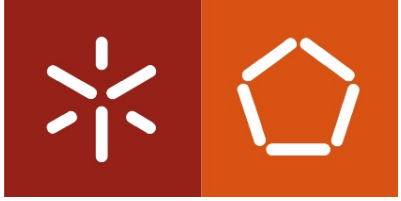
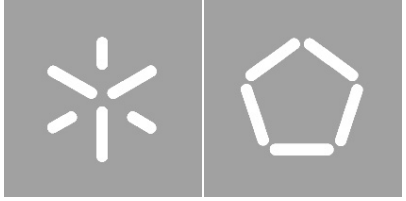


Universidade do Minho
Escola de Engenharia

Nuno Filipe Trovisco Fernandes

Cryptographic Library Support for a Certified Compiler

Janeiro de 2014



Universidade do Minho
Escola de Engenharia

Nuno Filipe Trovisco Fernandes

Cryptographic Library Support for a Certified Compiler

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor José Carlos Bacelar Almeida

Janeiro de 2014

Abstract

Cryptographic Library Support for a Certified Compiler

An essential component regarding the development of information systems is the compiler: a tool responsible for translating high-level language code, like *C* or *Java*, into machine code. The issue is, compilers are themselves big and complex programs, making them also vulnerable to failures that may be propagated to the compiled programs. To overcome those risks research on "certified compilers" has been made, and recently some proposals have appeared. That sort of compilers guarantees that the compilation process runs as specified.

In this dissertation is studied the applicability of the certified compiler *CompCert* in cryptographic software development. The first point being addressed was the use of support libraries, such as big number libraries. As a matter of fact, such libraries are an essential requisite for the considered type of application, therefore the study of different options for using these libraries, always considering the impact in program's performance and semantic preservation offered by the compiler.

The second point being addressed was the use of SIMD extensions available on recent processors. Here the objective was to demonstrate how one could overcome the current *CompCert*'s limitations as to discuss other solutions.

Keywords: *GMP*, *LIP*, certified compilers, cryptography, mathematical operations, big number libraries, cryptographic algorithms, proofs, *gcc*, *compcert*, *coq*, *AES*, *SSE*, processor extension, Intel

Resumo

Bibliotecas de Suporte Criptográfico para um Compilador Certificado

Um componente essencial na produção de sistemas informáticos é o compilador: a ferramenta responsável por traduzir o código numa linguagem de alto nível como o C ou o Java em instruções do processador que serão efectivamente executadas. Mas os compiladores são eles próprios programas grandes e complexos, vulneráveis a falhas que se podem propagar de forma incontrolável por todos os programas por eles processados. Com o objectivo de ultrapassar esse risco surgiram recentemente as primeiras propostas de "compiladores certificados" onde se garante que o processo de compilação está conforme o especificado.

Nesta dissertação é estudada a aplicabilidade do compilador certificado *CompCert* no desenvolvimento de software criptográfico. O primeiro aspecto abordado foi a utilização de bibliotecas de suporte, como as bibliotecas de números grandes. De facto, tais bibliotecas são um requisito essencial para tipo de aplicação considerado, estudando-se por isso diferentes alternativas para a utilização dessas bibliotecas, considerando quer o impacto na eficiência dos programas, quer as garantias de preservação semântica oferecidas pelo compilador.

Um segundo aspecto abordado foi a utilização de extensões SIMD disponibilizadas pelos processadores mais recentes. Aqui o objectivo foi o de mostrar como é possível ultrapassar as limitações da versão actual do *CompCert*, assim como discutir soluções mais abrangentes ao problema.

Palavras-chave: *GMP*, *LIP*, compiladores certificados, criptografia, operações matemáticas, bibliotecas de grandes números, algoritmos criptográficos, provas, *gcc*, *compcert*, *coq*, *AES*, *SSE*, extensões do processador, Intel

Contents

List of Figures

List of Tables

1 . Introduction

1.1 Context

There are several important factors to sustain while programing, but one of them not only becomes transparent to the programmer but a little neglected: the compiler. An excellent programmer may fail if the compiler has bugs. In a cryptographic context this matter became extremely relevant since bugs could propagate themselves to the final program, possibly compromising not only its behavior but also some vital information, specially when developing cryptographic software where information integrity and security are some of the main concerns. It's in situations like these that compilation bugs and errors must be avoided at all cost.

With this thought in mind, the need of a compiler capable of ensuring that the result of the compilation process was no other than the one expected, became a priority. And so researchers came with the concept of certified compilers, which basically consists of a compiler with the ability of generating assembly code semantically equivalent to its source program, and its correctness is entirely proved within a proof assistant [3]. There are some proof assistant systems (*ACL2*, *Agda*, *Coq*, *HOL Light*, *HOL4*, etc...) but in this case the focus goes to a particular one, *Coq*. It deserves special attention because it was used to develop a certified compiler named *CompCert*, which in its turn, is a crucial element in this dissertation study.

As it was already stated, cryptographic software development requires some special attention. This same type of development usually makes use of big number libraries, since not only it involves working with numbers which require a lot of bits (i.e: 2048) for their representation, but also because such numbers are used in complex mathematical operations so cryptographic operations could be performed (i.e: RSA and exponentiation). Following this reasoning, one can not start studying these kind of libraries without first analyzing the mathematical algorithms themselves by looking at their classical pseudocode

implementation and their respective asymptotic analysis. In a practical point of view, one library (*GMP*) was immediately chosen because of its completeness and performance. *GMP* evenly balances performance, both in time and in memory, with the amount of operations it provides on both mathematical and cryptographic contexts.

By using it, one's able to perform some basic arithmetic operations (i.e: addition, subtraction, multiplication), others a little bit more complex (i.e: exponentiation, great common divisor) and also some advanced ones (i.e: modular multiplication, modular exponentiation), obviously aside from many more. Of course that some of them consist only on combining more than one operations, nevertheless this only reinforces its completeness. Remaining only to chose another library to compare with *GMP* the choice fell on *LIP*, which is a library that offers a pretty reasonable complete, yet narrower, range of functions with a relatively good performance.

Finished all the research, analysis and benchmarking, it was time to use a library called *TrustedLib* in order to give an extra support to *CompCert*, which basically consisted in a *Coq* implementation of some low-level *GMP* functions, so the certified compiler would trust them, thus complementing its certification guarantees.

Having already been referred a few times, performance plays a key role practically in all areas, and off course cryptography is also one of them. In the best case scenario performance and security would walk side by side, unfortunately this is not the case. Encrypting/decrypting a text, generating a key/set of keys are operations that take their time, typically depending on the text size and number of key bits, respectively. A good way to achieve good performance, without explicitly using parallelism, is to resort to assembly instructions.

On 2010 Intel made a great advance on this field by natively implementing on their processors, assembly functions to perform AES encryption. Depending on how the instructions were used, several modes of encryption (like ECB, CBC and CTR) could be implemented. With this the opportunity to expand the study to a different, yet still connected area, came along. The goal was to develop a library, by making use of such assembly functions, so it could be possible to perform CBC, ECB and CTR encryption and decryption algorithms.

Also, the library would have to be modular enough so a program could be compiled by *CompCert* despite

of its limitations.

1.2 Objectives

This project covers some areas (cryptography, formal methods, program certification, computer architecture) that besides distinct could easily relate to each other. To do so the following objectives were defined:

- understand the concept of certified compilers, and their importance in cryptographic software development;
- search and chose a big number library to compare with *GMP*;
- evaluate *CompCert*'s support for libraries used in cryptographic operations, such as big numbers libraries, and the corresponding impact on its efficiency;
- extend *CompCert*'s semantic preservation result to make it aware of "trusted" external libraries for (a core) big number functionality;
- study the impact of recent SIMD assembly instructions, available in the new generation of processors, in cryptography particularly how they can be exploited in *CompCert* based developments.

1.3 Structure

Chapter 2 introduces not only the concepts of certified compilation and certified compilers but some detailed information on each certification step as well. It also presents some basic concepts of how *CompCert* was developed (used language to be more precisely) and a perspective from a user point of view regarding its limitations, usage, and even some misc information whose purpose is none other than to demonstrate the effort needed to develop a compiler of this kind.

Chapter 3 is devoted to big numbers. It contains all the information about big number libraries support. Beginning by mentioning the analysis of mathematical algorithms, and then presenting the results obtained by comparing *GMP* and *LIP* performance. To do so a *GMP* benchmark (GMPBench v0.2) was used. Obviously it was necessary to slightly modify the benchmark's source code so it could be used by the *LIP* library. The goal is to compare the programs performance when compiled with *GCC* (v4.7.3) and *CompCert* (v2.0), and then determine the impact of certified compilation. It's also considered the case where the *LIP* source code is compiled with *CompCert*, in order to compare the performance of a certified program when using a certified library. Finally are presented all details on how *TrustedLib* was used to extend *CompCert* so it could support some of *GMP* core functions.

At last but not least, chapter 4 states all the details, which involves both process and achieved results, on how the latest AES-SSE Instruction Set extensions were used to develop a cryptographic library. Such library consisted in combining the available SIMD extensions datatypes and assembly functions, and all together obtain three modes of AES encryption and decryption.

2 . Certified Compilation

Before presenting the most important concepts and details about certified compilation, it's important to begin with the basics. In this case such basic concept is none other than the *COQ* proof assistant. The reason why, it's because it was the chosen programming language to develop the certified compiler called *CompCert*, which will be an important subject of study in this project.

2.1 COQ

COQ is a proof assistant system implemented in *OCaml*. It implements a program specification and mathematical higher-level language called *Gallina*.

With this system it's possible to define functions (it only accepts functions that terminate otherwise it will give an error), predicates (that can be evaluated efficiently), state software specification, and via an enormous set of tactics it is possible to define theorems and lemmas. It is then possible to interactively develop formal proofs of these theorems. Such proofs can then be machine-check by a relatively small certification *kernel*.

There are two ways to use the system, by using the command *coqtop* to interactively construct proofs, or to write them in a file and then call the compiler using the command *coqc*. Just like common systems, *COQ* has some extensions. A particular important one is called *SSREFLECT*[1]

2.1.1 Examples

Defining a *Lemma*:

```
Lemma weak_peirce : ((((P->Q)->P)->P)->Q)->Q.
Proof.
intros H.
```

[1] http://www.msr-inria.inria.fr/Projects/math-components

```
6
apply H.
intros H0.
apply H0.
intros H1.
apply H.
intros H2.
apply H1.
Qed.
```

Defining a *Theorem*:

```
Theorem P_Q_nQ_nP : forall(P Q:Prop), (P->Q)->~Q->~P.
Proof.
intros P Q H H0.
intros H1.
apply H0.
apply H.
assumption.
Qed.
```

Let's see a specific example. Consider a function that tells if a given month has an even number of days. To do this, first a new inductive type must be defined:

```
Inductive month : Set :=
January : month | February : month | March : month | April : month |
May : month | June : month | July : month | August : month | Septembre : month |
Octobre : month |  Novembre : month | Decembre : month.
```

Now the only thing missing is the function definition. Since February could have either 28 or 29 days, the user must tell if the year is a leap year or not:

```
Definition month_even (leap:bool) (m:month) : bool :=
match m with
| February => if leap then false else true
| April | June | September | November => true
| _ => false
end.
```

Another example is a function that counts the number of times a natural number appears in a list. But this time we need to use recursion:

```
Fixpoint count_list (n:nat) (l:list nat):=
match l with
| nil => 0
| h::t => if beq_nat n h then 1 + count_list n t else count_list n t
end.
```

beq_nat is a native *COQ* library function that compares two natural numbers and returns a boolean value. To test these function the following can be done:

```
Eval compute in (month_even true February).
Eval compute in (count_list 2 (1::2::3::1::2::3::1::2::3::nil) ).
```

2.1.2 Final Remarks

The main goal of this section was neither to teach the *COQ* language or to present the available tactics to prove theorems/lemmas. It was only to give a basic notion of how it works, and more importantly, to

introduce now some syntax since later in this document more code will be shown. As it was said earlier, the specific details about certified compiler development will be presented next.

2.2 Certified Compilers

Compilers are seen as a black-box, but in reality they are a piece of complex software and vulnerable to bugs. A compiler error that could take a lot of time to be discovered when using "normal" compilation, would take less time if trusted compilation was used instead.

