The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Systems and Society

Formal Verification of ElGamal-based KEM IND-CCA1 Security and q-DDH Equivalence in Easy-Crypt

— Honours project (S2 2025)

A thesis submitted for the degree
Bachelor of Advanced Computing

By:
Xiuchen Gu

Supervisor:
Dr. Thomas Haines

October 2025

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the Academic Integrity Rule;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

October (2025), Xiuchen Gu

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Thomas Haines, for his guidance during the early stages of this work. His support in providing the initial EasyCrypt framework and proof structure laid the foundation for the formalization developed in this thesis.

Abstract

Public-key cryptography forms the foundation of modern secure digital communication, yet the informal nature of many security proofs can make them difficult to verify or susceptible to errors. This thesis presents a machine-checked formal proof, indicating its bilateral equivalence between the IND-CCA1 security of ElGamal based Key Encapsulaton Mechanism(KEM) and the q-DDH assumption.

Unlike the standard ElGamal public-key encryption, which is only IND-CPA secure, the KEM formulation encapsulates a session key derived from the ElGamal game, allowing a stronger IND-CCA1 guarantee. Both directions of the reduction, from an IND-CCA1 adversary to a q-DDH distinguisher and the reverse are mechanised in EasyCrypt, eliminating hidden assumptions and ensuring proof soundness.

Methodologically, we introduce a reusable operator-and-lemma library that represents the exponent linear algebra and its faithful reflection in the group, comprising operators such as `prodEx`, `addv`, `scalev`, `sumv`, and `shift_trunc`. This library provides a principled and verifiable oracle-simulation framework that captures the behavior of algebraic adversaries.

Keywords: bilateral equivalence, q-DDH, IND-CCA1, ElGamal, KEM, EasyCrypt, Algebraic Group Model, formal verification, tight reduction

Table of Contents

Chapter 1

Introduction

When we try to research on the topic about cryptography, a natural question is: how can we tell whether a cryptographic construction is "really secure" against real-world attackers? Broadly speaking there are two routes. One is destructive—designing algorithms and make experiments to find attacks and counterexamples. The other is constructive—providing security proofs that can be independently reproduced and proved. This paper follows the constructive approach and goes a step further by mechanizing the proofs: we translate security into executable games and reductions, and use a proof assistant to automatically check each step of the reasoning.

In particular, we study a classical ElGamal Key Encapsulation Mechanism (KEM) and its security in the attack model that allows decryption queries before the challenge is issued, denoted IND-CCA1. At the same time we consider a algebraic hardness assumption, the q-Decisional Diffie–Hellman problem (q-DDH). Our central claim is that, under accurate and suitable formalisation, breaking the IND-CCA1 security of the ElGamal KEM is essentially the same task as distinguishing the two distributions in the q-DDH problem for the same parameters. In other words, the two problems are equivalent and tight: given any adversary on one side we can covert it without loss to an adversary on the other side, preserving the adversary's success probability up to negligible loss.

Intuitively, IND-CCA1 allows the adversary to query a decryption oracle a limited number of times before receiving a single challenge ciphertext, but still requires that the adversary cannot distinguish a real session key from a random key. The q-DDH problem asks that, even if an adversary obtains several group elements corresponding to powers of a secret exponent, it cannot distinguish whether a given element corresponds to a true Diffie–Hellman product or to random generated. In the algebraic group setting, we want to trace adversarial group elements through their linear relations to known generators, which allows for constructing verifiable oracle responses. This technique enable us tightly mechanized correspondence between the two security frameworks.

1.1 Motivation

Public-key cryptography forms the foundation of modern secure cryptographic systems, enabling the confidential exchange of information over untrusted networks. Among its fundamental security notions, Key Encapsulation Mechanisms (KEM) play a crucial role in constructing shared symmetric keys for subsequent encrypted communication[1]. To remain secure against active attackers, these mechanisms require strong security guarantees, with indistinguishability under chosen-ciphertext attacks (IND-CCA) being among the most demanding and practically relevant. The IND-CCA1 variant, where adversaries have access to a decryption oracle only before seeing the challenge ciphertext, reflects many real-world attack scenarios while remaining trackable for theoretical analysis.

The ElGamal-based KEM, derived from the classical ElGamal construction, bases its security on the discrete logarithm problem in cyclic groups[2].

While ElGamal in its original public-key encryption form achieves only IND-CPA security [3], its KEM formulation enables stronger guarantees when analyzed under appropriate computational assumptions[2]. However, providing a fully formal and tight IND-CCA1 proof in the standard model remains challenging, often requiring non-trivial or idealized assumptions.

The Algebraic Group Model (AGM), introduced by Fuchsbauer, Kiltz, and Loss[4], provides a middle ground between the standard model and idealized models such as the random oracle model. In the AGM, adversaries are restricted to be *algebraic*, meaning they can only produce group elements for which they can provide explicit representations in terms of previously seen group elements. This restriction represents the intuition that adversaries cannot produce group elements arbitrarily but must construct them through the algebraic structure of existing elements.

The AGM has proven particularly valuable for analyzing the security of cryptographic schemes based on discrete logarithm assumptions. It enables proofs that would be infeasible in the standard model while avoiding the oversimplifications of purely idealized settings[5]. Recent work has highlights that many schemes, including digital signatures [6] and KEMs derived from ElGamal, can be proven secure under standard hardness assumptions when analyzed in the AGM.

Our work builds on this foundation by leveraging AGM-style techniques to indicates the IND-CCA1 security of ElGamal-based KEM under the q-DDH assumption. Although our proof does not require the full AGM framework, it employs the key insight that adversaries' group-element productions can be tracked through linear algebraic representations, enabling sophisticated oracle simulation techniques.

In addition, the mechanized proof formalizes a bilateral reduction between the IND-CCA1 security of the ElGamal-based KEM and the q-DDH assumption in EasyCrypt, establishing a computational equivalence that ensures the proof's tightness and soundness.

The proofs are mechanized in the EasyCrypt proof assistant[7], which is tailored for game-based cryptographic proofs at code level and gives support for common proof techniques, for instance, byequiv transformations, probabilistic reasoning, random variable transformations and algebraic manipulation techniques, it has been widely applied to build the security of the majority of cryptographic primitives and to connect with formalized proofs and their particular implementations.

Our main theorem can be informally phrased as follows:

> *The ElGamal-based Key Encapsulation Mechanism (*KEM*) is* IND-CCA1 *secure under the q-DDH assumption in cyclic groups of prime order and adversaries are modeled as algebraic in the sense of the AGM[4]. The reduction between an IND-CCA1 adversary against ElGamal-based* KEM *and a distinguisher for the q-DDh problem is tight, ensuring equality of advantages up to negligible terms. The security bound holds unconditionally with respect to the simulation, without loss beyond polynomial overhead.*

1.2 Problem Statement

The central question we address is:

> *Can we introduce a formal and bidirectional methodology, with tight concrete bounds, the equivalence between breaking IND-CCA1 security of ElGamal-based* KEM *and solving the q-DDH problem, with both reductions mechanized and machine-checked?*

This research question requires several non-trivial technical challenges that required to be carefully addressed in order to build a mechanized proof

1. **Reduction Construction**: The core of our work is to design precise and efficient reductions that bridge the IND-CCA1 security game with the q-DDH problem. Precisely, the reduction is required to translate any IND-CCA1 adversary against ElGamal-based KEM into a distinguisher for q-DDH, and conversely, it should also established an methodology to transform a q-DDH distinguisher into IND-CCA1 adversary. Achieving this requires a careful embedding of the q-DDH challenge into the Elgamal-based KEM experiment while preserving the advantage of adversary. In conclusion, the challenge is not only in defining the reduction accurately but also in ensuring that it do not tolerate any loss of advantage occurs.

2. **Oracle Simulation**: One of the most delicate and significant aspects of our security proofs is the establishment of a decryption oracle simulator. This orcale is required to be consistently answering adversarial queries while withholding information that would trivialize the underlying hard problems. In our setting, this needs simulating decryption queries in a way that is consistent with the q-DDH challenge without revealing the hidden exponent. This is particularly challenging under the AGM, where adversaries provide linear representations of ciphertexts.

The simulator is required to make leverage of these representations to provide faithful reflections while not leaking unintended information.

3. **Formal Verification**: Despite the high-level construction, our objective is to formalize and mechanize the entire reduction within the EasyCrypt, which requires us to encode all probabilistic reasoning, hybrid argument and algebraic manipulation in order to checked by the machine. In contrast to traditional paper proofs, where intuition often fills in the gaps. Constraints on length, representation validity and randomness-preserving transformations must all be explicitly formalized. In conclusion the key challenge is reconciling the intuitive reasoning of algebraic reductions with the strict requirements of mechanized verification framework.

4. **Tightness Analysis**: Finally, it is essential to illustrate the reduction is not only accurate but also tight in a concrete security sense. It involves proving that the adversarial is maintained exactly when transforming between the IND-CCA1 and q-DDH games. Constructing such concrete bounds is significant for the practical relevance of our results, while it ensures that the hardness of the q-DDH problem directly translates to the security guarantees for ElGamal-based KEM. Additionally, by examining tightness, it enables us to gain insight into the efficiency of the reductions and be able to verify that the proof has avoided artificial losses of security that might otherwise reduce its practical impact.

1.3 Main Contributions

Our work makes several significant contributions to both theoretical cryptography and formal verification:

1. **Reduction Theorem**: We prove the `qDDH_Implies_INDCCA1_ElGamal` and `INDCCA1_ElGamal_Implies_qDDH` lemma, establishing the equivalence between q-DDH hardness and IND-CCA1 security:

   $$\Pr[\text{IND\_CCA1\_P}(\text{ElGamal}, B).\text{main}() : \text{res}] = \Pr[\text{QDDH}(A_{\text{from\_INDCCA1}}(B)).\text{main}() : \text{res}]$$

2. **Oracle Simulation Technique**: We develop a sophisticated method for simulating the decryption oracle which utilizes linear algebraic representations to remain consistency without circular dependencies.

3. **Linear Algebra Library**: We contribute a reusable library of operators and their lemmas to EasyCrypt codebase that able to solve linear algebraic operations over exponents:

   - Core operators: `prodEx`, `addv`, `scalev`, `sumv`, `shift_trunc`
   - Over 30 lemmas of distributivity, scaling, base reduction, and vector operations
   - Reusable components for AGM proofs with linear combinations

4. **Complete EasyCrypt Formalization**: Our proof is entirely mechanized using EasyCrypt, comprising over 2000 lines code, with extensive documentation and modular design for future usage.

5. **Tight Security Analysis**: We indicate that our reduction is tight, without security loss, providing optimal concrete security bounds.

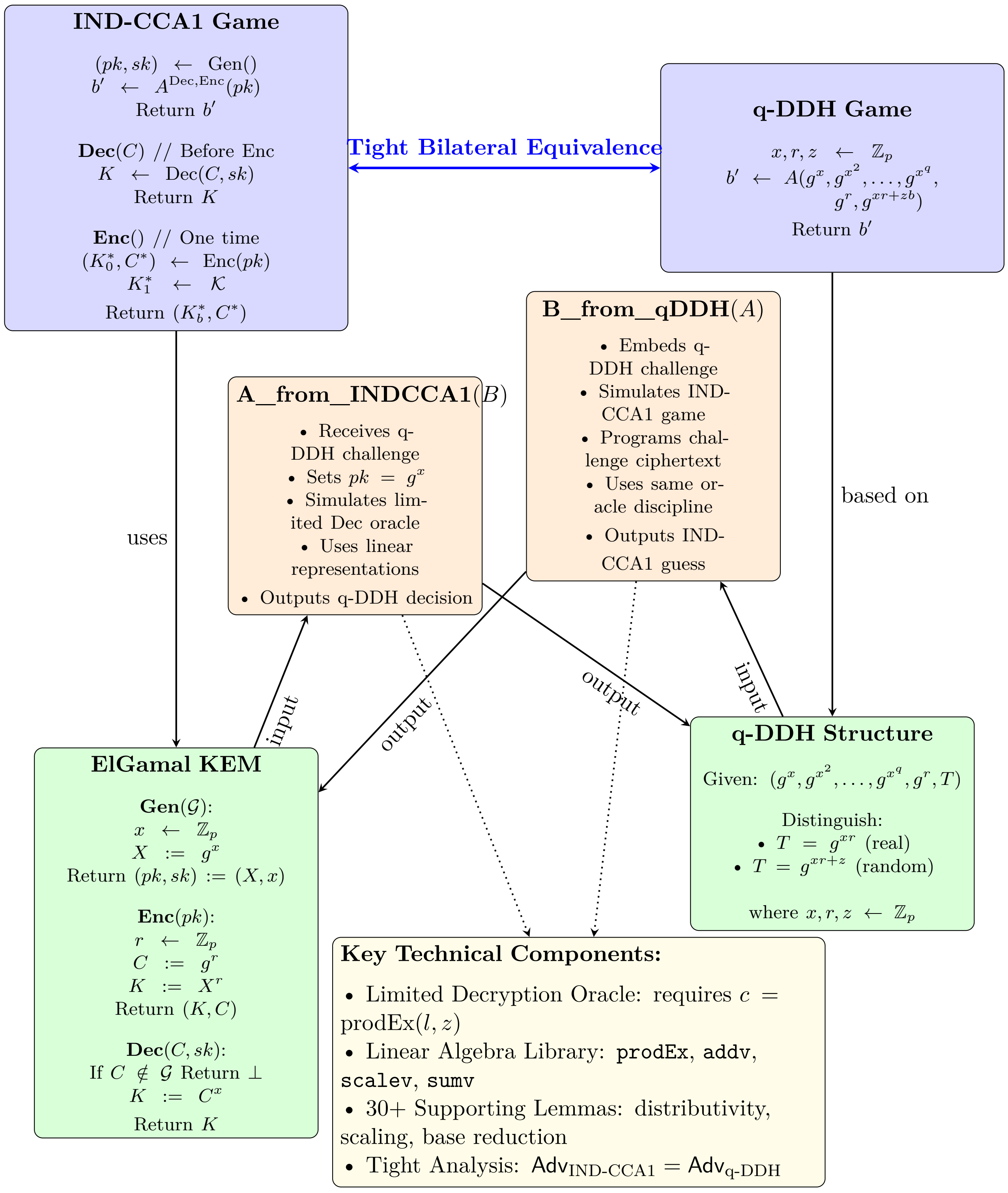
IND-CCA1 Game
$(pk, sk) \leftarrow \mathrm{Gen}()$
$b' \leftarrow A^{\mathrm{Dec,Enc}}(pk)$
Return b'
Dec(C) // Before Enc
$K \leftarrow \mathrm{Dec}(C, sk)$
Return K
Enc() // One time
$(K_0^*, C^*) \leftarrow \mathrm{Enc}(pk)$
$K_1^* \leftarrow \mathcal{K}$
Return (K_b^*, C^*)
Tight Bilateral Equivalence
q-DDH Game
$x, r, z \leftarrow \mathbb{Z}_p$
$b' \leftarrow A(g^x, g^{x^2}, \ldots, g^{x^q}, g^r, g^{xr+zb})$
Return b'
B_from_qDDH(A)
Embeds q-DDH challenge
Simulates IND-CCA1 game
Programs challenge ciphertext
Uses same oracle discipline
Outputs IND-CCA1 guess
A_from_INDCCA1(B)
Receives q-DDH challenge
Sets $pk = g^x$
Simulates limited Dec oracle
Uses linear representations
Outputs q-DDH decision
uses
based on
input
output
output
input
ElGamal KEM
Gen($\mathcal{G}$):
$x \leftarrow \mathbb{Z}_p$
$X := g^x$
Return $(pk, sk) := (X, x)$
Enc(pk):
$r \leftarrow \mathbb{Z}_p$
$C := g^r$
$K := X^r$
Return (K, C)
Dec(C, sk):
If $C \notin \mathcal{G}$ Return $\perp$
$K := C^x$
Return K
q-DDH Structure
Given: $(g^x, g^{x^2}, \ldots, g^{x^q}, g^r, T)$
Distinguish:
$T = g^{xr}$ (real)
$T = g^{xr+z}$ (random)
where $x, r, z \leftarrow \mathbb{Z}_p$
Key Technical Components:
Limited Decryption Oracle: requires $c = \mathrm{prodEx}(l, z)$
Linear Algebra Library: prodEx, addv, scalev, sumv
30+ Supporting Lemmas: distributivity, scaling, base reduction
Tight Analysis: $\mathsf{Adv}_{\text{IND-CCA1}} = \mathsf{Adv}_{\text{q-DDH}}$

Figure 1.1: Detailed bilateral reduction between IND-CCA1 security of ElGamal-based KEM and q-DDH hardness, showing the complete game structures and algorithmic details.

