Mechanising Proofs for Executable Code Using Hacspec & Coq

June 18, 2024

Christian Møller Jensen christianmjensen@outlook.com
Jonas Daniel Ancher jonasdancher@hotmail.com

Abstract

Growing concerns for information security has given rise to multiple different projects working towards formally proving executable code bases. In this thesis – to demonstrate the mechanising of a proof – we implement the ElGamal key exchange protocol, using the Hacspec DSL for Rust. To conduct the proof, we employ Hacspec's translator to translate the implementation to Coq, and create the proof therein. Despite the limitations of tools employed, a proof of correctness for a pseudocode implementation in addition to the translated executable implementation is created. In the end, we demonstrate the possibility of mechanising a proof for an executable implementation.

Acknowledgements

A thank you to the SSProve and Hacspec teams, for taking the time to answer questions on email, as well as creating the foundation for the project.

A heartfelt thank you to Markus Krabbe Larsen, for the immense amount of help and guidance you have contributed to this project. Without your help, much of the work with Coq, SSProve, and Nix would have been impossible for us to complete in time. We hope to have a chance to meet you in Scrollbar to return the favour!

An unquestionable thank you to our advisor and lecturer Carsten Schürmann. Having been in your classes on cryptography and later having you as our advisor, has been a fun and big learning experience.

And lastly, a thank you to our friends and family for being patient with us, while prioritising the project and spending a majority of our energy on working towards completing it.

Our Code Base

For the ease of our censor and examiner, and any future readers, we here link directly to the code base on GitHub: https://github.com/JonasDAncher/MasterThesis/

Contents

1 Introduction

The importance of writing secure applications is an ever increasing necessity in the world of software development. Cyber attacks are happening at an alarmingly increasing rate *(Forbes, 2024, AAG, 2024).* As the world we live in is becoming increasingly digital, with more and more things being reliant upon always on services, with critical infrastructure being controlled by computer systems, with the increasing complexity of software applications demanding pure trust from layman users, it is increasingly important that the software developers making these applications are expertly aware of the security and integrity of the applications. Even so, believing something to be secure and it actually being secure are two very different things. It can be a complex and difficult task for a cryptographer to *prove* an application to be secure. Many different tools and analysis frameworks have been made with this goal in mind: to better enable developers to write secure code, or at the very least be capable of proving a critical component to be secure.

Currently, protocol verification tools enable a developer to model a protocol in the verifier's language and query different kinds of attacks and/or leaks to determine the security of the protocol. But the downside of such tools, is that it cannot be applied directly on an executable implementation of a protocol only a developer's *model* of said protocol. This means, if the developer did not find a perfect way to model the protocol, whether the abstraction level of the algebraic relations is too high or a small detail is missed, one could believe the model of the protocol to be adequately reflective, while actually flawed. Even if the model perfectly reflects the theoretical protocol, there is no link from it to an executable implementation, and therefore the proof does not necessarily apply to the implementation. Instead, tools that can prove actual executable code is necessary.

Different ideas have been proposed to try and bridge the gap between protocol verifiers/assistants and actual executable implementations. One such project is the Rust domain specific language of *Hacspec*, further details on Hacspec can be found in subsubsection 4.1.2. In short, Hacspec allows a developer writing Rust code to make use of the Hacspec library to write any critical components, which can later be type-checked using Hacspec to verify the component to be type secure. Hacspec supports translation into the proof assistants F*, EasyCrypt, and Coq, enabling a proof engineer to create a formal proof of the implementation. Once a proof has been completed, using for example Coq, the code can be translated into an application language again, now as verified code.

However, tools for translation like Hacspec, are still very much new in the field of cryptography, and how easy they would be to incorporate into a production pipeline is still a sparse field of research. Therefore, we wish to research the following:

Is it possible to mechanise the proof of the correctness of an ElGamal implementation in Rust using Hacspec in conjunction with Coq?

With *mechanise* we mean *make tool-assisted.* That is can we use available technologies to make a proof of correctness for an executable ElGamal implementation, alleviating parts of the difficult process of formally proving an executable piece of code. By *proof of correctness* we mean a formal proof verifying that the implementation works to its specification. When saying *implementation*, we refer to a piece of code that is an executable implementation of a protocol/algorithm, in our case the ElGamal key exchange algorithm.

Lastly, with *Rust* we refer to the programming language, by *Hacspec* we refer to the DSL to Rust, and by *Coq* we refer to the proof assistant. All three will be introduced further in section 4, Technologies.

2 Example: ElGamal

For the purpose of writing an implementation that can be used with the tools, the ElGamal key exchange algorithm was chosen. Largely due to the simple nature of the algorithm, and it being well known in the field of cryptography, allowing a wide audience of readers that are familiar with the algorithm already, to readily understand the implementation, instead of working with another bleeding edge protocol/algorithm. Even so, for readers unfamiliar with the algorithm an explanation is presented below.

The ElGamal key exchange algorithm – as we will henceforth refer to as *ElGamal* – is an asymmetrical key encryption algorithm for public-key cryptography, described by Taher ElGamal in 1985. It is based on the widely known Diffie Hellman key exchange protocol.

It works in 3 separate steps; the key generation, the encryption, and lastly the decryption.

2.1 Key Generation

The party who receives the encrypted message, whom we shall call Alice, generates their key pair in the following way:

- A cyclic group G of order q with generator g, and e represents the identity element of G. The group and generator can be a combination known to be secure, as it does not need to be unique for each session.
- Chose x uniformly from $\{1, ..., q-1\}$.
- Compute $h := g^x$.
- Alice's public key is now (G, q, g, h), and x their private key. Alice freely publishes her public key.

2.2 Encryption

The encryption party, whom we shall call Bob, can encrypt a message M using Alice's public key (G, q, g, h) in the following way;

- Using a reversible mapping function, map M to m, which is an element of G.
- Chose a private key y uniformly from the same as Alice did, $\{1, ..., q-1\}$.
- Compute $s := h^y$ as the shared secret.
- Compute $c_1 := g^y$.
- Compute $c_2 := m \cdot s$
- Send the *ciphertext* (c_1, c_2) to Alice.

2.3 Decryption

Alice can now decrypt the *ciphertext* using her private key x:

- Compute $s := c_1^x$. This is the same shared key as Bob calculated because $c_1 = g^y$, meaning $c_1^x = g^{xy} = h^y$
- Compute the inverse of s in the group G as s^{-1}. Depending on the group used, this can be done in multiple ways.
- Compute m as $c_2 \cdot s^{-1} = (m \cdot s) \cdot s^{-1} = m \cdot e = m$
- Lastly, map m back to the plaintext message M.

3 Literature Review

In this section, we present a review of the current state of proof assistants as well as projects working towards mechanisation of formal proofs. Firstly, we present a section dedicated to programming languages that are specifically useful for the mechanisation of formal proofs for implemented code. This is followed by a brief section introducing a selection of research projects working on formal verification with Rust as the focus. Lastly, a section presenting the history and state of proof assistants, closing with a presentation of Geuvers' roles for a proof and goals for a proof assistant.

3.1 Programming Languages

For the current project, we have decided to work in Rust and Coq. For further details and a presentation of both, please see subsubsection 4.1.1 and subsubsection 4.2.2. However, another languages could also have been suitable for the work conducted.

Notable, F* (*"F*", 2024*) is a contender for a programming language as well as a proof assistant that could have been used. Had we chosen this language, we could have either decided to simply not use Hacspec, as we would have been able to implement ElGamal natively in F*. This option also exists with Coq. However, part of the projects interest lays in the possibility of writing an implementation in a commonly used language like Rust, and then later proving it by translating with Hacspec.

Hacspec also support translation to F*, like it does with Coq. Meaning, we could have chosen to implement in Rust, and prove it in F*. However, Coq has seen a lot of success within the field, perhaps because Coq has seniority over F*. Both factors contribute to the fact that there is a large amount of useful sources on Coq for us to utilise.

3.2 Formal Verification Projects

There are multiple projects currently working on – or recently ended – creating formal verification tools for Rust specifically. Many mention the motivation for utilising Rust being its strong type system, which can be used to greatly simplify formal verification.

The *Prusti Project* (*Astrauskas et al., 2022*), is a general purpose deductive verifier for Rust with a focus on "*Safe Rust*". By contrast, RustBelt (*Jung et al., 2017*), aimed at creating formal tools for verifying the supposedly safe encapsulation of *Unsafe Rust* code used internally in Rust. These "unsafe blocks" of code are necessary as Rust disallows aliasing mutable states, which is incompatible with certain low-level data structures. While the project formally ended in 2021, the team behind continues it *"in spirit"*, and are still working on formally verifying more of Rust.

3.3 Proof Assistants

Proofs are an incredibly important part of any mathematical field, essentially serving as the back-bone of knowledge. Can it be proved, then it is accepted as fact. However, the process of creating these proofs are incredibly time consuming, leading researchers to contemplate how they could use computer processing power to create proofs instead of making them by hand. A great example, is Hales' proof of the Kepler conjecture, (*Hales et al., 2024*), where Hales came to the conclusion that without mechanising parts of the proof, he was looking forward to more than 25 years of estimated work. Hales made the since famous *Flyspeck project* in 1998, where he used the proof assistant *HOL Light*, (*Harrison, 2024a*), *Coq* (*Paulin-Mohring, 2011*), and *Isabelle*, (*Harrison, 2024b*), managing to finish the project after 16 years in 2014.

Although there are more than a dozen different proof assistants available, there is a large difference in how developed they are, and how many features they support. Most notable are assistants like *Lean*, (*de Moura and et al., 2024*), *F**, (*"F*", 2024*), *Coq*, (*"Coq", 2024*), and *NuPRL*, (*"NuPRL", 2024*), with honourable mention of *Twelf*, (*Pfenning and Schürmann, 2024*).

In their 2009 paper, (*Geuvers, 2009*), Geuvers present an overview of the idea, history and future of the proof assistant technologies. Notably, they explain that a proof has two roles:

A proof ***convinces*** *the reader that the statement is correct, [and] a proof* ***explains*** *why the statement is correct.*

To the first role, Geuvers highlights an issue with proof checking programs: we must trust that the program is *correct.* To combat this issue, Geuvers presents 4 ways to increase confidence in proof checking programs.

Firstly, with a description of the logic, we can determine whether we believe those, and whether the definitions truly represents the desired expressions, as well as the steps the proof takes, and if they make sense or not. Secondly, if the program has a small enough kernel, it allows users to verify them themselves by checking the code. If all other proof rules are written as a composition of the basic steps, and the user (after verifying) trusts those steps, the user can likewise trust the composition of steps. Thirdly, and very importantly, as the proof checker program is in fact a program, it is possible to verify its correctness as well. Lastly, Geuvers talk of the *De Bruijn criterion.* This criterion essentially says, in Henk Barendregt words, *Barendregt and Geuvers, 2001*:

A proof assistant satisfies the de Bruijn criterion if it generates 'proof-objects' (of some form) that can be checked by an 'easy' algorithm.

Meaning, a proof assistant should make available some sort of proof object, that a sceptic user can verify the validity of to strengthen their believe in the programs correctness. Essentially saying, that the kernel of the proof assistant should be small enough, that it can be independently tested by a sceptical user.

Furthermore, Geuvers notes in their ideas for how the field should develop, that it would be preferable for the libraries and technologies to be simpler to use, and increase the amount of possible users, further motivating the present paper.

