 1 \mod n \). One way of finding modular inversion is to use Extended Euclidean Algorithm.

Another possibility is to use Euler’s theorem and modular exponentiation,

\[
a^{\phi(n)} \equiv 1 \mod n \quad (3.1)
\]

If both sides of the Equation (3.1) are multiplied by \( a^{-1} \) (assuming that the inversion exists), we get \( a^{\phi(n)-1} \equiv a^{-1} \mod n \), which gives the formula for \( a^{-1} \).

A disadvantage of this procedure is the implicit assumption of the existence of the inverse and also the fact that it is necessary to know \( \phi(n) \). Note that calculation of \( \phi(n) \) is computationally demanding if the prime factorization of the modulus is not known.

Therefore, for the library to be generally usable, modular inversion is implemented using Extended Euclidean Algorithm which can detect possible absence of the inverse.

The function has the following interface: \textbf{int bigint_mul_inv(A,B,RES)}. Parameter \textbf{A} is an element, whose inversion is sought, parameter \textbf{B} represents the modulus and parameter \textbf{RES} is used to store the result. If the inverse element is found, function returns 0. If the inversion does not exist, the return value is 1 and the parameter \textbf{RES} remains unchanged.

3.1.3 Advanced functions

The only advanced function implemented in the \textbf{bigi} library is the greatest common divisor (GCD). GCD of two integers \( a, b \) is the greatest positive integer \( d \), such that it holds \( d \mid a \) and \( d \mid b \).

This operation is implemented in a function \textbf{void bigint_gcd(A,B,RES)}. Parameters \textbf{A} and \textbf{B} represent numbers, the parameter \textbf{RES} stores the result. It must hold \( A \geq B \). Otherwise, the result in \textbf{RES} is equal to zero. If the inequality holds, the result is always greater than or equal to one, as arises from the definition of the CGD.

The algorithm 14.54 "Binary gcd algorithm" from [38] is used, its transcription is listed below as the Algorithm 5. This algorithm uses only addition, subtraction and bit shifts.
3. Implementation

Algorithm 5 Binary GCD

1: procedure gcd
2:  \( g \leftarrow 1 \)
3:  while \( x \) and \( y \) are odd do
4:      \( a \leftarrow a/2 \)
5:      \( b \leftarrow b/2 \)
6:      \( g \leftarrow 2g \)
7:  while \( a \neq 0 \) do
8:      while \( a \) is even do
9:          \( a \leftarrow a/2 \)
10:     while \( b \) is even do
11:          \( b \leftarrow b/2 \)
12:      \( t \leftarrow |a - b|/2 \)
13:     if \( a \geq b \) then
14:         \( a \leftarrow t \)
15:     else
16:         \( b \leftarrow t \)
17:    return \( g \cdot b \)

All variables in the implementation of the Algorithm 5 are multiprecision integers and arithmetic operations are performed by corresponding library functions.

3.2 Paillier cryptosystem

Paillier cryptosystem is implemented in two variants. The first variant uses the bigi library for modular exponentiation, the second variant employs the hardware RSA accelerator.

The whole Paillier cryptosystem can be simplified by the choice of parameter \( g \). In the original description of Paillier cryptosystem [6] \( g \) is any integer whose order in the group \( \mathbb{Z}_{n^2}^* \) is a nonzero multiple of \( n \). In [8] and [1], authors suggest to choose \( g = n + 1 \). With such a choice of the parameter \( g \), [1] defines decryption as follows

\[
m = (c^d \mod n^2 - 1) \cdot d^{-1} \mod n.
\]  

In such a case there is no need for calculation of the function \( L(x) = \frac{x-1}{n} \) which shortens decryption by one integer division.

This choice of the parameter \( g \) also allows to simplify the encryption process by using the binomial theorem for the specific case shown in the Equation (3.3).

\[
(n + 1)^a = 1 + an \mod n^2
\]  

38
With this knowledge, encryption equation (1.4) can be shortened from \( c = g^m \cdot r^n \mod n^2 \) to the following:

\[
c = (1 + m \cdot n) r^n \mod n^2.
\]

This change saves one exponentiation operation in exchange for one modular multiplication and addition.