1.4 Technical Overview

Figure 1.1 indicates an intuitive overview of our proof technique. At a high level, our goal is to show that the capability to break the IND-CCA1 security of the ElGamal-based KEM is computationally equivalent to handling the q-DDH problem. Instead of relying on a one-way reduction, our analysis establishes a *two-direction correspondence*—showing that both can be transformed into the other without any meaningful loss.

The proof process by building two designed reductions. In the **first direction**, we indicate that any adversary ability to breaking the ElGamal KEM under chosen-ciphertext attacks can be covert into an algorithm that handle the q-DDH challenge. In the **reverse direction**, we demonstrate that a successful q-DDH distinguisher can be used to build an adversary that breaks the KEM.

A key technical component is the leverage of **algebraic tracking**, where every group element generated during the simulation is represented as a linear combination of previously known ones. This idea, inspired by the Algebraic Group Model [8], ensures that simulated oracles behave consistently and ensures every adversarial action has a verifiable algebraic explanation.

Finally, the entire framework, containing oracle simulations, linear algebra operators (`prodEx`, `addv`, `scalev`, and `sumv`), and supporting lemmas—is implemented and mechanically verified in the EasyCrypt proof assistant, which guarantees the correspondence between the two games is not only theoretically sound but also *machine-checked.*

1.4.1 Key Technical Components

The success of our formalized proof of bilateral reduction relies on several sophisticated technical innovations:

Limited Decryption Oracle.

The basic of our approach is a constrained decryption oracle that requires adversaries to provide algebraic representations for their queries. Specifically, for any decryption query (C, z), the oracle will verify that $C = \text{prodEx}(l, z)$ where l is the current list of known group elements and z is a vector of exponents. This restriction ensures:

- The simulator will not be request to expose information of discrete logarithms
- All group elements have explicit linear representations by the generation of adversary
- The oracle simulation maintain consistency in both directions .

Linear Algebra Library.

We develop a comprehensive library of operators and lemmas for manipulating linear combinations in the exponent field:

- **Core Operators:** `prodEx` (product of exponentiations), `addv` (vector addition), `scalev` (scalar multiplication), `sumv` (vector summation), and `shift_trunc` (shift and truncate for alignment)
- **Algebraic Properties:** Over 30 supporting lemmas covering distributivity, scaling laws, base reduction, zero-vector properties, and shift-truncate compatibility
- **Consistency Guarantees:** All operations preserve the correspondence between linear algebra in the exponent field and group operations

Representation Tracking.

Throughout both direction of reductions, we maintain discipline of tracking the linear representations of all group elements that enables:

- Consistent oracle simulation without exposure of knowledge of secret keys
- Tight security analysis with no loss in adversarial advantages
- Modular proof structure that can be extended

1.4.2 Formal Verification Infrastructure

Our entire proof is fully mechanized in the EasyCrypt , cosists of over 2000 lines of verified code. The formalization provides not only a transformation of the paper proof [4] into mechanized, but also offering a methodological contribution in its own right. Specifically, it provides benefits specified below that strengthen both the soundness and reusability:

- **Complete Mechanization:** Every aspect of the security argument, such as game transformations, oracle simulations and algebraic manipulations has been expressed by EasyCrypt. Unlike paper proofs, there is no step left at the intuitive level, in particular, the boundary condition and random variable transformation is completely strictly formalized.
- **Modular Design:** The development makes contribution of a dedicated linear-algebra library with a reusable reduction components. These modules are designed to be generic and independent of ElGamal itself, enable them to be applied to a broader range of AGM-style proof, for instance, make proofs for digital signatures, polynomial commitments and so on.
- **Reproducible Results:** Due to the proof is fully mechanized, it can be independently rerun, checked, and extended by future researchers, that ensures the results are not only correct once, but keep verifiable as the EasyCrypt platform evolves. Future researchers can build on this code-base as a foundation for more complex cryptographic proofs on AGM.
- **Hidden Assumption Elimination:** Formal verification requires the proof to expose all implicit requirements. Subtleties that might be informally justified in

paper proof, for instance, vector length constraints and randomness-preserving bijections are discharged mechanically by EasyCrypt, which prevents the presence of unstated assumptions and ensures the security reduction is strictly formalized and accurate.

All in all, this aspects above indicates that the IND-CCA1 security of ElGamal and the hardness of the q-DDH problem are not only related through a reduction, but are computationally equivalent with perfect tightness. Additionally, our contribution is more than just a reimplementation of a known proof but a carefully formalized, reproducible and reusable approach.

Chapter 2

Background

This section introduces the EasyCrypt verification framework and provides essential background on the algebraic group model that forms our analysis.

2.1 The EasyCrypt Verification Framework

EasyCrypt [7] serves as our basic framework for mechanizing cryptographic security arguments. This tool specializes in formalizing and verifying security proofs for cryptographic constructions through rigorous mathematical reasoning. The coreof EasyCrypt is its Probabilistic Relational Hoare Logic (pRHL), which we utilize extensively throughout our bilateral equivalence proof to build exact correspondences between security games while maintaining control over probabilistic reasoning [9].

Probabilistic Relational Hoare Logic (pRHL) enables us to establish that corresponding executions across different game contexts maintain identical success probabilities without requiring detailed characterization of intermediate distributions. Such relational reasoning proves particularly powerful for our tight reduction, where the primary objective is demonstrating perfect correspondence between adversarial advantages rather than computing absolute probability values [10].

While EasyCrypt's standard reasoning strategies handle the majority of our proof obligations, certain technical steps in our bilateral reduction demand more sophisticated approaches that require global program analysis and explicit management of algebraic relationships. These include randomness-preserving transformations and complex oracle simulations that must maintain consistency across multiple game contexts. EasyCrypt addresses these requirements through specialized reasoning techniques and transformation rules, though their direct application requires careful orchestration in our specific proof context.

To address this complexity, the EasyCrypt ecosystem has evolved to include a comprehensive collection of generic libraries that encapsulate common cryptographic proof patterns into reusable components. These libraries abstract intricate reasoning steps into high-level "game transformation" lemmas and equivalence theorems that can be instantiated and composed within larger security arguments.

The formal verification process encompasses over 2000 lines of verified EasyCrypt code, representing one of the most substantial mechanizations of AGM-style reasoning to date. This level of formalization provides unprecedented confidence in the correctness of our security argument while establishing reusable infrastructure for future research in mechanized cryptographic verification.

2.1.1 Core Concepts

EasyCrypt operates on several key principles that make it particularly suitable for cryptographic analysis:

Probabilistic Relational Hoare Logic (pRHL). EasyCrypt's cornerstone is probabilistic Relational Hoare Logic (pRHL), which enables reasoning about relationships between probabilistic programs [9]. In our context, pRHL allows us to establish probability equalities between different cryptographic games, such as our main lemma:

$$\Pr[\text{IND\_CCA1\_P}(\text{ElGamal}, B).\text{main}() : \text{res}] = \Pr[\text{QDDH}(A_{\text{from\_INDCCA1}}(B)).\text{main}() : \text{res}]$$

Game Transformations. EasyCrypt provides several powerful tactics for transforming games while preserving their probability distributions:

- **byequiv transformations**: These are built upon pRHL and allow us to prove that two games have identical probability distributions by establishing a relational invariant between their executions.
- **bypr transformations**: These enable reasoning about probability bounds and are particularly useful for establishing security reductions.
- **rnd transformations**: These are pRHL-based tactics for reasoning about randomness. For example, in our proof we use:

```
rnd (fun k' => loge k' - sk0{1} * y{1})
    (fun z => g ^ (z + sk0{1} * y{1})).
```

  This transformation converts randomness from group elements to the exponent field, making the two games probabilistically equivalent.

Oracle Simulation. EasyCrypt excels at reasoning about oracle-based security games. In our work, pRHL is crucial for:

- **Correctness**: Ensuring that simulated oracles produce the same outputs as real oracles for valid queries
- **State Management**: Tracking the evolution of oracle state across game executions
- **Query Validation**: Verifying that adversarial queries satisfy the required algebraic constraints

2.2 Background: Algebraic Group Model

The Algebraic Group Model (AGM), introduced by Fuchsbauer, Kiltz, and Loss [4], provides a framework for analyzing cryptographic schemes by restricting adversaries to be "algebraic."

In particular, instead of assuming an all-powerful attacker, the AGM will restricts adversaries to be algebraic, meaning that whenever they generate a new group element, they must also explain how it was algebraically derived from elements they have already seen [8]. This model indicates the intuition that real cryptographic algorithms operate through concrete algebraic operations, rather than arbitrary black-box calculations. By constrain adversaries in certain way, the AGM provides a intermediate position between the idealized models, such as the Random Oracle Model, and the entirely general Standard Model, allowing proofs to achieve a balance between realistic and easy to formalize.

Definition 1 (Algebraic Adversary)**.** An adversary $\mathcal{A}$ is algebraic if, whenever it outputs a group element $h \in \mathcal{G}$, it also provides a representation vector $\mathbf{z} = (z_1, \ldots, z_k) \in \mathbb{Z}_p^k$ such that:

$$h = \prod_{i=1}^{k} g_i^{z_i}$$

where $g_1, \ldots, g_k$ are all group elements that $\mathcal{A}$ has seen so far.

The AGM assumption enables tighter security reductions by providing the reduction algorithm with algebraic representations of adversarial outputs, effectively giving the reduction "extraction" capabilities without explicit knowledge extraction.

Assumption 1 (AGM Assumption)**.** *All adversaries are algebraic in the sense of Definition 1.*

This assumption is particularly well-suited for analyzing discrete logarithm-based schemes like ElGamal, as it captures the natural algebraic structure that such schemes possess while remaining weaker than the generic group model.

2.3 Games framework

The foundation of our proof lies in the characterization of two computation problems:

2.3.1 IND-CCA1 Security Game for ElGamal.

The indistinguishability under chosen-ciphertext attacks (IND-CCA1) security game for ElGamal KEM proceeds as follows: The challenger first introduce a key pair Gen() $\rightarrow$ (pk, sk) where $pk = g^x$ and $sk = x$ for a randomly chosen $x \leftarrow \mathbb{Z}_p$. Then, the adversary A is given the public key pk and then make decryption queries to a oracle Dec(C, sk) that returns $K = C^x$ for valid ciphertexts $C \in \mathcal{G}$. At some point, the adversary requests a challenge by calling the encryption oracle only once. The challenger computes Enc$(pk) \rightarrow$ (K_0^*, C^*) where $C^* = g^r$ and $K_0^* = (g^x)^r = g^{xr}$ for a random $\mathbb{Z}_p \rightarrow r$, then selects a random key $\mathcal{K} \rightarrow K_1^*$, and finally return (K_b^*, C^*) for a random bit b. Finally, The adversary will outputs a guess b' and wins if $b' = b$.

The algorithm specification[4] is detailed in Figure 2.1.

IND-CCA1$^{A}_{\mathbf{EG},\mathcal{G}}$ Security Game for ElGamal

Algorithm:

```
00 $x \leftarrow \mathbb{Z}_p$
01 $X := g^x$
02 $b' \leftarrow A^{\text{Dec,Enc}}(X)$
03 Return $b'$
```

Oracles Available to Adversary A**:**

```
• Dec$(C)_a$ // Before Enc is called
  04 If $C \notin \mathcal{G}$ Return $\perp$
  05 $K := C^x$
  06 Return $K$

• Enc() // One time only
  07 $r \leftarrow \mathbb{Z}_p$
  08 $C^* := g^r$
  09 $K^* := X^r$
  10 $K_1^* \leftarrow \mathcal{K}$ // random key
  11 $b \leftarrow \{0,1\}$
  12 Return $(K_b^*, C^*)$
```

ElGamal KEM Operations:

- **Gen**$(\mathcal{G}) \rightarrow (pk, sk)$: $x \leftarrow \mathbb{Z}_p$; $X := g^x$; Return (X, x)
- **Enc**$(pk) \rightarrow (K, C)$: $r \leftarrow \mathbb{Z}_p$; $C := g^r$; $K := pk^r$; Return (K, C)
- **Dec**$(C, sk) \rightarrow K$: If $C \notin \mathcal{G}$ Return $\perp$; $K := C^{sk}$; Return K

Advantage Definition:

$\mathsf{Adv}^{\text{IND-CCA1}}_{\text{ElGamal}}(A) = \left|\Pr[b' = b] - \frac{1}{2}\right|$

where b is the random bit used in the Enc oracle

Security Goal:

Adversary A should not be able to distinguish between $K_0^* = g^{xr}$ (real key) and K_1^* (random key) even with access to decryption oracle before receiving the challenge (K_b^*, C^*)

Figure 2.1: IND-CCA1 security game for ElGamal encryption. The adversary has access to a decryption oracle before the challenge phase, then must distinguish between a real session key and a random key.

2.3.2 q-DDH Problem.

The q-Decisional Diffie-Hellman (q-DDH) problem will ask one to distinguish between two distributions over group elements. By given a tuple $(g^x, g^{x^2}, \ldots, g^{x^q}, g^r, T)$ where $x, r \leftarrow \mathbb{Z}_p$ are random, the attacker must determine whether $T = g^{xr}$ (real distribution) or $T = g^{xr+z}$ for a random $z \leftarrow \mathbb{Z}_p$ (random distribution).

The specification[4] is detailed below in Figure 2.2.

q-DDH$^{A}_{\mathcal{G},q}$ Problem

Algorithm:

```
00 $x, r, z \leftarrow \mathbb{Z}_p$
01 $b \leftarrow \{0, 1\}$
02 $T_0 := g^{xr}$, $T_1 := g^{xr+z}$
03 $b' \leftarrow A(g^x, g^{x^2}, \ldots, g^{x^q}, g^r, T_b)$
04 Return $b'$
```

Challenge Structure Given to Distinguisher A:

- **Powers of x:** $(g^x, g^{x^2}, g^{x^3}, \ldots, g^{x^q})$
- **Random element:** g^r where $r \leftarrow \mathbb{Z}_p$
- **Target element:** $T \in \{g^{xr}, g^{xr+z}\}$ where $z \leftarrow \mathbb{Z}_p$

Distinguishing Goal:

- **Real distribution $\mathcal{D}_0$:** $T = g^{xr}$ (DDH tuple)
- **Random distribution $\mathcal{D}_1$:** $T = g^{xr+z}$ (random element)

Distinguisher A must output a bit b' indicating which distribution the challenge comes from

Advantage Definition:

$\mathsf{Adv}^{\text{q-DDH}}_{\mathcal{G}}(A) = |\Pr[A(\mathcal{D}_0) = 1] - \Pr[A(\mathcal{D}_1) = 1]|$
where $\mathcal{D}_0 = (g^x, g^{x^2}, \ldots, g^{x^q}, g^r, g^{xr})$
and $\mathcal{D}_1 = (g^x, g^{x^2}, \ldots, g^{x^q}, g^r, g^{xr+z})$

Hardness Assumption:

For any probabilistic polynomial-time algorithm A:
$\mathsf{Adv}^{\text{q-DDH}}_{\mathcal{G}}(A) \leq \text{negl}(\lambda)$

The q-DDH assumption states that even given the first q powers of x,
it remains computationally hard to distinguish g^{xr} from g^{xr+z}

Relationship to Standard DDH:

- Standard DDH: Given (g^a, g^b, g^c), distinguish $c = ab$ from random c
- q-DDH: Given $(g^x, \ldots, g^{x^q}, g^r, T)$, distinguish $T = g^{xr}$ from $T = g^{xr+z}$
- q-DDH $\Rightarrow$ DDH but the converse may not hold

Figure 2.2: q-DDH (q-Decisional Diffie-Hellman) problem. The distinguisher receives the first q powers of a secret exponent x along with g^r and must distinguish between g^{xr} and g^{xr+z} for random z.