4 Technologies

In this section, we will present an overview of the most critical technologies that was used during the course of the project. It is separated into 2 different *environments*; Proof and Implementation.

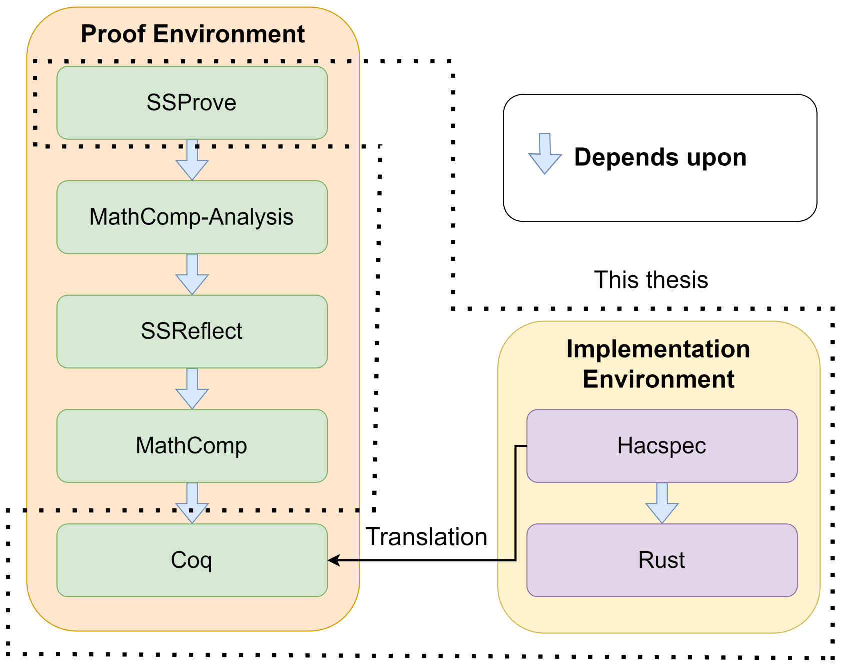


Figure 1: Visualisation of project dependencies. Objects inside the dotted line indicates what we worked directly with.

4.1 Implementation Environment

The *Implementation Environment* encapsulates the technologies used during the creation of the executable implementation of ElGamal, namely Hacspec and Rust. As visualised in Figure 1, Hacspec is dependent on Rust.

4.1.1 Rust

Rust prides itself on its productivity features, performance, and reliability. In a 2023 survey by StackOverFlow – with more than 90.000 developers voting – Rust was elected as the most admired programming language currently in use, where 85% of the developers using Rust expressed a desire to keep doing so *(StackOverFlow, 2024)*. The language has many benefits, due to its focus on performance, type-safety, and concurrency. However, the benefits relevant to this project is primarily its innate memory-safety features. In default – or *safe* – Rust these features are not something a developer chooses to use or not, as they are enforced due to being a part of the language's strong type system, and not an external package. In spite of that, if needed, a developer can omit these safeguards by using the *unsafe* keyword. This will tell the compiler to allow unsafe code in that part, which means the developer has to manually ensure the code is safe.

The memory safety in Rust is handled through its ownership model that prevents a variable from escaping the scope it is in. Meaning, memory unsafe operations are caught during compile time, and the code will not work with such operations. This does not alleviate all errors or unsafe operations though, as race conditions are still the developer's responsibility *(Rustnomicon, 2024)*.

Part of the memory safety features that Rust provides, *use-after-free* errors are completely avoided, as they are syntactically wrong. *Use-after-free* errors allows unused pointers to reveal data on the location they point to. The issue arises when the data the pointers were originally intended for is cleared, but the pointer itself remains. *C* and *C++* has no issue with the developer leaving these pointers around, but Rust simply will not compile with them.

4.1.2 Hacspec

High Assurance Cryptography SPECification, or Hacspec, is a Domain Specific Language, DSL, for Rust. It makes it possible to increase the confidence in the security of an implementation, due to its `secret_integer` type. Hacspec also provides a translation to languages such as F*, EasyCrypt, or Coq enabling a proof engineer to prove the code using the languages respective proof assistants.

One of the advantages of Hacspec compared to other alternatives is that it is made for Rust, granting the guarantees of its memory safety. Another big difference between Hacspec and other similar tools, is these tools only prove a model and not the code itself. This means that it is up to the developer to make an accurate model of the code and then prove the desired properties. In contrast to proving the model, Hacspec – in conjunction with a proof assistant – allows a proof engineer to prove an executable implementation. This is done by utilising the Hacspec translator, to translate the Rust/Hacspec code into either F*, EasyCrypt, or Coq. Working with the code in a proof assistant can then reveal whether the original code is correct and can be verified to the specifications, or if it contains errors and/or bugs.

This allows a developer – when writing the rest of the application in Rust – to continue expanding their applications with cryptographic elements without these being segmented or in other ways being removed from the rest of the code. It would also be rather easy to read since Hacspec is essentially basic Rust code in its semantics.

This point is even stronger for verification itself since the modelling is often done in a completely different language than what the original code is written in. Hacspec, in contrast, makes it so the proof is done on the executable code itself. Meaning, depending on the type of change, if new alterations is made to the code, the developer does not have to manually go and change the proof in addition to the code. Additionally, if made part of a pipeline, Hacspec makes it so there is no risk of the proof suddenly being outdated or otherwise forgotten in the future.

A limitation that Hacspec introduces on the implementation, is restrictions with concern to what can be type-checked and translated. You are forced to use Hacspec, and cannot make use of libraries that are not a part of it. Additionally, due to the fact that Hacspec must be able to translate to fully deterministic languages, it does not support things such as generation of random numbers. This can make proving certain cryptographic protocols very challenging, if not impossible, as the randomness is a critical component. All protocols that have a *key generation* as a part of them, is hard – or impossible – to implement fully in Hacspec.

To dive deeper into the details, The Hacspec paper, *Merigoux et al., 2021*, introduces the inner workings of Hacspec. As our thesis is most preoccupied with the *secret integers* and *translating to Coq* – these are the parts of Hacspec we rely most on – we keep the deep dive limited to these.

The translation from Hacspec to Coq is an important feature for this project. It serves as the crucial bridge between the commonly used language of Rust – and the less used Hacspec – to the proof environment. Unfortunately, documentation for exactly how Hacspec's translator accomplishes this, is lacking. While searching for the documentation, we contacted the developers behind the project, and were refereed to a paper, (*Haselwarter et al., 2023*), that only explains in broad terms how Hacspec translates to functional languages.

In an attempt to form a deeper understanding of how Hacspec translates into Coq, we researched how it translates into *other* languages, namely F* and EasyCrypt, hoping the translations to other functional languages would indeed be similar to the translation to Coq. Our reasoning for researching other translations, is that while still not well documented, they have at least some information on the translation. In the paper from 2021, section 4.1 describes the translation to F* in broad terms, detailing only a small part of the overall translation. an example of the depth to which they explain could be:

"The translation is very regular, as each hacspec assignment is translated to an F let-binding"*
Merigoux et al., 2021, pp. 8.

Limited information on the specifics are mentioned, only that they are translated. A bit more information is presented on the non-regular examples. An example showing translated code and a comparison between the two is described. However, once again very little information on the translator itself is given. Which is further supported by a correspondence with the developers, where they state the documentation is lacking in regard to Coq – and SSProve

– and is a focus point for the future development of the new project Hax. Furthermore, we found no evidence that this translation is certain to be correct, Which is supported by presentation notes from the developers of Hacspec, (*Holdsbjerg-Larsen et al., 2022*), that as of 2022 states the translations has no proof of the soundness.

The *secret integers* of Hacspec is implemented with the goal of being *secret independent*. That is, the code never branches on a secret value, and never accesses an array if a secret integer was used as the index. To accomplish this, Hacspec uses the standard Rust typechecker, together with having made a library of secret machine integers. These encompass all integer types, whether signed or unsigned, that Rust support. This library defines operations that are known to be constant-time and none other. Meaning only operations that compile to constant-time instructions by assembly architectures are present. Therefore, they only support certain arithmetic operations, and namely *not* division and power. Hacspec also introduces the functions of `classify` and `declassify` – which converts between Rust integers and Hacspec secret integers – purposefully making these very obvious, as declassifying a secret integer can result in the code losing secret independence.

4.2 Proof Environment

The *Proof Environment* encapsulates all the dependencies used in and needed by SSProve and Coq. As visualised in Figure 1, we do not directly interact with the dependencies except for Coq and SSProve. SSProve has 3 dependencies to bridge the gap down to Coq, and while we briefly interact with them, they are only there to make SSProve function.

4.2.1 Nix

The proof environment is built in/setup with Nix, (*"Nix", 2024*), a package manager made to make consistent environments. This means if a package works on one machine it works on every machine, and through specified versions should continue to work even when the programs and dependencies are updated. This is done by building each package in isolation from one another. Because of this, one package cannot break another, making it easy to both add new packages as well as upgrade the existing packages. This tool additionally makes it trivial to share environments, which in conjunction with working on every machine with no further modifications, makes it ideal for reproducibility as the environment works as is.

4.2.2 Coq

Coq is a formal proof management system created in 1989 by Gérard Huet and Thierry Coquand. Coq and other similar tools exist as a faster alternative to creating manual proofs by hand. The large benefit of Coq, in comparison to for example ProVerif, is its nature as an actual programming language. A developer can in Coq write entire applications. Meaning, it is possible to have an entire application that has been proven to have certain guarantees, for example in respect to security goals.

While Hacspec will be the language ElGamal will be implemented and type checked in, Coq and its Proof Assistant will be the tool used to create the proofs. As mentioned in subsubsection 4.1.2, Hacspec enables a translation into Coq. This is what enables the process of using Coq to prove the implementation.

Coq has a set of specific terms that we will be using throughout the rest of the paper. Therefore, we here present a short list of these terms, with an explanation of what they are.

Tactic: A small program that changes the goal
Goal: A sub-proof of an overall proof that must be proven using tactics
Lemma: An assertion that must be proven using tactics
Theorem: Like *Lemma*, an assertion that must be proven using tactics.

As stated in *Paulin-Mohring, 2011*, Coq as a programming language has 2 levels in its architecture.

First, we have a simple kernel that is implemented to handle simple functions, product types, (co)-inductive definitions, and sorts in conjunction with rules on type checking and computation. This kernel is used to represent objects, functions and proofs. The other level to the architecture, that is built on top of the kernel, mainly concerns itself with ease of use. This environment helps designing theories and proofs. It allows for users to extend the notation and tactics of the language. Additionally, it also allows for the existence of libraries. This environment can safely be used to extend the language due to the low level kernel checking any definitions and proofs.

Another reason for using a 2 level architecture, is it allows the user to write code in a level of abstraction that is much easier to understand. As an example, the user can write $1 + 3 = 4$ where the lower level kernel will receive `@eq Z (Zplus (Zpos xH) (Zpos (xI xH))) (Zpos (xO (xO xH)))`. This expression would have to be written manually if not for the 2 level architecture. Additionally, the 2 level architecture allows one to write code in both levels of abstraction. Meaning even if a number is represented with the higher levels of abstraction, the 2 levels allows one to specify or refer to the variable using the lower level. A reason for this is, in the higher level of abstraction the program has to make certain assumptions in regards to types, which in most cases work seamlessly. However, in certain cases an error can occur due to an incorrectly inferred type. In these cases, one can then specify the input types using the lower level syntax.