Paillier cryptosystem is implemented by the library `paillier` which implements all operations, namely initialization, encryption, decryption and addition, and their corresponding variants that use the hardware RSA accelerator.

Functions that use the microcontroller’s peripherals are wrapped in conditional compilation blocks, so that the code can be easily ported to a different platform.

### 3.2.1 Init

This function creates the public exponent \( n \), base \( g \) and the private key \( d \) based on primes \( p \) and \( q \), `int paillier_init(P, Q, DK, EK)`.

Parameters \( P \) and \( Q \) are primes. It is assumed that their product \( n = p \cdot q \) fits into an array representing a number. \( EK \) and \( DK \) are output parameters representing the public and private key, respectively.

In order to use the hardware acceleration for encryption and decryption, the Paillier cryptosystem must be initialized by the function `paillier_rsa_init(P, Q, DK, EK)`. This function has the same interface as `paillier_init`, the difference is that this function fills memory slots of the Public Key Engine (PKE) \[62\] with public and private keys respectively.

### 3.2.2 Encryption

Encryption is done according to Equation (3.4). Provided the fact that the encryption algorithm works only with public key, there is no need to protect it against side channel attacks. Two versions of this operation are implemented: one is using the hardware accelerator, the second one uses modular exponentiation from the `bigi` library.

The function implementing encryption in Paillier cryptosystem has the following interface: `int paillier_enc(ek, m, c)`.

Parameter \( m \) represents the plaintext, \( ek \) represents Paillier cryptosystem public key and the resulting ciphertext is stored in \( c \).

The return value of this function signalizes whether the encryption was successful, in which case it returns 0. If the plaintext violates \( m < n \), the return value is 1.

The function employing the hardware accelerator has the same interface as the function that uses software modular exponentiation: `paillier_rsa_enc(ek, m, c)`. This function is wrapped in conditional compilation block.
3. Implementation

#if __mikroc_PRO_FOR_ARM__, which identifies the compiler and if the compiler for the CEC1302 platform is used, the function is included in compilation. The RSA accelerator is used to speed up the computation of \( r^n \mod n^2 \).

3.2.2.1 Random Number Generation

Library `rng.h` implements a random number generator. It works as a wrapper interface for different random number generators on different platforms.

A random integer \( r \) for Paillier cryptosystem is generated by the function `rng_get_bigint(R, uint32_t words)`. Parameter \( R \) is the generated number of type `bigint_type` which has first `words` elements of its array representation filled with the output of the random number generator (RNG).

According to selected platform, the random number is generated either by a true random number generator (TRNG) on CEC1302 module, or by the function `random()`, if the platform is x86. The `random()` function is not considered secure, however, the security of Paillier cryptosystem does not rely on the randomness of parameter \( r \).

Prior to calling the function `rng_get_bigint`, it is necessary to initialize the RNG by calling the function `rng_init()`. In case of the CEC1302 platform, this function turns on the block clock and powers it. In case of x86 platform, the RNG is seeded by current time.

3.2.3 Decryption

The function that implements decryption of Paillier cryptosystem has the following interface: `paillier_dec(EK, DK, C, M)`. Because of the choice of parameter \( g \), decryption is done in a simplified way according to Equation (3.2).

Same as encryption, two versions of the decryption function are implemented. One uses the hardware RSA accelerator, the other uses modular exponentiation from the library `bigi`.

The function that uses the hardware accelerator has the same interface as the function using software modular exponentiation.

`paillier_rsa_dec(EK, DK, C, M)`

The accelerator is used for the computation of \( c^d \mod n^2 \).

Since the decryption algorithm works with the private key, it is protected against side channel attacks by hiding in power. During the decryption following peripherals are running:

- A/D converter,
- random number generator,
- timers.
The code that controls the peripherals is wrapped in a block of conditional compilation and the corresponding parts of the code are used only if the target platform is the CEC1302.

3.3 VeraGreg scheme

Implementation of the VeraGreg framework uses the algorithms mentioned in [18]. Critical parts of the algorithm are secured against side channel attacks.