Chapter 3

Related Work

IND-CCA (Indistinguishability under Chosen Ciphertext Attack) security is a core concept in public-key cryptography. It requires that even when attackers can query a decryption oracle, they still cannot distinguish the target plaintext, thus providing one of the strongest confidentiality guarantees for encryption schemes. IND-CCA security proofs are often complex and tedious, and traditional manual reduction processes are prone to errors. To reduce this risk, the academic community has increasingly focused on formal verification tools such as EasyCrypt in recent years [7]. Barthe et al. first conducted mechanized verification of some conclusions that were previously only argued intuitively in the Generic Group Model (GGM). Furthermore, EasyCrypt has been used for formal verification of modern encryption schemes, such as the IND-CCA security of Kyber during the NIST standardization process, whose proof was completed through machine verification using the Fujisaki-Okamoto transformation under the random oracle model [10].

Nevertheless, EasyCrypt also has limitations. Particularly, the library and ecosystem are still relatively underdeveloped, resulted in many commonly required basic lemmas and operators are not able to used directly, meaning that users are often required to implement basic algebraic machinery themselves in order to complete non-trivial proofs. This lack of readily reusable components increases the effort, but it also motivates the systematic development of reusable operators and lemmas, enriching EasyCrypt's ecosystem for future proof.

In this context, researchers have proposed some alternative models to obtain tighter or more concise security proofs. The most representative is the Algebraic Group Model (AGM) proposed by Fuchsbauer, Kiltz, and Loss [4]. In AGM, it is assumed that adversaries are algebraic, meaning that whenever they output a new group element, they must simultaneously provide a representation of how this element is formed by combining previous elements. This restriction provides additional structural information for reductions, making the proof process more concise and tight. A key result by the authors (Theorem 5.2) shows that breaking the IND-CCA1 security of ElGamal is tightly equivalent to solving the q-type DDH problem under AGM. This proof uses the algebraic representations provided by adversaries to construct consistent decryption oracle simulation, achieving an almost lossless security reduction.

The advantage of the AGM approach is that it provides tight reductions (no loss in attack advantage) and allows reasoning to avoid the complex guessing and technical rewinding common in standard model proofs. However, its limitations are also obvious: AGM relies on non-standard assumptions about adversaries, and existing proofs are still manual derivations that have not undergone mechanized verification. Although these proofs are highly convincing, some steps are still based on intuition rather than formal derivation, so potential subtle errors may still exist, which is precisely where formal methods can provide compensation.

Beyond encryption schemes, AGM has also been applied to more complex cryptographic protocols. In their 2023 work "From Polynomial IOP and Commitments to Non-malleable zkSNARKs," Faonio et al. [11] used AGM (combined with the random oracle model) to prove the simulation-extractability of the KZG polynomial commitment scheme, thereby constructing non-malleable zkSNARKs. This shows that AGM is also valuable in security analysis of complex protocols, especially in scenarios where direct proofs are difficult to provide in the standard model. However, similar to the ElGamal case, this work still remains at the manual proof level and has not achieved mechanized verification. This also indicates that there is still a gap between AGM-style reasoning and fully formal verification. How to combine intuitive algebraic reasoning with machine-verifiable formal methods remains an important challenge facing this field.

When it comes to the the ElGamal-based Key Encapsulation Mechanism (KEM), which has been widely studied under various computational assumptions and application settings. Galindo et al. [12] introduced a leakage-resilient variant of the ElGamal KEM, named BEG-KEM, which enhances resistance against side-channel attacks through masking and blinding techniques while maintaining practical performance on embedded platforms. Hashimoto et al. [13] further strengthened the theoretical foundations by providing a tight reduction proof for the Twin-DH hashed ElGamal KEM in multi-user environments, demonstrating its scalability and security. Earlier, Kiltz and Pietrzak [14] established one of the first leakage-resilient ElGamal constructions, proving that a CCA1-secure KEM combined with a one-time symmetric cipher can yield full IND-CCA1 security even under bounded key leakage. Finally, Lee et al. [15] introduced a decomposable KEM framework that achieves continuous key agreement with reduced bandwidth while retaining DDH-based security guarantees. Together, these works demonstrate the evolution of ElGamal-based KEMs from theoretical constructs toward efficient, leakage-resilient, and tightly secure primitives across a variety of cryptographic contexts.

Chapter 4

q-DDH Correctness and IND-CCA1 Security

This part establishes the mathematical and cryptographic foundations required for our analysis. We first introduce the ElGamal Key Encapsulation Mechanism (KEM), providing its algorithmic description and a formalized implementation in EasyCrypt; subsequently, we define the IND-CCA1 security model; and finally introduce the q-Decisional Diffie–Hellman (q-DDH) assumption. These components provide a structured foundation for subsequent reasoning about accuracy and security, enabling reductions to be rigorously expressed within our EasyCrypt code.

4.1 ElGamal Key Encapsulation Mechanism

Our analysis focuses on the KEM formulation of ElGamal, specifying three algorithms that operate over a cyclic group $\mathcal{G}$ of prime order p with generator g. This formal description facilitates distinguishing correctness from security: correctness ensures legitimate recipients accurately recover encrypted session keys, while security ensures attackers cannot distinguish keys from random ones.

4.1.1 Algorithmic Specification

The ElGamal KEM is defined by the following algorithms:

Key Generation.

```
1: Algorithm KeyGen($\mathcal{G}$) → $(pk, sk)$:
2: $x \leftarrow \mathbb{Z}_p$ {Sample random secret key}
3: $pk \leftarrow g^x$ {Compute public key}
4: return $(pk, sk) = (g^x, x)$
```

Key Encapsulation.

```
1: Algorithm Enc($pk$) $\to (K, C)$:
2: $r \leftarrow \mathbb{Z}_p$ {Sample random encapsulation exponent}
3: $C \leftarrow g^r$ {Compute ciphertext}
4: $K \leftarrow pk^r$ {Compute session key}
5: return $(K, C) = (g^{xr}, g^r)$
```

Key Decapsulation.

```
1: Algorithm Dec($C, sk$) $\to K$:
2: if $C \notin \mathcal{G}$ then return $\perp$
3: $K \leftarrow C^{sk}$ {Recover session key}
4: return $K$
```

4.1.2 EasyCrypt Implementation

The ElGamal KEM is implemented in EasyCrypt as follows:

```
(** ElGamal encryption scheme implementation **)
module ElGamal : Scheme = {
  (* Key generation: sk random, pk = g^sk *)
  proc keygen(): pk_t * sk_t = {
    var sk;
    sk <$ FD.dt;               (* Random secret key *)
    return (g ^ sk, sk);       (* Public key is g^sk *)
  }

  (* Encryption: return (pk^y, g^y) where y is random *)
  proc enc(pk:pk_t): key_t * ctxt_t= {
    var y;
    y <$ FD.dt;                (* Random encryption exponent *)
    return (pk ^ y, g ^ y);    (* Session key and ciphertext *)
  }

  (* Decryption: compute c^sk *)
  proc dec(c:ctxt_t,sk:sk_t): key_t option = {
    return Some (c ^ sk);      (* ElGamal decryption formula *)
  }
}.
```

Listing 4.1: ElGamal KEM Implementation in EasyCrypt

4.2 IND-CCA1 Security Model

Indistinguishability under non-adaptive chosen-ciphertext attacks (IND-CCA1) is a significant security notion for public-key encryption schemes.

Definition 2 (IND-CCA1 Security Game)**.** For a KEM scheme $\mathsf{KEM} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and adversary $\mathcal{A}$, the IND-CCA1 security game proceeds as follows:

```
1: $(pk, sk) \leftarrow \mathsf{KeyGen}()$ {Generate key pair}
2: $\mathcal{A}$ receives $pk$ and can query decryption oracle $\mathsf{Dec}(sk, \cdot)$
3: $(K_0, C^*) \leftarrow \mathsf{Enc}(pk)$ {Generate challenge}
4: $K_1 \leftarrow \mathcal{K}$ {Sample random key}
5: $b \leftarrow \{0, 1\}$ {Choose challenge bit}
6: $b' \leftarrow \mathcal{A}(K_b, C^*)$ {Adversary distinguishes}
7: return $(b' = b)$ {Adversary wins if correct}
```

The adversary's advantage is defined as:

$$\mathsf{Adv}^{\text{IND-CCA1}}_{\mathsf{KEM}}(\mathcal{A}) = \left| \Pr[\text{IND-CCA1}^{\mathcal{A}}_{\mathsf{KEM}} = 1] - \frac{1}{2} \right|$$

In the algebraic group model, the decryption oracle requires adversaries to provide algebraic representations for their queries, ensuring that all ciphertexts can be expressed in terms of previously seen group elements.

4.2.1 Relationship to Other Security Notions

Conceptually, IND-CCA1 strengthens IND-CPA by granting a decryption oracle only *before* the challenge, while weaker than full adaptive IND-CCA2 [4].

In modern use, IND-CCA1 still occurs in settings where post-challenge oracle access is implausible (e.g., certain KEM/DEM designs under one-shot decryption exposure) or as a intermediate step to IND-CCA2. Formal treatments and exercises with complete games, oraclesand reductions can be found in standard texts and course notes [16, 17].

4.2.2 Algebraic Oracle Implementation

The key contribution in our formalization is the algebraic oracle that enforces AGM constraints through a two-layer design: interface abstraction and implementation with validation.

Interface Layer: Decryption Oracle Abstraction under AGM. We encode the phase-based constraints of the IND-CCA1 game directly into the type and effect system. The decryption oracle interface `Oracles_CCA1i` provides initialization `init` and representation-based decryption `dec(c, z)`, where vector z represents the algebraic representation of ciphertext group element c relative to the visible base list l. The adversary interface `Adv_INDCCA1` is decomposed into a "scouting" phase `scout(pk)` (allowing calls to `O.dec`) and a "distinguishing" phase `distinguish(k, c)` (prohibiting calls to `O.dec`).

This layered abstraction provides two benefits: First, the game rules (decryption allowed before challenge, forbidden after challenge) are enforced through effect annotations at the type level, avoiding the need to manually exclude illegal calls. Second, requiring representation z makes explicit the core assumption of the Algebraic Group Model (AGM)—that adversaries must provide linear/exponential combinations relative to the "known set" l for every queried group element, therefore strictly limiting the adversary's accessible information to the linear space generated by l and enabling precise algebraic reasoning in subsequent reductions.

```
(** Interface for oracles used in CCA1 security games **)
module type Oracles_CCA1i = {
  proc init(sk_init : sk_t) : unit                       (*
Initialize with secret key *)
  proc dec(c : ctxt_t, z : exp list) : key_t option      (*
Decryption with representation *)
}.

(** Adversary interface for IND-CCA1 game **)
module type Adv_INDCCA1 (O : Oracles_CCA1i) = {
  proc scout(pk : pk_t) : unit {O.dec}          (* Phase 1:
explore with queries *)
  proc distinguish(k : key_t, c : ctxt_t) : bool {}  (* Phase 2:
distinguish challenge *)
}.
```

Listing 4.2: Algebraic Oracle Interface

Implementation Layer: Limited Oracle with Query Quota and Representation Validation. The module `O_CCA1_Limited` enforces both query quota and representation consistency validation at the implementation level. In `dec(c, z)` calls, a query is considered valid only when the quota is not exceeded and the condition $c = \texttt{prodEx}(l, z)$ is satisfied. For valid queries, the oracle calls the underlying scheme's decryption algorithm `S.dec` to obtain the session key and incorporates it into the base list l and query log qs, thereby monotonically expanding the adversary's visible linear space. For invalid queries, a placeholder value `witness` is returned to avoid additional leakage.

This implementation realizes the minimal principle of "queryable implies learnable": adversaries can only incorporate new group elements that are genuinely obtained through valid queries into subsequent representations. Meanwhile, the interface layer already prohibits calling `dec` after the challenge phase, strictly consistent with IND-CCA1's non-adaptive constraints.

```
(** Limited CCA1 Oracle Implementation **)
module O_CCA1_Limited (S : Scheme)  : Oracles_CCA1i= {
  var sk : sk_t                                        (* Secret key *)
  var qs : (ctxt_t * key_t option) list              (* Query history
*)
  var l : group list                                   (* List of group
elements seen *)

  (* Initialize oracle with secret key *)
  proc init(sk_init : sk_t) = {
    sk <- sk_init;
    qs <- [];           (* Empty query list *)
    l <- [];            (* Empty group element list *)
  }

  (* Decryption oracle with representation checking *)
  proc dec(c : ctxt_t, z : exp list) : key_t option = {
    var p : key_t option;
    var invalid_query : bool;

    (* Check if query is valid:
       - Haven't exceeded q queries
       - Ciphertext matches expected representation *)
    invalid_query <- (q < size qs + 2  \/ c <> prodEx l z);

    (* Perform actual decryption using scheme *)
    p <@ S.dec(c, sk);

    (* Update state only if query was valid *)
    if (!invalid_query) {
      l <- oget p :: l ;            (* Add decrypted key to list *)
      qs <- (c, p) :: qs;       (* Record query *)
    }

    (* Return result or witness if invalid *)
    return (if !invalid_query then p else witness);
  }
}.
```

Listing 4.3: Limited Oracle Implementation with Validation

4.3 q-DDH Assumption

The q-Decisional Diffie-Hellman (q-DDH) assumption generalizes the standard DDH assumption to handle multiple powers of a secret exponent.

In particular, the *Decisional Diffie–Hellman (DDH)* assumption is one of the fundamental hardness assumptions in public-key cryptography, which states that, given a group generator g and elements g^x, g^r, it is computationally infeasible to determine whether a third element is g^{xr} or just a random group element.

The *q-Decisional Diffie–Hellman (q-DDH)* assumption extends this idea to more structured setting, where the adversary additionally observes multiple powers of the same secret exponent. In particular, even with access to $g^x, g^{x^2}, \ldots, g^{x^q}$, it should be infeasible for any efficient adversary to distinguish a true Diffie-Hellman pair from a randomly generate one [18].

Definition 3 (q-DDH Problem)**.** Let $\mathcal{G}$ be a cyclic group of prime order p with generator g. The q-DDH problem asks to distinguish between the following distributions:

$$\mathcal{D}_0 = (g, g^x, g^{x^2}, \ldots, g^{x^q}, g^r, g^{xr}) \tag{4.1}$$

$$\mathcal{D}_1 = (g, g^x, g^{x^2}, \ldots, g^{x^q}, g^r, g^{xr+z}) \tag{4.2}$$

where $x, r, z \leftarrow \mathbb{Z}_p$ are chosen uniformly at random.

Assumption 2 (q-DDH Assumption)**.** *For any polynomial-time algorithm $\mathcal{B}$, the q-DDH advantage*

$$\mathsf{Adv}_{\mathcal{G}}^{q\text{-}DDH}(\mathcal{B}) = |\Pr[\mathcal{B}(\mathcal{D}_0) = 1] - \Pr[\mathcal{B}(\mathcal{D}_1) = 1]|$$

is negligible.