While this 2 level abstraction is one of the strengths of Coq, it also comes with the downside of making code difficult to both read and debug due to the code base in certain cases suddenly changing abstraction level, or in other cases when writing code, one might find a tactic that seemingly fits perfectly, however, it incurs an error due to a hidden type mismatch. Even though these examples have workarounds, they still significantly raises the barrier of entry to the tool.

As stated in *Paulin-Mohring, 2011*, at the core of Coq is a logical system that implements higher order logic in conjunction with a simple functional programming language. Furthermore, Coq is intended to be an environment to build libraries of definitions and facts, with a tool-set that allows for querying the environment for relevant definitions and lemmas.

Coq implements a natural deduction system. That is, you first define a *proposition* (lemma, theorem or goal) that you intend to prove. This proof is then done by using the natural deduction elimination or introduction tactics. One important note is, that Coq is based on intuitionistic logic instead of classical logic. This means there are no implementation for the *law of the excluded middle* or *double negation elimination.* However, Coq does allow for creating custom axioms, meaning when needed it is possible to introduce these missing logical rules. This tool of creating custom axioms is one the user needs to be thoughtful of using, as it without any repercussions lets you state something to be true, meaning the program trust this statement explicitly.

As mentioned earlier in this section, Coq allows for libraries of tactics and proofs. This means that instead of only using the basic tactics of natural deduction one can also use any proofs or tactics that someone else has already made. This, in essence, does not change anything functionally as these fundamentally are a set of the predefined natural deduction tactics applied in a specific order, that someone else has found to eliminate or manipulate a goal. However, while not being functionally different from manually using the natural deduction tactics, it does significantly differ in the amount of work needed by the proof engineer.

With the proposition and a proof made, the proof can then be evaluated through the judgement $\Gamma \vdash p : A$ where Γ is the environment or more precisely a list of names and its associated values, also known as the context. This expression can be read as the term p of type A is provable in the context Γ.

4.2.3 SSProve

SSProve is a large library to help a proof engineer with a larger set of tactics to utilise when proving. more specifically, SSProve is a project that bases itself on the Coq Proof Assistant. The project attempts to create tool support for the *State-separating proofs*, SSP, methodology. The idea being, that the high-level modular way of structuring game-based cryptographic proofs is incredibly useful when combined with a probabilistic relational program logic for formalizing the lower-level details. SSProve does exactly this, enabling the construction of fully machine-checked crypto proofs using the Coq Proof Assistant. While SSProve is a significant library, we only scratch the surface of it in this project, as it allows us to work on the proof of correctness. And as such we have not gone into further details on the projects inner workings.

5 Methodology

In this section, we will present the methodology for the project, and we begin by explaining the three phases the project was separated into. Phase 1 largely revolves around the setup of the different tools, while phase 2 concerns the implementation in Hacspec together with translation to Coq, and phase 3 is the proving phase generally meaning the work in Coq. Lastly, the section ends with a description of the most important complications that made us reevaluate certain specifics throughout the project.

5.1 Phases

Our original plan was to first install Hacspec and through its library implement ElGamal in Rust. This includes an encryption, a decryption and a key-generation function. When done we would use Hacspec to translate the implementation to Coq. Our goal is to then create a proof of correctness for this translated code.

Our methodology can be separated into 3 distinct phase:

- Installation and setup phase
- Implementation, type-checking, and translation phase
- Proving and verification phase

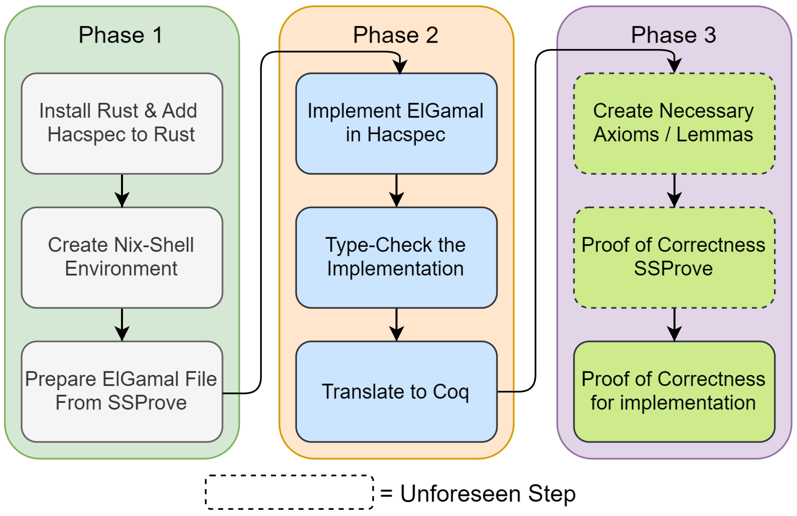

Phase 1
Install Rust & Add Hacspec to Rust
Create Nix-Shell Environment
Prepare ElGamal File From SSProve
Phase 2
Implement ElGamal in Hacspec
Type-Check the Implementation
Translate to Coq
Phase 3
Create Necessary Axioms / Lemmas
Proof of Correctness SSProve
Proof of Correctness for implementation
= Unforeseen Step

Figure 2: Visualisation of the 3 phases and what they entail.

5.1.1 Phase One: Installation and Setup

Phase one starts with utilizing Nix to set up the environment to get all the tools in the toolchain to function correctly. Our reason for doing so, is that

Nix creates an identical environment on whichever computer it is used, and as such makes our results significantly easier to reproduce. Details on what Nix is and how it works can be found in section 4.2.1.

Due to the toolchain containing multiple tools that are still in development – and some have since the start of this project been archived – many roadblocks were experienced during the initial set up phase of the project. However, Nix did eventually allow us to have such strict control of the environment, that all tools where made to function as desired. Our nix shell environment include all the tools and dependabilities to get Coq to function correctly. Hacspec was install directly into Rust as a *Cargo* crate, which is Rust's word for what is known as a library or package in many other languages. However, this too was a challenge, as the installation instructions from the Hacspec repository contains incorrect instructions.

5.1.2 Phase Two: Implementation, Type-checking, & Translation

Once the tools are installed and capable of working together, we move into the implementation and translation phase. The tools used in this phase are both Rust and Hacspec, further details can be found in subsubsection 4.1.1 and subsubsection 4.1.2 respectively. This phase is where ElGamal is implemented, and a main focus from the start is to use the Hacspec secret integers to benefit from the Hacspec type-checking to provide the guarantees of Hacspec; secret independence. Of course, as the programming language is Rust, we also benefit from the memory safety guarantees of it. A presentation of the final implementation of ElGamal can be found in section 6.

5.1.3 Phase Three: Proving and Verification

Post implementation and successful type-checking, the implementation can be translated into Coq using Hacspec's translation feature. With the translation correctly done, the project moved into the last phase; proving and verifying the implementation. This phase entirely depends on Coq as our proof assistant of choice. Our intended goal is to provide a proof of correctness for the implementation. A popular library for Coq named SSProve already contains a selection of proofs for ElGamal. As such, we decided to extend this file with our own implementation and proofs. This is due to much of the setup we would otherwise be required to make ourselves already being present here.

The idea is to prove *perfect indistinguishability* between a function that returns just the mapped message `m` – or the *ideal* function – and a function that behaves like the real ElGamal encryption/decryption does, or the *real* function. More detailed, the *real* function firstly *encrypts* the mapped message `m` by calling the `enc` function we implemented Hacspec and translated to Coq, and secondly calls the `dec` function – similarly implemented and translated – to *decrypt* the encrypted `m` and then return the result. As a lemma in Coq it is written as

```
Lemma indistinguishability : real ≈₀ ideal.
```

The symbol "$\approx_o$" denotes *perfect indistinguishability* in SSProve.

To do this, *real* and *ideal* has to be made as *packages*. For an explanation of what a package is[1]:

"In SSProve a package is a finite set of procedures that might contain calls to external procedures. The set it implements is called its export interface and the set on which it depends its import interface"
Haselwarter et al., 2023, pp. 4.

In our case, the *real* package would be the set of procedures that encrypts, decrypts, and then returns an `m`. And for *ideal*, it simply returns an unaltered `m`. The finished packages can be found in appendix 11.6 and 11.5, while the pseudo code version of *real* in appendix 11.4. All of which are presented in further details in subsection 6.3 and subsection 6.4.

5.2 Complications

As for complications in the project, when writing the ElGamal code in Hacspec we originally wanted to include a proper key generation function. However, when we finished writing the method and tried to translate the code to Coq, we got an error message *"unable to translate to Hacspec due to out-of-language errors"*. After researching this, we realised that no external libraries could be used in the translation. This, in addition to finding that the Hacspec library had no functionality for random functions, resulted in us deciding to have a static key in the code, that would be replaceable with a proper random function. This unfortunately means that our later proofs will be void of key generation.

A similar thing happened with choosing the group for our generator. The provided Hacspec examples had static values instead of real generators, further cementing our belief that Hacspec support no randomness. Because of this, we were left with 2 options. First, implement a library for choosing generators and key generation. Second, chose an arbitrary value that we know to be within specification and specify that our proof lacks this facet. While obviously the first option is most desirable, it is also out of scope for a project of the present one's size. Therefore, due to time constraints we decided on the second option, as we believe the proof retains most, if not all, of its value even with these modifications. Another master thesis, (*Schmidt and Bjerg, 2023*), similarly found Hacspec's restrictive nature to disallow easily using randomness, and likewise argued that their focus was on correctness, and therefore, believed this to have limited effect on their paper too.

In addition, we decided that our proof should be created in such a way that no further modifications to the proof should be needed if a random value is chosen.

[1]For more details on SSProve packages please see *Haselwarter et al., 2021, pp. 20-22.*

This means that we cannot rely on the specific value to prove any lemmas, and instead have to prove it using only strategies that rely on the types.

Many of the complications found in this project, stems from Hacspec being an unfinished product, as functions and types might not be well documented and therefore difficult to approach from the perspective of being a new user of the tool. An example would be, when we used the `bytes!` macro to create our types. As very little information on this macro is present and the implementation being quite obfuscated behind several calls to other classes and also being referred in classes without any direct indication of where to search for the method.

Even more challenging, was the fact that certain functions are not implemented yet. Such a case was present when we tried to raise g to the power of sk. Here the `pow` function was not implemented, and as such neither is the translation of the power function to Coq. For this we decided that the best approach is to create a Coq file that aliases the already present Coq implementation of pow. However, to make this automatic, we additionally needed to change the translation to include an import of the aforementioned file. Another possible approach for this is to change the translation layer to call the Coq pow directly. Looking through the file indicates it would be a larger rewrite than we believed would fit into the project, and as such the former option was chosen.

The last complication for the project is time. As proving lemmas is a significantly time consuming process, we decided to make use of the `admit` keyword in Coq. This allows one to remove a goal without proving it, which is both a strength and weakness of Coq, since you end up with an incomplete proof. For this project to finish in time, we allowed ourselves to make use of this functionality, however, only sparingly, and only when we had a good reason to believe the goal would be both time consuming to prove and obviously provable.