Formally a compiler is a function that for a given source either returns nothing (in case of an error) or a compiled code:

$$Compiler : Source \rightarrow CompiledCode \cup \{None\} \tag{2.1}$$

When developing critical/high-assurance and security relevant applications, the compiler has an important and relevant role. With this in mind the use of formal methods like model checking and deductive verification is extremely important and necessary. Such methods are applied to the source code in order to guarantee some safety properties.

Although developing such kind of compilers is hard, mostly because of the lack of automated proof mechanisms, the outcome is a totally fail-safe applications compiler.

2.2.1 Correctness Property

The concept of simulation has been established with the purpose to maintain the program translation process transparent. More precisely, the compilation process as to simulate the source language behavior. There's a simulation relation between the source and the compiled code. Such relation is established by the compiler and its made once. For every behavior in the source exists an equivalent behavior in the compiled code and vice-versa.

Next there are five correctness properties between the source and the compiled code that must be

met in order to ensure the application safety properties:

- Source and compiled code are observationally equivalent;
- If the source has well-defined semantics, then source and compiled code are observationally equivalent;
- If the source has well-defined semantics and satisfies some functional specification, then the compiled code also satisfies that same specification;
- If the source is type-safe and memory-safe , then so is the compiled code;
- Compiled code is type-safe and memory-safe;

Generally the certification of a compiler guarantees that all safety properties, previously proved in the source code, must hold for the compiled code as well.

2.2.2 Certified Compilation

Certified compilation ensures that a program runs accordingly to its behavior. To do so it is needed to introduce the concept of certified/verified compilers, since it's through this that certified compilation its attainable.

2.2.2.1 Certified Compilers

Regarding this definition a compiler is a function that for a given source code returns some compiled code that satisfies a desired correctness property:

$$\forall s \in Source, c \in CompiledCode : Compiler(s) = Some(c) \Rightarrow Property(s, c) \tag{2.2}$$

On a certified compiler both algorithm and implementation are verified, which is perfectly suited to compile critical software systems. In the end, if all proofs are correct and complete, then the result is

a bug-free compiler. Although it seems easy, implementing this kind of compiler is a difficult and long process. If by any chance the proving process can be automated, putting up the proof obligations for each transformation (reminding that every single transformations requires a proof) is still a manual process.

Given this hindrance it became necessary to explore other ways to achieve certified compilation.

2.2.2.2 Certifying Compilers

It is known that its easier to prove the correctness of a computation result rather than the compilation algorithm itself. This was the start-point to the concept of certifying compilers which basically consists on equip a compiler with a certifier that checks the result of a translation. If by any chance the returned proof doesn't work, it can be checked in a second step:

$$\forall s \in S, c \in C, \pi \in Proof : CComp(s) = Some(c, \pi) \wedge \pi correct \Rightarrow Prop(s, c) \qquad (2.3)$$

2.2.2.3 Proof-carrying Code

A solution that uses a certifying compiler its the concept of proof-carrying code (PCC). The idea is to eliminate the need for trust between code consumer, code provider and a secure communication. In order to do so it is the code that carries a proof on predefined safety properties. PCC does not make a statement about who generates the proof, it can be a compiler or even by hand. In case of a malicious code or code manipulation, when trying to execute the code the proof would fail, stoping the program's execution. The same happens for proof manipulation. This method comes with an overhead, which refers to the first execution, where the proof needs to be checked. After such verification the code runs without any performance drawback.

Concerning compilation, the program's execution approach works only for properties that are checked during the compilation process and can be used to detect bugs in a compiler. Although it can not guarantee a correct translation or even a simulation relation between source and compiled code.

2.2.2.4 Translation Validation

In the translation validation approach the compiler function presented earlier, is now complemented with a verifier:

$$\forall s \in S, c \in C : Comp(s) = (Some(c) \wedge Verify(s,c) = true) \Rightarrow Prop(s,c) \tag{2.4}$$

Such verifier consists in two components: an analyzer and a proof checker (a simple theorem prover). The analyzer receives both source and compiled code, it then finds corresponding sections on the code and generates proof scripts. Such scripts are then sent to the proof checker in order to verify their correctness.

Given the fact that each compilation result is automatically checked against the source, compilation errors can be detected almost immediately. The translation validation method is very flexible because it can verify any kind of translator, whether it is a complete compiler or a single compiler pass. It monitors each transformation and ensures that the compiled code still simulates the source code. If such relation can not be verified the transformation is aborted, otherwise, there is a correct simulation of the source and therefore certified compilation is achieved.

Given this, it is possible to say that not every correct program must verify, but every verified program must be correct. The only downside of translation validation is that correct translations may be rejected due to deficits in the verifier implementation.

2.2.3 Back-end

This section is divided in two, the first half explains the intermediate languages used in *CompCert* whilst the second one explains the translation steps between those languages. The following image shows the intermediate languages and their translation flow.

2.2.3.1 Languages

The first source language is *Cminor*, which is a simplified and low-level language based on *C*. The basic structure is expressions, statements, functions and programs. The *Cminor* possesses a very weak typing

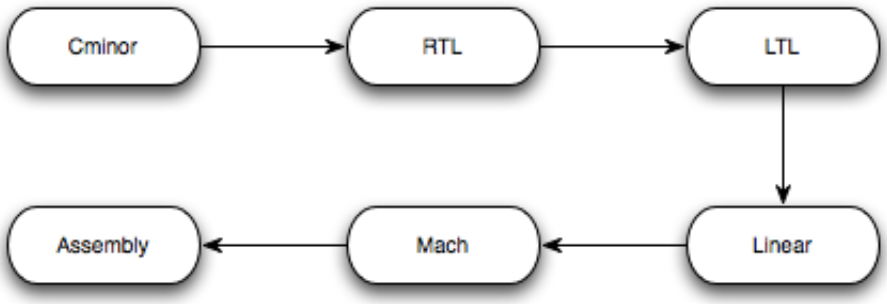


Figure 2.1 Languages and their translation flow

system, with only *int* and *float* types, although it is useful for later transformations (see next section), it is also very susceptible to run time errors, therefore the programmer must be very careful. Besides this, function definitions/calls are annotated with signatures which gives information about the number and type of arguments and an optional type for the result. An important remark must be made, there are two types of *Cminor*, the *external* and the *internal* one. The first one is processor independent and it's the front-end interface. The second one refers to the back-end and includes a set of operators that reflect what the target processor can do, in other words, processor dependent. The conversion from external to internal is automatically made.

The first intermediate language is *RTL (Register Transfer Language)*. Here functions are represented as a control-flow graph (*CFG*) of abstract instructions operating on (temporary) pseudo-registers. Each and every function has an unlimited supply of these registers. The nodes of the *CFG* consist in individual instructions, instead of basic blocks, because it simplifies semantics without slowing down the compilation process. This language has the same trivial and static typing system as *Cminor*. From now on, several variations of *RTL* will be used as intermediate steps towards the final goal, the assembly code. This whole process will progressively refine the notion of pseudo-register until they are mapped to hardware registers and stack slots.

In this next language, called *LTL (Location Transfer Language)*, control is still represented by a graph but now the nodes contain basic instructions blocks. This change is due to the fact that the transformations from *RTL* to *LTL* insert reload and spill instructions, and such insertion is easier to perform on instruction

blocks instead of single instructions. Stack slots are not yet mapped to memory locations, instead there is a mapping of locations to values. This happens because distinct slots may overlap, what could later result in an overlap between memory areas.

The *Linear* is a variant of the *LTL*. Here the *CFG* and the basic blocks are replaced by a list of instructions with explicit labels and branches, where non-branch instructions continue at the next instruction in the list. This language has three infinite supplies of stack slots, namely local, incoming, outgoing. This slots will later be mapped to actual memory locations.

The last intermediate language is called *Mach*. It's the last step before the assembly code be generated. This language is a variant of *Linear*, where the three infinite supplies of stack slots are mapped to actual locations on the stack frame. The local and outgoing slots go to the callee stack frame while the incoming slot goes to the caller stack frame. All hardware registers are global and shared between caller and callee.

2.2.3.2 Language's Translation Steps

The first of several translation steps is the translation from external *Cminor* to internal *Cminor*, which is called instruction selection and re-associaton. It matches the processor known operations and addressing modes. The operators that do not correspond to a processor instruction are accordingly encoded.

The next phase is the *RTL* generation, which consists on the translation from internal *Cminor* to *RTL*. Here, the *Cminor* structured control is encoded as a flow graph. Every expression is decomposed into sequences of *RTL* instructions, such decomposition is easily made because of the previous conversion from external to internal *Cminor*, where every operation became one instruction. Then pseudo-registers are generated and they'll hold the values of *Cminor* variables, as well as the values of intermediate results of expression evaluations. The major difficulty here is the need to generate "fresh" pseudo-registers as well as "fresh" *CFG* nodes and fill them with new instructions and branch them properly to another graph node.

The translation from *RTL* to *LTL*, called register allocation, is based on coloring an inference graph. Such graph is obtained by inverting the edges of the *CFG*. There are two ways to express these interferences,

it can refer to an interference between two pseudo-registers, or, an interference between a pseudo-register and an hardware register. Although this coloring procedure isn't certified, the returned coloring is. This highlights the fact that it's simpler to verify the result rather than the computation. Being more specific, the procedure itself consists in:

1. Creating an interference graph. Preference edges (meaning that two pseudo-registers should preferably be allocated to the same/to a specific location) are also recorded. This extra information has no impact on the correctness of the register allocation, only in its quality;

2. Perform type reconstruction on the input code. It basically consists in associating an *int/float* type to every pseudo-register. Graph coloring will then choose the hardware registers and stack slots of the appropriate class;

3. Coloring of the interference graph, which then returns a mapping from pseudo-registers to locations;

This translation mechanism replaces references to pseudo-registers by the reference to their mapping. Also the values of all registers are preserved by the registers allocation process, although this does not hold for dead registers, for the simple fact that their values are totally irrelevant.

After the register allocation phase comes the linearization phase, where a translation from *LTL* to *Linear* is performed. This step basically linearizes the *CFG* by:

1. Rewriting the *CFG* by eliminating branches-to-branches ("*sequences of branches with no intervening computations*"). This elimination process is know by control flow graph tunneling [2];

2. Producing an ordered list that contains the enumeration of the reachable nodes of the graph;

3. Putting the *CFG* instructions in a list according to the previously established order. It adds a *goto* to every instruction that points to the label of its successor in the *CFG*. If a *goto* branches to an immediately following label, such *goto* is eliminated.

[2] `http://compcert.inria.fr/doc-1.6/html/Tunneling.html`

The *Linear* to *Mach* translation consists on laying out the stack frame. It's the only time where changes in the memory layout occur. This peculiarity makes this step very hard to prove. The compiler plays an important role, since its job is to compute the number of slots of each kind and the number of callee-save registers that need to be used. In order to do so it scans the *Linear* code and then determines the size and layout of the stack frame. After this, references to stack slots are translated into actual $loads$ and $stores$. Besides this, it also adds function prologues[3] and epilogues[4] so they can, respectively, save and restore the values of used callee-save registers. It ensures that, if nothing fails (program doesn't get stuck at runtime), locations of type *int* always contain either *integer* or *pointer* or *undef* values, and that locations of type *float* always contain either *float* or *undef* values.

The last step is the generation of assembly code. It translates the *Mach* code to *assembly*. This simple process consists in an expansion of *Mach* instructions, operators and addressing modes to instruction sequences.

2.2.4 Final Remarks

Compilation is a critical step in software development. As it was seen certified/verified compilation is a better and safer solution. The downside is that it is very hard to achieve. It implies a whole reformulation in compilers development to incorporate the use of formal methods.

CompCert[5] is a verified compiler that uses the *Coq*[6] proof assistant. For more details on this matter, [3] and [7] might be a very good reading. Now that the technical details have been introduced, it is time to see *CompCert* from a user point of view. The next section contains precisely that.

[3] Assembly language specific code lines placed at the beginning of a function. They prepare the stack and registers.

[4] Placed at the end of a function. Used to restore the stack and registers to the previous state (before function call).