4.3.1 EasyCrypt Formalization of q-DDH

The q-DDH game is implemented in EasyCrypt as follows:

```
(** q-DDH adversary interface **)
module type A_qDDH = {
  proc guess(gtuple : group list) : bool  (* Distinguish q-DDH
tuple *)
}.

(** q-DDH game: distinguish (g, g^x, g^{x^2}, ..., g^{x^q}, g^r,
g^{xr})
    from (g, g^x, g^{x^2}, ..., g^{x^q}, g^r, g^z) where z is
random **)
module QDDH (A : A_qDDH)  = {
  proc main() : bool = {
      var x, r, z , b_int;
      var gtuple : group list;
      var challenge : group;
      var b, b' : bool;

      (* Sample random values *)
      x <$ dt;   (* Secret base *)
      r <$ dt;   (* Random for challenge *)
      z <$ dt;   (* Random alternative *)

      b <$ {0,1};  (* Bit determining real or random *)

      (* Convert boolean to integer for computation *)
      b_int <- (if b then ZModE.zero else ZModE.one);

      (* Create q-DDH tuple: g^x, g^{x^2}, ..., g^{x^q} *)
      gtuple <- map (fun i => g^(exp x i)) (range 1 (q+1));

      (* Challenge element: either g^{xr} (real) or g^{xr+z}
(random) *)
      challenge <- g^((x * r) + (z * b_int));

      (* Give adversary the tuple: [g^x, ..., g^{x^q}, g^r,
challenge] *)
      b' <@ A.guess(gtuple ++ [g^r] ++ [challenge]);

      (* Adversary wins if they distinguish correctly *)
      return b = b';
  }
}.
```

Listing 4.4: q-DDH Game Implementation

The program first defines the adversary interface `A_qDDH`, where the single procedure `guess(gtuple)` receives a list of group elements and outputs a boolean value, representing the adversary's binary judgment about the source of the input vector.

Subsequently, the main procedure of the game module `QDDH(A)` is presented. It first declares exponent field elements x, r, z, *b_int*, a group element list *gtuple*, a single group element *challenge*, and boolean variables b, b'. The program then independently and uniformly samples from distribution *dt* to obtain the secret exponent x, the random exponent r used for constructing the challenge, and the additional random quantity z, and flips a coin to get bit b.

The program converts the boolean value b to an element *b_int* in the exponent field, following the rule that when b is true, it takes zero, and when b is false, it takes one. Subsequently, it generates the list *gtuple* using `map (fun i => g^(exp x i)) (range 1 (q+1))`, which computes term by term according to the exponential powers $x^1, x^2, \dots, x^q$ to obtain $[g^x, g^{x^2}, \dots, g^{x^q}]$.

The challenge element *challenge* is constructed through an exponential linear combination: $g^{(x \cdot r)+(z \cdot b\_int)}$. Therefore, when b is true ($b\_int = 0$), the challenge is g^{xr}, and when b is false ($b\_int = 1$), the challenge is g^{xr+z}.

The complete vector is then concatenated as `gtuple ++ [g^r] ++ [challenge]`, which appends g^r and the challenge element sequentially after the prefix $[g^x, \dots, g^{x^q}]$. This string of group elements is passed to the adversary procedure `A.guess` to obtain output b'.

Finally, the game returns the boolean value $b = b'$ as the result of one trial: if the adversary's judgment b' matches the hidden bit b, it returns true, indicating that the adversary successfully distinguished in this round; otherwise, it returns false.

Overall, this process implements: sampling key-related vectors and a challenge element controlled by a bit, passing them to the adversary to make a guess, and using whether the adversary guesses correctly as the output of this round of the game.

4.4 Mathematical Foundations

Our formalization builds upon a comprehensive linear algebra library that enables precise manipulation of algebraic representations while maintaining faithful correspondence with group operations. These mathematical structures are fully implemented and verified within the EasyCrypt proof assistant, giving a machine-checked foundation for the algebraic reasoning used in subsequent sections.

4.4.1 Core Linear Algebra Library

The heart of our approach lies in a comprehensive library for manipulating linear combinations in the exponent field while maintaining consistency with group operations.

Basic Group Operations.

```
(* Product of a list of group elements *)
op prod (elements : group list) = foldr ( * ) e elements.

(* Exponentiation: compute bases[i]^exps[i] for all i, return as
list *)
op ex (bases : group list)(exps : exp list) =
  (map (fun (i : group * exp) => i.`1 ^ i.`2) (zip bases exps)).

(* Product of exponentiation: compute product of ex(bases, exps)
*)
op prodEx (bases : group list)(exps : exp list) =
  prod (ex bases exps).
```

Listing 4.5: Basic Group Operations

The `prodEx` operator is our core abstraction, computing:

$$\mathsf{prodEx}(\mathbf{g}, \mathbf{a}) = \prod_{i=0}^{n-1} g_i^{a_i}$$

where $\mathbf{g} = (g_0, g_1, \ldots, g_{n-1})$ and $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$.

Vector Operations in the Exponent Field.

Zero Vector (`zerov`) The zero vector operation creates a vector of length $q + 1$ filled with zero elements:

$$\texttt{zerov} = \underbrace{[0, 0, \ldots, 0]}_{q+1 \text{ elements}} \tag{4.3}$$

```
(* Zero vector of length q+1 *)
op zerov = nseq (q+1) zero.
```

Listing 4.6: Zero Vector Definition

This serves as the additive identity for vector operations and the initial value for vector accumulation.

Vector Addition (`addv`) Vector addition performs pointwise addition of corresponding elements from two vectors:

$$\texttt{addv}(a, b) = [a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n] \tag{4.4}$$

where $a = [a_1, a_2, \ldots, a_n]$ and $b = [b_1, b_2, \ldots, b_n]$.

```
(* Vector addition: pointwise addition *)
op addv (a b : exp list) =
  map (fun (x : exp * exp) => x.`1 + x.`2) (zip a b).
```

Listing 4.7: Vector Addition

This operation is commutative and associative, forming the basis for linear combinations of algebraic representations.

Vector Multiplication (`mulv`) Vector multiplication performs pointwise (Hadamard) multiplication of corresponding elements:

$$\texttt{mulv}(a, b) = [a_1 \cdot b_1, a_2 \cdot b_2, \ldots, a_n \cdot b_n] \tag{4.5}$$

```
(* Vector multiplication: pointwise multiplication *)
op mulv (a b : exp list) =
  map (fun (x : exp * exp) => x.`1 * x.`2) (zip a b).
```

Listing 4.8: Vector Multiplication

This operation is used for combining exponent vectors when computing products of group elements.

Scalar Multiplication (`scalev`) Scalar multiplication multiplies each element of a vector by a scalar value:

$$\texttt{scalev}(a, s) = [s \cdot a_1, s \cdot a_2, \ldots, s \cdot a_n] \tag{4.6}$$

where s is a scalar and $a = [a_1, a_2, \ldots, a_n]$.

```
(* Scalar multiplication: multiply vector by scalar *)
op scalev (a : exp list)(b : exp) = map (fun x => x*b) a.
```

Listing 4.9: Scalar Multiplication

This operation enables scaling of algebraic representations, crucial for constructing linear combinations in AGM proofs.

Vector Sum (`sumv`) Vector sum computes the sum of a list of vectors using fold-right with vector addition:

$$\texttt{sumv}([v_1, v_2, \ldots, v_k]) = v_1 + v_2 + \cdots + v_k \tag{4.7}$$

where each v_i is a vector and $+$ denotes vector addition (`addv`).

```
(* Sum of vectors: fold addition over list of vectors *)
op sumv (a : exp list list) = foldr addv zerov a.
```

Listing 4.10: Vector Sum

This operation aggregates multiple algebraic representations into a single representation, essential for oracle simulation.

Shift and Truncate (`shift_trunc`) The shift and truncate operation prepends a zero to the vector and then takes the first $q + 1$ elements:

$$\texttt{shift\_trunc}([v_1, v_2, \ldots, v_n]) = [0, v_1, v_2, \ldots, v_q] \tag{4.8}$$

This ensures the result has exactly $q+1$ elements, maintaining consistent vector dimensions.

```
(* Shift and truncate operation for oracle simulation *)
op shift_trunc (v : exp list) = take (q+1) (zero :: v).
```

Listing 4.11: Shift and Truncate Operation

This operation is particularly important in oracle simulation where new group elements must be incorporated into the existing algebraic representation while maintaining the proper dimensionality for the q-DDH structure.

Key Algebraic Properties and Lemmas The correctness of our reduction relies on several fundamental algebraic properties that we establish through formal lemmas. These lemmas are essential for the oracle simulation and reduction construction.

Lemma 1 (Linear Distributivity for prodEx)**.** *For group element list bases and exponent lists a, b of matching sizes:*

$$\mathsf{prodEx}(bases, \mathsf{addv}(a, b)) = \mathsf{prodEx}(bases, a) \cdot \mathsf{prodEx}(bases, b)$$

Proof: Section A.1.

Lemma 2 (Product Exponentiation Distributivity)**.** *For any group list gs and exponent n:*

$$(\prod gs)^n = \prod(map(\lambda g \to g^n, gs))$$

Proof: See Section A.1.

Lemma 3 (Scaling Consistency (Primary))**.** *For any group element list bases, exponent list exp, and scalar scala:*

$$\mathsf{prodEx}(bases, exp)^{scala} = \mathsf{prodEx}(bases, \mathsf{scalev}(exp, scala))$$

Proof: See Section A.1.

Lemma 4 (Scaling Consistency (Alternative))**.** *For any group element list bases, exponent list exp, and scalar scale:*

$$\mathsf{prodEx}(bases, exp)^{scale} = \mathsf{prodEx}(\mathsf{ex}(bases, \mathsf{nseq}(\mathsf{size}(bases), scale)), exp)$$

Proof: See Section A.1.

Lemma 5 (Zero Vector Identity)**.** *The product with a zero vector yields the identity element:*

$$\mathsf{prodEx}(bases, \mathsf{nseq}(n, zero)) = e \quad \text{for } n \geq 0$$

Proof: Section A.2.

Lemma 6 (Shift-Truncate Property)**.** *For bases and exponents of size $q + 1$ where the last element of exps is zero:*

$$\mathsf{prodEx}(bases, \mathsf{shift\_trunc}(exps)) = \mathsf{prodEx}(\mathsf{behead}(bases), exps)$$

Proof: Section A.3.

Lemma 7 (Empty Vector Properties)**.** *Products with empty vectors yield the identity:*

$$\mathsf{prodEx}(bases, []) = e \tag{4.9}$$
$$\mathsf{prodEx}([], exps) = e \tag{4.10}$$
$$\mathsf{prodEx}([], []) = e \tag{4.11}$$

Proof: Section A.2.

Lemma 8 (Vector Addition with Empty Operands)**.** *Vector addition with empty operands:*

$$\mathsf{addv}([], b) = [] \tag{4.12}$$
$$\mathsf{addv}(a, []) = [] \tag{4.13}$$
$$\mathsf{sumv}([]) = \mathsf{zerov} \tag{4.14}$$

Proof: Section A.4.

Lemma 9 (Size Preservation)**.** *Vector operations preserve or predictably modify sizes:*

$$\mathsf{size}(\mathsf{scalev}(v, s)) = \mathsf{size}(v) \tag{4.15}$$
$$\mathsf{size}(\mathsf{addv}(a, b)) = \min(\mathsf{size}(a), \mathsf{size}(b)) \tag{4.16}$$
$$\mathsf{size}(\mathsf{shift\_trunc}(v)) = \min(q + 1, \mathsf{size}(v) + 1) \tag{4.17}$$

Proof: Section A.5.

Lemma 10 (Cons Operation for prodEx)**.** *For a group element g, group list gs, exponent e, and exponent list es:*

$$\mathsf{prodEx}(g :: gs, e :: es) = g^e \cdot \mathsf{prodEx}(gs, es)$$

$$\mathsf{prodEx}(g :: gs, zero :: es) = \mathsf{prodEx}(gs, es)$$

Proof: Section A.6.

Lemma 11 (Double prodEx Composition)**.** *For any base list b and exponent lists $e1$, $e2$:*

$$\mathsf{prodEx}(\mathsf{ex}(b, e1), e2) = \mathsf{prodEx}(b, \mathsf{mulv}(e1, e2))$$

This lemma establishes the equivalence between nested prodEx operations and pointwise multiplication of exponent vectors. In mathematical notation:

$$\prod_{i=0}^{n-1} \left(b_i^{e1_i}\right)^{e2_i} = \prod_{i=0}^{n-1} b_i^{e1_i \cdot e2_i}$$

Proof: See Section A.7.

Lemma 12 (Ex-Map-prodEx Equivalence)**.** *For bases, exps, and constant c with matching sizes:*

$$\mathsf{ex}(\mathit{map}(\mathsf{prodEx}(\mathit{bases}), \mathit{exps}), \mathsf{nseq}(\mathsf{size}(\mathit{exps}), c)) = \mathit{map}(\mathsf{prodEx}(\mathsf{ex}(\mathit{bases}, \mathsf{nseq}(\mathsf{size}(\mathit{bases}), c))), \mathit{exps})$$

This lemma shows the commutativity between mapping and exponentiation operations. In expanded form:

$$\prod_{j=0}^{m-1} \left(\prod_{i=0}^{n-1} \mathit{bases}_i^{\mathit{exps}_j[i]}\right)^c = \text{map}\left(\lambda \mathit{exp\_vec} \to \prod_{i=0}^{n-1} (\mathit{bases}_i^c)^{\mathit{exp\_vec}[i]}, \mathit{exps}\right)$$

Proof: See Section A.7.

Lemma 13 (prodEx Map with Range)**.** *For generator g, exponent x, and range* $[s, e)$*:*

$$\mathit{map}(\lambda i \to g^{x^i}, \mathit{range}(s, e)) = \mathsf{ex}(\mathsf{nseq}(e - s, g), \mathit{map}(\lambda i \to x^i, \mathit{range}(s, e)))$$

This lemma provides an algebraic representation for geometric sequences of group elements:

$$[g^{x^s}, g^{x^{s+1}}, \ldots, g^{x^{e-1}}] = \mathsf{ex}([g, g, \ldots, g], [x^s, x^{s+1}, \ldots, x^{e-1}])$$

Proof: See Section A.7.

4.4.2 Advanced Algebraic Manipulation Lemmas

The oracle simulation and reduction construction require sophisticated algebraic manipulations beyond basic vector operations. The following lemmas establish advanced properties for range operations, vector transformations, and structural manipulations that are essential for the correctness of our bilateral reduction.

Lemma 14 (Range Simplification)**.** *For any secret key sk:*

$$g :: \mathit{map}(\lambda i \to g^{sk^i}, \mathit{range}(1, q+1)) = \mathit{map}(\lambda i \to g^{sk^i}, \mathit{range}(0, q+1))$$

Proof: See Section B.1.

Lemma 15 (Ex Cons Distribution). *For group element x, group list xs, exponent e, and exponent list es:*

$$\mathsf{ex}(x :: xs, e :: es) = (x^e) :: \mathsf{ex}(xs, es)$$

Proof: See Section B.2.

Lemma 16 (Ex Range Shift Property). *For secret key sk:*

$$\mathsf{ex}(map(\lambda i \to g^{sk^i}, range(0, q+1)), \mathsf{nseq}(q+1, sk)) = map(\lambda i \to g^{sk^i}, range(1, q+2))$$

Proof: See Section B.3.

Lemma 17 (Scalev of Nseq). *For integer n, exponents x and c:*

$$\mathsf{scalev}(\mathsf{nseq}(n, x), c) = \mathsf{nseq}(n, x \cdot c)$$

Proof: See Section B.4.

Lemma 18 (Drop-Addv Commutativity). *For vectors u, v of equal size and integer n:*

$$\mathsf{drop}(n, \mathsf{addv}(u, v)) = \mathsf{addv}(\mathsf{drop}(n, u), \mathsf{drop}(n, v))$$

Proof: See Section B.5.

Lemma 19 (Addv Size Inequality). *For vectors a, b with $\mathsf{size}(a) \leq \mathsf{size}(b)$:*

$$\mathsf{addv}(a, b) = \mathsf{addv}(a, \mathsf{take}(\mathsf{size}(a), b))$$

Proof: See Section B.6.

Lemma 20 (Drop-Sumv Commutativity). *For non-empty list xs of uniform-sized vectors and valid index n:*

$$\mathsf{drop}(n, \mathsf{sumv}(xs)) = \mathsf{sumv}(map(\mathsf{drop}(n), xs))$$

Proof: See Section B.7.