6 Results

In this section, we will present the results and findings the project revealed. Firstly, we will present the Rust/Hacspec implementations and specify how they have been implemented. Secondly, we present the Coq implementation. Following, two sections on the proofs of correctness that was completed is explained, the latter of the two also contains the axioms we found to be necessary. Lastly, this section ends on a brief description of certain unexpected learnings.

The results of the project is a standard implementation of ElGamal except for 2 things. Firstly, we used the secret integer type from Hacspec on the variables that has to be secret, and therefore the code also includes some conversions to and from said secret integer. Secondly, a modification on the `keygen` function to consistently returns 4 as the secret key instead of a randomly generated key was made, due to Hacspec not supporting any library that includes random number generation. Next, we have a translation from the aforementioned ElGamal implementation to an equivalent implementation in Coq. The translated `keygen`, `enc`, and `dec` functions can be found in the appendix 11.10, 11.11, and 11.12.

6.1 The Rust/Hacspec Implementation

As described in our methodology subsubsection 5.1.1, an implementation of ElGamal in Rust/Hacspec was made. The full implementation can be seen in appendix 11.13. In this implementation we have used Hacspec's `secret_integer` type, see subsubsection 4.1.2, for details on the motivation for using it. To make it clear when this type is used, we employ the naming convention of prepending `secret` to the variable name. An example of could be:

```
Let secret_m = U128::classify(m);
```

where we use the classify function to convert `m` to the secret type.

As described in section 2, ElGamal has 3 main parts; key generation, encryption and decryption. The key generation function we implemented can be seen in Figure 3.

```
pub fn keygen() -> (u128, U128){
    let secret_sk = U128::classify(sk);
    let pk =
    ↪  (SECRET_G.pow_mod(secret_sk,SECRET_Q)).declassify();
    (pk,secret_sk)
}
```

Figure 3: Hacspec `KeyGen` Function

In this function, we unfortunately had to use a static key instead of a random one. As previously mentioned, this is due to Hacspec not allowing any external libraries. Meaning no random functions are supported in the current version of Hacspec. However, if such functionality should be added to Hacspec, one could use it instead of the `sk` variable on line 2.

To describe the function: we first convert the secret key `sk` to the secret integer type. Next, we create the public key by raising g to the power of the secret key. This variable is then declassified, meaning we convert it back to none secret unsigned integer as the public key has no reason to use the secret type. The two variables – the public key and secret key – is then returned as a pair.

Second we have the encryption function:

```
1   pub fn enc(target_pk: u128, m: u128) -> (u128, u128){
2           let (source_pk,secret_source_sk) = keygen();
3           let secret_target_pk = U128::classify(target_pk);
4           let secret_m         = U128::classify(m);
5
6           let secret_s =
             ↪  secret_target_pk.pow_mod(secret_source_sk,SECRET_Q);
7           let c1 = source_pk;
8           let c2 = (secret_m * secret_s).declassify();
9
10          (c1, c2)
11  }
```

Figure 4: Hacspec Encryption Function

Here we first call the `keygen` function to get our secret key, followed by type conversions to the secret integer type. Even the public variables are converted, as the type needs to be consistent to use mathematical operations. On line 6-8, we calculate the session key and the ciphertext. For further information on the mathematics see subsection 2.2. Finally, on line 10 we return the ciphertext as a pair.

Lastly we have the decryption function, seen in Figure 5 Similar to the earlier

```
1   pub fn dec(secret_target_sk: U128, c: (u128,u128)) -> u128 {
2           let (c1,c2)          = c;
3           let secret_c1        = U128::classify(c1);
4           let secret_c2        = U128::classify(c2);
5
6           let secret_s_inverse =
             ↪  secret_c1.pow_mod((-secret_target_sk),SECRET_Q);
7           let secret_m         = secret_c2 * secret_s_inverse;
8           let m                = secret_m.declassify();
9           m
10  }
```

Figure 5: Hacspec Decryption Function

functions, the decryption function starts with type conversion to the secret type. Then the modular multiplicative inverse of the session key is calculated. Finally, the message is calculated and returned. The mathematics can be found in further detail in subsection 2.3.

Currently, the `pow` function we used in ElGamal has no implementation in Hacspec. This means that until this is implemented, the Rust/Hacspec ElGamal file will not execute. Similarly to the case regarding key generation, we were restricted from using any other implementation as Hacspec requires one to only use its libraries.

Luckily, the translation to Coq is still possible. However, this translation will call a function that does not exist. Therefore, we have created a Coq file that handles the power function, by simply redirecting to the inbuilt `pow` function in Coq. This file can be seen in the appendix 11.1. In addition, we have made a change in Hacspecs translation code to include an import of the `powmod.v` file:

```
From ElGamal Require Import powmod. \n\
```

This line was added to the file `rustspec_to_coq.rs` file from Hacspec[2] which handles the part of the translation from Hacspec to Coq that writes the `imports`. It specifies that the `powmod` file shall be imported, from the *ElGamal* library, which is the library we have made.

6.2 The Coq Implementation

With the translation to Coq completed, we then create a proof of correctness for the pseudo code implementation of ElGamal as a proof of concept for the more complicated executable implementation. However, before looking at the lemma, one need to understand the `Enc`, `Dec_open` and `KeyGen` functions from the SSProve file.

The `KeyGen` function, see figure 6, uses the sample uniform function on line 4 to generate a random secret key. On line 5: a type conversion from `Ordinal` to the `'fin` type, and lastly on line 6 the pair of g^x (`pk`) and x is returned, where x is the secret key. Before the return statement both variables are converted back to the `Ordinal` type. This conversion between `Ordinal` and `'fin` is due to ordinal being the desired type, however no mathematics can be made on this type, and as such a temporary conversion is needed.

```
1  Definition KeyGen {L : {fset Location}} :
2      code L [interface] (chPubKey × chSecKey) :=
3      {code
4        x ← sample uniform i_sk ;;
5        let x := otf x in
6        ret (fto (g^+x), fto x)
7      }.
```

Figure 6: Translated KeyGen function

[2]https://github.com/hacspec/hacspec/blob/master/language/src/rustspec_to_coq.rs#L1876.

The `Enc` function takes the parameter `pk` and `m`, and similar to the `KeyGen` function, generates a secret key y using the sample uniform function, and on line 6 returns the pair of g^y and $pk^y \cdot m$ where pk is g^x. This function like `KeyGen` changes types to `'fin` for the mathematics and then changes back to `Ordinal`. See figure 7.

```
1 Definition Enc {L : {fset Location}} (pk : chPubKey) (m :
  ↪  chPlain) :
2     code L [interface] chCipher :=
3     {code
4       y ← sample uniform i_sk ;;
5       let y := otf y in
6       ret (fto (g^+y, (otf pk)^+y * (otf m)))
7     }.
```

Figure 7: Translated Encryption function

The `Dec` function, see figure 8, like the other two changes the type to `'fin` to do the mathematics and changes back to `Ordinal` type before returning the output. This function returns $c1^{-sk} \cdot c2$ which in practice is $g^{y^{-x}} \cdot g^{y^{x}} \cdot m$ which is equivalent to returning m. See subsection 2.3 for an explanation of the mathematics.

```
Definition Dec_open {L : {fset Location}} (sk : chSecKey) (c
  ↪  : chCipher) :
    code L [interface] chPlain :=
    {code
      ret (fto ((fst (otf c))^-(otf sk) * ((snd (otf c)))))
    }.
```

Figure 8: Translated Decryption function

Now with an understanding of the underlying functions, we can look at the proof of correctness. This is done by proving the following lemma:

`Lemma Enc_Dec_Perfect : Enc_Dec_real` $\approx_o$ `Enc_Dec_ideal.`

Here `Enc_Dec_ideal` is a package that returns a randomly generated message `m` without any manipulation, while `Enc_Dec_real` is a package we built that uses the `Enc`, `Dec` and `KeyGen` functions from SSProve's ElGamal implementation. This can be seen in figure 9. `Enc_Dec_real` takes the same randomly generated message `m` as the `Enc_Dec_ideal` package, and encrypts it using a public and private key generated from the `KeyGen` function. This encryption is then decrypted with the same key by the decryption function. This together means that we have to prove that a message that is encrypted, then decrypted is perfectly indistinguishable from the original message, exactly as described in section 5.1.3.

```
Definition Enc_Dec_real :
  package fset0 [interface]
    [interface #val #[10] : 'plain → 'plain ] :=
    [package
      #def #[10] (m : 'plain) : 'plain
      {
        '(pk, sk) ← KeyGen ;;
        c ← Enc pk m ;;
        m ← Dec_open sk c ;;
        ret m
      }
    ].
```

Figure 9: Enc_Dec_real package

6.3 Proof of Correctness: Pseudo Code ElGamal

The proof can be roughly split into 3 stages. The first one can be seen on figure 10. The main functionality here is to expand `Enc_Dec_real` and `Enc_Dec_ideal` to their actual content instead of just names referring to the content.

```
eapply eq_rel_perf_ind_eq.
simplify_eq_rel m.
apply r_const_sample_L.
2: intros sk.
2: apply r_const_sample_L.
1,2: apply LosslessOp_uniform.
intros pk.
apply r_ret.
intros s0 s1 e1.
```

Figure 10: Setup stage.

The second stage regards the removal of type conversions or otherwise change the form into clear mathematics. This can be seen in figure 11. An example of this could be a goal that has `fto (otf 4)` as a part of it. Here `fto` and `otf` are type conversions that cancel each other out, and as such can be removed.

```
split.
2: apply e1.
repeat rewrite otf_fto.
simpl.
```

Figure 11: Removal of type conversions.

The last stage is where the mathematics are proven. This stage can be seen in figure 12. Here general mathematical rules can be applied until the two sides of the equation are reflexive. The specifics of the mathematics can be seen in figure 13.

```
rewrite -2!expgM.
rewrite mulnC.
rewrite mulgA.
rewrite mulVg.
rewrite mul1g.
rewrite fto_otf.
reflexivity.
```

Figure 12: Mathematics stage.

With this, the proof has reached the state of $m = m$ and, thus, the proof is done. For a detailed step-by-step explanation for each tactic used, please see appendix 11.16.

While creating this proof, we discovered an error in SSProve's pseudo code implementation of ElGamal. Specifically, the $Dec_o pen$ part of the algorithm was incorrect. It was implemented as

```
m = (fto ((fst (otf c)) * ((snd (otf c)) ^ -(otf sk))))
```

Which translated into mathematical notation would be equivalent with

$$m = c_1 \cdot c_2^{-sk}$$

However, remembering the variables to be

$$c_1 = g^y \qquad c_2 = g^{x^y} \cdot m \qquad sk = x$$

And applying the mathematics from their implementation, we end with

$$\begin{aligned} m &= g^y \cdot (g^{xy} \cdot m)^{-x} \\ &= g^{y + xy^{-x}} \cdot m^{-x} \end{aligned}$$

The issue here is obviously that c_2 should not be raised to the power of $-sk$. It is a simple mistake to make, however, it could have been realised early, had SSProve also had a proof of correctness in its ElGamal implementation. Having spotted this error, and seeing the obvious need for a proof of correctness, we decided to correct both of these issues.

