

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

November 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.

November (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

Formal verification plays a crucial role in validating the security and correctness of cryptographic constructions. However, complex reductions and game based proofs also make mechanized verification a challenge.

This project represents formalization of the bilateral equivalence between the Indistinguishability under Non-Adaptive Chosen-Ciphertext Attack(IND-CCA1) security of ElGamal Key Encapsulation Mechanism(KEM) and the q -determined Diffie-Hellman(q -DDH) assumption in the Algebraic Group Model(AGM).

Using the EasyCrypt proof assistant, we formally prove both directions of this equivalence relation by mechanizing algebraic reasoning and oracle simulations and linear algebra operations.

This thesis indicates the feasibility of machine-checked complex cryptographic proofs within the framework of an algebraic group model (AGM) and documents the technical methods and practical lessons I learned throughout the formalization process.

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

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Statement	2
1.3	Main Contributions	4
1.4	Technical Overview	5
2	Background	7
2.1	Cryptographic Computational Models	7
2.1.1	The Standard Model	7
2.1.2	Idealized Models	7
2.1.3	Algebraic Group Model	8
2.2	Public key cryptography	9
2.2.1	Key Encapsulation Mechanisms	9
2.2.2	Indistinguishability under non-adaptive chosen-ciphertext attack (IND-CCA1 Security)	9
2.3	The ElGamal Cryptosystem	10
2.4	IND-CCA1 Security Game for ElGamal.	10
2.5	q-Decisional Diffie-Hellman (q-DDH) Problem.	11
2.6	The Bilateral Equivalence	12
2.7	Formal Verification	13
2.7.1	General Formal Verification Process	13
2.8	The EasyCrypt Verification Framework	13
2.8.1	Core Concepts	14
3	Related Work	17
3.1	Formal Verification of Cryptographic Proofs using EasyCrypt	17
3.2	the Algebraic Group Model in cryptography	18
3.3	ElGamal Encryption and Key Encapsulation Mechanism	19
3.3.1	Variants and Extensions of ElGamal-Based KEMs	19

4	q-DDH Correctness and IND-CCA1 Security	21
4.1	ElGamal Key Encapsulation Mechanism	21
4.1.1	Algorithmic Specification	21
4.1.2	EasyCrypt Implementation	22
4.2	IND-CCA1 Security Model	23
4.2.1	Relationship to Other Security Notions	23
4.2.2	Algebraic Oracle Implementation	23
4.3	q-DDH Assumption	25
4.3.1	EasyCrypt Formalization of q-DDH	26
4.4	Mathematical Foundations	28
4.4.1	Core Linear Algebra Components	28
4.4.2	Key Algebraic Properties and Lemmas	31
4.4.3	Advanced Algebraic Manipulation Lemmas	34
5	Evaluation : Formalizing the Bilateral Reduction	37
5.1	Overview of bidirectional reduction	37
5.1.1	Forward Reduction: A_from_INDCCA1.	37
5.1.2	Backward Reduction: B_from_qDDH.	38
5.2	Forward Reduction: IND-CCA1 to q-DDH	38
5.2.1	Module Structure and State Variables	38
5.2.2	Internal Oracle Simulation	39
5.2.3	Main Reduction Procedure	40
5.3	Backward Reduction: q-DDH to IND-CCA1	42
5.3.1	Module Structure and Scout Phase	42
5.3.2	Distinguish Phase	43
5.4	Main Theorem: Bilateral Equivalence	44
5.4.1	Forward Direction.	44
5.4.2	Backward Direction.	44
5.5	Overall Analysis of INDCCA1_ElGamal_Implies_qDDH Code	45
5.5.1	Key components of Proof Structure	45
5.6	Overall Analysis of qDDH_Implies_INDCCA1_ElGamal Code	45
5.6.1	Key components of Proof Structure	45
5.7	Case Study 1: Mechanizing Algebraic Manipulations and Self evaluation	47
5.7.1	The Proof Goal	47
5.7.2	Proof Structure	47
5.8	Case Study 2: ProdEx Split Last Zero Lemma	58
5.8.1	The Lemma Statement	59
5.8.2	Proof Structure	59
5.9	Lessons Learned from case study and formaliztion procedure	63
5.10	Main Results	63
5.10.1	Lessons Learned from case study and formaliztion procedure	64
5.10.2	enhance the rigorous and the readability of algebraic reduction . .	64
5.10.3	show the power of Algebraic Group Model	64