Lemma 21 (Sumv of Zero Vectors). *For non-negative integer n:*

$$\mathsf{sumv}(\mathsf{nseq}(n, \mathsf{zerov})) = \mathsf{zerov}$$

Proof: See Section B.8.

Lemma 22 (Sumv of Singleton Zero Vectors). *For positive integer n:*

$$\mathsf{sumv}(\mathsf{nseq}(n, \mathsf{nseq}(1, zero))) = \mathsf{nseq}(1, zero)$$

Proof: See Section B.9.

Lemma 23 (Zip Concatenation Distributivity). *For lists a_1, a_2, b_1, b_2 with $\mathsf{size}(a_1) = \mathsf{size}(b_1)$:*

$$\mathsf{zip}(a_1 \mathbin{++} a_2, b_1 \mathbin{++} b_2) = \mathsf{zip}(a_1, b_1) \mathbin{++} \mathsf{zip}(a_2, b_2)$$

Proof: See Section B.10.

Lemma 24 (Product Concatenation). *For group lists xs and ys:*

$$\prod(xs \mathbin{++} ys) = \prod(xs) \cdot \prod(ys)$$

Proof: See Section B.11.

Lemma 25 (prodEx Split with Last Zero). *For equal-sized lists bases and exps where the last exponent is zero:*

$$\mathsf{prodEx}(bases, exps) = \mathsf{prodEx}(\mathsf{take}(\mathsf{size}(bases) - 1, bases), \mathsf{take}(\mathsf{size}(exps) - 1, exps))$$

Proof: See Section B.12.

Lemma 26 (Behead-Drop Equivalence). *For any group list base:*

$$\mathsf{behead}(base) = \mathsf{drop}(1, base)$$

Proof: See Section B.13.

Chapter 5

Evaluation : Formalizing the Bilateral Reduction

This chapter shows the formal construction and verification of our bilateral equivalence between IND-CCA1 security of ElGamal and the q-DDH assumption. We indicates both directions of the reduction with complete EasyCrypt implementations and proofs first.

5.1 Technical overview of bidirectional reduction

Our core technical contribution is the establishment of two reductions that bridge computational equivalence between:

5.1.1 Forward Reduction: A__from__INDCCA1.

This reduction will convert all IND-CCA1 adversary B against ElGamal into a q-DDH distinguisher.,which works as follows:

1. **Challenge Reception:** It will first receiving a q-DDH challenge $(g^x, g^{x^2}, \ldots, g^{x^q}, g^r, T)$, then set the public key as $pk = g^x$.
2. **Oracle Simulation:** Then it will leverage a limited decryption oracle that only processes queries (C, z) where adversary provides an explicit linear representation z such that $C = \text{prodEx}(l, z)$, while $l = [g, g^x, \text{previous\_results}]$ is the list of group elements already known.
3. **Challenge Programming:** Finally the adversary is expected to request the encryption challenge, that sets $C^* = g^r$ and $K^* = T$, giving the q-DDH challenge directly into the IND-CCA1 game.

4. **Decision Extraction:** The reduction outputs the adversary's guess as its q-DDH decision.

5.1.2 Backward Reduction: B_from_qDDH.

This reduction converts any q-DDH distinguisher A into an IND-CCA1 adversary:

1. **Challenge Embedding:** The reduction embeds the received IND-CCA1 challenge into a q-DDH problem by generating the tuple $(g^x, g^{x^2}, \ldots, g^{x^q}, C^*, K^*)$ where x is the secret key and (K^*, C^*) is the challenge key-ciphertext pair.
2. **Game Simulation:** Then it will simulates the q-DDH game for the attacker by the same oracle discipline as the forward reduction.
3. **Advantage Preservation:** The distinguisher's decision is directly translated into an IND-CCA1 guess.

5.2 Forward Reduction: IND-CCA1 to q-DDH

The forward reduction constructs a q-DDH adversary `A_from_INDCCA1` that uses any IND-CCA1 adversary as a subroutine. This reduction demonstrates that if ElGamal's IND-CCA1 security can be broken, then the q-DDH assumption can also be broken.

5.2.1 Module Structure and State Variables

```
module (A_from_INDCCA1 (A : Adv_INDCCA1) : A_qDDH) = {

  (* State variables for the reduction *)
  var gxs : group list         (* Powers g^x, g^{x^2}, ..., g^{x^q} *)
  var l : group list           (* List of group elements (oracle state) *)
  var reps : exp list list     (* Linear representations of l in basis (g::gxs) *)
```

Listing 5.1: A_from_INDCCA1 Module Structure

The module takes an IND-CCA1 adversary `A` as parameter and implements the q-DDH adversary interface `A_qDDH`;

`gxs` stores the powers $[g^x, g^{x^2}, \ldots, g^{x^q}]$ extracted from the q-DDH challenge. This forms the basis for our algebraic representation system;

`l` maintains the list of group elements that the adversary has "seen" through oracle queries, similar to the oracle state in the original IND-CCA1 game;

`reps` stores the corresponding linear representations of elements in `l` with respect to the basis $(g, g^x, g^{x^2}, \ldots, g^{x^q})$. This enables algebraic manipulation without knowing the discrete logarithm.

5.2.2 Internal Oracle Simulation

The core innovation is the internal oracle `O_Internal` that simulates the IND-CCA1 decryption oracle using only the q-DDH challenge, without access to the actual secret key.

```
  (* Internal oracle that simulates CCA1 oracle using q-DDH
challenge *)
  module O_Internal : Oracles_CCA1i = {
    var sk : sk_t
    var qs : (ctxt_t * key_t option) list
    var challenge_c : ctxt_t

    proc init(sk_init : sk_t) = {
    sk <- sk_init;
       qs <- [];

      l <- [];
    }

    (* Core oracle: simulate decryption without knowing secret
key *)
    proc dec(c : ctxt_t, z : exp list) : key_t option = {
      var p : key_t option;
      var rep_c, rep_p;
      var invalid_query : bool;

      (* Validity check: query limit and representation
consistency *)
      invalid_query <- (q < size qs + 2  \/ c <> prodEx l z);

      (* Compute representation of ciphertext in basis (g::gxs) *)
      rep_c <- sumv (map (fun x : exp list * exp => scalev x.`1
x.`2)
         (zip reps z));
           (* Prepend zero for g^0 term *)
       rep_p <- ( shift_trunc rep_c);

      (* Compute corresponding group element *)
      p <- Some (prodEx (g :: gxs) (rep_p));

      (* Update state if query was valid *)
          if (!invalid_query) {
        reps <- rep_p :: reps ;
        l <-   oget p :: l ;           (* Add to group element
list *)
        qs <- (c, p) :: qs;       (* Record query *)
             (* Store representation *)
      }

      (* Return result *)
      return (if invalid_query then witness else p);
    }
  }
```

Listing 5.2: Internal Oracle Implementation

Module declaration implementing the `Oracles_CCA1i` interface with state variables for secret key, query history, and challenge ciphertext, and an initialization procedure that sets up the oracle state; note that when `sk` is stored, it's not actually used in the simulation, the oracle operates entirely using algebraic manipulation, and the key decryption procedure that must simulate decryption without knowing the secret key proceeds by introducing local variables for the computation: `p` for the result, `rep_c` and `rep_p` for algebraic representations, and `invalid_query` for validity checking, after which a validity check integrating two conditions is performed, namely (i) the query limit `q < size qs + 2` that ensures we don't exceed the allowed number of queries and (ii) the representation consistency `c <> prodEx l z` that verifies the provided representation `z` correctly represents ciphertext `c`. Then the algebraic magic happen: we compute the representation of ciphertext `c` in the basis $(g, g^x, \ldots, g^{x^q})$ by taking each previously seen element's representation from `reps`, scaling each representation by the corresponding coefficient in `z`, and summing all scaled representations using `sumv`; next, the `shift_trunc` operation prepends a zero and truncates to maintain proper dimensionality, which accounts for the fact that the decryption result c^x has a different representation structure, and then compute the actual group element corresponding to the representation using `prodEx` with the full basis $(g, g^x, \ldots, g^{x^q})$.

Finally, when it comes to state update for valid queries, we add the computed representation to `reps`, add the computed group element to `l`, and record the query–response pair in `qs`, and we return the computed result for valid queries, or a witness value for invalid queries.

5.2.3 Main Reduction Procedure

```
  (* Main reduction procedure *)
  proc guess(gtuple : group list) : bool = {
    var c : ctxt_t;
    var k : key_t;
    var b' : bool;
    var x_exp : exp;
```

```
7
8      (* Parse q-DDH challenge: gtuple = [g^x, g^{x^2}, ...,
   g^{x^q}, g^r, T] *)
9      gxs <- take q gtuple;              (* Extract [g^x, ...,
   g^{x^q}] *)
10     c <- nth witness gtuple q;         (* Extract g^r (ciphertext)
   *)
11     k <- nth witness gtuple (q + 1);   (* Extract T (challenge
   key) *)
12
13     (* Initialize internal oracle *)
14     O_Internal.init(witness);
15
16       (* Set initial state: adversary has seen g and g^x *)
17     l <-   head witness gxs :: g::   [];
18     (* Corresponding representations in basis (g::gxs) *)
19     reps <- (* g = g^1 * (g^x)^0 * ... *)
20       (zero :: one :: nseq (q-1) zero) ::[(one :: nseq q zero)];
   (* g^x = g^0 * (g^x)^1 * ... *)
21
22
23
24     (* Run IND-CCA1 adversary *)
25     A(O_Internal).scout(head witness gxs);      (* Scout phase *)
26     b' <@ A(O_Internal).distinguish(k, c);     (* Challenge phase
   *)
27
28     return b';
29   }
```

Listing 5.3: Main Reduction Procedure

From the first 6 lines, it procedure signature and local variable declarations. The procedure receives a q-DDH challenge tuple and must return a boolean guess.

After that(line 9), it extract the first q elements $[g^x, g^{x^2}, \ldots, g^{x^q}]$ from the challenge tuple using `take q`.

And it will extract the $(q+1)$-th element, which represents g^r (the ElGamal ciphertext component); extract the $(q+2)$-th element, which is either g^{xr} (real) or g^{xr+z} (random) depending on the q-DDH challenge bit.

In the next step it will initialize the internal oracle with a witness value (since we don't have a real secret key) and set up the initial state where the adversary has seen g^x (the public key) and g (the generator).

Additionally it will also initiate the representation vectors:

- g is represented as $(0, 1, 0, \ldots, 0)$ in basis $(g, g^x, \ldots, g^{x^q})$

- g^x is represented as $(1, 0, 0, \ldots, 0)$ in the same basis

After that the reduction will run the adversary's scout phase, giving it access to the public key g^x and the simulated oracle and run the adversary's distinguish phase with the challenge key k and ciphertext c.

Finally, it return the adversary's guess, which becomes our q-DDH distinguisher's output.

5.3 Backward Reduction: q-DDH to IND-CCA1

The backward reduction constructs an IND-CCA1 adversary `B_from_qDDH` that uses any q-DDH adversary as a subroutine. This reduction demonstrates that if the q-DDH assumption can be broken, then ElGamal's IND-CCA1 security can also be broken.

5.3.1 Module Structure and Scout Phase

```
(** Alternative adversary construction **)
module (B_from_qDDH (A  : A_qDDH) : Adv_INDCCA1)(O :
Oracles_CCA1i) = {
  var gxs : group list

  (* Scout phase: build up powers of x using decryption oracle *)
  proc scout(pk:pk_t) : unit ={
     var i   : int;
     var p   : key_t option;
     gxs <- [pk];
     i <- 1 ;

     while (i <= q-1) {

     p <@ O.dec(last witness gxs, ( one :: nseq i zero));

     gxs <- gxs ++ [oget p];
     i <- i + 1;
   }}
```

Listing 5.4: B_from_qDDH Module Structure

Module declaration taking a q-DDH adversary `A` and implementing the IND-CCA1 adversary interface with oracle access. Then we state variable `gxs` to store the powers of x that we'll construct through oracle queries. It will then scout phase procedure where we can make decryption queries to build up our knowledge. The Local variables are `i` for loop counter and `p` for oracle responses.

Before the loop start, we first initialize `gxs` with the public key $pk = g^x$. Then we start loop counter at 1 (we already have g^x, now we need $g^{x^2}, g^{x^3}, \ldots$) and loop to construct powers $g^{x^2}, g^{x^3}, \ldots, g^{x^q}$.

We can gain insight from quering the oracle with ciphertext `last witness gxs` (the highest power we have so far) and representation vector `(one :: nseq i zero)`:

- This represents the query "decrypt g^{x^i} with representation $(1, 0, 0, \ldots, 0)$"
- The oracle will return $g^{x^{i+1}}$ (since decryption multiplies by x)

Finally we append the new power to our list: `gxs` now contains $[g^x, g^{x^2}, \ldots, g^{x^{i+1}}]$ and increase counter to build the next power.

5.3.2 Distinguish Phase

```
  (* Convert to q-DDH challenge *)
   proc distinguish(k: key_t, c: ctxt_t) : bool = {
      var b'  : bool;

      (* Pass challenge to q-DDH adversary *)
      b' <@ A.guess( gxs ++ [c] ++ [k]);
      return b';
   }
```

Listing 5.5: B_from_qDDH Distinguish Phase

Detailed Analysis:

Firstly, the procedure is expected to receiving the challenge key `k` and ciphertext `c`. And at the initial stage, it will initiate a local variable `b'` for the adversary's response.

After that, which is the crucial step of this phase: construct a q-DDH challenge tuple by concatenating:

- `gxs`: The powers $[g^x, g^{x^2}, \ldots, g^{x^q}]$ we built in the scout phase
- `[c]`: The challenge ciphertext g^r
- `[k]`: The challenge key, which is either g^{xr} (real) or random

Finally, it will return the q-DDH adversary's guess as the IND-CCA1 distinguisher's result.

5.4 Main Theorem: Bilateral Equivalence

Our main result establishes a tight bilateral equivalence between the two security notions:

Theorem 1 (Bilateral Equivalence)**.** *Under the algebraic group model, for any cyclic group $\mathcal{G}$ of prime order p:*

5.4.1 Forward Direction.

For any IND-CCA1 adversary $\mathcal{B}$ making at most q decryption queries, there exists a q-DDH adversary A_from_INDCCA1($\mathcal{B}$) such that:

$$\Pr[\mathit{IND\_CCA1\_P}(\mathit{ElGamal}, \mathcal{B}).\mathit{main}() : \mathit{res}] = \Pr[\mathit{QDDH}(\mathit{A\_from\_INDCCA1}(\mathcal{B})).\mathit{main}() : \mathit{res}]$$

5.4.2 Backward Direction.

For any q-DDH adversary $\mathcal{A}$, there exists an IND-CCA1 adversary B_from_qDDH($\mathcal{A}$) such that:

$$\Pr[\mathit{QDDH}(\mathcal{A}).\mathit{main}() : \mathit{res}] = \Pr[\mathit{IND\_CCA1\_P}(\mathit{ElGamal}, \mathit{B\_from\_qDDH}(\mathcal{A})).\mathit{main}() : \mathit{res}]$$

Both reductions are tight with no security loss.

And the main theorem are formalized as the following EasyCrypt lemmas:

```
(* Forward direction *)
lemma qDDH_Implies_INDCCA1_ElGamal &m :
  Pr[IND_CCA1_P(ElGamal,B).main() @ &m : res] =
  Pr[QDDH(A_from_INDCCA1(B)).main() @ &m : res].

(* Backward direction *)
lemma INDCCA1_ElGamal_Implies_qDDH &m :
  Pr[QDDH(A).main() @ &m : res] =
  Pr[IND_CCA1_P(ElGamal,B_from_qDDH(A)).main() @ &m : res].
```

Listing 5.6: Main Bilateral Equivalence Lemmas

This completes our formalization of the bilateral equivalence between IND-CCA1 security of ElGamal and the q-DDH assumption, providing the first EasyCrypt machine-checked proof of this fundamental result.