Therefore, we firstly changed the SSProve implementation to correctly reflect the decryption of ElGamal, namely

The code that then rightly reflects this correction would be

```
m = (fto ((fst (otf c))^-(otf sk) * ((snd (otf c)))))
```

With this correction we were able to finish the proof of correctness for the pseudo code implementation of ElGamal. The proof can be found in *ElGamal.v*, and is the lemma `Enc_Dec_Perfect`, as well as in appendix *11.2*. Afterwards, the rest of the proofs created by the SSProve team were verified to still compile.

$$\begin{aligned} m &= c_1^{-sk} \cdot c_2 \\ &= g^{y^{-x}} \cdot g^{x^y} \cdot m \\ &= g^{-xy+xy} \cdot m \\ &= 1 \cdot m \\ &= m \end{aligned}$$

Figure 13: Showing m can be decrypted from c_1, $-sk$, and c_2

6.4 Proof of Correctness: Implemented ElGamal

Next we have the proof of correctness for the Coq implementation. This proof is inserted into the proof file for the pseudo code ElGamal from SSProve due to this shortening the time needed to set up before implementing proofs. As this proof is significantly larger than the previous, we will avoid going into details on the specifics. We will instead present the required knowledge to read the code, while the entirety of the proof can be found in appendix 11.3.

The code roughly mirrors the previous proof, but with a much larger cleanup phase. To stay consistent with the document, we decided that `m` should have the same type as in the other proofs. Due to this, we have `m` as a `choiceType`, or more specifically the `Ordinal` type. This `Ordinal` type has an important trait to know for our project. That is, you cannot do mathematics directly on this type. And as such, we require the `fintype` instead. Luckily, there is a cast between these types, and as such no further work is needed for this transition. While closely related to the `Ordinal` type, the `fintype` allows for mathematics, additionally this type has a built in modulo, meaning we can use this to enforce `mod q` but on the type itself instead of using the modulo function.

Now that we have a bridge between `fin` and `ordinal`, we need to move closer to the `MachineInteger` type that our `enc`, `dec` and `keygen` functions use. Here, we make use of the fact that `Ordinal` lets the prover use a coercion to the `Nat` type, where *"coercion"* is a type of casting that happens automatically, and only exists for types that are equivalent without any further proofs. With this, we have a direct link between `fin`, `Ordinal` and `nat`. Next, we have that `nat` has a tactic that translate to and from Coq's `BinNums.Z` type, which the `MachineIntegers` library has a cast to and from. This means, we can cast from the `Ordinal` type – which `m` is – to the `MachineInteger` type that our ElGamal implementation uses.

As some of the coercions make this translation cumbersome, we decided to create functions that translate directly. This is due to coersions not directly showing up in goals, making it much less clear what is happening and why. The different castings we use can be seen in figure 14. Here the green colour represents the functions we created, the red colour is SSProve functions, orange is from BinInt.Z, and lastly, blue is from `MachineIntegers`.

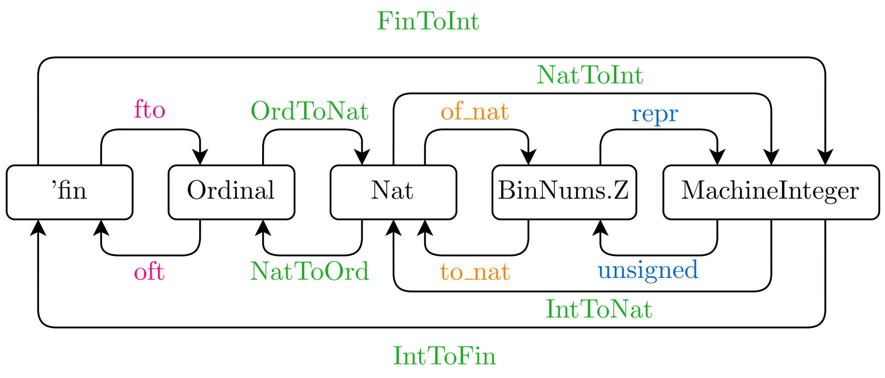


Figure 14: Casting.

In addition to the conversions, we created lemmas that the reader needs to be aware of to understand the proof. The first one, `FinToInt_IntToFin_Eq` can be seen on figure 15. This lemma enables the removal of a conversion from `int` to `fin`, and then back to `int`. However, to do so it first has to be proven that the value converted is smaller than what the `fintype`'s inbuilt modulo value is, as otherwise this conversion would change the value.

```
Lemma FinToInt_IntToFin_Eq {k:nat} `{Positive k}
  {H1: BinInt.Z.le (BinInt.Z.of_nat k)
  (@MachineIntegers.max_unsigned MachineIntegers.WORDSIZE128)}:
    ∀ {n: MachineIntegers.int128},
      @FinToInt k _ (IntToFin n) =
      MachineIntegers.modu n (NatToInt k).
```

Figure 15: `FinToInt_IntToFin_Eq` lemma.

Similarly, we created the lemma `NatToInt_IntToNat_Eq`. This one, in contrast to the previous one, does not require proving of any additional goals as none of the types inherently change the value. As such, the following lemma was created to remove any instance of converting to `int` from `nat`, then back to `int`. The lemma can be seen in Figure 16

```
  Lemma NatToInt_IntToNat_Eq:
    ∀ {n: Hacspec_Lib.uint128} ,
      NatToInt (IntToNat n) = n.
```

Figure 16: `NatToInt_IntToNat_Eq` lemma.

Next, the lemma `reprmod`, which can be seen in figure 17. Here we prove that if the `MachineIntegers.repr` function is used on a number modulo `q`, then you can remove the modulo `q` part because `repr` is already defined to use modulo. This lemma leaves you with two additional goals to prove. One is that the value is larger than 0, and the other is that the number is smaller than `maxint`. These are fairly simple to prove as any variable we intent to use are within this range.

```
Lemma reprmod :
∀ {n: BinNums.Z} {q: nat} {h: Positive q}
{H1: BinInt.Z.le BinNums.Z0 n} {H2: BinInt.Z.le n
↪  (@MachineIntegers.max_unsigned
↪  MachineIntegers.WORDSIZE128)},
  IntToFin (MachineIntegers.repr n) =
  IntToFin (MachineIntegers.repr
      (BinInt.Z.modulo n (BinInt.Z.of_nat q))).
```

Figure 17: `reprmod` lemma.

Finally, we have the lemma `g_gt_eq`, which together with the definition `q_eq` lets us set the modulus of the `fintype` to be `q`. This lemma and definition can be seen in figure 18.

```
Parameter q_eq : BinInt.Z.of_nat #[g] =
                 MachineIntegers.unsigned q_v.
Lemma g_gt_eq  : #|gT| = #[g].
```

Figure 18: `g_gt_eq` lemma and `q_eq` definition.

Last part needed to understand the proof is the axioms. Unlike proofs, these are assertions we define to be true and as such does not need any work to be proven. The first 3 simply removes wrappers that has no definitions other than being a wrapper, these can be seen in figure 19.

```
Axiom Remove_Classify   :
  ∀ {n: MachineIntegers.int128},
    Hacspec_Lib.uint128_classify(n) = n.

Axiom Remove_Declassify  :
  ∀ {n: MachineIntegers.int128},
    Hacspec_Lib.uint128_declassify(n) = n.

Axiom Remove_Secret :
  ∀ {n: MachineIntegers.int128},
    Hacspec_Lib.secret n = n.
```

Figure 19: Remove `Classify`, `Declassify` and `Secret`.

Next, we have the `unsigned_repr` axiom, that can be seen in figure 20. As `MachineIntegers.unsigned` is a conversion from `MachineIntegers.int` to `BinNums.Z` and `MachineIntegers.repr` is a conversion from `BinNums.Z` to

`MachineIntegers.int`, we decided to create an axiom that removes instances of them being next to each other. That is, if anywhere in the proof we find `MachineIntegers.unsigned (MachineIntegers.repr n)` it can then be converted to `n` without any further proof needed.

```
Axiom unsigned_repr :
  ∀ {n: BinNums.Z},
    @MachineIntegers.unsigned MachineIntegers.WORDSIZE128
    ↪  (MachineIntegers.repr n) = n.
```

Figure 20: The `Unsigned_repr` axiom.

The second to last axiom can be seen on figure 21. Here we remove modulo `#|gT|` as it is equivalent with `q` as per the `g_gt_eq` lemma, and `n` is defined to be less than `q`.

```
Axiom Remove_mod_gT :
  ∀ {n: BinNums.Z},
    (BinInt.Z.modulo n (BinInt.Z.of_nat #|gT|)) = n.
```

Figure 21: The `Remove_mod_gT` axiom.

Finally we have the axiom `mul_inv`. This axiom can be seen in figure 22. It states that if a number is multiplied with its inverse, it can be replaced with the number 1.

```
Axiom mul_inv:
  ∀ {n z: BinNums.Z},
  BinInt.Z.mul
    (OrdersEx.Z_as_OT.pow n z)
    (BinInt.Z.pow n (BinInt.Z.opp z)) =
        (BinNums.Zpos 1%AC).
```

Figure 22: The `mul_inv` axiom.

Hereby, everything needed to understand the proof of correctness is explained. The proof can be found in the appendix 11 or on our GitHub here[3] due to its length.

With the proof as close to complete as time allows us, using only axioms we are reasonably confident can be proven, we show that it is actually feasible to mechanise the proof of an *executable* implementation.

[3]Ancher and Jensen, 2024

7 Discussion

In this section we will discuss the various results and observations we made in the during the project. While we set out to analyse the viability of mechanising a proof of correctness for an ElGamal implementation, we made separate learnings along the project which too are worthwhile reflections on the project as a whole. We begin the section with highlighting the benefits and limitations of Hacspec, followed by reflecting on proofs made in Coq. Lastly, we discuss proving executable implementations, and its less obvious benefits.

7.1 Benefits & Limitations of Hacspec

Hacspec as it is right now requires the user to only use base Rust and the Hacspec libraries. This greatly limits the kind of implementations one can make. As an example, in the ElGamal implementation, generation of a random key is needed. However, there is no such generator in Hacspec and if attempted with a different library one will meet the error message `"unable to translate to Hacspec due to out-of-language errors"` when translating. Because of these limitations, we were left with 2 options.

One: implement the random function ourselves, or two: use a static key instead. We decided not to implement the randomness, as it would be both a time consuming process and it would increase the complexity of the proof significantly. Instead we chose the second option of using a static key. This option is the least time consuming one, however, it has its own downsides as well. When we later want to prove the implementation, we have to make sure that the proof holds even for more general cases than what the code itself requires. As an example, in the implementation we have to prove that the public key multiplied with the message is less than `maxint`, which when using a low enough static key is trivial. However, should we in a real implementation wish to use a random key, and not have proven the general case, the proof would have little to no value.

Therefore, when using Hacspec, it is important to be mindful of the limitations of it in such a way, that the proof works despite the limitations and not *because* of the limitations. This – in a more practical sense – means that we, in the proof, have to work with tactics that manipulate types instead of ones that rely on specific values.

If one wish to replicate our project, the aforementioned points should be central when proving. However, it is noteworthy that in our proof we never felt required to rely on any tactics that were for specific numbers. And as such, at least in our case, the proof worked in a more general case even without additional alterations.

The before mentioned points regarding lack of key generation additionally mean, that even when the proof is completed, one has to blindly trust the key generation methods if one cannot prove or otherwise validate this part of the implementation. In spite of the fact that the proof should work with this new value, the key generation function itself might have bugs or security risks.