5.10.4 Formalizing Algebraic Adversaries in EasyCrypt	65
6 Conclusion and Future work	67
6.1 Conclusion	67
6.1.1 Scope and Limitations	67
6.1.2 Challenges and Effort in Mechanizing AGM-Based Proofs	68
6.2 Future Directions	68
A Formal Proofs of Key Lemmas	71
A.1 Distributivity and Scaling Proofs	71
A.2 Zero Vector and Identity Proofs	73
A.3 Shift-Truncate Operation Proof	74
A.4 Vector Addition Properties	75
A.5 Size Preservation Proofs	76
A.6 prodEx cons Proofs	77
A.7 prodEx sizele Proofs	77
A.8 sumv cons Proofs	78
A.9 prodEx map Proofs	78
B Formal Proofs of Advanced Algebraic Lemmas	81
B.1 Range Simplification Proofs	81
B.2 Ex Cons Distribution Proofs	81
B.3 Ex Range Shift Property Proofs	82
B.4 Scalev of Nseq Proofs	83
B.5 Drop-Addv Commutativity Proofs	83
B.6 Addv Size Inequality Proofs	83
B.7 Drop-Sumv Commutativity Proofs	84
B.8 Sumv of Zero Vectors Proofs	85
B.9 Sumv of Singleton Zero Vectors Proofs	86
B.10 Zip Concatenation Distributivity Proofs	87
B.11 Product Concatenation Proofs	88
B.12 prodEx Split with Last Zero Proofs	89
B.13 Behead-Drop Equivalence Proofs	90
B.14 Addv Nth Distribution Proofs	90
Bibliography	93

Introduction

When we try to research on the topic about cryptography, a fundamental question is: how can we tell whether a cryptographic construction is “really secure” against real-world attackers? In particular, there are two routes. One is to experiments to find attacks and counterexamples. The other is to provide security proofs that can be independently reproduced and proved. Our project follows the second approach: we translate a bidirectional security into executable games and reductions, and use a proof assistant to automatically check each step of the reasoning.

In particular, we study a bilateral equivalence established in [1], between the ElGamal Key Encapsulation Mechanism (KEM) and its security in the attack model that allows decryption queries before the challenge is issued, denoted IND-CCA1 and a algebraic hardness assumption, the q -Decisional Diffie–Hellman problem (q -DDH) in the Algebraic Group Model (AGM). All the terms mentioned above is detailed explained in the Section 2.2.

The central claim of our work is that, under accurate and suitable formalisation, is it possible to prove that breaking the IND-CCA1 security of the ElGamal KEM is basically the same task as distinguishing the two distributions in the q -DDH problem for the same parameters.

1.1 Motivation

In cryptographic security analysis, the *computational model* provides a formal framework which defines the adversaries’ capabilities and constraints of adversaries when attacking a cryptographic scheme to determine the security of a cryptosystem.

In particular, the *standard model* (explained in subsection 2.1.1) imposes minimal restrictions which refers to the computational framework where the adversary is assumed to be

1 Introduction

bounded only by the time and computational power, while *idealized models*(explained in subsection 2.1.2) such as the random oracle model or generic group model impose additional constraints, where the adversaries are assumed to only accessible to a random encoding of group elements, rather than the concrete and efficiently computable representations. The Algebraic Group Model (AGM)(explained in subsection 2.1.3), introduced by Fuchsbauer, Kiltz, and Loss[1], provides a middle ground between the standard model and idealized models mentioned above. The full explanation of these models lies in the section 2.1.

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[2]. Recent work has highlights that many schemes, including digital signatures [3] 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.