Complete EasyCrypt Implementation: The full formal verification code, including all lemmas and proofs, is available at: https://github.com/GuSheldom/INDCCA-QDDH

5.5 Overall Analysis of INDCCA1_ElGamal_Implies_qDDH Code

This part indicates the core of the proof for the reduction from IND-CCA1 ElGamal to q-DDH. The entire proof uses the **byequiv** strategy, constructing equivalence between two games to complete the reduction proof.

5.5.1 Key components of Proof Structure

1. **Code Alignment**: Using `swap` operations to reorder instructions in both games, ensuring the related random sampling and computation steps align accurately.
2. **Loop Invariant Maintenance**: Establishing a loop-for-while invariant that ensures at each iteration, we initiate counter `i` remains within valid range and the group element list `gxs` is introduced to correctly corresponds to the exponent sequence. In addition, oracle's base list `l` is maintained and ensuring the query count limitations are satisfied.
3. **List Operation Equivalence**: Proving that `map`, `range`, `rev`, and other list operations produce same results in both games, particularly building equivalence between the last element of the lists and their `prodEx` representations.
4. **Random Variable Transformation**: Applying the crucial random variable transformation `rnd (fun z => g ^ (z + sk0{2} * y{2}))(fun k' => loge k' - sk0{2} * y{2})`, which is a bijective transformation preserving distributional consistency.

5.6 Overall Analysis of qDDH_Implies_INDCCA1_ElGamal Code

This part represents the reverse direction of the reduction, indicating that q-DDH hardness can imply to IND-CCA1 security of ElGamal. The proof leverage the **byequiv** strategy to establish game equivalence through sophisticated algebraic manipulations and oracle simulations.

5.6.1 Key components of Proof Structure

1. **Code Alignment and Setup**:
   - Uses `swap` operations to align random variable sampling between games
   - Establishes the foundational structure with `proc. inline*. swap{1} 11 -9. swap{1} 14 -10`
   - Brings random coins to the front for proper synchronization
2. **Oracle Invariant Maintenance**: The proof first introduce an 8-condition invariant for oracle simulation:
   - **Query List Consistency**: `IND_CCA1_P.OS.l{1} = A_from_INDCCA1.l{2}`
   - **Query Count Synchronization**: Between different oracle implementations
   - **Representation-Ciphertext Mapping**: `A_from_INDCCA1.l{2} = map (prodEx (g :: A_from_INDCCA1.gxs{2})) reps`

- **Base Elements Construction**: From secret key powers
- **Uniform Representation Size**: All vectors have size `q + 1`
- **Trailing Zeros Constraint**: For valid algebraic representations
- **Representation Count Relationship**: Links query count to representation count

3. **Algebraic Transformations**:
    - **prodEx Flattening**: Transform nested `prodEx` operations leveraging the lemma `ex_map_prodEx`
    - **Distributivity Applications**: Uses `prodEx_addv_distributive` and `prodExConsGeneral`
    - **Zero Suffix Proofs**: Proving that representation vectors have trailing zeros through `drop_scalev` and `scalev_nseq_zero`
4. **Advanced List Manipulation Techniques**:
    - **Range Splitting**: Decomposes ranges, for instance, `range 1 (q + 2) = range 1 (q + 1) ++ [q + 1]`
    - **Take/Drop Operations**: Uses `take_cat`, `drop_sumv`, and `nth_drop` for precise list manipulation
    - **Zip and Map Compositions**: Complex operations on paired lists
5. **Random Variable Transformation**:
    - Applies `rnd (fun k' => loge k'- sk0{1} * y{1}) (fun z => g ^ (z + sk0{1} * y{1}))`
    - Establishes bijectivity through logarithm-exponentiation inverse pairs
    - Keep uniform distribution over the key space

5.6.2 Proof Complexity Analysis

It is obvious to see the reverse direction proof is more complex than the forward one due to: Firstly, we must generate valid and provable representations from IND-CCA1 queries. Additionally, we are required to preserve various invariants across different data structures. Then, it is important to make careful analysis of boundary case in list operations and algebraic manipulations. Finally, accroding to the code, we use complex reduction on list structures to construct universal properties.

5.7 Comparison with Traditional Paper-Based Proofs

Our formal verification approach demonstrates significant advantages over traditional paper proofs, as exemplified by the reduction proof shown in the referenced paper (Theorem 5.2):

Our formal proof indicates several advantages compared to traditional paper proofs, as shown in the referenced paper [4].

5.7.1 Rigor and Precision Comparison

The paper proof [4] states "First note that given an adversary A_{alg} against q-ddh$_G$ one can easily construct an adversary B_{alg} against ind-cca1..." which represents a high-level, intuitive description that leaves implementation details unspecified. By comparison, the formalization we provide, indicates a complete, machine-verified construction of the reduction through the `B_from_qDDH` module, with any step of the oracle simulation fully defined and verified.

5.7.2 Oracle Simulation Accuracy

The paper states that "B_{alg} can express C as $C = \prod_{j \geq 0} g^{a_j x^j}$" and utilize this for decryption queries, but the exact approach that maintaining consistency and handling boundary cases is not specified. However, our `O_CCA1_Limited` oracle provides precise query counting, algebraic representation validation, and state consistency checks. Every decryption query is handled with mathematically verified accuracy through the `prodEx`.

5.7.3 Distribution Equivalence

In the paper [4], claims like "Clearly, $(g^x, g^{x^2}, \ldots, g^{x^q}, C^*, K^*)$ is correctly distributed" has been specified in our approach by establishing distributional equivalence through rigorous random variable transformations with machine-checked bijectivity proofs, eliminating possibility of distributional errors.

5.7.4 Security Loss Analysis

The paper proof states advantage equations like $\mathbf{Adv}^{q-ddh}_{G,B_{alg}} = \mathbf{Adv}^{ind-cca1}_{EG,G,A_{alg}}$ without detailed step-by-step verification of tightness. In contrast, every probability transformation is mechanically verified to main equality, providing concrete assurance of the reduction claim.

5.7.5 Elimination of Proof Gaps

Traditional cryptographic proofs, while providing valuable insight, often contain implicit steps, to eliminate this effect, our EasyCrypt proof:

- Requiring explicit construction of all reduction components
- Machine-verifying every step and probability calculation
- Ensuring that all algebraic manipulations are mathematically sound

- Offering executable reduction algorithms with precise specifications

5.8 Main Results

5.8.1 enhance the rigorous and the readability of algebraic reduction

In the traditional paper proof, it is frequently seen that some paper proof summarize this idea in one sentence, ' we can extract its representation since it is algebraic. However, in our proof, all such reasoning must be explicit in the game logic, to be specific, we are required to specify the rule that 'invalid queries should return a fixed value', while formally indicate that this behavior do not increase the attackers' success probability.

As a result, in our project, the proof becomes entirely transparent, for instance, we specify the quantity of the queries that adversary can make; what conditions the oracle should update state. And every step has a precise relationship between code and theorem. This coherence ensures that the formal proof not only eliminate the hidden flaws, but also offers a reproducible blueprint for future reductions and research.

5.8.2 motivate the research on Algebraic Group Model

In AGM(Algebraic Group Model), we make a formalized proof about the equivalence between security of IND-CCA1 of ElGamal and q-DDH assumption. In the paper [4], they prove a tight equivalence showing that, under the AGM, breaking the IND-CCA1 security of ElGamal is equivalent to solving a q-DDH problem.

Our EasyCrypt formalization confirms this result: we build a reduction showing that any algebraic adversary breaking ElGamal's IND-CCA1 security can be converted into a q-DDH distinguisher, with both advantages exactly equal.

We contribute to this theory in two main ways. First, it offers a verification that the equivalence fully holds under the AGM, eliminating possible mistakes or hidden assumptions that may exist in paper proofs, which strengthens the AGM as an analytical framework that, by restricting adversaries to algebraic behavior, allows tight security proofs that are difficult to achieve in the standard model.

Second, it indicates the power of the AGM itself: due to adversaries must be "algebraic" (i.e., every group element they output must come with a linear representation), indicates that the unprovable results in the standard model can often be proven in the AGM. For example, Fuchsbauer et al. [4] showed that several assumptions (such as CDH, SDH, and interactive LRSW) become equivalent to the discrete logarithm problem under the AGM.

Our formalization adds a tool-verified instance to this line of results, demonstrating that the AGM can generate clear and theoretically meaningful security statements. In addition, the formal proof highlights the tightness and efficiency of reductions in the AGM, which is important for future theoretical research. When integrating with known attacks in the generic group model, the AGM-based proof characterizes the complexity edge for breaking ElGamal's CCA security. All in all, this work is not only a validation of a single security result but also a relation between intuitive reasoning and formal verification in cryptographic theory.

5.8.3 Formalizing Algebraic Adversaries in EasyCrypt

One of the key contribution of our work is emphasizing how the special adversary model of the AGM can be captured fully using the EasyCrypt framework. The AGM is a notion that lies between the standard model and the generic group model: it allows adversaries to perform group operations but requires they provide its algebraic representation whenever they output a new group element. In the code, the oracle enforces this restriction by checking whether `c = prodEx l z` holds, where l is the list of group elements already known to the adversary, and z is the vector of exponents given by the query. Then, if the query fails the condition, the oracle is expected to return a fixed value witness and do not update the state, which directly reflects the constraint of the AGM that the adversary must be able to express each one as a linear combination of previously observed elements rather than generate new group elements.

To implement this, we define a oracle interface that extends the traditional decryption oracle by introducing a exponent-vector parameter. Then the exploration and the distinguishing phases of the adversary are designed to interact with this algebraic oracle. By this modeling, the security game is embedded within EasyCrypt, assuring that every reasoning step are able to verified mechanically. This indicates that even non-standard adversary models can be precisely express and represented and formally verified through appropriate modular design in EasyCrypt.

Chapter 6

Conclusion and Future work

6.1 Conclusion

Upon reflection, the formalization of algebraic adversaries and oracle simulation in EasyCrypt we introduced not only used for verifying the IND-CCA1 security of ElGamal, but also offers a general and reusable basic structure for future theoretical analysis for Algebraic Group Model (AGM).

The core operator we introduced, `prodEx`, with its supporting lemmas on distributivity, scaling, and vector operations, enables almost all manipulation of exponent-linear relations within EasyCrypt, give an essential capability for reasoning about algebraic adversaries in a mechanized setting.

Additionally, this framework can be directly extended to formalize and verify other results from the AGM paper by Fuchsbauer, Kiltz, and Loss [4]. For instance, their proof of the *Algebraic Computational Diffie–Hellman (aCDH)* assumption is equivalent to the standard *Computational Diffie–Hellman (CDH)* assumption relies on representing every group element as a linear combination of previously known elements. In their paper proofs, this reasoning is expressed informally through algebraic extraction arguments. In the future analysis, by using our `prodEx` operator, such relations can be explicitly encoded as:

$$h = \mathsf{prodEx}([g_1, g_2, \dots, g_n], [a_1, a_2, \dots, a_n]),$$

which allows EasyCrypt to mechanically verify whether the adversary's output h follows from known bases. This transformation turns the intuitive statement 'since the adversary is algebraic, we can extract its coefficients' in the original paper into a formally verified lemma within EasyCrypt.

6.2 Future Directions

Looking forward, this approach provides several promising research directions. First, it could support the development of a **mechanized AGM lemma library**, covering results such as a CDH–CDH equivalence, algebraic SDH reductions [19], and algebraic LRSW assumptions, providing a reusable foundation for future mechanized cryptographic proofs. Additionally, by combining these algebraic representations with probabilistic reasoning in EasyCrypt, it might be possible to make proof on **hybrid models** that mix algebraic reasoning with post-quantum assumptions (e.g., MLWE [20] or isogeny-based systems [21]).

Such extensions would not only strengthen the theoretical rigor of the AGM but also provide a concrete bridge between algebraic reasoning and post-quantum formal verification. All in all, our work make contribution to the long-term goal of creating a unified, machine-verified code approach for a accurate and reproducible cryptographic security reductions.

Appendix A

Formal Proofs of Key Lemmas

This appendix contains the complete EasyCrypt proofs for the key lemmas used in our vector operations and algebraic manipulations. These proofs are extracted directly from our `INDCCA.ec` file without modification.

A.1 Distributivity and Scaling Proofs

```
(* Distributivity: (prod gs)^n = prod (map (^n) gs) *)
lemma prod_exp_distributive (gs : group list) (n : exp) :
  prod gs ^ n = prod (map (fun g => g ^ n) gs).
proof.
  elim: gs => [|g gs IH] //=.
  smt(@G @GP @List).
  rewrite /= !prod_cons.
  have exp_mult_dist: (g * prod gs) ^ n = g ^ n * (prod gs) ^ n.
    smt(@G @GP).
  rewrite exp_mult_dist. rewrite IH. smt().
qed.
```

Listing A.1: Proof of Product Exponentiation Distributivity

```
(* Scaling lemma: prodEx with scaled exponents *)
lemma prodExScale1 bases exp scala :
    prodEx bases exp ^ scala = prodEx bases (scalev exp scala).
proof.
  rewrite /prodEx. rewrite /ex. rewrite !prod_exp_distributive.
congr.
  rewrite /scalev. search zip. rewrite zip_mapr.
  rewrite -!map_comp. congr. apply fun_ext => xy.
  by case: xy => [x y] /=; rewrite /(\o) /= expM.
qed.
```

Listing A.2: Proof of Scaling Consistency (Primary)

```
(* Alternative scaling lemma using ex *)
lemma prodExScale2 bases exp scale :
    prodEx bases exp ^ scale = prodEx (ex bases (nseq (size
bases) scale)) exp.
proof.
  rewrite prodExScale1. rewrite !/prodEx /ex. congr.
  elim: bases exp => [|base_h base_t IH] [|exp_h exp_t] //=.
  rewrite !zip_nil_l. smt().
  rewrite !zip_nil_l. smt().
  rewrite zip_nil_r. rewrite /scalev. rewrite zip_nil_r. smt().
  rewrite /scalev /=. simplify.
  have nseq_cons: nseq (1 + size base_t) scale = scale :: nseq
(size base_t) scale.
    rewrite -nseqS. smt(size_ge0). smt(@G).
  rewrite nseq_cons. simplify.
  split. smt(@G @GP). rewrite IH. smt().
qed.
```

Listing A.3: Proof of Scaling Consistency (Alternative)

```
(* Addition distributivity for prodEx *)
lemma prod_ex_addv (base : group list) (a b : exp list) :
    size a = size base => size b = size base =>
    prod (ex base (addv a b)) = prod (ex base a) * prod (ex base
b).
proof.
  move=> size_a size_b.
  rewrite /addv.
  move: a b size_a size_b.
  elim: base.
  move=> a b size_a size_b. rewrite /ex. rewrite !zip_nil_l.
    by smt(@List @G).
  move=> g gs IH a b size_a size_b.
  case: a size_a => [|a_h a_t] size_a; first by smt(@List).
  case: b size_b => [|b_h b_t] size_b; first by smt(@List).
  rewrite /ex !zip_cons !map_cons !prod_cons /=.
  rewrite prod_cons expD /ex. rewrite IH. smt(@G @GP). smt().
rewrite /ex.

  pose A := prod (map (fun (i : group * GP.exp) => i.`1 ^ i.`2)
(zip gs a_t)).
  pose B := prod (map (fun (i : group * GP.exp) => i.`1 ^ i.`2)
(zip gs b_t)).
  pose X := g ^ a_h.
  pose Y := g ^ b_h.
  rewrite mulcA. rewrite mulcA. congr.

  have com : Y * A = A * Y. rewrite mulcC. smt().
  rewrite -mulcA. rewrite com.
  smt(@G @GP).
qed.

(* ProdEx distributes over vector addition *)
lemma prodEx_addv_distributive (bases : group list) (a b : exp
list) :
   size a = size bases => size b = size bases =>
   prodEx bases (addv a b) = prodEx bases a * prodEx bases b.
proof.
    move => ha hb.
    rewrite /prodEx prod_ex_addv. smt(). smt(). smt().
qed.
```

Listing A.4: Proof of Linear Distributivity

A.2 Zero Vector and Identity Proofs

```
(* ProdEx with all zero exponents is identity *)
lemma prodEx_nseq_zero (bases : group list) (n : int) :
  0 <= n =>
  prodEx bases (nseq n zero) = e.
proof.
  move=> ge0_n.
  rewrite /prodEx /ex.
  elim: bases n ge0_n => [|g gs IH] n ge0_n.
  - (* bases = [] *)
    by rewrite zip_nil_l map_nil prod_nil.
  - (* bases = g :: gs *)
    case: (n = 0) => [n_zero | n_pos].
    + (* n = 0 *)
      rewrite n_zero nseq0.
      by rewrite zip_nil_r map_nil prod_nil.
    + (* n > 0 *)
  have n_gt0: 0 < n by smt().
  have nseq_cons: nseq n zero = zero :: nseq (n-1) zero.
  rewrite -nseqS. smt(). simplify. trivial.
  rewrite nseq_cons zip_cons map_cons prod_cons /= exp0 IH.
  smt(). smt(@G @GP).
qed.