Operations Init, Grant, Encrypt and Decrypt are implemented for the CEC1302 platform, their prototypes can be found in `veragreg.h`. System parameters $\lambda$, $\mu_1$, $\mu_2$ and data types are defined in `veragreg_defs.h`.

The Add operation is implemented in C only for the x86 platform because it is not intended for the microcontroller side.

3.3.1 Init

Init generates keys for the underlying cryptographic primitives. Keys for the Paillier cryptosystem are computed from predefined primes $p$ and $q$ that are stored in the firmware. Following, it initializes the RSA and AES accelerators.

The keys are stored in structures that are passed to the function

\[
\text{int veragreg\_init(}\text{veragreg\_priv\_key } \ast \text{priv, veragreg\_pub\_key } \ast \text{pub})\]

as output parameters.

\[
\text{typedef struct veragreg\_priv\_key } \{
\begin{align*}
\text{key\_type ahe\_dec\_key;}
\text{key\_type se\_key;}
\text{key\_type m1;}
\text{key\_type m2;}
\end{align*}
\} \text{veragreg\_priv\_key;}
\]

\[
\text{typedef struct veragreg\_pub\_key } \{
\text{key\_type ahe\_enc\_key;}
\} \text{veragreg\_pub\_key;}
\]

3.3.2 Grant

The function that implements the Grant operation has the following interface:

\[
\text{int veragreg\_grant(}\text{MESS, ID})\]

Grant checks, whether the data comply with the policy, in this case if they are exactly 32 bits long. The data identifier is stored in the output parameter ID. The identifier is the value of a counter that is increased with each granted identifier. The return value is 0 if the data complies with the policy, otherwise it is equal to 1.
3. Implementation

3.3.3 Encrypt

Encryption is implemented according to the Equation (3.5) by the function with following interface:

\[
\text{int veragreg_enc(M,C,ID,veragreg_pub_key *p,veragreg_priv_key *s)}
\]

Parameter \( M \) represents the plaintext, \( C \) represents corresponding identifier (output of the \text{varagreg_grant} function). \( C \) is the output parameter for the ciphertext. Parameters \( p \) and \( s \) represent the public and private key structures, respectively.

\[
c = \text{paillier_enc}(\text{AES}(b) \cdot m_1 + d \cdot m_2)
\]

(3.5)

Paillier cryptosystem encryption uses the accelerated variant. Corresponding hardware accelerator is also used for AES encryption. Both accelerators are used only if the target platform is CEC1302, otherwise software implementations of Paillier cryptosystem encryption and AES are used.

This function is protected against side channel attacks by hiding in power and time, the following peripherals are turned on during encryption:

- A/D converter,
- random number generator,
- timers.

Running peripherals increase the power consumption, thereby masking the information in the power consumption channel.

Random swapping of operations during encryption also protects this function against side channel attacks. Encryption only uses addition and multiplication, the encryption Equation (3.5) can be broken down as follows

\[
c = \text{paillier_enc}(\text{AES}(b) \cdot m_1 \cdot m_2 + d \cdot m_2)
\]

(3.6)

As it can be seen from the Equation 3.6, the product \( AES(b) \cdot m_1 \cdot m_2 \) can be computed in three different ways, in each case the stored intermediate value is different. The computation \( d \cdot m_2 \) can be done before or after the computation of the product \( AES(b) \cdot m_1 \cdot m_2 \). This results in six different ways of computing encryption formula.

It can be assumed that the different intermediate values, which appear in different time during encryption, make the DPA attack harder because this kind of attack requires the intermediate values to be aligned in time in all measured power traces.
3.3.4 Add

Addition of ciphertexts is implemented by function

\[
\text{veragreg\_add(biggint\_type } C[\text{][NUM\_SIZE}], \text{ crypto\_type LIST[\text{][NUM\_SIZE}], uint32\_t LEN, SUM)}.
\]

Parameter \(C\) is a pointer to an array of length \(LEN\) containing ciphertexts, \(LIST\) is a pointer to an array of identifiers corresponding with the ciphertexts.