In addition, as the main foundation of our work, Fuchsbauer, Kiltz, and Loss [1] established a bilateral reduction between the IND-CCA1 security of ElGamal-based KEM and the q-DDH assumption, providing a paper proof of this equivalence. Based on their theoretical framework, our work aims to provide a machine-checked verification of this reduction using the EasyCrypt. We want our formalization translates their high-level cryptographic arguments into entire mechanized proofs, making explicit all reasoning that is implicit in the original paper proof and ensuring correctness.

The proofs are mechanized in the EasyCrypt proof assistant[4], 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. The detailed easycrypt introduction is in the Section 2.8

1.2 Problem Statement

The central question we address is:

Can we introduce a formal and bidirectional methodology, to prove the equiva-

lence between breaking IND-CCA1 security of ElGamal-based KEM and solving the q-DDH problem, with both reductions machine-checked?

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

1. Reduction Construction:

The reduction from [1] provides bidirectional transformations between IND-CCA1 adversaries and q-DDH distinguishers. The first important part of our work is to formalize both reductions using EasyCrypt. Precisely, the reduction is required to convert any IND-CCA1 adversary against ElGamal-based KEM into a distinguisher for q-DDH, and conversely, it should also established an approach to transform a q-DDH distinguisher into IND-CCA1 adversary. This reduction embeds the q-DDH instance into the KEM security game, relating the adversarial advantages in both problems .

2. Oracle Simulation:

An important aspect of the reduction is the decryption oracle simulator, the oracle is required to consistently answer the queries from adversaries while avoiding information leakage that would trivialize the q-DDH challenge. In our formalization, we should mechanize how the simulators handles decryption queries in a way that is consistent with the q-DDH challenge while preserving the hidden exponent. We should follow the AGM setting [5], where adversaries should provide linear representations of ciphertexts, allowing the simulator to compute the responses based on this representations. Our easycrypt formaliztion should verify this simulation correctly preserve consistency without information leakage.

3. Formal Verification:

Beyond the high-level reduction, a key challenge lies in mechanizing the proof within EasyCrypt, which requires converting the theoretical reduction proof from A.3 in [1] into a entirely machine-checked one, it requires us to making explicit all reasoning that is left implicit in the paper proof. For instance, we need to encode all probabilistic reasoning, hybrid argument and algebraic manipulation correctly. In contrast to the traditional paper proofs where intuition often fill in the gaps, mechanized verification requires complete formal rigor for each step such as constraints on length and the randomness-preserving transformations. This part is the most significant part of our work and the main challenge is reconciling the intuitive algebraic reasoning with the strict requirements of EasyCrypt.

4. Learning EasyCrypt:

An additional main challenge is mastering the EasyCrypt proof assistant itself, which requires understanding both its proof logic (pRHL) and its tactical language. At the initial stage of this project, EasyCrypt was an unfamiliar tool for me. Mastering the EasyCrypt requires extensive reading of the reference manual

and sample code, repeated experimentation with different proof strategies, and continuous learning and adjustment during the ongoing formalization process.

1.3 Main Contributions

This thesis provides an entire machine-checked formalization of the bidirectional reduction between IND-CCA1 security of ElGamal KEM and q-DDH from [1]. The mechanization process handles the key challenges in formal verification and indicates the feasibility of machine checking the AGM based cryptographic proofs. The main contributions are:

1. Complete Bilateral Reduction Formalization:

We provide a machine-checked formalization in EasyCrypt of both directions of the reduction from [1], to be more specific, we try to construct the computational equivalence by the following code format:

We formalize the `qDDH_Implies_INDCCA1_ElGamal` and `INDCCA1_ElGamal_Implies_qDDH` lemma, verifying the equivalence between q-DDH hardness and IND-CCA1 security from [1]:

$$\Pr[\text{IND_CCA1_P}(\text{ElGamal}, B).\text{main}() : \text{res}] = \Pr[\text{QDDH}(A_{\text{from_INDCCA1}}(B)).\text{main}() : \text{res}] \quad (1.1)$$

The formalization we construct verify the correctness of the relationship claimed in the original paper [1]

2. Oracle Simulation Technique:

We formalized the decryption oracle simulation technique from [1] using EasyCrypt, which leverages algebraic representations to answer the adversaries' queries. In doing so, we handle the challenge in encoding the extraction of representation while ensuring the simulator preserving the consistency invariants.