An important detail to remember regarding Hacspec, is that it is still a work in progress. Therefore, one might run into functions that are not yet implemented. Such a scenario happened in this project, where the `pow` function was missing. This means that while the program was compiling and translating to Coq, it had translated using a function that did not exist and, therefore, the Coq code could not compile. To combat this, we changed the code of the translator to import a function we had created which handles the power and modulo functions. While this may at first glace seem like an easy solution, it further complicates the process and increases the points of failure. Because not only must you trust the Hacspec translation but also your own modifications.

Similar to the key generation problem, we likewise had the option of implementing `pow` in both Hacspec and the translation. However, one of the attributes of the Hacspec libraries is that they protect from timing attacks. They do so by only allowing arithmetic expressions that is known to be constant time, and as such the pow function would be much more complicated to implement than what we originally thought. Furthermore, we have a suspicion that this method is even more so complicated than other similar methods (addition multiplication etc.) as they have constant time calculations whereas power functions only has this attribute, in cases where arbitrary precision is acceptable. And in cases regarding security, we believe the precision to be vital, and as such creating a power function that is both acceptable in regard to precision and constant time is both way out of scope for this project, and it is currently uncertain if it is even possible.

Due to these concerns, we decided to only implement power in Coq, meaning our Hacspec code does not function. However if Hacspec updates their library with a power implementation, our Hacspec ElGamal code should work as is. While timing attacks are a concern for the security of the program, this project only concerns itself with the correctness of the code, and as such we do not have any problem in regards to the Coq code having a power function, even with the knowledge that the security might be in jeopardy.

7.2 The Proofs Made In Coq

After implementing ElGamal in Hacspec and translating to Coq, the next step was to prove this implementation. For this, we decided to make use of the pseudo code implementation of ElGamal that is provided in SSProve itself. By inserting our implementation of ElGamal in SSProve's ElGamal file, much of the setup required would already be provided. Our initial approach was to implement a proof of correctness on our Hacspec implementation. However, we discovered that the SSProve implementation lacked such a proof, and due to the pseudo code implementation being much simpler than an executable one, we decided to implement the proof for this first. Furthermore, if the pseudo code proof is impossible to make then working on an executable one makes close to no sense.

While working on the proof of correctness for the executable implementation, a goal persisted that require us to prove $sessionkey \cdot m \leq maxint$. Which we very likely can prove due to sk being static as 4 in the code. However, if we were to use a proper key generation function, it would be impossible to prove as the variables are of a finite type. After doing some research on why this goal occurred, we realised that in our code when we multiply m with session key, there is nothing that prevents the code from overflowing, as Rust handles variables larger than maxint with a reduction modulo maxint. This means there are no security issues with using too large numbers, however, it will result in the algorithm returning a wrong message even with input that is within specifications, if those numbers are large enough.

We decided to solve this with the axiom that utilises the inbuilt modding in the fin type to automatically mod with q. And as such, the variables are smaller than maxint for the purpose of the proof. However unless an infinite datatype is implemented in Hacspec, the error will persist either in the form of incorrect axioms or error prone code in the area of large keys. With $session\_key$ being reduced modulo the order of the group, and m being defined to be less than the order, an alternative solution, is to specify the cyclic group to be small enough that $session\_key \cdot m$ is smaller than maxint. However, as this is outside of this paper's scope we can only leave it as a suggestion and warning for anyone attempting to work with key generation in the Hacspec library.

When writing proofs in Coq, a way to reach the end is to use the `admit` keyword on goals that seem provable. This removes the goal, but requires that when the proof is finished, one must go back and prove the admitted goals. In the proof of correctness for executable implementation of ElGamal, we used this `admit` keyword on certain goals. Due to time constraints, these goals have been left to be proven. Although, we believe these to indeed be provable. In the first example of an `admit`, we need to prove that a natural number is larger than 0, which it is by definition. In the second example, we must prove that `q` is less than or equal `max-unsigned-int`. This should be provable, due to q being an unsigned integer and, therefore, by definition is less than or equal max int. For the next example, we need to prove that `sk` is less than `q`, which it is by definition of `sk` being chosen from `{1, ..., q-1}`. Lastly, we need to prove that `m` is less than max unsigned int. This is true as `m` is bound by `q`, which in turn is an unsigned integer, and thereby less than max unsigned integer by definition. While all of these axioms seem logical, easily provable, and they have certain type conversions that can be removed, this process is time consuming and, therefore, unreachable in the time frame of this project.

As first introduced in subsection 3.3, it is interesting to discussing how Coq manage the roles of a proof; firstly, to convince a reader the proof is correct. To do this, we can see how well Coq manages to follow the 4 ways Geuvers describes proof assistants can increase confidence from the user. The Coq proof assistant follows all four ways of increasing trust in its correctness. Firstly, Coq's specification of the logic is detailed and well documented. Secondly, Coq has a relatively small kernel. Thirdly, Coq has been verified inside of Coq itself, which, as Geuvers notes, increases confidence twice, as it means the specification of the logic must also be available. And lastly, as hinted to in the second point, Coq's small kernel enables it to follow the Bruijn criterion, as any user can test the small kernel themselves.

To the second role of a proof, that being a proof should explain why the statement is correct, proof assistants have, and are still struggling to do this. For example in Coq, while each proof can be executed line by line by any user, understanding why each step has been taken is not at all simple. Although attempts to make the tactic names *somewhat* indicative of what they do, learning to decipher these can certainly be a challenge. Certain libraries have tactics which names are completely unintuitive, while tactics that automatically attempt to move the proof further towards the goal(s), have names like `done` and `lia`, while the tactic `simpl` could indicate a simplification of the goal(s), though at times it does the opposite.

Following a proof in Coq can be difficult and the tactics can appear almost *magical* at times. However, the CoqIDE – the interactive Coq proof assistant – has certain features to alleviate some of the annoyances, like its *context* window, *goal(s)* window, and the ability to search for and look-up tactics, all help to explain what is going on. However, Geuvers second role of a proof is still difficult for these proof assistants to do, meaning a proof engineer can be necessary to understand the steps taken.

7.3 Proving Implementations & Its Hidden Benefits

Writing code that compiles is much easier than correct code, and even more so secure code, as there is a clear indication when it compiles. However, correct and secure code has little to no indication. Which is why using proofs to find bugs and improve code is a great tool, due to it allowing for compile time errors based on the lemmas written. One could argue, that this simply moves the problem from writing correct code to writing correct lemmas. While this point has merit, it is important to remember that this tool is something you can use in parallel with other methods of writing correct code, and, as can be seen in the above mentioned examples, gives you an additional way of finding bugs or other ways improve your code.

Having the ability to prove implementations have a lot of positives. However, we find a few downsides as well. If you, for example, for any reason want to change the implementation, it can potentially make the rewrite take much longer, as the proof could have to be changed as well. As an example of this, during our project a change from `uint128` to an infinite data type was discussed. However, we inevitably chose to not change type due to `uint` being central to the secret type in Hacspec. We also realised that changing this type would require us to either start over with the proof or at least change a substantial part of it. With this example, we realised that having a proof bound to your code can significantly increases the workload when updating code, which, increasing with the size of the code base, can serve as a disincentive to make changes. Therefore, a balancing act of when to use this tool is essential. However, in a security setting we believe this to be worth the extra time to implement as having proofs gives you certainty in the implementation being correct.

Two areas that closely relate to our project, are the formal proofs and verification tools. However, one key difference is that traditional formal proofs often task themselves with proving something theoretical, while verification tools – for example ProVerif – often only regard a *model* of the real code and not the code itself. Because of this, even when traditional theoretical proofs have been proven, it is easy to find one self in a situation where the result and what impact the proof has on the implementation, can be difficult to discern. Proving the theoretical implementation, only determines if the algorithm *can* be implemented correctly, not if an implementation *is* implemented correctly. And yet, you have not tested issues such as the chosen data types perhaps having an unforeseen impact on the implementation.

Which brings the stark contrast of looking at our toolchain. It not only allows for a proof on an executable implementation, but also gives you certainty in what this proof actually means. As in our example, it is very clear what exactly the proof means and equally important; what it *does not* mean. Here we have a proof that shows that when you use our encryption function followed by our decryption function, you will always get the same message out that you inserted (if using a message and key generation function small enough to stay within max integer). As of right now, we have only showed that a proof of correctness is possible using our toolchain. However, we see no reason that other proofs should not be equally possible in regards to, for example, security. Which would let someone prove everything we already know about a theoretical implementation on a concrete one. This would potentially not only give developers a great tool, but also remove a common security concern; errors in the implementation and not the protocol itself. Furthermore, it would also make it much clearer exactly what the implementations does and does *not* do, which can be used both as a selling point, but also simply for communication as knowing exactly what an implementation does, with a proof to back it up gives you certainty in arguments for and against using different protocols.

7.4 Unexpected Learnings

While not directly related to our research question, other important learnings were discovered during the project. Such as learning that, while proving executable implementations is not a widely used practice, it has some major benefits insofar that it can help identify problems or bugs in the implementation that would otherwise have been difficult to detect. Furthermore, it helps the proof engineer to understand the intricacies of the implementation.

This is exemplified when we in the proof found a tactic that perfectly fit the current state of the given goal, except we used modulo at one point where the tactic did not. This made us look at the mathematics behind this small part of the code, leading us to find that using modulo here is an error. In this example, we found a bug due to the proof being infeasible with how the code was implemented at that time. However, while this example clearly shows finding a bug and learning through proving, we believe the strength in finding bugs through proofs lie even stronger in the ability to find bugs caused by side effects of decisions, instead of specification errors that can also easily be found by creating a proper testing suite. An example of this is described in the paragraph below, where we used a finite datatype instead of an infinite one.

In the process of proving our ElGamal implementation, we reached an impasse when the code asked us to prove that $g^{x^y} \cdot m$ is less than max unsigned integer. This appears possible to prove with the current implementation of using 4 as the secret key, however impossible with a real key generation function due to us using machine integers which are finite and therefore makes the program incorrect in large enough numbers. The fix for this issue is to either use an infinite data type such as the `bigInt_integers` type that Hacspec provides. Unfortunately, we found no evidence that the translator translates `bigInt_integers` and, therefore, we are left with the second option of trusting that the key generation function used in a real example, will use small enough numbers that reaching max unsigned integer is impossible. More specifically, we need $g^{x^y} \cdot m$ to be less than max unsigned integer as this value is stored as is and not reduced modulo q unlike the other variables that could go beyond. We temporarily fixed this with the axiom that defines `fin` to be `mod q` and, thereby, bound this calculation to be much smaller than max unsigned integer.

8 Conclusion

We have in this thesis shown a clear example of mechanising a proof of correctness of an ElGamal implementation written in Hacspec. Post implementation, the proof was made in the Coq Proof Assistant, utilising aspects from the SSProve library. A limitation of Hacspec restricted the implementation to a static key, instead of a randomly generated one. Despite this, we believe our proof of correctness to still be correct and valid for randomly generated keys, as we did not allow ourselves to utilise strategies in the proving process to be for specific numbers.

While Hacspec provides desirable benefits, a developer not familiar with proving in Coq would be greatly challenged to make good use of the tools. The steep learning curve as well as the cumbersome challenges relating to the process' many types and casting is a significant barrier of entry.

Additionally, Hacspec made us compromise in our implementation, and such compromises might change what can be proven. Indeed, it might make a proof incorrect, as it could only relate to the specific compromise, and not the general case. This is definitely a consideration one must have when assessing the usefulness of the tools, and their benefit for proving executable code.