3.3.5 Decrypt

The function that implements decryption has the following interface:

\[
\text{int veragreg\_dec(crypto\_type } *C, \text{ crypto\_type LIST[\text{][NUM\_SIZE}],}
\text{ unsigned int LIST\_LEN, crypto\_type } *\text{DEC, veragreg\_pub\_key } *\text{pub,}
\text{ veragreg\_priv\_key } *\text{priv)}
\]

According to the Definition the decryption algorithm first checks whether the list of identifiers complies with the policy. Furthermore, it ensures that the aggregate contains only values, whose identifiers are in the list and if both conditions are satisfied, it returns decrypted aggregate.

The decryption process is secured against side channel attacks in the same way as encryption. During decryption, following peripherals are running:

- A/D convertor
- random number generator,
- timers.

During the decryption, several equations listed in the Section must be checked, it is not possible to randomize the order of operations.

3.4 Communication

The communication interface is implemented using the tiny–asn.1 C library. The communication goes through serial interface of the microcontroller.

3.5 Naive symmetric scheme

The naive scheme consists of a random number generator, AES hardware accelerator wrapper and a simple interface for sending the measured data over the serial line.

As stated in the Section the naïve scheme is implemented only for the purpose of comparison with the VeraGreg framework. In order to minimize the communication overhead and make the measurement as fast as possible, the naïve scheme encrypts random data using the AES accelerator and measures the execution time of encryption. Then the resulting ciphertext
3. Implementation

is decrypted and the time of the decryption operation is measured and the
times of encryption and decryption are sent over the serial line. This is done
several times, according to user requirements.

3.6 PC demo app

A demo application for PC is implemented in C. Works as a server that sends
request for encrypted data to the CEC1302 platform, then adds received ci-
phertexts using the VeraGreg Add operation and sends the aggregate back for
decryption.

The PC application uses the ASN.1 based communication protocol imple-
mented with the tiny-asn.1 library \cite{56}. 
Firstly, individual libraries were tested, followed by the entire implementation. Testing was done on the x86 platform using Python scripts and a golden model based on gmpy2 \cite{44}. The structure of the test environment is depicted in Figure 4.1.

The correctness of the implementation on the target platform was verified by comparing the encryption results with the results calculated on the x86 platform. For this purpose, a randomized component of Paillier cryptosystem was fixed. In this way, the multiplatform implementation and its easy configuration was also tested.

During implementation the individual components of the system were tested, integration of all components on the system level was also tested.

All components were tested using randomized tests and corner cases.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{test_environment_structure.png}
\caption{Test environment structure.}
\end{figure}
4. Testing

4.1 Arithmetic library

Correctness of the implemented library bigi was verified against existing Python library gmpy2 [44]. The testing was done on the x86 platform.

Testing was done by random values — random operands were generated and passed to bigi functions and corresponding gmpy2 functions. Results were then compared.

4.2 Paillier cryptosystem

in order to verify the correctness of the implementation of Paillier cryptosystem, a random $r$ was fixed. Results were compared with implementation in Python which uses gmpy2 library. Both x86 and CEC1302 platforms were tested.

On the CEC1302 platform, versions using hardware accelerator and exponentiation in the Montgomery domain were also tested.

4.3 Communication

Communication has been verified by the process itself. The correctness of designed ASN.1 protocol was tested with ASN1 Playground tool [63] which verifies the correctness of protocol syntax and allows BER encoding and decoding.

4.4 System level

The correctness of the whole VeraGreg implementation was verified by encryption of a random value, decryption of the resulting ciphertext and checking that the decrypted value is equal to the input of encryption.

For testing purposes, the policy was changed so that it allowed decryption of single ciphertext.

After testing the correctness of encryption and decryption, the policy was also tested. Aggregates created with too much data and with few values were sent to the platform for decryption. The response to aggregate with wrong identifier in the list of identifiers was also tested.
Experimental Evaluation

Experimental evaluation has several parts. The first part deals with a comparison of the execution time of implemented variants of Paillier cryptosystem. In the second part of this chapter, an attempt to evaluate implemented side channel attack countermeasures is described. The last part deals with a comparison of the VeraGreg framework and the implemented naïve scheme based on a symmetric cipher.

5.1 Measured Metrics

To compare the different implementations of Paillier cryptosystem as well as the VeraGreg framework and the naïve scheme, execution time and the memory requirements were measured, namely the size of compiled code and required amount of RAM.