3. Linear Algebra Manipulation framework:

To make the formalization more efficient, we develop a framework consists of a collection of EasyCrypt operators and corresponding lemmas 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 Machine-checked proof:

The entire proof of theorem is mechanized using the EasyCrypt, consists of over 2000 lines of code, including lemmas, definitions and proof scripts. Our work shows

that sophisticated AGM-based reductions and oracle simulations and proof process can be successfully translated into code format.

5. Insights from Formalization:

Through the entire process of formalization, I obtained detailed understanding of the reduction's structure and the challenges of formalizing AGM-based proofs. In addition, I think it will be more easy for me to using EasyCrypt to do future formalizing, which may be valuable for future efforts to mechanize similar cryptographic proofs.

1.4 Technical Overview

Figure 1.1 indicates an overview of the reduction framework from [1] that we formalize in EasyCrypt.

At a high level, our goal is to using EasyCrypt 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. Our formalization mechanizes both directions of this equivalence, verifying that adversaries can be transformed in either direction while preserving their computational advantages.

In the **first direction**, the reduction converts any adversary of ElGamal KEM under chosen-ciphertext attacks into an algorithm that handle the q-DDH challenge. In the **reverse direction**, the reduction transforms an IND-CCA1 adversary from any q-DDH distinguisher by embedding the KEM challenge into a q-DDH challenge.

A key technical point is the leverage of **algebraic tracking**, inspired by the Algebraic Group Model [1]. Where every group element generated during the simulation is represented as a linear combination of previously known ones, ensures that simulated oracles answer consistently and guarantees every adversarial action has a verifiable algebraic explanation.

Finally, the entire framework, containing oracle simulations, linear algebra operators (`prodEx`, `addv`, `scalev`, `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*.

By completing all the preparation work we mentioned above, making our machined-checked proof stage become efficiency and easy to prove the equivalence we mentioned: equation 1.1