[5] `http://compcert.inria.fr`

[6] `http://coq.inria.fr/`

2.3 CompCert

Technical details aside, it is now time to see *CompCert* from a user point of view. The next section intents to show that. Other details can bee seen in [4]

2.3.1 Limitations

According to the official website, *CompCert* supports all of ISO C 99 standard specifications, with some exceptions:

- switch statements must be structured as in MISRA-C; unstructured "switch", as in Duff's device, is not supported.
- Unprototyped function types are not supported. All functions must be prototyped;
- Variable-argument functions cannot be defined.;
- *longjmp* and *setjmp* are not guaranteed to work;
- Variable-length array types are not supported;
- Designated initializers are not supported;
- Consequently, *CompCert* supports all of the MISRA-C 2004 subset of C, plus many features excluded by MISRA (such as recursive functions and dynamic heap memory allocation).

Besides these, it also has some limitations regarding flags passed in compilation time, and it can only produce code for IA32 (x86 32-bits) architectures.

Still according to the website, several extensions to ISO C 99 standard are indeed supported:

- The *_Alignof* operator and the *_Alignas* attribute from ISO C2011.
- Pragmas and attributes to control alignment and section placement of global[7] variables.

[7] only in a global context, local variables are not assured

2.3.2 Misc Information and Further Improvements

In order to give a better perspective about how hard it is to write a certified compiler, the numbers describing *CompCert*'s constitution were also retrieved from the official website: 16% of code, 8% of semantics, 17% of claims, 53% of proof scripts and 7% of miscellaneous information. It's not much of a surprise that more than a half of the compiler consists on proof scripts. Now, how big can this 53% can be? Well, based on information retrieved from the website, it's about 50 000 lines of *Coq* code.

Besides its limitations, some improvements could also be done:

- Low proof automation;
- More assurance;
- More optimizations;
- Shared-memory concurrency;
- Connections with hardware verification;
- Support other source languages (i.e: Object Oriented);
- Verifying program provers and static analyzers.

2.3.3 Final Remarks

The purpose of this section was to present *CompCert* from an end user perspective, mainly by exhibiting some of its limitations. Of course that since it's still in development, new functionalities are being added. Meanwhile, expecting it to support everything that, for example, *GCC* supports is a long shot bet. It's not impossible, but it's an extremely hard work. As an example, simply adding new types of data and/or features, will likely imply a great effort on revisiting the respective files and implement proofs regarding correctness, memory preservation and more. There are cases (as will be seen later in this document) where adding a specific feature would imply reproducing such modification until assembly level conversions. As

was seen, numbers speak for themselves and developing a compiler with this kind of reliability requires a tremendous amount of effort. Curiosities apart and moving towards another phase of this dissertation project, the next chapter describes all work and investigation done regarding cryptographic library support for a certified compiler.

3 . Cryptographic Library Support

3.1 Efficient Mathematical Algorithms

The following chapter consists of an overall view about some of the most heavier mathematical operations. All of them refer to multiple-precision arithmetic. These algorithms were taken from [5]. Before introducing the algorithms themselves, it's important to show the numbers representation.

3.1.1 Radix Representation

Although there are innumerous ways to represent a number, the most common are base/radix 10 (also known as decimal) and base/radix 2 (also known as binary). While the first one is most used in math, science and "normal life", the other is more suited for machine computations. For example, if $x = 1724$ in a radix 10 representation, it means that $x = 1 * 10^3 + 7 * 10^2 + 2 * 10^1 + 4 * 10^0$. In a radix 2 representation, $x = 1724$ equals $x = 11010111100$, which means that $x = 1 * 2^{10} + 1 * 2^9 + 0 * 2^8 + 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$. More formally, a positive integer number *x* can be represented as $x = a_n * b^n + a_{n-1} * b^{n-1} + ... + a_1 * b + a_0$, each integer digit being between 0 and b, and $a_n \neq 0$. This is called the *radix b* representation of x. On the next algorithms, this same representation is written as $x = (a_n a_{n-1} ... a_1 a_0)_b$. The signed-magnitude representation will be used to represent any negative integer number.

3.1.2 Multiplication

Not different from the basic and traditional pencil and paper method. Consider two positive integer numbers x and y in a radix b representation. The product $w = x * y$, with $x = (x_n, ..., x_0)$, $y = (y_t, ..., y_0)$,

and $w = (w_{t+n+1}, ..., w_0)$, is calculated by the following algorithm:

do i = 0, t $\{$

$c = 0$

do j = 0, n $\{$

$(uv) = w_{i+j} + x_j * y_i + c$

$w_{i+j} = v$

$c = u$

$\}$

$w_{i+n+1} = u$

$\}$

return(w)

Computational efficiency: This algorithm requires $(n+1)(t+1)$ multiplications. These, lets say, small multiplications are performed in single precision.

3.1.3 Squaring

Consider one positive integer number x in a radix b representation. The squaring $w = x * x$, with $x = (x_{t-1}, ..., x_0)$ and $w = (w_{2t-1}, ..., w_0)$, is calculated by the following algorithm:

do i=0, t - 1$\{$

$(uv) = w_{2i} + x_i * x_i$

$w_{2i} = v$

$c = u$

do j = i + 1, t - 1 $\{$

$(uv) = w_{i+j} + 2x_j * x_i + c$

$w_{i+j} = v$

$c = u$

$\}$

$w_{i+t} = u$

$\}$

$(uv) = w_{2t-2} + x_{t-1} * x_{t-1}$

$w_{2t-2} = v$

$w_{2t-1} = u$

return(w)

Computational efficiency: This algorithm requires $(t^2+t)/2$ single precision multiplications, which is close to one half of the ones needed by the multiplication algorithm. Generally speaking, performing a squaring instead of a multiplication can bring a speedup factor of 2.

3.1.4 Division

while ($x \geq yb^{n-t}$) $\{$

$q_{n-t} = q_{n-t} + 1; x = x - yb^{n-t}$

$\}$

do i = n, t+1, -1 $\{$

if($x_i == y_t$)$\{$ $q_{i-t-1} = b-1$ $\}$

else$\{$ $q_{i-t-1} = \lfloor (x_i b + x_{i-1})/y_t \rfloor$ $\}$

while ($q_{i-t-1}(y_t b + y_{t-1}) > x_{i-1}b + x_{i-2}$)$\{$

$q_{i-t-1} = q_{i-t-1} - 1$

$\}$

$x = q_{i-t-1} y b^{i-t-1}$

if($x < 0$)$\{$ $x = x + yb^{i-t-1}; q_{i-t-1} = q_{i-t-1} - 1$ $\}$

$\}$

$r = x$

return(q, r)

Computational efficiency: This algorithm requires at most $n-t$ single precision divisions.

3.1.5 Modular Multiplication

3.1.5.1 Classic

This algorithm combines the previous multiple precision multiplication and division. The result, $x*y$ mod m, is obtained by first multiplying x and y and then divide the result by m. The division's reminder is the algorithm's result.

Computational efficiency: This algorithm requires $(n+1)(t+1)$ single precision multiplications and $n-t$ single precision divisions.

3.1.5.2 Montgomery Reduction

$A = T$
do i=0, n-1{
$u_i = a_i m'$ mod b
$A = A + u_i m b^i$
}
$A = A/b^n$
if($A \geq m$){$A = A - m$}
return(A)

Computational efficiency: This algorithm requires $n(n+1)$ single precision multiplications and 0 single precision divisions. Not only it is faster than the previous one, it also takes advantage when the input values are the same (squaring).

3.1.6 Greater Common Divisor

3.1.6.1 Binary (Euclidean based)

Given two numbers x and y, so that $x \geq y$, the $gcd(x,y)$ is computed by the following algorithm:

g = 1
while(is_even(x) && is_even(y)){
    $x = x/2$
    $y = y/2$
    $g = g/2$
}
while($x \neq 0$){
    while(is_even(x))
        $x = x/2$
    while(is_even(y))
        $y = y/2$
    $t = |x-y|/2$
    if($x \geq y$){$x = t$}
    else{ $y = t$ }
}
return ($g * y$)

Computational efficiency: If numbers x and y are in radix 2, all divisions by 2 become right-shifts, which are extremely fast.

3.1.6.2 Binary Extended (Euclidean Extended based)

Given two numbers x and y, so that $x \geq y$, the $gcd(x,y)$ is computed by the following algorithm:

g = 1

while(is_even(x) && is_even(y)){

$x = x/2; y = y/2; g = g/2$

}

$u = x; v = y; A = 1; B = 0; C = 0; D = 1$

while ($u \neq 0$) {

while (is_even(u)){

$u = u/2$

if(((A mod 2) $== 0$) && ((B mod 2) $== 0$)){ $A = A/2; B = B/2$ }

else{ $A = (A+y)/2; B = (B-x)/2$ }

}

while(is_even(v)){

$v = v/2$

if(((C mod 2) $== 0$) && ((D mod 2) $== 0$)){ $C = C/2; D = D/2$ }

else{ $C = (C+y)/2; D = (D-x)/2$ }

}

if($u \geq v$){ $u = u-v; A = A-C; B = B-D$ }

else{ $v = v-u; C = C-A; D = D-B$ }

}

$a = C;$

$b = D;$

return($a, b, g * v$)

Computational efficiency: The only multiple precision operations are addition and subtraction. Divisions by 2 are right-shifts. As the number of bits needed to represent u and/or v decreases at least by 1, the number of iterations taken by the algorithm is at most 2(lg x + lg y + 2).

3.1.7 Exponentiation

3.1.7.1 Sliding Window

Given g, $e = (e_t, ..., e_0)$ in base 2 representation with $e_t = 1$, and a window size $k \geq 1$. The algorithm computes g^e.

$g_1 = g$
$g_2 = g^2$
do i=1, $(2^{k-1} - 1)$ { $g_{2i+1} = g_{2i-1} * g_2$ }
$A = 1$
$i = t$
while ($i \geq 0$) {
    if($e_i == 0$) {
      $A = A^2$
      $i = i - 1$
    }
    else {
      find longest bit string $(e_i, ..., e_l)$ that verifies $((i - l + 1 \leq k)$ && $(e_l == 1))$ {
          $A = A^{2^{i-l+1}} * g(e_i, ..., e_l)$
          $i = l - 1$
      }
    }
}
return A

Computational efficiency: It requires, at most, $(2^{k-1} - 1) + (l - 1)$ multiplications and $1 + (lk + h_l)$ squarings.

3.1.7.2 Montgomery Exponentiation

Given $m = (m_{l-1}, ..., m_0)$, $R = b^l$, $m' = -m^{-1}$ mod b, $e = (e_t, ..., e_0)$ in base 2 representation and with $e_t = 1$. The result is x^e mod m.

$\tilde{x}$ = $MontMul(x, R^2$ mod $m)$

$A = R$ mod m

do i=t, 0, -1 $\{$

$A = MontMul(A, A)$

if($e_i == 1$)$\{$ $A = MontMul(A, \tilde{x})$ $\}$

$\}$

$A = MontMul(A, 1)$

return A

Computational efficiency: This algorithm requires $2l(l+1) + 3tl(l+1) + l(l+1)$single precision multiplications, and $\frac{7}{2}l$ Montgomery multiplications.

3.1.8 Final Remarks

As it was possible to see, and a little bit of expected, not all algorithms possess the same computational efficiency. This is due to many facts, being the main ones, the type of operation being either basic like multiplication or more complex like exponentiation, and at last but not least, more than one algorithm to perform the same operation, for example the classic and Montgomery algorithms that perform the modular multiplication, which is the result of improving the performance of prior algorithm(s). With all of this mathematical algorithms, the hardest part is done, remaining only the coding, which varies depending of the chosen programming language, and the machine specs in which the code will be executed. All of these are important factors to take in account.

Introduced the most important and relevant arithmetic algorithms and their execution time, it is now time to show the benchmark results and compare the chosen big number libraries performance regarding the compiler (normal and certified).

3.2 Cryptographic Libraries Benchmarking

The *GMP*[1] and *LIP*[2] where the chosen libraries to analyse. The GNU Multiple Precision arithmetic library is a vast, complex and an optimized implementation of various mathematical operations. The *LIP* library can be called a lightweight version of *GMP*. The objective is to analyse the impact of certified compilation versus normal compilation. A benchmark consisting of performing some heavy mathematical and cryptographic operations involving numbers with a significant size (number of bits) was used.

3.2.1 Experimental Setup

Some modifications were made to the GMPBench code so it could be used by the *LIP* library. Such modifications include both type definitions and function definitions. The parameters order is different (e.g. in *GMP* the variable that stores the result is the first argument, while in the *LIP* is the last one).