The execution time measurement should have been implemented by 32-bit timer with 1kHz clock. On the CEC1302 platform, it is not possible to read whole 32 bits from the timer register, only the lowest 16 bits. This is caused probably by some bug in the platform. Due to this bug, the 32 bit timer could not be used for the time measurement.

Due to the aforementioned reasons, the time measurement is implemented by increment of volatile variable, which is triggered each milisecond. The time measurement resolution is hence 1 ms.

The mikroC for ARM profiling tools were used for the measurement of memory requirements.

5.2 Experiment setup

All metrics were measured on the implementation platform or using MicroC for ARM IDE profiler tools. No external equipment was used for the experimental evaluation.
5. Experimental Evaluation

Table 5.1: Comparison of execution times of Paillier cryptosystem.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Encryption time</th>
<th>Decryption time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montgomery</td>
<td>2162 ms</td>
<td>1503 ms</td>
</tr>
<tr>
<td>Accelerator</td>
<td>711 ms</td>
<td>1241 ms</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison of memory requirements of Paillier cryptosystem.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Code size</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montgomery</td>
<td>18807 B</td>
<td>18003 B</td>
</tr>
<tr>
<td>Accelerator</td>
<td>10055 B</td>
<td>16355 B</td>
</tr>
</tbody>
</table>

5.3 Paillier on RSA

This section deals with a comparison of the implementation of the Paillier cryptosystem on the CEC1302 platform using the software modular exponentiation and the hardware RSA accelerator.

For the measurement of the execution time, 20 random values were encrypted and decrypted. The fact that the data was generated by the platform removed the need for communication and thus speeded up the measurement.

Following from the Table 5.1, the use of RSA hardware accelerator speeds up the encryption roughly three times, when random $r$ is 960 bits long and the modulus is 2048 bits long. The standard deviation of encryption and decryption times is less than 1%.

The speedup for decryption is not so significant since modular exponentiation is only a small part of the decryption process.

Memory requirements of both variants are depicted in Table 5.2. The size of the code for the accelerated version is significantly smaller since the compiler does not include unused functions into compilation.

The implementation of Paillier cryptosystem with the hardware RSA accelerator outperforms the pure software implementation in all measured categories, mainly due to significant reduction of encryption time and program size.

5.4 VeraGreg Framework vs. Naïve Scheme

The result of a comparison of the VeraGreg framework and the naïve scheme based on AES can be easily predicted from the definition of VeraGreg encryption and decryption operations which both use AES along with modular exponentiation and integer division. The aim of the comparison is to quantify the difference, not to find out if the VeraGreg framework is faster or not.

The decryption time of VeraGreg framework depends on the number of values in the aggregate. Table 5.3 depicts time for decryption of aggregate
Table 5.3: Comparison of execution times of the naïve scheme and VeraGreg framework.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Encryption time</th>
<th>Decryption time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive scheme</td>
<td>5 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>VeraGreg</td>
<td>1032 ms</td>
<td>1504 ms</td>
</tr>
</tbody>
</table>

Table 5.4: Comparison of memory requirements of the naïve scheme and VeraGreg framework.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Code size</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive scheme</td>
<td>8032 B</td>
<td>760 B</td>
</tr>
<tr>
<td>VeraGreg</td>
<td>15055 B</td>
<td>19553 B</td>
</tr>
</tbody>
</table>

consisting of three values (ciphertexts), the standard deviation of encryption and decryption times is less than 1%. For more ciphertexts, the decryption time can be easily estimated by adding the appropriate multiple of naïve scheme encryption time, which corresponds with the time of AES encryption using CEC1302 hardware accelerator. This can be seen from Equation (1.18).

The difference in the size of the compiled code and occupied RAM in Table 5.4 is not surprising since the VeraGreg framework needs significantly more libraries, namely the `bigi` library, `rng` library and `aes_wr` library.

### 5.5 Side channel attack countermeasures

I have also attempted to evaluate the implemented side channel attack countermeasures, namely against power analysis attacks. During examination of the development board Clicker 2 for CEC1302, I have found out that it is not possible to measure the power consumption without significant interference with the board, which can possibly destroy the board.