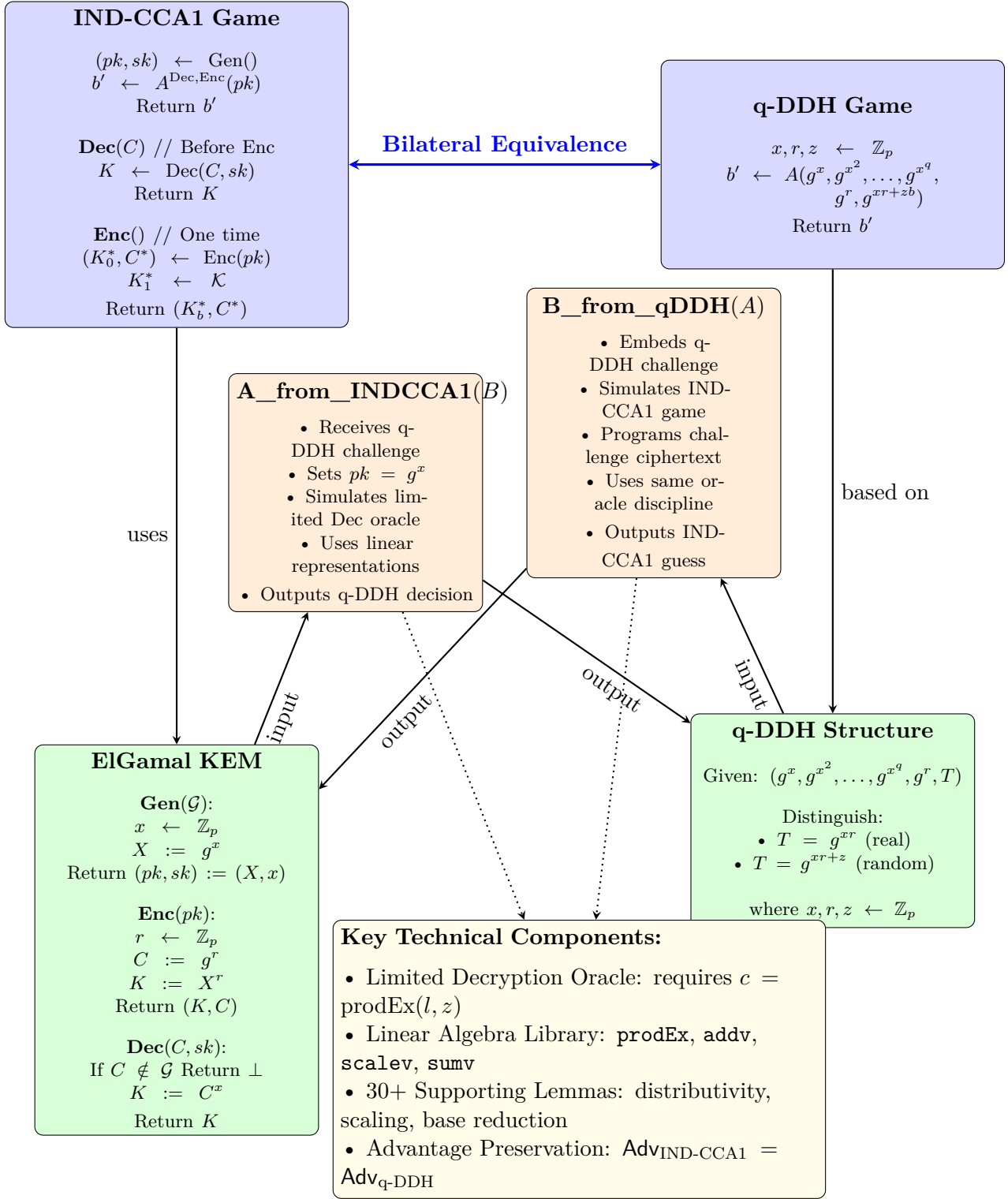


Figure 1.1: Detailed bilateral reduction between IND-CCA1 security of ElGamal-based KEM and q-DDH hardness, showing the complete game structures and algorithmic details.

Background

This chapter shows the cryptographic foundations and formal verification framework of this work. We will first show an overview of cryptographic computational models, present the specific security definitions and cryptographic primitives, and introduce the EasyCrypt proof assistant used for mechanization.

2.1 Cryptographic Computational Models

Here are the explanation and the instances of some important models:

2.1.1 The Standard Model

The *standard model* is the most fundamental approach, imposing minimal constraints which refers to the computational framework where the adversary is assumed to be bounded only by the computational feasibility. In particular, adversaries of standard model are modeled as probabilistic polynomial-time (PPT) algorithms that can perform any computation executed within polynomial time [6]. In other words, the standard model provides the strongest guarantees due to there is no idealized assumptions about cryptographic primitives or mathematical structures. As a result, it is challenging to provide formal proof using standard model for schemes like ElGamal [7].

2.1.2 Idealized Models

To enable security proofs for practical schemes, cryptographers have developed various *idealized models* that provides additional structure on adversarial capabilities as ideal objects, for instance:

- **Random Oracle Model (ROM)** [8]: Hash functions are modeled as truly random functions accessible only through oracle queries. This model enable us to

2 Background

simplify the proofs of many practical schemes, but introduces assumptions that do not hold in real hash function implementations.

- **Generic Group Model (GGM)** [9]: In this model, adversaries can only interact with group elements through a generic interface that does not reveal any information about the specific representation of the group elements. While this model captures certain intuitive assumptions about computational hardness, it is generally often considered overly restrictive, in real world, real adversaries may exploit specific structure of group representations to attack.

2.1.3 Algebraic Group Model

The Algebraic Group Model (AGM), introduced by Fuchsbauer, Kiltz, and Loss [1], provides a framework for analyzing cryptographic schemes by restricting adversaries to be "algebraic." As we mentioned in the section 1.1, this model provides a intermediate ground between two models mentioned above.

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 [5].