The number of bits of program's input was also changed to a more adequate one. Table 3.1 shows the most relevant modifications as well as the functions correspondence.

[1] `http://gmplib.org/`

[2] `http://www.win.tue.nl/~klenstra/lip.html`

GMP	LIP
mpz_t	verylong
mpz_urandomb	zrandomb
mpz_tdiv_q	zdiv
mpz_gcd	zgcdeucl
mpz_mul	zmul
mpz_setbit	zsetbit
mpz_mod	zmod
mpz_powm	zexpmod
mpz_sub	zsub
mpz_add	zadd
mpz_cmp_ui	zcompare
mpz_invert	zinvmod
mpz_init/clear	verylong var assigned to 0

Table 3.1 Differences between *GMP* and *LIP*.

The following piece of code is extracted from the benchmark and it is used to determine the number of iterations, which means, the number of times that the algorithm will be executed. This value considers the function elapsed time. The fact that this is not a fixed value ensures a more fair benchmarking, because it considers different execution times for the several algorithms.

```
TIME (t, mpz_tdiv_q (z, x, y));
  printf ("done\n");

  niter = 1 + (unsigned long) (1e4 / t);
```

An example of how the time required to run a function is measured:

```
  t0 = cputime ();
```

```
for (i = niter; i > 0; i--)
  {
    mpz_tdiv_q (z, x, y);
  }
ti = cputime () - t0;
printf ("done!\n");

ops_per_sec = 1000.0 * niter / ti;
```

3.2.1.1 Machine Specifications

The performance tests were conducted in a machine with the following specifications:

Manufacturer: Apple

Model: MacBook6,1

CPU: Intel Core 2 Duo P7550 (Penryn) @ 2.26GHz

Main memory: DDR3 PC3-8500, 4GB with 14ns latency

CPU #cores / #threads: 2/2

CPU peak FP performance: 18 GFLOPS

Cache details : L2 3072 KB

3.2.2 Results

The following charts show the obtained results after running the benchmark three to eight times, and the selected results are in a range no larger than 5% of the other 2 best values. They contain the cases where the programs using the *GMP* library were compiled with *GCC* and *CompCert*. The same for the *LIP*

library. Both libraries were compiled with *GCC*. Only the charts shown in the *LIPCERT* sub-section contain the results for the *LIP* compiled with *CompCert*.

Example of parameters interpretation:

multiply 32 32 - multiplies a 32-bit random number for a 32-bit random number;

multiply 32 - squares a 32-bit random number;

rsa 32 - signs a 32-bit random message.

3.2.2.1 Division Algorithm

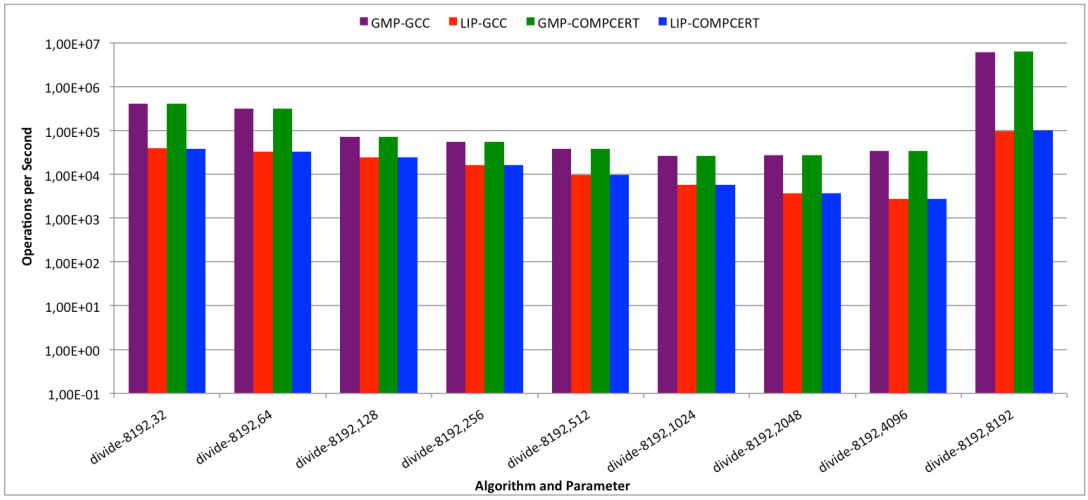


Figure 3.1 Divide

As expected, *GMP* achieves better performance than *LIP*. Increasing the size of input data, in this case the number of bits, there is a performance decrease. For *GMP* this happens until dividing a 8192 bit number for a 2048 bit number, since in the next two cases there is a improvement, especially in the last case, where performance achieves excellent results. The same thing happens using *LIP*, but this time only in the last test. For the discrepancy noticed on the last case, some other experience was made consisting on executing the division algorithm with the same range of values, but both input values were the same (*divide x,x*, where x varies from 32 to 8192). The results obtained were much more higher, and pointed

out the fact that the algorithm's performance is higher when dividing numbers with an equal amount of bits.

3.2.2.2 Great Common Divisor Algorithm

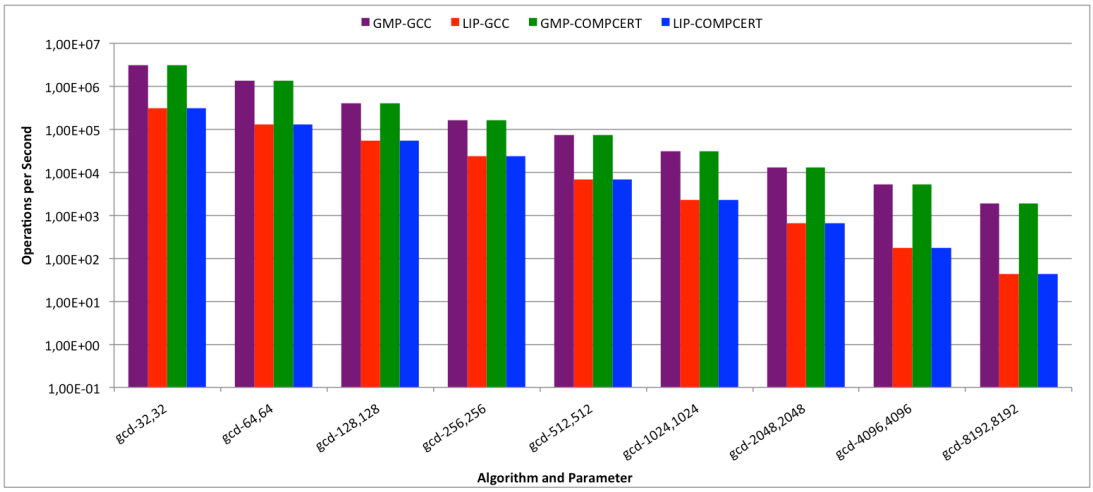


Figure 3.2 Great common divisor

Without surprises *GMP* achieves better performance than *LIP*. Also increasing the number of bits results in a performance decrease, as expected. Although it's important to say that along with this increase, the gap between *GMP* and *LIP* enlarges. Meaning that the *LIP*'s performance decreases faster than *GMP*'s.

3.2.2.3 Extended Great Common Divisor Algorithm

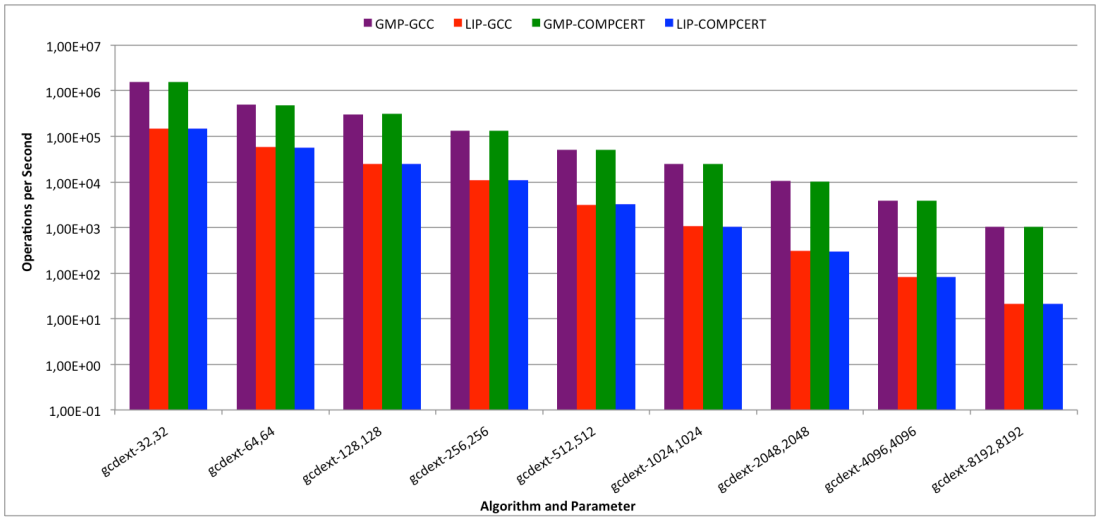


Figure 3.3 Great common divisor extended

GCD's extended version has the same behavior as the previous one. The only difference is that, in the overall this algorithm's performance is slightly worst than the previous one. Such thing does make sense because the extended version of the Euclidean algorithm besides finding the greatest common divisor of two integers, say a and b, it also finds other two integers, say x and y (one of which is typically negative) that satisfy *Bézout's identity*

$$ax + by = gcd(a, b) \tag{3.1}$$

3.2.2.4 RSA Algorithm

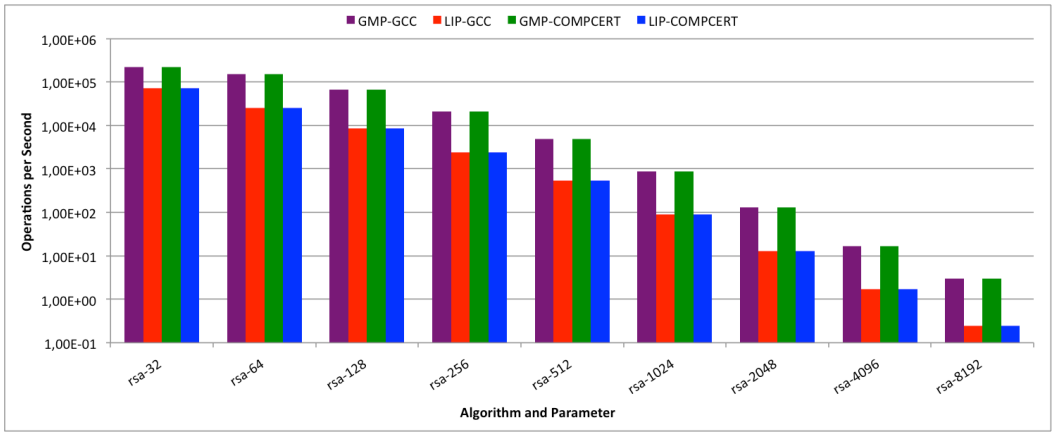


Figure 3.4 RSA

The performance of both libraries also decreases along with the increase of algorithms data input size. Without any surprises since this was the expected behavior. Once again *GMP* achieves higher levels of performance. *LIP*'s number of operations per second drops faster than *GMP*'s.

3.2.2.5 Multiplication Algorithm

Through the several multiplication tests although the conclusion is the same, and that is *GMP* showing better results than *LIP*, there are a few interesting remarks to be said.

For an instance, consider the case where the multiply algorithm is squaring a n-bit random number and one where it multiplies a n-bit random number for another n-bit random number. Despite the fact that both have the same input size, there is a minimal difference in the number of operations per second. This results in the squaring method having a better performance. Such difference resides in the fact that squaring uses only one number instead of two different ones. This shows that both libraries possess mechanisms that take advantage of multiplying two equal numbers.