In conclusion, this thesis presents a working example of a toolchain that allows proving specific facets of an *executable* implementation, and not simply a model, showing the promising future of mechanising proofs.

9 Further Work

Due to the obvious time and man-hour limitations a master thesis imposes, multiple directions of research have been identified as worthwhile endeavours to pursue that this thesis was unable to do.

Firstly, using the other proof assistants – that Hacspec is able to convert to – has not been attempted in this thesis. The focus was early chosen to be Coq, leaving EasyCrypt and F* under-researched. We cannot rule out, that the effort of proving an implementation could be both easier but also simpler and quicker with these other tools.

Secondly, while working on this thesis, the Hacspec project was archived from GitHub. The developers behind it, has moved to the wider project of *Hax*, which aims to alleviated some of the difficulties of being bound to a single language and library. Hax uses Hacspec, and as such expands on what the tools are capable of. While this would of course have been preferable if it happened before we began the thesis project, this thesis could be the groundwork for further analysing the expanded Hax, researching how some of the pain points that troubled this thesis have been addressed in Hax.

Thirdly, implementing other proofs than a proof of correctness, namely proofs regarding security is obviously of great value as this would enable having a running implementation of a cryptographical protocol that is proven to be both correct and secure could be reached.

Finally, the nix environment does not include the Hacspec installation. As this is not a package in the nix package manager, however this should still be possible through manually inserting the relevant command lines to clone and install the git repository, and with specifying the commit, it should still keep with Nix's core values of working even when programs have been updated.

10 References

Forbes. (2024, February 28). *Cybersecurity stats: Facts and figures you should know*. Retrieved May 24, 2024, from https://www.forbes.com/advisor/education/it-and-tech/cybersecurity-statistics/

AAG. (2024, May 1). *The latest 2024 cyber crime statistics (updated may 2024)*. Retrieved May 24, 2024, from https://aag-it.com/the-latest-cyber-crime-statistics/

*F**. (2024, May 21). Retrieved May 21, 2024, from https://www.fstar-lang.org/

Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., & Summers, A. J. (2022). The prusti project: Formal verification for rust. In J. V. Deshmukh, K. Havelund, & I. Perez (Eds.), *Nasa formal methods* (pp. 88–108). Springer International Publishing.

Jung, R., Jourdan, J.-H., Krebbers, R., & Dreyer, D. (2017). Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, *2*, 1–34. https://doi.org/10.1145/3158154

Hales, T., et al. (2024, May 21). *Flyspeck github*. Retrieved May 21, 2024, from https://github.com/flyspeck/flyspeck

Harrison, J. (2024a, May 21). *Hol light*. Retrieved May 21, 2024, from https://www.cl.cam.ac.uk/~jrh13/hol-light/

Paulin-Mohring, C. (2011). Introduction to the coq proof-assistant for practical software verification. In *Laser summer school on software engineering* (pp. 45–95). Springer.

Harrison, J. (2024b, May 21). *Isabelle*. Retrieved May 21, 2024, from https://isabelle.in.tum.de/

de Moura, L., & et al., S. U. (2024, May 21). *Lean*. Retrieved May 21, 2024, from https://lean-lang.org/

Coq. (2024, May 21). Retrieved May 21, 2024, from https://coq.inria.fr/

Nuprl. (2024, May 21). Retrieved May 21, 2024, from https://nuprl-web.cs.cornell.edu/

Pfenning, F., & Schürmann, C. (2024, May 21). *Twelf*. Retrieved May 21, 2024, from https://twelf.org/

Geuvers, J. (2009). Proof assistants : History, ideas and future. *Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)*, *34*(1), 3–25. https://doi.org/10.1007/s12046-009-0001-5

Barendregt, H., & Geuvers, H. (2001). Proof-assistants using dependent type systems. In *Handbook of automated reasoning* (pp. 1149–1238). Elsevier Science Publishers B. V.

StackOverFlow. (2024, May 24). *Stack overflow survey*. Retrieved May 24, 2024, from https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages

Rustnomicon, T. (2024, May 24). *Rust races*. Retrieved May 24, 2024, from https://doc.rust-lang.org/nomicon/races.html

Merigoux, D., Kiefer, F., & Bhargavan, K. (2021). *Hacspec: Succinct, executable, verifiable specifications for high-assurance cryptography embedded in rust* (Technical Report). Inria. https://inria.hal.science/hal-03176482

Haselwarter, P. G., Hvass, B. S., Hansen, L. L., Winterhalter, T., Hritcu, C., & Spitters, B. (2023). The last yard: Foundational end-to-end verification of high-speed cryptography [https://eprint.iacr.org/2023/185]. https://doi.org/10.1145/3636501.3636961

Holdsbjerg-Larsen, R., Milo, M., & Spitters, B. (2022). A verified pipeline from a specification language to optimized, safe rust.

Nix. (2024, May 27). Retrieved May 27, 2024, from https://nixos.org/

Haselwarter, P. G., Rivas, E., Muylder, A. V., Winterhalter, T., Abate, C., Sidorenco, N., Hritcu, C., Maillard, K., & Spitters, B. (2021). Ssprove: A foundational framework for modular cryptographic proofs in coq [https://eprint.iacr.org/2021/397]. https://doi.org/10.1145/3594735

Schmidt, L. B., & Bjerg, R. T. (2023). High assurance specification of the halo2 protocol.

Ancher, J. D., & Jensen, C. M. (2024, May 12). *Masterthesis github*. Retrieved May 12, 2024, from https://github.com/JonasDAncher/MasterThesis

11 Appendix

11.1 The `powmod` functions.

```
Definition pow
  (g_3: int128)
  (x_4: int128)

  : int128 :=
      repr (unsigned g_3 ^ unsigned x_4).

Definition uint128_modulo
        (n_5: int128)
        (n_6: int128)

        : int128 :=
                MachineIntegers.modu n_5 n_6.

Definition uint128_pow_mod
  (g_0: int128)
  (x_1: int128)
  (n_2: int128)

  : int128 :=
    uint128_modulo (pow g_0 x_1) n_2.
```

11.2 SSProve's ElGamal proof of correctness

Proof of correctness for the SSProve ElGamal:

```
Lemma Enc_Dec_Perfect :
  Enc_Dec_real ≈₀ Enc_Dec_ideal.
Proof.
(* We go to the relation logic using equality as invariant. *)
  eapply eq_rel_perf_ind_eq.
  simplify_eq_rel m.
  apply r_const_sample_L.
  2: intros sk.
  2: apply r_const_sample_L.
  1,2: apply LosslessOp_uniform.
  intros pk.
  apply r_ret.
  intros s0 s1 e1.
  split.
  2: apply e1.
  repeat rewrite otf_fto.
  simpl.
  rewrite -2!expgM.
  rewrite mulnC.
  rewrite mulgA.
  rewrite mulVg.
  rewrite mul1g.
  rewrite fto_otf.
  reflexivity.
Qed.
```

11.3 Hacspec proof of correctness

Proof of correctness for the Hacspec type-checked and translated Rust code, written and proven in Coq:

```
Lemma Hacspec_Enc_Dec_Perfect :
  Hacspec_Enc_Dec_real ≈₀ Enc_Dec_ideal.
Proof.
  eapply eq_rel_perf_ind_eq.
  simplify_eq_rel m.
  apply r_ret.
  intros s0 s1.
  intros e1.
  split.
  2: apply e1.
  repeat rewrite otf_fto.
  repeat rewrite fto_otf.
  repeat rewrite Remove_Declassify.
  repeat rewrite Remove_Classify.
  assert (BinInt.Z.of_nat #|gT| = MachineIntegers.unsigned secret_q_v).
  1: rewrite g_gt_eq.
  1: rewrite q_eq.
  1: unfold secret_q_v.
  1: rewrite Remove_Secret.
  1: done.
  repeat rewrite FinToInt_IntToFin_Eq.
  3,4: rewrite H.
  3,4: unfold secret_q_v.
  3,4: rewrite Remove_Secret.
  3,4: apply MachineIntegers.unsigned_range_2.
  2: admit.
  unfold MachineIntegers.modu.

(*   unfold secret_q_v, secret_g_v. *)
  repeat rewrite Remove_Secret.

  unfold MachineIntegers.modu, MachineIntegers.mul, MachineIntegers.sub,
    powmod.uint128_pow_mod, powmod.pow, powmod.uint128_modulo, MachineIntegers.modu.

  repeat rewrite unsigned_repr.
  repeat rewrite Znat.nat_N_Z.
  rewrite Remove_mod_gT.
  rewrite (Zdiv.Zmod_small (BinNums.Zpos (BinNums.xO (BinNums.xO 1%AC)))).
  2: split.
  2: done.
  2 : admit.

  repeat rewrite Zdiv.Zmod_mod.
  rewrite -BinInt.Z.mul_assoc.
  repeat rewrite OrdersEx.Z_as_OT.pow_pos_fold.
  Import BinInt.Z.
  rewrite reprmod.
  2: {
  repeat rewrite -Zpow_facts.Zpower_mod.


  rewrite H.
  repeat rewrite -Zpow_facts.Zpower_mod.
  2,3,4,5: unfold secret_q_v.
  2,3,4,5: repeat rewrite Remove_Secret.
  2,3,4,5: done.
  simpl.
  rewrite Zdiv.Zmod_0_l.
  repeat rewrite mul_0_r.
  done.
}
  2: {
  repeat rewrite -Zpow_facts.Zpower_mod.
  rewrite H.
  repeat rewrite -Zpow_facts.Zpower_mod.
  2,3,4,5: unfold secret_q_v.
  2,3,4,5: repeat rewrite Remove_Secret.
  2,3,4,5: done.
  simpl.
  rewrite Zdiv.Zmod_0_l.
  repeat rewrite mul_0_r.
  done.
}
  rewrite Zdiv.Zmult_mod.
  rewrite H.
  repeat rewrite -Zpow_facts.Zpower_mod.
  2,3,4,5: unfold secret_q_v.
  2,3,4,5: rewrite Remove_Secret.
  2,3,4,5: done.
  rewrite -(Zdiv.Zmult_mod ((MachineIntegers.unsigned secret_g_v ^ 4) ^ 4)).
  rewrite -Zdiv.Zmult_mod.
  rewrite -H.
  rewrite -reprmod.
  2: {rewrite asd.
  rewrite mul_1_r.
  eapply Znat.Nat2Z.is_nonneg.
  }
  2:{
  unfold secret_g_v.
  rewrite Remove_Secret.
  rewrite asd.
  rewrite OrdersEx.Z_as_OT.mul_1_r.
  admit.
}
  rewrite asd.
  erewrite BinInt.Z.mul_1_r.
  unfold IntToFin, NatToOrd, IntToNat.
  eapply ord_inj.
  simpl.
  rewrite MachineIntegers.Z_mod_modulus_eq.
  rewrite -(Znat.Z2Nat.id MachineIntegers.modulus).
  2: done.
  rewrite -ssrZ.modnZE.
  2: done.
  rewrite Znat.Nat2Z.id.
  rewrite (@modn_small m).
  2:{ eapply ltn_trans.
  1: eapply ltn_ord.
  simpl.
  rewrite -(Znat.Nat2Z.id (#|gT|)).
  rewrite H.
  unfold secret_q_v.
  rewrite Remove_Secret.
  done.
  }
  rewrite (@modn_small m).
  2: apply ltn_ord.
  reflexivity.
```

```
Qed.
```