This model indicates the intuition that real cryptographic algorithms operate through concrete algebraic operations, rather than arbitrary black-box calculations, 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 $\mathbf{z} = (z_1, \dots, z_k) \in \mathbb{Z}_p^k$ such that:

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

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

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. As a result, it is crucial for our work as it allows to construct an oracle simulator which can preserve the consistency when answering the adversaries without access to the secret key, enables the reduction can be fully mechanized in the EasyCrypt.

2.2 Public key cryptography

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[10].

2.2.1 Key Encapsulation Mechanisms

A Key Encapsulation Mechanism consists of three algorithms [11]:

- $\text{Gen}(1^\lambda) \rightarrow (pk, sk)$: Generates a public key pk and secret key sk pair.
- $\text{Enc}(pk) \rightarrow (K, C)$: Takes a public key and outputs a randomly chosen session key K and encapsulation C .
- $\text{Dec}(C, sk) \rightarrow K$: Decrypts a encapsulation C using secret key sk to recover the session key K or fails.

2.2.2 Indistinguishability under non-adaptive chosen-ciphertext attack (IND-CCA1 Security)

To preserve secure against active attackers, that 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.

Formally, the IND-CCA1 security game $\text{ind-cca1}_{\text{KEM}, par, b}^A$ for a $\text{KEM} = (\text{Gen}, \text{Enc}, \text{Dec})$ with parameters par and adversary \mathcal{A} proceeds in three phases [1]:

1. **Setup Phase:** The challenger generates $(pk, sk) \xleftarrow{\$} \text{Gen}(par)$ and provides pk to the adversary.
2. **Query Phase:** The adversary \mathcal{A} makes queries to the decryption oracle $\text{Dec}(C)$, which computes $K \xleftarrow{\$} \text{Dec}(C, sk)$ and returns K (or \perp if C is invalid). Note that this oracle is only available *before* the challenge phase.
3. **Challenge Phase:** The adversary invokes the encryption oracle $\text{Enc}()$ exactly once. The challenger will:
 - Computes $(K_0^*, C^*) \xleftarrow{\$} \text{Enc}(pk)$ (real session key)
 - Samples $K_1^* \xleftarrow{\$} \mathcal{K}$ (random key from key space)
 - Returns (K_b^*, C^*) where b is the secret challenge bit
4. **Output:** The adversary outputs a guess b' and wins if $b' = b$.

2 Background

The adversary's *advantage* is defined as:

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

2.3 The ElGamal Cryptosystem

The ElGamal cryptosystem is a fundamental establishment of public-key cryptography [12]. The basic ElGamal operates in a cyclic group \mathcal{G} of prime order p with generator g . The scheme generates a key pair by selecting a random secret exponent $x \leftarrow \mathbb{Z}_p$ and computing the public key as $X = g^x$. Encryption of a message m involves choosing a random $r \leftarrow \mathbb{Z}_p$ and computing the ciphertext as $(C_1, C_2) = (g^r, m \cdot X^r)$. Decryption recovers $m = C_2 / C_1^x$.

The ElGamal-based KEM, derived from the classical ElGamal construction, bases its security on the discrete logarithm problem in cyclic groups[7]. While ElGamal in its original public-key encryption form achieves only IND-CPA security [13], its KEM formulation enables stronger guarantees when analyzed under appropriate computational assumptions[7]. It can be divided into three phases [1]:

- $\text{Gen}(\mathcal{G})$: Choose $x \leftarrow \mathbb{Z}_p$ uniformly at random. Set $X := g^x$. Return $(pk, sk) := (X, x)$.
- $\text{Enc}(pk)$: Choose $r \leftarrow \mathbb{Z}_p$ uniformly at random. Compute $C := g^r$ and $K := X^r = g^{xr}$. Return (K, C) .
- $\text{Dec}(C, sk)$: If $C \notin \mathcal{G}$ return \perp . Otherwise compute $K := C^x$ and return K .

Correctness follows from $C^x = (g^r)^x = g^{rx} = g^{xr} = X^r = K$.