For the last method, multiplying a 8192-bit random number for a n-bit random number, there is a significant drop in performance compared with the other two. But those were not strange news, since the

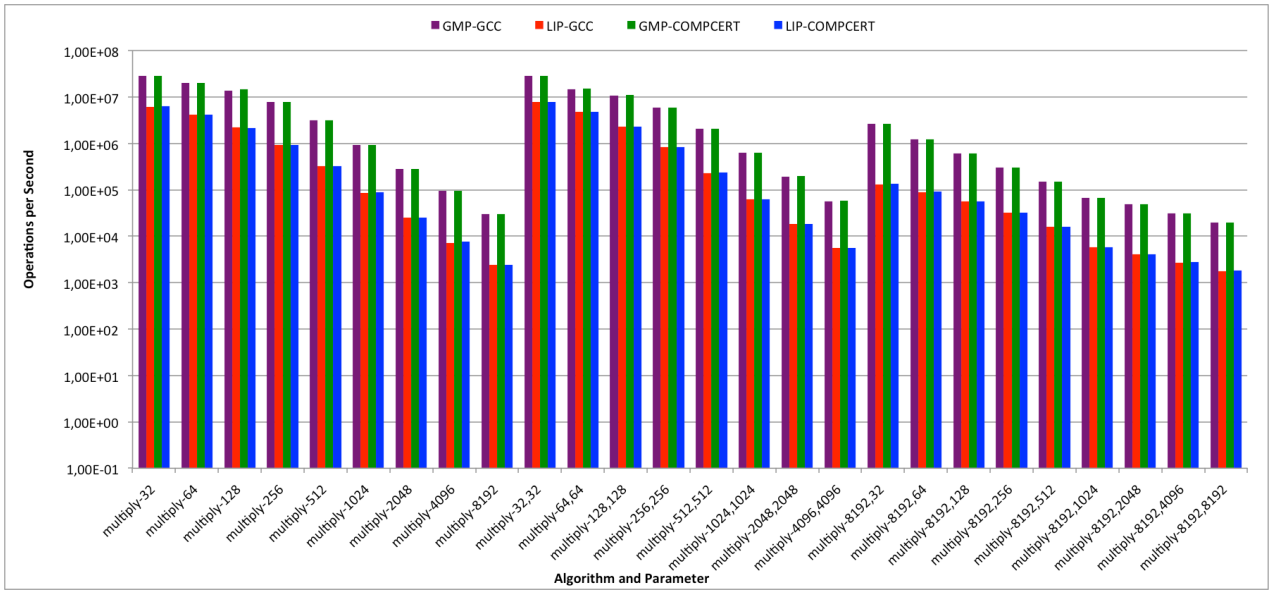


Figure 3.5 Multiply

size of the input data is bigger than the other ones. What is more important here is the gap between both libraries and the fact that it is larger than the ones on the previous two methods. This shows the benefits from using *GMP* over *LIP*, since its performance is better.

Bridging all three methods, several conclusion can be took: i) considering the *GMP* library, multiplying a 8192-bit number for a 32-bit number has approximately the same number of operations as squaring a 512-bit number; ii) considering the *LIP* library multiplying a 8192-bit number for a 32-bit number has approximately the same number of operations as multiplying two 512-bit numbers;

3.2.2.6 RSA - *GMP* power-modulo calculation modification

The *mpz_powm* is the normal RSA mode. The *mpz_powm_sec* function performs exponentiation with the same time and same cache access patterns for any arguments with the same size. This function was specially designed for cryptographic purposes, where resilience to side-channel attacks is desired.

In both cases the *mpz_powm_sec* performance is a little worst than the normal one. This result was already expected, because of its cache access methodology. The compilers choice doesn't seem

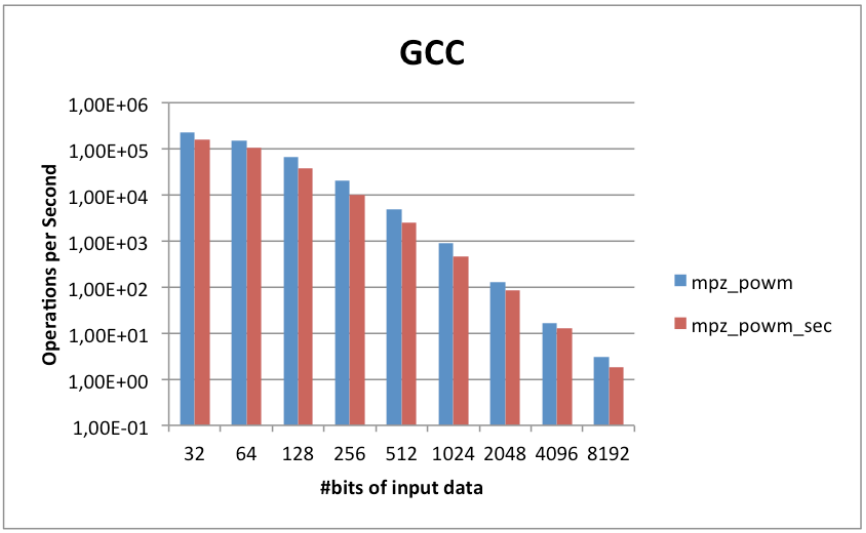


Figure 3.6 Comparing mpz_powm and mpz_powm_sec

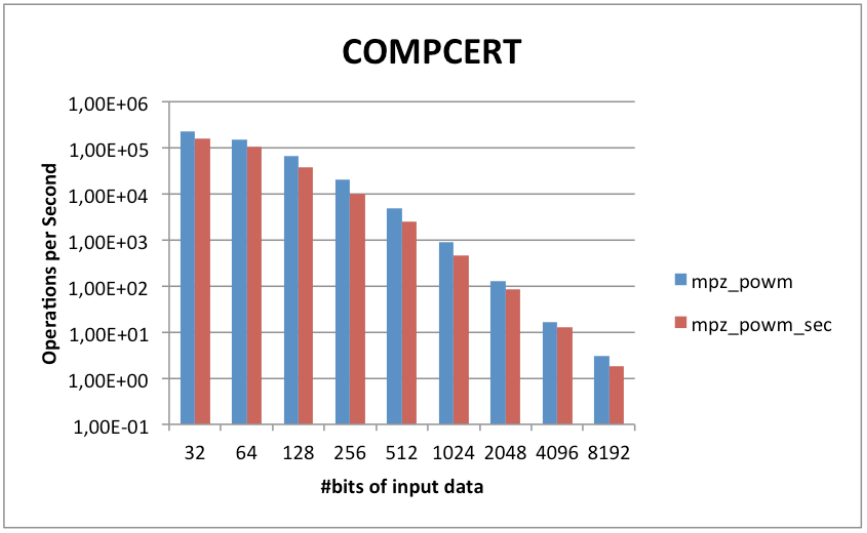


Figure 3.7 Comparing mpz_powm and mpz_powm_sec

to significantly influence performance. Considering the security properties offered by the *mpz_powm_sec* function, and the not so significant loss of performance, choosing this function over the other one seems to be a very good decision. This statement is reinforced by the fact that, the test algorithm is a cryptographic one.

3.2.2.7 RSA - *LIP* power-modulo calculation modification

The normal calculation mode uses the *zexpmod* function. But there are two more functions worth exploring, the *zmod_m_ary* which performs exponentiation in a size *m* window, and the *zmontexp* which uses the Montgomery modulo exponentiation (it also requires the Montgomery number conversion, so *LIP* can use the Montgomery arithmetic functions).

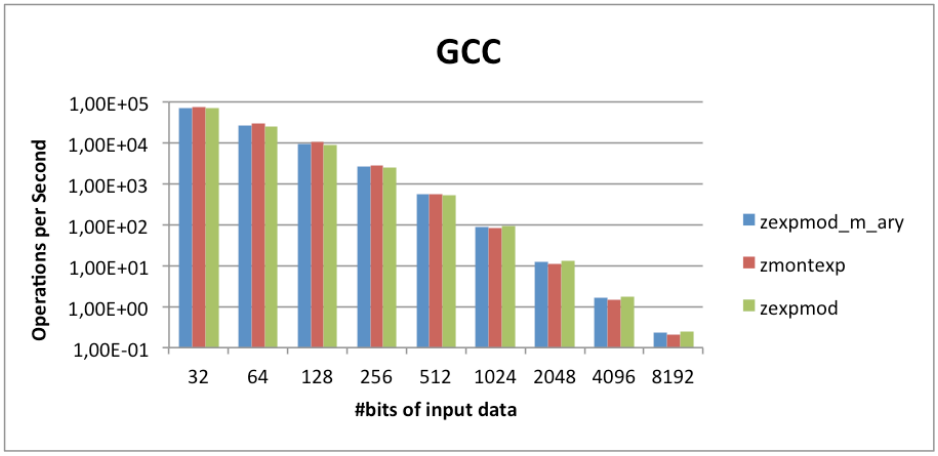


Figure 3.8 Comparing zmod_m_ary, zmontexp and zexpmod

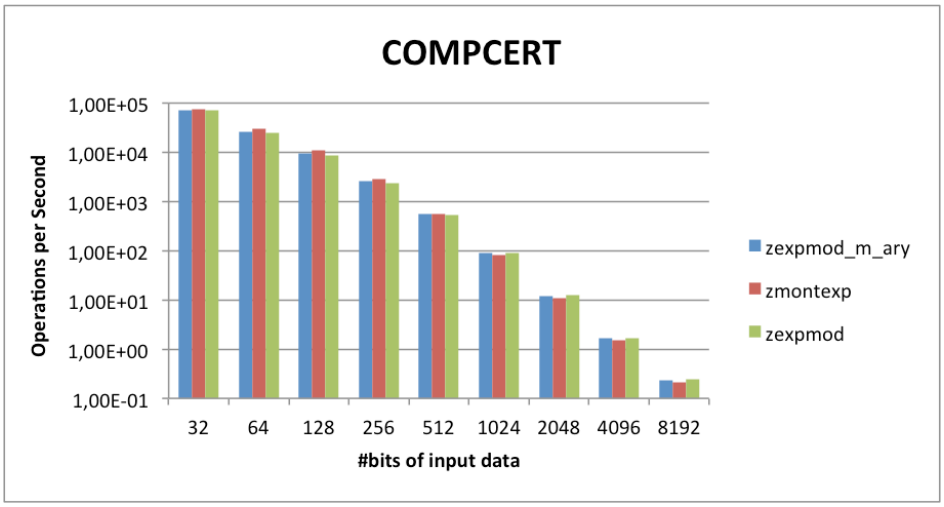


Figure 3.9 Comparing zmod_m_ary, zmontexp and zexpmod

Overall the use of a certain method instead of another didn't seem to be relevant. For data input

sizes between 32 and 512 bits (considering the chart's scale), the best choice is the Montgomery method, followed by the m_ary method, using the default value (see Table 3.2) of *m* accordingly to exponent number of bits and then the normal one (perform exponentiation with the *zexpmod* function. From 1024 to 8192 bits the Montgomery method offers the worst performance, followed by the m_ary method, and at last but not least, achieving better results the normal method. Despite the turn of events the gap between all three functions is very small.

e	**m**
< 2	2
< 3	3
< 7	4
< 16	5
< 35	6
< 75	7
< 160	8
< 340	9
otherwise	10

Table 3.2 Default values for *m*, when using the *zmod_m_ary* function. *m = zdefault_m(e)*

3.2.2.8 LIPCERT

On the following chart, *Cert_CompCert* means "*LIP* library compiled with *CompCert* and benchmark files also compiled with *CompCert*", and *Uncert_CompCert* means "*LIP* library compiled with *GCC* and benchmark files compiled with *CompCert*". It is only shown one example, simply because the remaining results do not differ much from one compiler to another, except for this one where the difference is more visible.

Generally speaking these results do not differ from the previous ones at all. But the main focus here is whether or not to choose using the *LIP* library compiled with *CompCert*. All results are really close. There

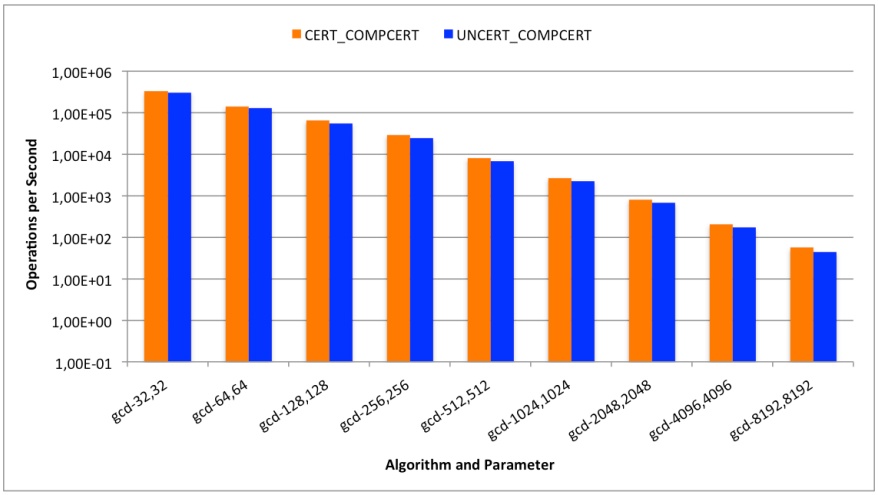


Figure 3.10 Great common divisor

are some tests where *CERT_COMPCERT* shows better performance, and same tests where it does not. Given the small gap between both cases, and the safety properties of using a certified library, one can choose using a full certified (library plus code) application, instead if a partial (code only) one.