```

```
(* ProdEx with empty exponent list is identity *)
lemma prodEx_nil_r (bases : group list) :
  prodEx bases [] = e.
proof.
  rewrite /prodEx /ex.
  rewrite zip_nil_r.
  rewrite map_nil.
  rewrite prod_nil.
  trivial.
qed.

lemma prodEx_nil_l (exps : exp list) :
  prodEx [] exps = e.
proof.
  rewrite /prodEx /ex.
  rewrite zip_nil_l.
  rewrite map_nil.
  rewrite prod_nil.
  trivial.
qed.

lemma prodEx_nil :
  prodEx [] [] = e.
proof.
  by rewrite prodEx_nil_l.
qed.
```

Listing A.5: Proof of Zero Vector Properties

A.3 Shift-Truncate Operation Proof

```
(* Shift-truncate lemma: key property for oracle simulation *)
lemma prodExShiftTrunce bases exps:
       size bases = q +1   =>
       size exps = q + 1 =>
       drop q exps = nseq 1 zero =>
       prodEx bases (shift_trunc exps) = prodEx (behead bases)
exps.
proof.
      move=>  size_bases size_exps last_zero.
      rewrite /shift_trunc.
      have take_eq : (take (q+1) (zero :: exps)) = zero :: (take
q exps).
      smt(@List gt0_q).
      rewrite take_eq.
      have exps_split: exps = take q exps ++ [zero].
      rewrite -(cat_take_drop q).
      have h : take q (take q exps ++ drop q exps) = take q exps.
      have size_take_q : size(take q exps) = q by smt(@List
gt0_q).
      rewrite take_size_cat. smt(). smt(). rewrite h.
      rewrite last_zero. smt(@List @G @GP).
      rewrite exps_split.
      have size_take_q : size(take q exps) = q by smt(@List
gt0_q).
      rewrite take_size_cat. smt().
      have base_cons : (head witness bases) :: (behead bases) =
bases.
      have : bases <> []. smt().
      apply head_behead.
      have h : (prodEx bases (zero :: take q exps)) =
      prodEx (head witness bases :: behead bases) (zero :: take q
exps).
      smt().
      rewrite h.
      rewrite prodExCons.
      pose oversize := (take q exps ++ [zero]).
      pose a := (behead bases).
      rewrite (prodEx_sizele a oversize). smt().
      have size_a:  size a = q by smt(@G @List @GP gt0_q).
      rewrite size_a.   rewrite /oversize.
      rewrite take_size_cat. smt(@List @G gt0_q). smt().
qed.
```

Listing A.6: Proof of Shift-Truncate Property

A.4 Vector Addition Properties

```
(* Vector addition with empty left operand *)
lemma addv_nil_l (b : exp list) :
  addv [] b = [].
proof.
  rewrite /addv.
  rewrite zip_nil_l.
  rewrite map_nil.
  trivial.
qed.

(* Vector addition with empty right operand *)
lemma addv_nil_r (a : exp list) :
  addv a [] = [].
proof.
  rewrite /addv.
  rewrite zip_nil_r.
  rewrite map_nil.
  trivial.
qed.

(* Vector addition with empty operand *)
lemma addv_empty (a b : exp list) :
  a = [] \/ b = [] =>
  addv a b = [].
proof.
  case => [a_empty | b_empty].
  - by rewrite a_empty addv_nil_l.
  - by rewrite b_empty addv_nil_r.
qed.

(* Sum of empty vector list is zero vector *)
lemma sumv_nil :
  sumv [] = zerov.
proof.
  rewrite /sumv. simplify. smt().
qed.

(* sumv concat lemma*)
lemma sumv_cons (v : exp list) (vs : exp list list) :
    sumv (v :: vs) = addv v (sumv vs).
proof.
  rewrite /sumv. simplify. smt().
qed.
```

Listing A.7: Proof of Vector Addition Properties

A.5 Size Preservation Proofs

```
(* Size of scaled vector equals original size *)
lemma size_scalev (v : exp list) (s : exp) :
  size (scalev v s) = size v.
proof.
  rewrite /scalev.
  by rewrite size_map.
qed.

(* Size of sum of vectors with uniform size *)
lemma size_sumv (l : exp list list) :
  (forall v, v \in l => size v = q + 1) =>
  size (sumv l) = q + 1.
proof.
  elim: l => [|v vs IH] uniform_size.
  - rewrite sumv_nil /zerov.
```

```
    by rewrite size_nseq ler_maxr // gt0_q.
  - rewrite sumv_cons.
    have size_v: size v = q + 1.
      by apply uniform_size; rewrite in_cons.
    have size_sumv_vs: size (sumv vs) = q + 1.
      apply IH => w w_in_vs.
      by apply uniform_size; rewrite in_cons w_in_vs orbT.
    rewrite /addv size_map size_zip.
    by rewrite size_v size_sumv_vs minrr.
qed.
```

Listing A.8: Proof of Size Preservation Properties

A.6 prodEx cons Proofs

```
(* Cons with zero exponent lemma *)
lemma prodExCons bs e b :
  prodEx (b :: bs) (zero :: e) = prodEx bs e.
proof.
  rewrite /prodEx /ex. rewrite zip_cons. rewrite map_cons.
rewrite prod_cons.
  rewrite /= exp0. smt(@G @GP @List).
qed.

(* General cons lemma for prodEx *)
lemma prodExConsGeneral (g : group) (gs : group list) (e : exp)
(es : exp list) :
  prodEx (g :: gs) (e :: es) = g ^ e * prodEx gs es.
proof.
  rewrite /prodEx /ex.
  rewrite zip_cons map_cons prod_cons.
  trivial.
qed.
```

Listing A.9: Proof of Size Preservation Properties

A.7 prodEx map Proofs

```
(* Double prodEx composition *)
lemma prodExEx b e1 e2 :
    prodEx (ex b e1) e2 = prodEx b (mulv e1 e2).
proof.
  rewrite /prodEx /ex /mulv. rewrite zip_mapl. rewrite zip_mapr.
  rewrite -!map_comp. congr. rewrite /(\o) /=.
  elim: b e1 e2 => [|g_h g_t IH] [|e1_h e1_t] [|e2_h e2_t] //=.
  split. smt(@G @GP). by apply IH.
qed.



(* Ex with map and constant scaling *)
lemma ex_map_prodEx bases exps size_bases size_exps c :
    size_bases = size bases => size_exps = size exps =>
    ex (map (prodEx bases) exps) (nseq size_exps c) =
    map (prodEx (ex bases (nseq size_bases c))) (exps).
proof.
  move => h h0. apply (eq_from_nth g). smt(@List).
  move => i hi. rewrite /ex.
  rewrite (nth_map (witness, witness) g). smt(@List size_ge0).
  rewrite nth_zip. smt(size_ge0 @List). simplify.
  rewrite (nth_map witness). smt(size_ge0 @List).
  rewrite nth_nseq. smt(size_ge0 @List).

  have inner_ex:
    map (fun (i0 : group * GP.exp) => i0.`1 ^ i0.`2) (zip bases
(nseq size_bases c)) =
      ex bases (nseq size_bases c).
    rewrite /ex //.
  rewrite inner_ex.
  rewrite (nth_map (witness ) g). smt(size_ge0 @List).
  rewrite h.
  rewrite -prodExScale2. smt().
qed.

(* Map exponentiation with range *)
lemma prodExMap g (x : exp) (s e : int) :
    map (fun i => g ^ exp x i) (range s e) =
    ex (nseq (e-s) g) (map (fun i => exp x i) (range s e)).
proof.
  apply (eq_from_nth g). smt(@List).
  move => i hi. rewrite (nth_map 0). smt(@List).
  move => @/ex. rewrite (nth_map (g,zero)). smt(@List).
  simplify. smt(@List).
qed.
```

Listing A.10: Proof of Size Preservation Properties

Appendix B

Formal Proofs of Advanced Algebraic Lemmas

This appendix contains the complete EasyCrypt proofs for the advanced algebraic manipulation lemmas. These proofs are extracted directly from our `INDCCA.ec` file without modification and demonstrate the sophisticated reasoning required for oracle simulation.

B.1 Range Simplification Proofs

```
(* Simplification: prepending g to powers equals mapping from 0 *)
lemma q0_simp (sk : exp) :
      (g :: map (fun (i : int) => g ^ exp sk (i))(range 1 (q+1
))) =
      map (fun (i : int) => g ^ exp sk (i))(range 0 (q + 1 )).
 proof.
  have range_split: range 0 (q + 1) = 0 :: range 1 (q + 1).
  smt(@ G @GP @List gt0_q). rewrite range_split map_cons.
  congr. smt(@G @GP).
qed.
```

Listing B.1: Proof of Range Simplification

B.2 Ex Cons Distribution Proofs

```
(* Ex operation distributes over cons structure *)
lemma ex_cons_general:
  forall (x : group) (xs : group list) (e : ZModE.exp) (es :
ZModE.exp list),
  ex (x :: xs) (e :: es) = (x ^ e) :: ex xs es.
proof.
  move => x xs e es .
  rewrite /ex /=. trivial.
qed.
```

Listing B.2: Proof of Ex Cons Distribution

B.3 Ex Range Shift Property Proofs

```
(* Ex with range shift property *)
lemma ex_range_shift (sk : exp) :
  ex (map (fun (i : int) => g ^ exp sk i) (range 0 (q + 1)))
(nseq (q + 1) sk) =
  map (fun (i : int) => g ^ exp sk i) (range 1 (q + 2)).

proof.
  rewrite /ex.
  have Hlen: size (range 0 (q + 1)) = size (nseq (q + 1) sk) by
rewrite size_range size_nseq.
  rewrite zip_mapl -map_comp /(\o).
  apply (eq_from_nth witness).
  rewrite size_map size_zip size_range size_nseq.
  smt(@List @G @GP).
  move=> i hi.
  rewrite !(nth_map witness) //. smt(@List). smt(@List @G @GP).
  case (0 <= i < q + 1) => [valid_i|]; last smt(@List).
  simplify.
  have zip_nth: nth witness (zip (range 0 (q + 1)) (nseq (q + 1)
sk)) i = (i, sk).
  have range_i: nth witness (range 0 (q + 1)) i = i.
  rewrite nth_range //.
  have nseq_i: nth witness (nseq (q + 1) sk) i = sk.
  rewrite nth_nseq_if valid_i. smt().
  (* Zip structure property: nth element of zip equals pair of
nth elements *)
  have zip_structure: forall j, 0 <= j < min (size (range 0 (q +
1))) (size (nseq (q + 1) sk)) =>
  nth witness (zip (range 0 (q + 1)) (nseq (q + 1) sk)) j =
  (nth witness (range 0 (q + 1)) j, nth witness (nseq (q + 1) sk)
j).
  move=> j hj. smt(@List).
  rewrite zip_structure. smt(@List @G @GP).
  rewrite range_i nseq_i. smt().
  rewrite zip_nth. rewrite /= nth_range //.
  simplify. rewrite -expM. congr.
  smt(@G @GP @ZModE).
```

```
qed.
```

Listing B.3: Proof of Ex Range Shift Property

B.4 Scalev of Nseq Proofs

```
(** Scalev of nseq **)
lemma scalev_nseq (n : int) (x c : ZModE.exp) :
  scalev (nseq n x) c = nseq n (x * c).
proof.
  rewrite /scalev.
  by rewrite map_nseq.
qed.

(* Scalev of nseq 0 *)
lemma scalev_nseq_zero (n : int) (c : ZModE.exp) :
  scalev (nseq n zero) c = nseq n zero.
proof.
  rewrite scalev_nseq. smt(@ZModE).
qed.
```

Listing B.4: Proof of Scalev of Nseq

B.5 Drop-Addv Commutativity Proofs

```
(** Drop commutes with addv **)
lemma drop_addv (n : int) (u v : ZModE.exp list) :
  size u = size v => drop n (addv u v) = addv (drop n u) (drop n
v).
    proof.
    move => size_eq.
    elim: u v n size_eq => [|u_head u_tail IH] [|v_head v_tail] n
//=. smt(). smt().
    move=> size_eq.
    case: (n <= 0) => [le0_n|/ltzNge gt0_n]. smt(). smt().
qed.
```

Listing B.5: Proof of Drop-Addv Commutativity

B.6 Addv Size Inequality Proofs

```
(* Addv with size inequality a <= b*)
```

```
lemma addv_neq a b :
    size a <= size b => addv a b = addv a (take (size a) b).
    proof.
    move=> size_lt.
    rewrite /addv.
    congr. elim: a b size_lt => [|x xs IH] [|y ys] //= size_lt. smt().
qed.
```

Listing B.6: Proof of Addv Size Inequality

B.7 Drop-Sumv Commutativity Proofs

```
(* Drop commutes with sumv: dropping elements from sum equals sum of dropped elements
    This is a key lemma for vector operations in the reduction, showing that
    drop and sumv operations can be interchanged under certain conditions. *)
lemma drop_sumv (n : int) (xs : ZModE.exp list list) :
  xs <> [] =>                                (* xs is non-empty *)
  all (fun v => size v = q + 1) xs =>       (* all vectors have uniform size q+1 *)
  0 <= n <= q =>                            (* drop index is valid *)
  drop n (sumv xs) = sumv (map (drop n) xs).
proof.
  move=>  xs_nonempty all_size_eq [ge0_n len_q].
  (* Establish that xs has at least one element *)
  have  xs_largeT : 1 <= size xs by smt(@List).
  move : xs_largeT.
  (* Induction on the list xs *)
  elim: xs xs_nonempty all_size_eq => [|v vs IH] xs_nonempty all_size.
  - (* Base case: xs = [] - contradiction with non-empty assumption *)
    by [].
  (* Inductive case: xs = v :: vs *)
  have v_size: size v = q + 1 by smt(@List ). (* v has size q+1 *)
  have vs_all_size: all (fun u => size u = q + 1) vs by smt(@List). (* vs elements have size q+1 *)

  (* Rewrite using sumv_cons and map_cons *)
  rewrite sumv_cons map_cons sumv_cons.
  (* Key step: drop commutes with addv since both operands have same size *)
  rewrite drop_addv. rewrite size_sumv.  smt(@List @ZModE ).
smt().
  move=> size_ge1.
  have : 0<= size vs.  smt(@List). move => size_ge0.
  (* Case analysis on whether vs is empty or not
     Case 1 : vs has at leat 1 elements *)
  case: (1 <=size vs ) => [vs_ge1|vs_eq0]. rewrite IH.
  smt(). smt(). smt(). trivial.
  (* Case 2: vs is empty *)
  have vs_empty : vs = [] by smt(@List). rewrite vs_empty map_nil.
  (* When vs is empty, sumv vs = zerov *)
  rewrite sumv_nil.  rewrite /zerov.
  (* Use addv_neq to handle size mismatch between drop n v and
zerov *)
  rewrite (addv_neq (drop n v) (nseq (q + 1) zero)).
  rewrite size_drop. smt(). rewrite v_size size_nseq. have rev :
0 < q + 1 -n  by smt().
  rewrite !StdOrder.IntOrder.ler_maxr. smt(). smt().  smt().
  (* Establish size equality for the final step *)
  have : size (drop n v) = q + 1 -n. rewrite size_drop. smt().
rewrite v_size. smt(@GP @G).
  move => size_dd. rewrite size_dd.  congr.
  (* Final simplification using drop_nseq_eq_take *)
  rewrite drop_nseq_eq_take. smt(). trivial.
qed.
```