For the purpose of power analysis attacks it is necessary to measure power consumption of the microcontroller only. The best places for doing so are microcontroller power pins. These, however, are not accessible, because the microcontroller is in a packaging with pins on the bottom and is placed on a four layer circuit board.

Another suitable place for the power consumption measurement is the board voltage regulator. The regulator LTC3586 is located on the bottom part of the board, the microcontroller is on the top. Moreover, there are six balancing capacitors connected between the voltage regulator and the microcontroller.

Balancing capacitors are used to stabilize the voltage on the power supply pins of the microcontroller. However, the little changes in voltage are the source of information for the DPA attack. For the attack to be successful,
the balancing capacitors must be removed and the working frequency of the microcontroller must be lowered.

During the removal of capacitors, there is a risk of damaging the board. As it can be seen in the Figure 5.1, the balancing capacitors are connected to the inner layers of the printed circuit board.

Another risk of removing the balancing capacitors is that the board might not be possible to program without external programmer, or the board will not be able to boot since the voltage regulator will not be able to cover the increased power consumption during this process.

![Figure 5.1: Capacitor attachment to the board](image)

5.6 Results

The use of hardware acceleration for modular exponentiation accelerates the encryption of the Paillier cryptosystem approximately three times compared to the variant implementing the exponentiation in software.

Modular exponentiation acceleration speeds up decryption 1.5 times compared to the software variant. The reason for the smaller difference is that decryption consists of multiple operations, including division, which is computationally demanding, and modular exponentiation takes smaller part of execution time than in encryption algorithm.
The comparison of the VeraGreg implementation with the naïve scheme based on the AES cipher yielded a quantification of the difference, i.e., the use of the VeraGreg framework represents a significant time and memory overhead. The time needed for encryption and decryption of VeraGreg is considerably higher than that of the naïve scheme.

When trying to evaluate the implemented protection against side channel attacks, it was found that without significant intervention in the development kit the consumption of the CEC1302 microcontroller cannot be measured appropriately. The first reason is that the printed circuit board of the development board has multiple layers and the power is distributed in the inner layers. This prevents connection to the microcontroller’s power pins. The LTC3586 voltage regulator is also mounted on the development board. Voltage regulators usually represent an unwanted source of noise that makes it difficult to identify a useful part of a signal when a DPA attack is used.

The last obstacle for a power measurement is the cascade of balancing capacitors, which serve to cover peak power consumption. After removing it there is a risk that the board could not be programmed over the USB interface, or might not even boot.
This chapter outlines possible further improvements to the implementation of the VeraGreg framework in order to deploy VeraGreg in practice.

6.1 Side channel attacks

In the future research, it would be useful to verify the effectiveness of implemented countermeasures against side channel attacks. To do this, it will be necessary to perform certain modifications on the board and test whether it is possible to measure the consumption on this board during the execution of the cryptographic algorithm.

If this is not possible, the development board with the CEC1702 microcontroller [64] can be used. Or there is an option of designing special development board with CEC1302 that is customized to easily measure the power consumption.

Another topic is the evaluation of the platform’s security, in particular that of the RSA and AES hardware accelerators.

6.2 Usable implementation

The problem that needs to be solved before putting the implementation into practice is key security, thus resolving their storage in a secure NVM, or transferring to a platform that contains secure NVM on the chip.

6.3 VeraGreg framework

Other areas are porting the implementation to other platforms, testbed automation and an automated unified multiplatform build system.
6. Future work

On the theoretical level, it is possible to look for new algorithms meeting the VeraGreg framework definition and also to answer the question whether VeraGreg is compromised, if attacker knows the SE key.
In this work, I dealt with an implementation of the VeraGreg framework on a low-cost secure microcontroller and evaluated its effectiveness in comparison to a naïve scheme based on the AES cipher.

The first step was to select a secure microcontroller. The most important criteria were price of the microcontroller and development tools, furthermore the implementation platform had to include RSA and AES hardware accelerators. Secure microcontroller CEC1302 manufactured by Microchip was selected as the implementation platform. This microcontroller has both RSA and AES accelerators and hardware random number generator. The documentation and development tools are publicly available, the microcontroller itself costs $1.93 and the development board costs $39.