11.4 *Real* Package for the Pseudo Code

The pseudo code SSProve ElGamal Package:

```
Definition Enc_Dec_real :
  package fset0 [interface]
    [interface #val #[10] : 'plain → 'plain ] :=
    [package
      #def #[10] (m : 'plain) : 'plain
      {
        '(pk, sk) ← KeyGen ;;
        c ← Enc pk m ;;
        m ← Dec_open sk c ;;
        ret m
      }
    ].
```

11.5 The *Ideal* Package

The *Ideal* Package:

```
Definition Enc_Dec_ideal :
  package fset0 [interface]
    [interface #val #[10] : 'plain → 'plain ] :=
    [package
      #def #[10] (m : 'plain) : 'plain
      {
        ret m
      }
    ].
```

11.6 Hacspec *Real* Package

The *real* package using the Hacspec translated implementation.

```
Definition Hacspec_Enc_Dec_real :
  package fset0 [interface]
    [interface #val #[10] : 'plain → 'plain ] :=
    [package
      #def #[10] (m : 'plain) : 'plain
      {
        '(pk, sk) ← hacspec_gen ;;
        c ← hacspec_enc pk m ;;
        m ← hacspec_dec sk c ;;
        ret m
      }
    ].
```

11.7 Hacspec gen Interface

```
Definition hacspec_gen {L : {fset Location}} :
  code L [interface] (chPubKey × chSecKey) :=
  {code
    let '(c1Int,c2Int) := El_Gamal.keygen in
    let c1Key := IntToFin c1Int in
    let c2Key := IntToFin c2Int in
    ret (c1Key,c2Key)
  }.
```

11.8 Hacspec enc Interface

```
Definition hacspec_enc {L : {fset Location}} (pkKey : chPubKey) (m : chPlain) :
  code L [interface] chCipher :=
  {code
    let pkInt     := FinToInt pkKey in
    let mInt      := FinToInt m in
    let cipherInt := El_Gamal.enc pkInt mInt in
    let cipher    := fto(
                       otf (IntToFin cipherInt.1),
                       otf (IntToFin cipherInt.2)) in
    ret cipher
  }.
```

11.9 Hacspec dec Interface

```
Definition hacspec_dec {L : {fset Location}} (skKey : chSecKey) (cipher : chCipher) :
  code L [interface] chPlain :=
  {code
    let skInt  := FinToInt skKey in
    let cipher := (
                     FinToInt (fto (fst (otf cipher))),
                     FinToInt (fto (snd (otf cipher)))) in
    let mInt   := El_Gamal.dec skInt cipher in
    ret (IntToFin(mInt))
  }.
```

11.10 Hacspec gen Function

```
Definition keygen   : (int128 '× uint128) :=
  let secret_sk_0 : uint128 :=
    uint128_classify (sk_v) in
  let pk_1 : int128 :=
    uint128_declassify (uint128_pow_mod (secret_g_v) (secret_sk_0) (
        secret_q_v)) in
  (pk_1, secret_sk_0).
```

11.11 Hacspec enc Function

```
Definition enc (target_pk_2 : int128) (m_3 : int128)  : (int128 '× int128) :=
  let '(source_pk_4, secret_source_sk_5) :=
    keygen  in
  let secret_target_pk_6 : uint128 :=
    uint128_classify (target_pk_2) in
  let secret_m_7 : uint128 :=
    uint128_classify (m_3) in
  let secret_s_8 : uint128 :=
    uint128_pow_mod (secret_target_pk_6) (secret_source_sk_5) (secret_q_v) in
  let c1_9 : int128 :=
    source_pk_4 in
  let c2_10 : int128 :=
    uint128_declassify ((secret_m_7) .* (secret_s_8)) in
  (c1_9, c2_10).
```

11.12 Hacspec dec Function

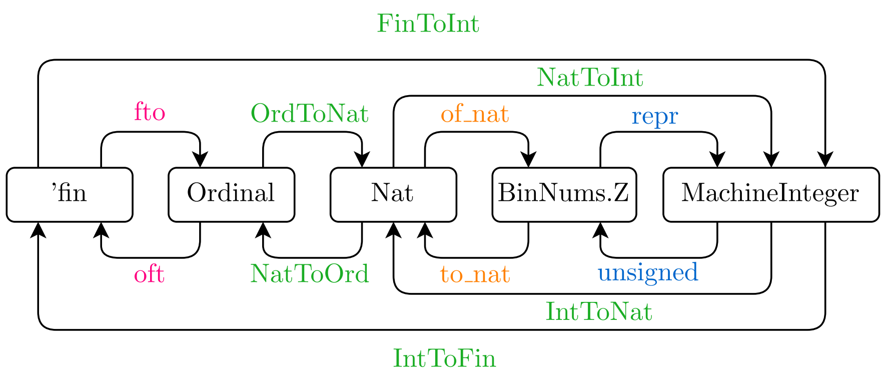
```
Definition dec
  (secret_target_sk_11 : uint128)
  (c_12 : (int128 '× int128))

  : int128 :=
  let '(c1_13, c2_14) :=
    c_12 in
  let secret_c1_15 : uint128 :=
    uint128_classify (c1_13) in
  let secret_c2_16 : uint128 :=
    uint128_classify (c2_14) in
  let secret_s_inverse_17 : uint128 :=
    uint128_pow_mod (secret_c1_15) (- (secret_target_sk_11)) (secret_q_v) in
  let secret_m_18 : uint128 :=
    (secret_c2_16) .* (secret_s_inverse_17) in
  let m_19 : int128 :=
    uint128_declassify (secret_m_18) in
  m_19.
```

11.13 Hacspec ElGamal

```rust
use hacspec_lib::*;

const Q : u128 = 29u128;
const G : u128 = 2u128;
const SECRET_Q : U128 = U128(Q);
const SECRET_G : U128 = U128(G);
const sk:u128 = 4u128;


pub fn keygen() -> (u128, U128){
    let secret_sk = U128::classify(sk);
    let pk        = (SECRET_G.pow_mod(secret_sk,SECRET_Q)).declassify();
    (pk,secret_sk)
}

pub fn enc(target_pk: u128, m: u128) -> (u128, u128){
    let (source_pk,secret_source_sk) = keygen();
    let secret_target_pk = U128::classify(target_pk);
    let secret_m         = U128::classify(m);

    let secret_s = secret_target_pk.pow_mod(secret_source_sk,SECRET_Q);
    let c1       = source_pk;
    let c2       = (secret_m * secret_s).declassify();

    (c1, c2)
}

pub fn dec(secret_target_sk: U128, c: (u128,u128)) -> u128 {
    let (c1,c2)   = c;
    let secret_c1 = U128::classify(c1);
    let secret_c2 = U128::classify(c2);

    let secret_s_inverse = secret_c1.pow_mod((-secret_target_sk),SECRET_Q);
    let secret_m  = secret_c2 * secret_s_inverse;
    let m         = secret_m.declassify();
    m
}
```

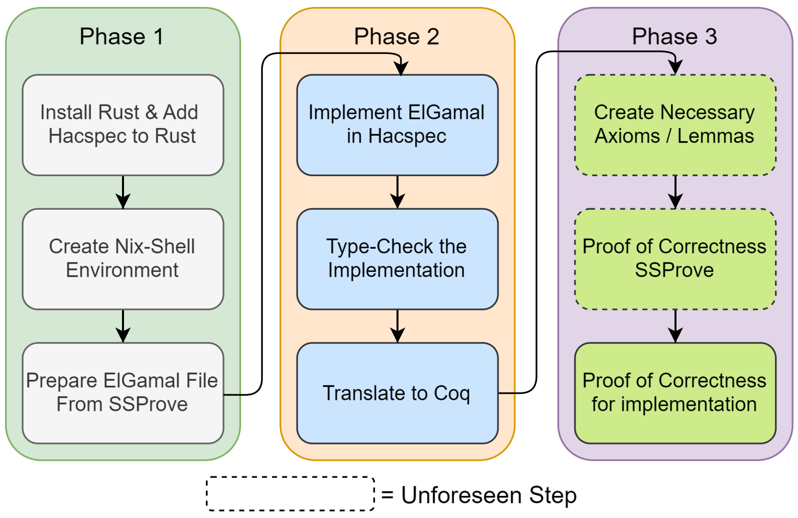
11.14 Types and Casting

Visualising the types and how we cast between them



11.15 Dependencies

Visualising the types and how we cast between them


Phase 1
Install Rust & Add Hacspec to Rust
Create Nix-Shell Environment
Prepare ElGamal File From SSProve
Phase 2
Implement ElGamal in Hacspec
Type-Check the Implementation
Translate to Coq
Phase 3
Create Necessary Axioms / Lemmas
Proof of Correctness SSProve
Proof of Correctness for implementation
= Unforeseen Step

11.16 Step-By-Step Explanation of the Pseudo Code ElGamal Proof of Correctness

`eapply eq_rel_perf_ind_eq .`	sets up the equation
`simplify_eq_rel m.`	simplifies stuffs
`apply r_const_sample_L .`	Introduces a function that generates a sample value losslessly and uniformly

`2:intros sk.`	Introduces `sk` in the second goal to the context
`2:apply r_const_sample_L` `.`	Introduces a function that generates a sample value losslessly and uniformly in the second goal
`1,2: apply LosslessOp_uniform` `.`	Removes the first goal and second goal, this can be done due to an introduction to a new sample value does not have anything new to prove.
`intros pk.`	Introduces `pk` to the context
`apply r_ret.`	Changes the format of goal one from
`intro s0 s1 e1.`	Introduces `statement 1, statement 2` and `expression 1 (so = s1)` from goal 1 to the context
`split.`	separates the expression `statement 1 ^ statement 1` into 2 goals: `goal 1: statement 1, goal 2: statement 2`
`2: apply e1.`	applies the expression `e1 (s0 = s1)` on goal 2, this makes the goal disappear due to the goal being equal to e1 from the context
`repeat rewrite otf_fto` `.`	Removes instances where `otf` is followed by `fto`, this can be done without introducing new goals due to casting between the `Ordinal` and `'fin` type not having any side effects
`simpl.`	This simplifies the expression, in this instance it changes `(expression 1, expression 2).1` into `expression 1`. This has no functional change, however it does makes the expression easier to understand.

`rewrite -2!expgM.`	$expgM$ changes $g^{x \cdot y}$ to g^{x^y}. 2! mean we do this twice. $-$ means we do the opposite, that is g^{x^y} to $g^{x \cdot y}$. The entire expression means that we rewrite the first 2 instances of g^{x^y} to $g^{(x \cdot y)}$
`rewrite mulnC.`	This function is commutative multiplication, that is it changes the expression $g^{(x \cdot y)}$ to $g^{(y \cdot x)}$
`rewrite mulnA.`	This function is associative multiplication, that is it changes the expression $x \cdot (y \cdot z)$ to $x \cdot y \cdot z$
`rewrite mulVg.`	This function rewrites $g^x \cdot g^{-x}$ to 1 due to the product rule and the zero exponent rule
`rewrite mul1g.`	Identity property of multiplication with 1, $1 \cdot m = m$
`rewrite fto_otf.`	Removes casts from 'fin to ord followed by ord to 'fin
`reflexivity.`	Indicates that the proof is finished, as $m = m$