Listing B.7: Proof of Drop-Sumv Commutativity

B.8 Sumv of Zero Vectors Proofs

```
(** Sum of n copies of zero vector equals zero vector
    This lemma establishes that summing any number of zero
vectors gives the zero vector,
    which is fundamental for vector arithmetic in the reduction.
**)

lemma sumv_nseq_zerov (n : int) :
     0 <= n =>
     sumv (nseq n zerov) = zerov.
proof.
         elim/natind: n => [n le0_n|n ge0_n IH].
  - (* Base case: n <= 0 *)
    move=> _.
    (* When n <= 0, nseq gives empty list, sumv of empty list is
zerov *)
     rewrite nseq0_le // sumv_nil //.

    (* Inductive case: n >= 0 *)
    move=> ge0_n1.
    rewrite nseqS // sumv_cons. rewrite IH //.
    have addv_zero_identity: addv zerov zerov = zerov.
    rewrite /addv /zerov.
    (* Prove equality by showing each element is equal *)
    apply (eq_from_nth zero).
  - rewrite !size_map !size_zip !size_nseq. smt().
  - move=> i hi.
    rewrite size_map size_zip !size_nseq in hi.
    have hi_simplified: 0 <= i < q + 1. smt(). rewrite (nth_map (zero, zero)) //.
    rewrite size_zip !size_nseq. smt().
    (* Each element of zip is (zero, zero), and zero + zero = zero *)
    rewrite nth_zip //.  rewrite /=.
    rewrite !nth_nseq_if. simplify. smt(@GP @ZModE @G).
    (* Apply the zero identity to complete the proof *)
    rewrite addv_zero_identity. smt().
qed.
```

Listing B.8: Proof of Sumv of Zero Vectors

B.9 Sumv of Singleton Zero Vectors Proofs

```
(** Sum of n singleton zero vectors equals one singleton zero vector
    This lemma shows that summing any positive number of singleton zero vectors
    [nseq 1 zero] results in a single singleton zero vector. This is important
    for handling uniform-sized vector operations in the reduction. **)
lemma sumv_nseq_zero_singleton (n : int) :
     1 <= n =>
     sumv (nseq n (nseq 1 zero)) = nseq 1 zero.
proof.
  move=> ge1_n.
  (* Induction on n *)
  elim/natind: n ge1_n => [m le0_m|m ge0_m IH] ge1_n.
  - (* Base case: m <= 0 *)
    (* Contradiction: we have 1 <= n and n = m, but m <= 0 *)
    smt().

  - (* Inductive case: m >= 0 *)
  case: (m = 0) => [m_eq_0|m_neq_0].

  + (* Case m = 0, so n = 1 *)
  rewrite m_eq_0. rewrite nseqS. smt().  rewrite nseq0. rewrite sumv_cons sumv_nil. rewrite /addv.
  rewrite /zerov.  have : (zip (nseq 1 zero) (nseq (q + 1) zero)) =   (zip (nseq 1 zero) (nseq (1) zero)).
  rewrite nseqS. smt(gt0_q).  rewrite nseq1.   rewrite zip_cons. rewrite zip_nil_l. smt().

  move =>H. rewrite H. apply (eq_from_nth zero). rewrite size_map size_zip !size_nseq.

  smt(). move=> i hi.
  rewrite size_map size_zip !size_nseq in hi.
  (* hi: 0 <= i < 1, i = 0 *)
  have i_eq_0: i = 0 by smt().
  rewrite i_eq_0.  rewrite (nth_map (zero, zero)) //.
  + (* prove index valid *)
  rewrite size_zip !size_nseq. smt(). rewrite !nth_zip //.
  rewrite /= !nth_nseq_if /=.
  smt(@ZModE). have ge1_m: 1 <= m by smt(). rewrite nseqS //.



  rewrite sumv_cons. rewrite IH. smt(). rewrite /addv.
  apply (eq_from_nth zero).
  rewrite !size_map !size_zip !size_nseq.
  smt().
  move=> i hi.
  rewrite size_map size_zip !size_nseq in hi.
  have i_eq_0: i = 0 by smt().
  rewrite i_eq_0.
  rewrite (nth_map (zero, zero)) //.
  + rewrite size_zip !size_nseq. smt().

  rewrite nth_zip //. rewrite /= !nth_nseq_if /=.
  smt(@ZModE).
qed.
```

Listing B.9: Proof of Sumv of Singleton Zero Vectors

B.10 Zip Concatenation Distributivity Proofs

```
lemma zip_cat_distributive (a1 a2 : 'a list) (b1 b2 : 'b list) :
  size a1 = size b1 =>
  zip (a1 ++ a2) (b1 ++ b2) = zip a1 b1 ++ zip a2 b2.
proof.
  move=> size_eq.
  apply (eq_from_nth witness).

  - (* size equal *)
  rewrite size_zip size_cat.  smt(@G @GP @List).
  - (* equal each element *)
    move=> i hi.
    rewrite size_zip size_cat in hi.

    (* dicuss the locate of i *)
    case: (i < size a1) => [i_lt_a1|i_ge_a1]. rewrite nth_cat.
  - smt(@List). smt(@List).

qed.
```

Listing B.10: Proof of Zip Concatenation Distributivity

B.11 Product Concatenation Proofs

```
lemma prod_cat (xs ys : group list) :
  prod (xs ++ ys) = prod xs * prod ys.
proof.
  elim: xs => [|x xs IH].
  - (* Base case: xs = [] *)
    rewrite cat0s.
    (* prod ([] ++ ys) = prod ys *)
    (* prod [] * prod ys = 1 * prod ys = prod ys *)
    rewrite prod_nil.
    smt(@G @GP).
  - (* Inductive case: xs = x :: xs *)
    rewrite cat_cons.
    (* prod ((x :: xs) ++ ys) = prod (x :: (xs ++ ys)) *)
    rewrite prod_cons.
    (* = x * prod (xs ++ ys) *)
    rewrite IH.
    (* = x * (prod xs * prod ys) *)
    rewrite prod_cons.
    (* prod (x :: xs) * prod ys = (x * prod xs) * prod ys *)
    algebra.
qed.
```

Listing B.11: Proof of Product Concatenation

B.12 prodEx Split with Last Zero Proofs

```
(** Product split with last zero exponent
    This lemma shows that when the last exponent in a list is zero, the product
    can be computed by ignoring the last base-exponent pair. This is crucial for
    oracle simulation where we need to handle cases with trailing zero exponents.

    Intuitively: prodEx([g1, g2, ..., gn], [a1, a2, ..., an-1, 0])
                = prodEx([g1, g2, ..., gn-1], [a1, a2, ..., an-1])

    This property allows the reduction to efficiently handle representations
    that have zero coefficients in the last position. **)
lemma prodEx_split_last_zero (bases : group list) (exps : ZModE.exp list) :
  size bases = size exps =>
  0 <size exps  =>
  nth witness exps (size exps - 1) = zero =>
  prodEx bases exps =
  prodEx (take (size bases - 1) bases) (take (size exps - 1) exps).
proof.
  move=> size_eq size_gt0 last_zero.

  (* split exps into prefix and last element*)
  have exps_split: exps = take (size exps - 1) exps ++ [zero].
  rewrite -(cat_take_drop (size exps - 1)).
  have drop_last: drop (size exps - 1) exps = [nth witness exps (size exps - 1)]. smt( @List). smt(@List).


  (* split bases into prefix and last element *)
  have bases_split: bases = take (size bases - 1) bases ++ [nth witness bases (size bases - 1)].
  rewrite -(cat_take_drop (size bases - 1)).
  have drop_last_base: drop (size bases - 1) bases = [nth witness bases (size bases - 1)]. smt(@List). smt(@List).
  have tran : prodEx bases exps = prodEx (take (size bases - 1) bases ++
             [nth witness bases (size bases - 1)]) (take (size exps - 1) exps ++ [zero]). smt(). rewrite tran.
  rewrite /prodEx /ex. rewrite -bases_split.
            (* split zip and map *)
  have zip_split: zip bases (take (size exps - 1) exps ++ [zero]) =
               zip (take (size bases - 1) bases) (take (size exps - 1) exps)
```

```
                ++ zip (drop (size bases - 1) bases) [zero].

  rewrite -zip_cat_distributive. rewrite !size_take.

  smt(). smt(). simplify. smt(). smt(@List). rewrite zip_split.
rewrite map_cat. rewrite prod_cat.
  have : prod (map (fun (i : group * GP.exp) => i.`1 ^ i.`2)
               (zip (drop (size bases - 1) bases) [zero])) = e.
  have drop_single: drop (size bases - 1) bases = [nth witness
bases (size bases - 1)].
  smt(@List). rewrite drop_single.
  simplify. search exp.

  have exp_zero: (nth witness bases (size bases - 1)) ^ zero = e.
smt(@G @GP).
  rewrite exp_zero. smt(@G @GP). smt(@G @GP).

qed.
```

Listing B.12: Proof of prodEx Split with Last Zero

B.13 Behead-Drop Equivalence Proofs

```
(* my version of relation between behead and drop*)
lemma my_behead_drop (base : group list) : behead base = drop 1
base.
    case: base => [|x xs].
  - (* base = [] *)
    rewrite behead_nil -drop0.
    by [].
  - (* base = x :: xs *)
    rewrite behead_cons drop_cons. simplify.
    smt(@G @GP @List).
qed.
```

Listing B.13: Proof of Behead-Drop Equivalence

B.14 Addv Nth Distribution Proofs

```
(* distributive of nth and addv*)
lemma addv_nth (a b : ZModE.exp list) (pos : int) :
  size a = size b =>
  0 <= pos < size a =>
  nth witness (addv a b) pos = nth witness a pos + nth witness b
pos.
```

```
proof.
  move=> size_eq pos_valid.
  rewrite /addv. rewrite(nth_map witness witness). smt(@List).
smt( @List).
qed.
```

Listing B.14: Proof of Addv Nth Distribution

Bibliography

[1] Setareh Sharifian and Reihaneh Safavi-Naini. Information-theoretic key encapsulation and its application to secure communication. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 2393–2398, 2021. URL https://arxiv.org/pdf/2102.02243. [Cited on page 2.]

[2] Sven Schäge. New limits of provable security and applications to ElGamal encryption. Cryptology ePrint Archive, Paper 2024/795, 2024. URL https://eprint.iacr.org/2024/795. [Cited on page 2.]

[3] Atul Pandey, Indivar Gupta, and Dhiraj Kumar Singh. Improved cryptanalysis of a elgamal cryptosystem based on matrices over group rings. *Journal of Mathematical Cryptology*, 15(1):266–279, 2021. doi: doi:10.1515/jmc-2019-0054. URL https://doi.org/10.1515/jmc-2019-0054. [Cited on page 2.]

[4] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, volume 10992 of *Lecture Notes in Computer Science*, pages 33–62, Santa Barbara, United States, 2018. Springer. URL https://inria.hal.science/hal-01870015. [Cited on pages 2, 3, 8, 13, 14, 15, 17, 21, 45, 46, and 49.]

[5] Balthazar Bauer, Georg Fuchsbauer, and Antoine Plouviez. The one-more discrete logarithm assumption in the generic group model. Cryptology ePrint Archive, Paper 2021/866, 2021. URL https://eprint.iacr.org/2021/866. [Cited on page 2.]

[6] Manuel Fersch. *The provable security of elgamal-type signature schemes*. PhD thesis, Dissertation, Bochum, Ruhr-Universität Bochum, 2018, 2018. [Cited on page 2.]

[7] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 71–90. Springer, 2011. URL https://www.iacr.org/archive/crypto2011/68410071/68410071.pdf. [Cited on pages 2, 11, and 17.]

[8] Cong Zhang, Hong-Sheng Zhou, and Jonathan Katz. An analysis of the algebraic group model. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022*, pages 310–322, Cham, 2022. Springer Nature Switzerland. ISBN 978-3-031-22972-5. [Cited on pages 7 and 13.]

[9] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction*, pages 1–6, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31113-0. [Cited on pages 11 and 12.]

[10] João Bártolo Almeida, Sergio A. Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Victor Laporte, Jean-Christophe Léchenet, Chengxiao Low, Tiago Oliveira, Henrique Pacheco, Marco A. B. de Almeida Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying kyber episode v: Machine-checked ind-cca security and correctness of ml-kem in easycrypt. Cryptology ePrint Archive, Paper 2024/843, 2024. URL `https://eprint.iacr.org/2024/843`. [Cited on pages 11 and 17.]

[11] Alessandro Faonio, Daniele Fiore, Markulf Kohlweiss, Luca Russo, and Michał Zając. From polynomial iop and commitments to non-malleable zksnarks. Cryptology ePrint Archive, Paper 2023/569, 2023. URL `https://eprint.iacr.org/2023/569`. [Cited on page 18.]

[12] David Galindo, Johannes Großschädl, Zhe Liu, and Praveen Kumar Vadnala. Implementation of a leakage-resilient elgamal key encapsulation mechanism. *Journal of Cryptographic Engineering*, 6(3):229–240, 2016. doi: 10.1007/s13389-016-0124-5. URL `https://eprint.iacr.org/2014/835.pdf`. [Cited on page 18.]

[13] Yuji HASHIMOTO, Koji NUIDA, and Goichiro Hanaoka. Tight security of twin-dh hashed elgamal kem in multi-user setting. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E105.A, 08 2021. doi: 10.1587/transfun.2021CIP0008. [Cited on page 18.]

[14] Eike Kiltz and Krzysztof Pietrzak. Leakage resilient elgamal encryption. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*, volume 6477 of *Lecture Notes in Computer Science*, pages 595–612. Springer, 2010. doi: 10.1007/978-3-642-17373-8_34. URL `https://www.iacr.org/archive/asiacrypt2010/6477599/6477599.pdf`. [Cited on page 18.]

[15] Joohee Lee, Jihoon Kwon, and Ji Sun Shin. Efficient continuous key agreement with reduced bandwidth from a decomposable kem. *IEEE Access*, 11:33224–33235, 2023. doi: 10.1109/ACCESS.2023.3262809. [Cited on page 18.]

[16] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014. ISBN 1466570261. [Cited on page 21.]

[17] Princeton University. Lecture 22: The cramer–shoup cryptosystem (notes on cca1/cca2 definitions), 2007. URL `https://www.cs.princeton.edu/courses/archive/fall07/cos433/lec22.pdf`. Course notes for COS 433: Cryptography. [Cited on page 21.]

[18] Emmanuel Bresson, Yassine Lakhnech, Laurent Mazaré, and Bogdan Warinschi. A generalization of ddh with applications to protocol analysis and computational soundness. In *Annual International Cryptology Conference*, pages 482–499. Springer, 2007. [Cited on page 24.]

[19] Syh-Yuan Tan, Ioannis Sfyrakis, and Thomas Gross. A q-SDH-based graph signature scheme on full-domain messages with efficient protocols. Cryptology ePrint Archive, Paper 2020/1403, 2020. URL `https://eprint.iacr.org/2020/1403`. [Cited on page 50.]

[20] Tuy Tan Nguyen, Tram Thi Bao Nguyen, and Hanho Lee. An analysis of hardware design of mlwe-based public-key encryption and key-establishment algorithms. *Electronics*, 11(6):891, 2022. [Cited on page 50.]

[21] Luca De Feo. Mathematics of isogeny based cryptography. *arXiv preprint arXiv:1711.04062*, 2017. [Cited on page 50.]