3.2.3 Final Remarks

High performance and high security are very hard to achieve. *GCC* proves to be slightly more efficient than *CompCert*, although the difference is not that big. Therefore and considering the benefits from using a certified compiler, one can choose *CompCert* over *GCC*. The major drawback is that *CompCert* may not compile some code that *GCC* compiles (this has to do with the own compiler's source code), which leads to one of two things, put a tremendous effort to modify the source code so it can be compiled by *CompCert*, or simply dispose from the benefits of certified compilation and use *GCC*.

3.3 TrustedLib Support

In this chapter are shown some of changes made to *CompCert*'s source code, so it could be able to formally support some *GMP* low-level functions. This new level of support is achieved due to the inclusion of the TrustedLib [3][4] library. The implementation of this functions in the compiler's code makes it possible to *CompCert* to understand that although such functions come from an external library, they are to be trusted. Besides the whole list referenced in the document, only an implementation is exhibited, for content cleaning purpose.

3.3.1 *TrustedLib* Context

CompCert's formalizations only support calls to system libraries that leave the memory state unchanged. This results in the compiler not dealing with modifications in the memory state on calls to external functions. Given that *GMP* functions does cause memory modifications, a problem arises. With this in mind, the need for "something" that allows a programmer to specify functions that could, a) get data from the environment and fill input memory regions with it; and b) set the environment with data from output memory regions; came up. That "something" is *TrustedLib* and it ensures to *CompCert* that such modifications can indeed be trusted and even tells it the expected result (i.e: adding two numbers)

Resuming, *TrustedLib* is nothing but a "mechanism for declaring external functions that may impact the memory" [1] and its role is to tell *CompCert* that the specified external functions can indeed be trusted.

3.3.2 Adding *TrustedLib* to *CompCert*

In order to give *CompCert* the correct support for a specific external function, one must first correctly specify the function arguments in the "TrustedLibBuiltin" file and then the proof code itself in the "TrustedLibBuiltinSem" file.

[3] This work was mostly developed by professor José Bacelar Almeida, and was slightly improved in this thesis context.

[4] It uses the *SSREFLECT COQ* language extension

3.3.2.1 List of newly supported functions

In the source code each function has the prefix “TLGMP” which stands for *TrustedLib* GMP.

- mpn_add_n
- mpn_add_1
- mpn_add
- mpn_sub_n
- mpn_sub_1
- mpn_sub
- mpn_neg
- mpn_mul_n
- mpn_mul
- mpn_sqr
- mpn_mul_1
- mpn_addmul_1
- mpn_submul_1
- mpn_tdiv_qr
- mpn_divrem_1
- mpn_divexact_by3c
- mpn_mod_1
- mpn_lshift
- mpn_rshift
- mpn_cmp
- mpn_gcd
- mpn_gcd_1
- mpn_gcdext
- mpn_sqrtrem

3.3.3 Examples

3.3.3.1 Some relevant pre-requisites

Before showing the developed code for the functions above, it’s important to present some other relevant functions which are the core of the low level *GMP* functions specification, since they deal with the memory

representations of 32 *bit* limbs[5].

First the `split_limb32` function, which splits an integer in a limb part and the remainder:

```
Definition split_limb32 (z:Z) : Z*Z := (Zmod z (2^32), z / (2^32)).
```

The `encodeZ32'` function it's used to get the memory representation of an integer and the respective carry:

```
Fixpoint encodeZ32' nbytes (z:Z) : (nbytes.-tuple byte) * Z :=
 match nbytes
 with
   | nbytes'.+4 =>
     let (l,h) := split_limb32 z in
     let (bytes, r) := (encodeZ32' nbytes' h) in
     (cat_tuple (limb_of_Z l) bytes, r)
   | nbytes' => bytes_of_Z nbytes' z
 end.
```

Analogous to the previous function, except that the `encodeZ32` ignores the carry:

```
Definition encodeZ32 nbytes (z:Z) : nbytes.-tuple byte :=
 (encodeZ32' nbytes z).1.
```

At last but not least, the `decodeZ32` which is used to read an integer from a memory representation:

```
Fixpoint decodeZ32 (l:list byte) : Z :=
 match l with
 | [:: b1, b2, b3, b4 & l'] => Z_of_limb [tuple b1;b2;b3;b4] + 2^32 * decodeZ32 l'
 | l' => int_of_bytes l'
 end.
```

[5] part of a multi-precision number that fits into a single word

Finished introducing the basic functions to get both memory representation of an integer, and the integer itself, it's time to discuss some low-level *GMP* implementation examples.

The next example is organized in the following way: first is presented the function header with the respective arguments and then the function definition itself. For both cases there's the respective and detailed explanation.

3.3.4 mpn_add_n

According to the *GMP* documentation, the function signature is:

```
mp_limb_t mpn_add_n (mp_limb_t* rp, const mp_limb_t* s1p, const mp_limb_t* s2p,
       mp_size_t n)
```

and it adds $\{s1p, n\}$ and $\{s2p, n\}$, and write the *n* least significant limbs of the result to *rp*.

The function header signature accordingly to *TrustedLib* pragmas is:

```
mpn_add_n : OutPtr (4 * #3), InPtr (4 * #3), InPtr (4 * #3), Int -> Int
```

where each argument is separated by a semicolon, and the argument count starts at zero: *OutPtr (4 * #3)*, output pointer multiplies by 4 the 3^{rd} argument content. This characterizes the memory region of the result *gmp_integer*.
The first *InPtr (4 * #3)*, input pointer, multiplies by 4 the 3^{rd} argument content. This characterizes the memory region of the input *gmp_integer*.
The second *InPtr (4 * #3)* multiplies by 4 the 3^{rd} argument content. This characterizes the memory region of the input *gmp_integer*.
The first *Int* specifies the size (number of bits) of the other arguments.

The function's semantics is added to *CompCert* through the following *Coq* definition:

```
Definition TLGMP_mpn_add_n (outs: list nat) (ins:list (int + list byte)) :
   option (tl_outdata' outs * Z) :=
match outs, ins with
| [:: n] , [:: inr o1; inr o2; inl o3] =>
  let (res, carry) := encodeZ32' n (decodeZ32 o1 + decodeZ32 o2)
  in Some([::: res], carry)
| _, _=> None
end.
```

outs: list of natural numbers.

ins: either a integer number or a list of bytes.

inr o1: matches *o1* variable with the list of bytes from the *ins* argument.

inr o2: matches *o2* variable with the list of bytes from the *ins* argument.

inl o3: matches *o3* variable with the integer from the *ins* argument.

In the success case scenario, both *o1* and *o2* integers are decoded, accordingly to the previously characterized input memory regions, and added. The result is then encoded and stored in the (previously characterized) output memory region. Otherwise the result is *None*.

3.3.5 Final remarks

Despite of the relevant number of the functions that were listed in this chapter, there is much work yet to be done. Not only because of the total amount of functions, but mostly because of their complexity. All the functions mentioned in this chapter are low-level, therefore, adding this to the dimension of the *GMP* library, one can see the true complexity and effort needed to give full support for *GMP*.

Covered one of the main topics of this thesis, an agreement between both student and supervisor was made, so the study could proceed to a new and promising area. Such area is the core of the next chapter, and it relates to the *AES (short for Advanced Encryption Standard) Instruction Set* extension for the *x86 Instruction Set Architecture* proposed by Intel in 2008, and incorporated in the 2010 processoros. The extension adds a new set of instructions that can be combined to implement/run the *AES* algorithm.

4 . SIMD Extensions

This last chapter explains the type of support given to *CompCert* regarding the *AES* extensions Instruction Set and how it was attained through a developed library. Such library was created so it was easy to use the assembly functions in a more high-level and intuitive way, and more important so it could be used by a program that is intended to be compiled by both *GCC* and *CompCert*.

4.1 Context and Problems

Cryptographic operations are know to work with large number of bits, either for messages, ciphertexts, keys or, depending on the algorithm, initialization vectors and nonces. Usually one uses these libraries to perform these types of operations in a software level. Therefore being able to work with such datatypes as closer to the machine level as possible, the best. SIMD - *Single Instruction Multiple Data* [6]- processor extensions support precisely that. Although the types were always there, the required instructions were still missing. This was until the year 2010[1] where Intel expanded the Instruction Set[2] and implemented directly in hardware, functions that could perform parts of AES algorithm operations. Combining such functions, different modes of AES could be implemented. The main reason was to increase the speed of programs/applications using AES to encrypt and/or decrypt information. A library that uses these same instructions was developed.

Using only *GCC* did not brought much problems, but since the main topic of this thesis is certified compilers, and therefore *CompCert*, necessarily it must also be used. As was expected some problems arose. An assessment of such problems is described next.

[1] Although the model was proposed in 2008, processors with such extension only came out in early 2010, more precisely the Westmere processors family.

[2] created the AES-NI which is short for Advanced Encryption Standard New Instructions

4.1.1 Alignment

Since *CompCert* only guarantees alignment in global variables, the test vectors used in the main program were declared as global variables instead of local. Only by doing this, the local alignment limitation was overcome. Since this is a small and controlled programing environment having global variables (with this level of importance) wasn't much of a big deal, therefore the hazards involving the use of global variables were, in certain a way, irrelevant.

4.1.2 ABI - *CompCert* Arguments

Trying to use the *__m128i* data type (intrinsic type of SSE2 extension) as the type of arguments for each function, would result in the main program only being compiled with *GCC*. The reason why is simple, *CompCert* can not, yet, support these extensions types. The adopted solution was to wrap the library functions so the parameters were types know by both *GCC* and *CompCert*, such as array of integers.

4.1.3 Native Support

The former limitations could also be overcome by modifying *CompCert*'s code and specifying the new kind of alignment and the new types. However, besides hard this was a relatively intrusive solution since probably would imply refactoring parts of the code. Exploring other less complicate solutions some approaches could be considered. For instance, in the first case correcting *CompCert* to contemplate 16 *bit* data alignment should be enough.

Regarding the Application Binary Interface case, another solution could be represent such data types in the compiler's code, and then create structures (this time in the program's code) that would resemble to the ones used by the extensions. Aside from this would also be needed to specify the same type in *Coq*. With this there was no need to include the respective header files.

In order to being able to use the extensions function calls a feasible solution, but also kind of a cheating one, could be: first declare the same functions used by the extensions (with same signature) and in the

function's body explicitly use the extensions registers (i.e: %xmm0) and/or processor's registers (i.e: %eax) depending on the instruction. Resuming, rebuild the important functions resorting to inline assembly. Then the only thing still missing is setting up the environment to call and test the functions. In order to perform a better linking with *CompCert*'s purpose, such functions could be declared in the *TrustedLib* module. Probably the major drawback of this solution is code replication. Although the above is a feasible solution it is clear that the correct one is giving full support to the compiler. However such process would be extremely hard to perform.

4.2 Library Development

A modular and portable library was developed[3] so it could behave exactly like *GMP* in chapter 3.2, compiled by *GCC* and used by *CompCert*. In this particular case the main program, which runs the encryption and decryption functions, was compiled by *CompCert* and uses the previously compiled (by *GCC*) library functions.

List of used AES-SSE extensions functions/data types:

- __m128i
- _mm_shuffle_epi32
- _mm_slli_si128
- _mm_xor_si128
- _mm_loadu_si128
- _mm_aeskeygenassist_si128
- _mm_shuffle_pd
- _mm_aesenc_si128
- _mm_aesenclast_si128
- _mm_aesenclast_si128_si128
- _mm_aesdec_si128
- _mm_aesdeclast_si128

[3] most of the functions content was retrieved from [2], but the portability so it could be compiled by *CompCert*, which implied some modifications to the previous referenced code, was made in this thesis context.