In order to implement the VeraGreg framework on the CEC1302 microcontroller, it was necessary to design and implement a large number library optimized for the ARM platform. This library was named bigi and it is available as open source software. The operations used by cryptographic algorithms are implemented with respect to side channel attacks.

Furthermore, it was necessary to implement Paillier cryptosystem which is the basis of the VeraGreg scheme. Paillier cryptosystem was implemented using the library bigi and the RSA hardware accelerator. To the best of my knowledge, this is the first published implementation of Paillier cryptosystem using an RSA accelerator. Using the accelerator, encryption enjoys 3 times speedup compared to a software implementation, decryption is accelerated twice. To generate random numbers for encryption, the implementation uses a hardware random number generator which is a part of the CEC1302 platform.

The implementation of the VeraGreg framework is based on Paillier cryptosystem that employs the RSA accelerator and the random number generator. Furthermore, it uses the AES accelerator to encrypt identifiers. The entire implementation is protected against side channel attacks by hiding in consumption, hiding in time, and by random swapping of operations during encryption.
The naïve scheme, which serves for efficiency comparison, is based on the AES cipher and it uses the AES accelerator on the CEC1302 platform. This simple scheme assumes knowledge of the key to the data-processing party. The naïve scheme, unlike the VeraGreg framework, does not preserve privacy and does not give the user the option to verify the operations that were performed with her data.

Comparing the VeraGreg scheme to the naïve AES-based scheme has shown that using VeraGreg introduces a significant time and space overhead: encryption takes 200 times longer than the naïve scheme. Decryption time for VeraGreg depends on the number of values in the resulting aggregate, where each identifier of each value must be encrypted with AES. For an aggregate of 3 values, decryption takes 250 times longer compared to the naïve scheme.

The comparison shows that VeraGreg should be used in applications where privacy and verification of the operations over encrypted data are crucial for the user, since the space and time overhead is significant compared to the naïve scheme.

Evaluation of the implemented side channel attack countermeasures has not been done since there is no suitable way of measuring power consumption of the microcontroller on the development board. In order to measure power consumption, the board requires significant changes which could possibly destroy the board or prevent its correct function.
Bibliography


Bibliography


Bibliography


Acronyms

**FHE** Fully homomorphic encryption
**PHE** Partially homomorphic encryption
**AHE** Additive homomorphic encryption
**AES** Advanced Encryption Standard
**RSA** Rivest–Shamir–Adleman encryption algorithm
**NVM** Non-volatile memory
**gcd** Greatest common divisor
**ADU** Application data unit
**PDU** Protocol data unit
**ASN.1** Abstract syntax notation one
**IoT** Internet of things
Running the VeraGreg Demo

In order to compile the VeraGreg implementation and to upload it to the target platform, PC with Windows and full license for MikroC for ARM are required.

1. Import the project from enclosed electronic media
2. Compile project
3. Run the MikroE mikroprog bootloader version 2.4.1
4. Connect the board to the computer via micro USB cable
5. Select appropriate COM port in the mikroprog menu
6. Reset the platform
7. During 5 seconds after the reset press the connect button in the mikroprog window
8. Select the desired .hex file
9. Program the platform
10. After finishing programming, reset the platform again
11. Run server application and proceed according to the instructions on screen
C

Contents of the Enclosed
Electronic Media

thesis.................. The directory of \LaTeX\ source codes of the thesis
  text ........................................ Thesis text
  pics ........................................ Pictures
src ........................................ the directory of source codes
  tests ......................... Tests for all libraries, integration tests
  veragreg ...................... VeraGreg framework source
    core ...................................... VeraGreg library
    crypto ........................ Paillier cryptosystem, AES, RNG sources
    bigi ................................. multiprecision arithmetic library
    policy ................................... VeraGreg policy
  CEC1302 ........................ Projects for the MicroC for ARM IDE
    naive_experiment ............. Measurement of naïve scheme
    paillier_experiment ...... Measurement of Paillier cryptosystem
    veragreg_experiment ...... Measurement of VeraGreg framework
    veragreg_demo ............. Demo application for VeraGreg