- _mm_set_epi32
- _mm_setr_epi8
- _mm_setzero_si128
- _mm_insert_epi64
- _mm_insert_epi32
- _mm_srli_si128
- _mm_shuffle_epi8
- _mm_add_epi64
- _mm_add_epi32
- _mm_aesimc_si128

4.2.1 Key Expansion Algorithms

The following algorithms are used to prepare the cipher key for the encryption algorithm. Such process is called key expansion and it is necessary to encrypt the message. In the decryption process the expanded key is re-processed, but this time by a invert mix column function (*_mm_aesimc_si128*).

4.2.1.1 Algorithm for 128 *bits* key size

```
__m128i AES_128_ASSIST (__m128i temp1, __m128i temp2) {
__m128i temp3;
     temp2 = _mm_shuffle_epi32 (temp2 ,0xff);
     temp3 = _mm_slli_si128 (temp1, 0x4);
     temp1 = _mm_xor_si128 (temp1, temp3);
     temp3 = _mm_slli_si128 (temp3, 0x4);
     temp1 = _mm_xor_si128 (temp1, temp3);
     temp3 = _mm_slli_si128 (temp3, 0x4);
     temp1 = _mm_xor_si128 (temp1, temp3);
     temp1 = _mm_xor_si128 (temp1, temp2);
     return temp1;
}
```

4.2.1.2 Algorithm for 192 *bits* key size

```
void KEY_192_ASSIST(__m128i* temp1, __m128i* temp2, __m128i* temp3) {
     __m128i temp4;
     *temp2 = _mm_shuffle_epi32 (*temp2, 0x55);
     temp4 = _mm_slli_si128 (*temp1, 0x4);
     *temp1 = _mm_xor_si128 (*temp1, temp4);
     temp4 = _mm_slli_si128 (temp4, 0x4);
     *temp1 = _mm_xor_si128 (*temp1, temp4);
     temp4 = _mm_slli_si128 (temp4, 0x4);
     *temp1 = _mm_xor_si128 (*temp1, temp4);
     *temp1 = _mm_xor_si128 (*temp1,* temp2);
     *temp2 = _mm_shuffle_epi32(*temp1, 0xff);
     temp4 = _mm_slli_si128 (*temp3, 0x4);
     *temp3 = _mm_xor_si128 (*temp3, temp4);
     *temp3 = _mm_xor_si128 (*temp3,* temp2);
}
```

4.2.1.3 Algorithm for 256 *bits* key size

```
void KEY_256_ASSIST_1(__m128i* temp1, __m128i* temp2){
     __m128i temp4;
     *temp2 = _mm_shuffle_epi32(*temp2, 0xff);
     temp4 = _mm_slli_si128 (*temp1, 0x4);
     *temp1 = _mm_xor_si128 (*temp1, temp4);
     temp4 = _mm_slli_si128 (temp4, 0x4);
     *temp1 = _mm_xor_si128 (*temp1, temp4);
     temp4 = _mm_slli_si128 (temp4, 0x4);
```

```
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    *temp1 = _mm_xor_si128 (*temp1,* temp2);
}

void KEY_256_ASSIST_2(__m128i* temp1, __m128i* temp3) {
    __m128i temp2,temp4;
    temp4 = _mm_aeskeygenassist_si128 (*temp1, 0x0);
    temp2 = _mm_shuffle_epi32(temp4, 0xaa);
    temp4 = _mm_slli_si128 (*temp3, 0x4);
    *temp3 = _mm_xor_si128 (*temp3, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp3 = _mm_xor_si128 (*temp3, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp3 = _mm_xor_si128 (*temp3, temp4);
    *temp3 = _mm_xor_si128 (*temp3, temp2);
}
```

4.2.2 ECB mode

Before talking about the implemented functions, lets recall the ECB encryption and decryption processes represented in the following images.

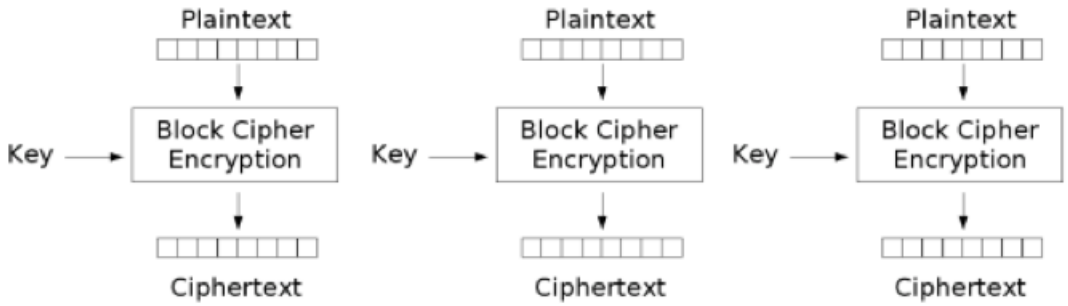


Figure 4.1 ECB encryption schema mode

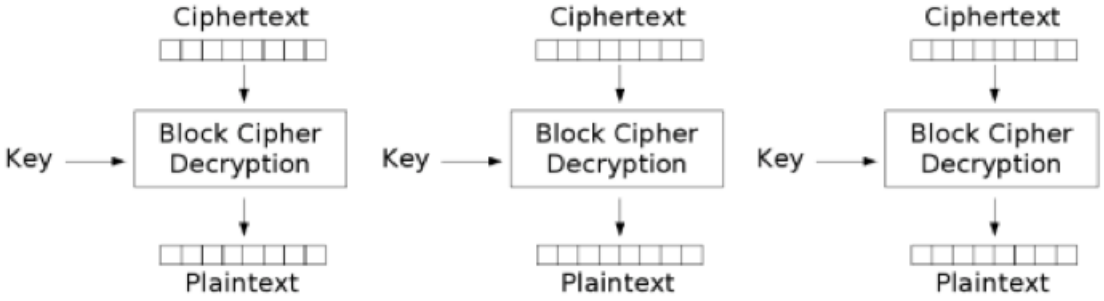


Figure 4.2 ECB decryption schema mode

The next functions execute both encryption and decryption (of a plaintext and a ciphertext respectively) using the *ECB* (short for *Electronic Codebook*) encryption/decryption mode of *AES*. The first argument is a pointer to the cipher key, the second a pointer for the message and the third a pointer for the ciphertext (where to store the result), followed by a pointer to the expected ciphertext. The last argument simply tells whether to print or not the information.

The difference for all three functions is the number of *bits* used in the respective key, since every key size has it own key expansion algorithm.

```
void ECB128(unsigned int* _k, unsigned int* _m, unsigned int* _out,
     unsigned int* Expected_CIPHERTEXT, int verbose);

void ECB192(unsigned int* _k, unsigned int* _m, unsigned int* _out,
     unsigned int* Expected_CIPHERTEXT, int verbose);

void ECB256(unsigned int* _k, unsigned int* _m, unsigned int* _out,
     unsigned int* Expected_CIPHERTEXT, int verbose);
```

4.2.3 CBC mode

Lets now recall the CBC encryption and decryption processes represented in the following images.

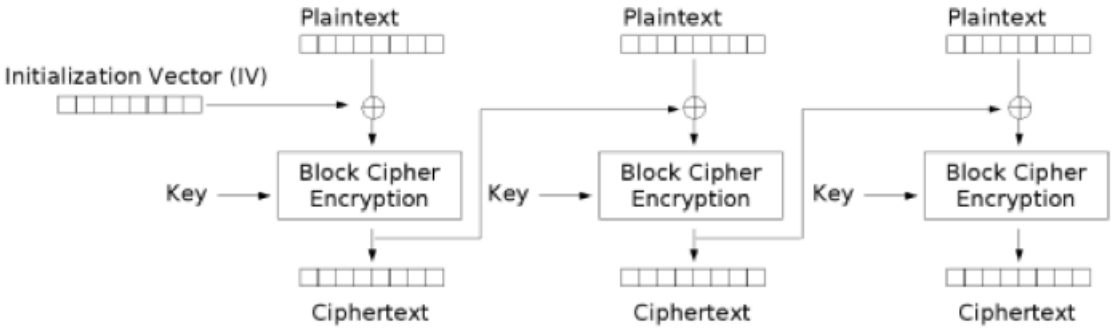


Figure 4.3 CBC encryption schema mode

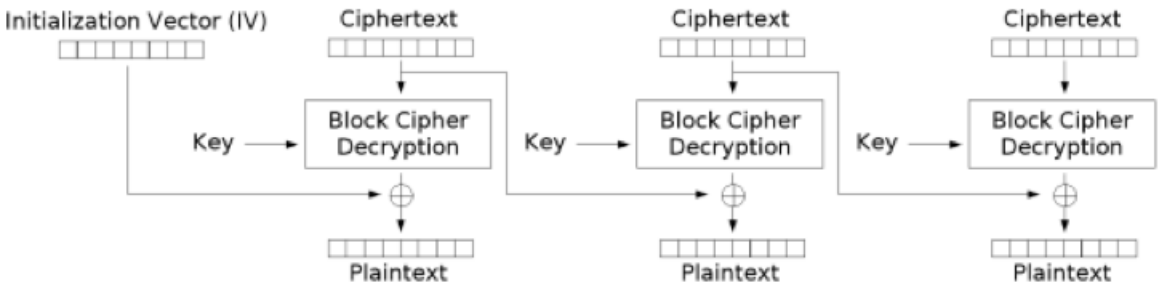


Figure 4.4 CBC decryption schema mode

The following code refers to functions that perform both encryption and decryption using the *CBC* (short for *Block Cipher Mode*) encryption/decryption mode of *AES*. The arguments are, a pointer to the cipher key, a pointer for the message/plaintext to be encrypted, a pointer for the ciphertext, a 16 *bits* initialization vector, the expected ciphertext and a flag to print or not the results. Again the difference for all three functions is the number of *bits* used in the respective key.

```
void CBC128(unsigned int* _k, unsigned int* _m, unsigned int* _out,
```

```
    unsigned int ivec[16], unsigned int* Expected_CIPHERTEXT, int verbose);

void CBC192(unsigned int* _k, unsigned int* _m, unsigned int* _out,
    unsigned int ivec[16], unsigned int* Expected_CIPHERTEXT, int verbose);

void CBC256(unsigned int* _k, unsigned int* _m, unsigned int* _out,
    unsigned int ivec[16], unsigned int* Expected_CIPHERTEXT, int verbose);
```

4.2.4 CTR mode

The CTR encryption and decryption processes are as follow:

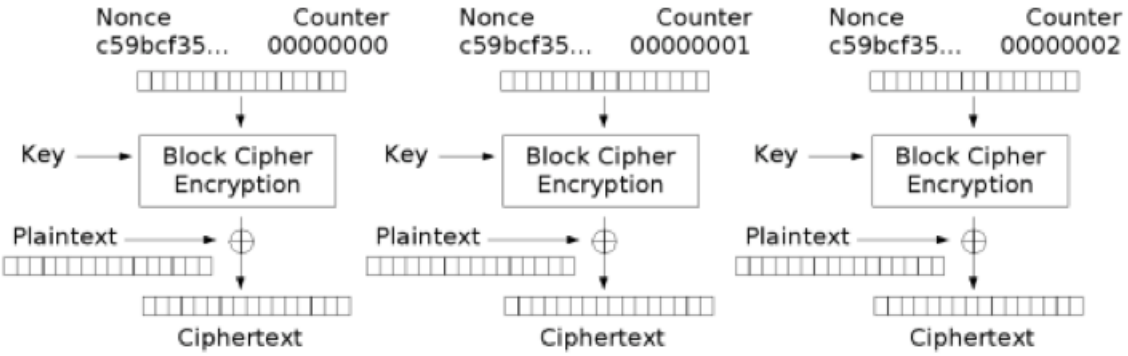


Figure 4.5 CTR encryption schema mode

At last but not least, the functions that perform both encryption and decryption using the *CTR* (short for *Counter mode*) encryption/decryption mode of *AES*. The arguments are, a pointer to the cipher key, a pointer for the message/plaintext to be encrypted, a pointer for the ciphertext, a 8 *bits* initialization vector, a 4 *bits nonce*, the expected ciphertext and a flag to print or not the results. Once again the difference for all three functions is the number of *bits* used in the respective key.

```
void CTR128(unsigned int* _k, unsigned int* _m, unsigned int* _out,
  unsigned int ivec[8], unsigned int nonce[4], unsigned int* Expected_CIPHERTEXT,
```

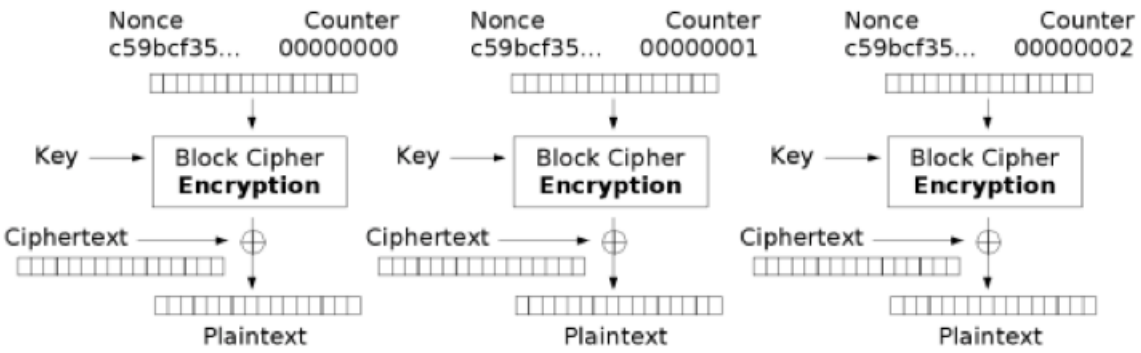


Figure 4.6 CTR decryption schema mode

```
   int verbose);

void CTR192(unsigned int* _k, unsigned int* _m, unsigned int* _out,
  unsigned int ivec[8], unsigned int nonce[4], unsigned int* Expected_CIPHERTEXT,
   int verbose);

void CTR256(unsigned int* _k, unsigned int* _m, unsigned int* _out,
  unsigned int ivec[8], unsigned int nonce[4], unsigned int* Expected_CIPHERTEXT,
   int verbose);
```

4.3 Developed Library Performance

The performance tests were conducted in a machine with the following specifications:

Manufacturer: Apple

Model: MacBook6,1

CPU: Intel Core 2 Duo P7550 (Penryn) @ 2.26GHz

Main memory: DDR3 PC3-8500, 4GB with 14ns latency

CPU #cores / #threads: 2/2

CPU peak FP performance: 18 GFLOPS

Cache details : L2 3072 KB

Given the above specs, one must have already realized that Intel's extensions were not present in the machine's processor. The solution was to use *SDE*, an extensions emulator developed by Intel [4]. This incurred in a small (and undetermined) overhead.

For a better comprehension of the following charts, an iteration represents the key expansion for encrypting, the encryption process, the key expansion for decrypting and decryption process itself.

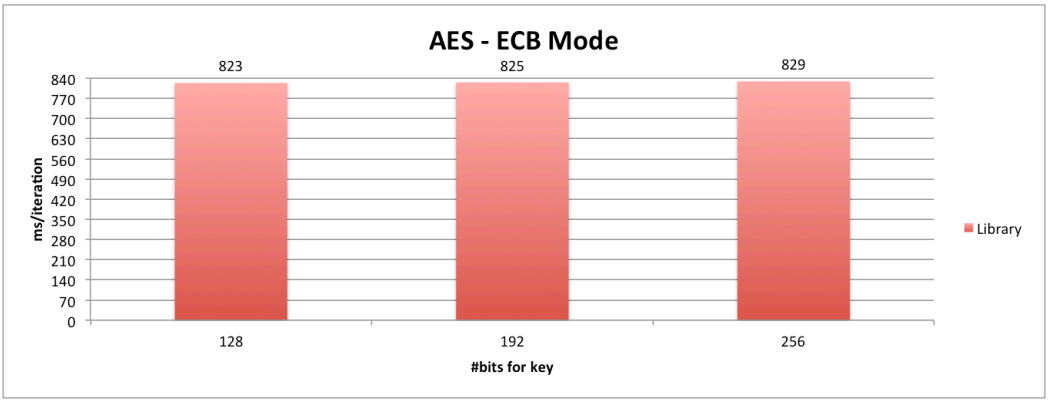


Figure 4.7 Comparing libraries's performance for the different key sizes for ECB mode

[4] http://software.intel.com/en-us/articles/intel-software-development-emulator

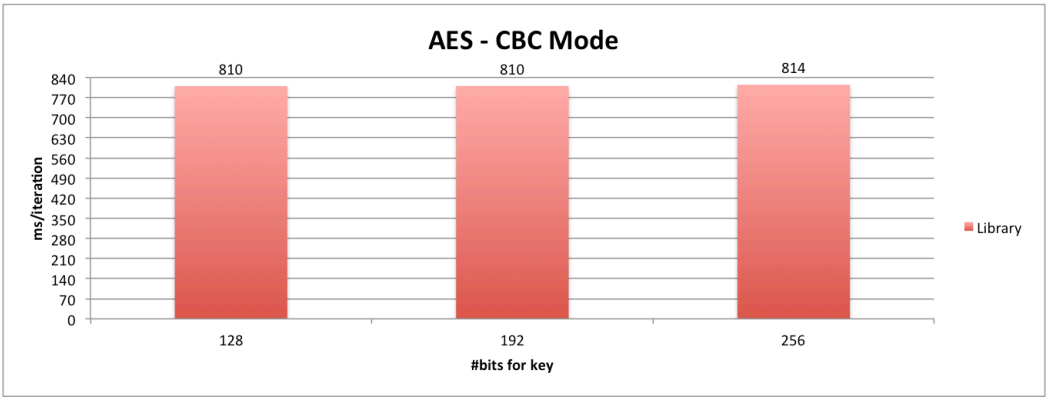


Figure 4.8 Comparing libraries's performance for the different key sizes for CBC mode

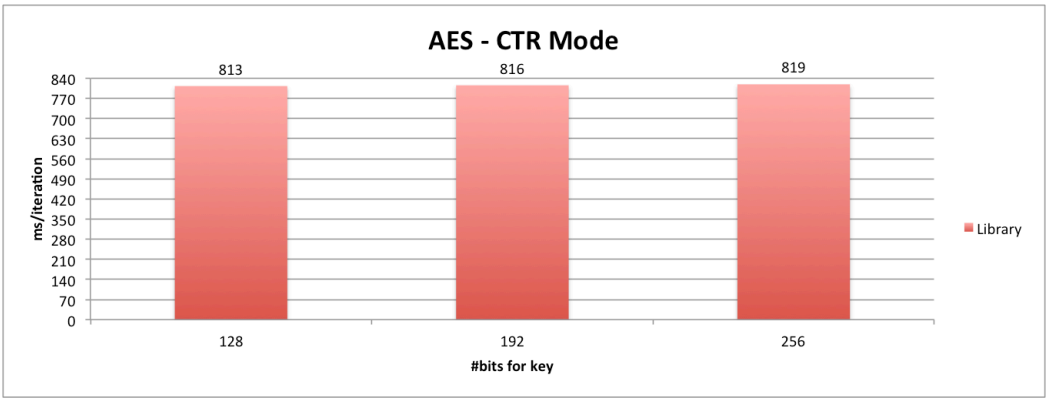


Figure 4.9 Comparing libraries's performance for the different key sizes for CTR mode

4.4 External library VS Assembly incorporation

Another way to achieve this kind of support would be incorporating the extensions assembly directives directly in the *CompCert* assembly code. Although it doesn't sound very difficult, as a matter of fact it is. Mostly because it implies adding new information, like the new 128 *bit* data structures, on the most top files of the compiler (first files to be compiled when compiling *CompCert*'s source code) so they could be "transferred" along the compilation processes into the files below which of course, means having the remaining files featuring the same support. Besides this, a full specification of the used functions should also be included.

Resuming, it would imply a total (or almost total) file content refactoring so the newly added information would be recognized and finally reach the generated assembly code, then and only then the extension assembly functions would be known and therefore we would have a fully support for *AES* extensions.

4.5 Final remarks

Considering the two options and their respective complexity/time-effort ratio against the remaining time, a choice was made and it consisted on only giving the same type of support as given to *GMP*. Although the fully assembly support was a more challenging and interesting approach, it would require a great amount of time, time that did not exist. The execution times were acceptable, but once again most work was being done by *GCC* compiled code, since *CompCert* could only compile the main program, which basically consists in a few lines of code. This results in only have certifying the program's entry point, which obviously does not give as much guarantees as expected. One matter that must be taken into account, is the fact that an emulator was used, which only by itself incurs in a performance bottleneck.

5 . Conclusions

Throughout this entire document some concepts were explained, some work was made and the respective results were presented and discussed. Among them one idea was always kept in mind, comparing *GCC* with *CompCert*. Not to tell which's the best compiler, but to show that in certain and particular cases, *CompCert* was a good bet. Mainly when high assurance is an important requisite.

In order to attend such level of assurance, it's necessary to resort to formal methods at the compilation time. Since compilation is a critical step in software development, certified/verified compilation came to be a better and at same time a safer solution. As it was seen, the major downside is that it's very hard to achieve since it implies a whole reformulation in the compilers source code so the use of formal methods could be integrated.

After analyzing some of the most commonly used mathematical algorithms in cryptography, it was easily seen that not all of them possess the same computational efficiency. Obviously such variation depends of the type of operation used, whether it's a basic one like multiplication or a bit more complex one like exponentiation, or even the combination of more than one operation, as it happens for example with the Classic and Montgomery algorithms for the modular multiplication.

Already taking execution times into account and perfectly knowing that high performance and high security are two concepts hard to combine, all charts presented in section 3.2 demonstrate that despite the fact of *GCC* being slightly more efficient than *CompCert*, the entire set of benefits that come alongside with the use of a certified compiler, choosing *CompCert* instead of *GCC* seems not a bad idea at all. This is principally valid for the *LIPCERT* case, since in the other ones *CompCert* only treated the main program's entry point leaving the rest of the work to be made by code compiled with *GCC*. This results in a program being only partially verified.

Adding support for some of the *GMP* core functions turned to be a challenging work that also allowed a better insight of how *CompCert's* source code is organized and more important, how it uses *Coq* to perform function's formal verifications. It also shows how complex a well coded and complete big number library can be.

When developing the *SSE-AES* library some problems appeared along the way. Whether the solution

was to wrap the library and use it with a higher level interface or declaring testing data in global variable space they were all overcome. Besides the adopted solutions and the obvious one (which is native support) other possible good options were slightly studied.

Based an all of the previous points although *CompCert* seems a good option when compared to *GCC*, it is important to keep in mind a great limitation: *CompCert* can not compile code that *GCC* can. Good examples are the *GMP* library and the developed *SSE-AES* library. *CompCert* could not compile them either because of the use of certain flags (in the second case) or because of certain functions. However in some cases simpler than the ones stated in this document, given the advantages and assurance offered by *CompCert*, it is a good option to use it instead of *GCC*. Other solution is to do the same that was made here, and use *CompCert* to only compile the main program that uses libraries compiled with *GCC*. In order of this to work such libraries must be compiled for a 32*bits* architecture since *CompCert* cannot yet support the 64*bits* one.

5.1 Future Work

It was clearly seen that a lot of work is yet to be done in *CompCert*, not only implementing other *GMP* core functions, but high-level functions as well. Since it would require a lot of time, only part of the first one was made in this thesis context, while the second one was not even considered.

Another thing that could still be done was to make *CompCert* able to support the *SSE-AES* directives directly in it assembly code. Although it was part of the plan, as it was said before, such option was discarded due to some unexpected problems regarding the *SSE-AES* library development. Also, the developed library could be expanded so other modes could be implemented.

Bibliography

[1] J.B Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: Efficient provably secure machine code from high-level implementations. *CCS*, 2013.

[2] Shay Gueron. *Intel® Advanced Encryption Standard (AES) New Instructions Set*, 2012.

[3] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *POPL*, 2006.

[4] Xavier Leroy. *The CompCert C verified compiler - Documentation and user's manual*. INRIA Paris-Rocquencourt, version 2.1 edition, October, 28 2013.

[5] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[6] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface*. MK, 4th edition, November 2011.

[7] Sebastian Prehn. Formally certified and certifying compiler back-ends. *WS*, 2008.