2.4 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 $\text{Gen}(par) \rightarrow (pk, sk)$ 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 an oracle $\text{Dec}(C, sk)$ 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 $\text{Enc}(pk) \rightarrow (K_0^*, C^*)$ 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 from paper[1] is detailed in Figure 2.1.

2.5 q -Decisional Diffie-Hellman (q -DDH) Problem.

IND-CCA1 _{EG, G} ^A Security Game for ElGamal
Algorithm: 00 $x \leftarrow \mathbb{Z}_p$ 01 $X := g^x$ 02 $b' \leftarrow A^{\text{Dec}, \text{Enc}}(X)$ 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: $\text{Adv}_{\text{ElGamal}}^{\text{IND-CCA1}}(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.

2.5 q -Decisional Diffie-Hellman (q -DDH) Problem.

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.

The q -DDH problem will ask one to distinguish between two distributions over group elements. By given a tuple $(g^x, g^{x^2}, \dots, g^{x^q}, g^r, T)$ where $x, r \leftarrow \mathbb{Z}_p$ are random, the

2 Background

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 from paper[1] is detailed below in Figure 2.2.

q-DDH_{G,q}^A 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 $b' \leftarrow A(g^x, g^{x^2}, \dots, g^{x^q}, g^r, T_b)$	
04 Return b'	
Challenge Structure Given to Distinguisher A:	
• Powers of x: $(g^x, g^{x^2}, g^{x^3}, \dots, g^{x^q})$	
• 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:	
$\text{Adv}_G^{\text{q-DDH}}(A) = \Pr[A(\mathcal{D}_0) = 1] - \Pr[A(\mathcal{D}_1) = 1] $	
where $\mathcal{D}_0 = (g^x, g^{x^2}, \dots, g^{x^q}, g^r, g^{xr})$	
and $\mathcal{D}_1 = (g^x, g^{x^2}, \dots, g^{x^q}, g^r, g^{xr+z})$	
Hardness Assumption:	
For any probabilistic polynomial-time algorithm A :	
$\text{Adv}_G^{\text{q-DDH}}(A) \leq \text{negl}(\lambda)$	
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 $(g^x, \dots, g^{x^q}, g^r, T)$, 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 .

2.6 The Bilateral Equivalence

Fuchsbauer, Kiltz, and Loss [1] constructs a bilateral equivalence between the IND-CCA1 security of ElGamal-based KEM and the q-DDH assumption in the Algebraic Group Model. This equivalence indicates that breaking the IND-CCA1 security of ElGamal KEM is computationally equivalent to handling the q-DDH problem, where adversaries are restricted to be algebraic.

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. Additionally, q-DDH problem asks that, for any adversary obtains certain group elements corresponding to powers of a secret exponent, still unable to distinguish whether a given element corresponds to a true Diffie–Hellman product or to random generated.

In the algebraic group setting, we are required trace adversarial group elements through their linear relations to known generators, which allows for constructing verifiable oracle responses. This technique enable us mechanized correspondence between the two security game frameworks.

2.7 Formal Verification

Formal verification is the leverage of mathematical mechanisms to ensure that a design conforms to some precisely expressed notion of functional correctness. Concretely, given (1) a model of a system, (2) a description of the environment that the system is supposed to operate in, and (3) properties that the system is intended to fulfill, formal verification can be used to search for scenarios that violate the properties and proving that the properties always hold when no such violations exist [14].

2.7.1 General Formal Verification Process

Recent progress in computer-aided cryptography highlights that a general formal verification process can be applied to real-world cryptographic systems. Given a scheme’s specification and (hand-written) security proof, this process allows us to formally verify both the scheme and its implementations. While the process does not depend on any specific tools, the choice of tool may significantly impact the difficulty, depending on factors such as the properties to verify or the type of proof used [15]. In our works, we select EasyCrypt to formalize the equivalence. We mainly focus on formalizing the bilateral reduction between IND-CCA1 and q-DDH from [1], and verifying that the reduction correctly establishes the equivalence between the two problems using the formal verification.

2.8 The EasyCrypt Verification Framework

EasyCrypt is a formal verification framework specifically developed for verifying the security of cryptographic designs [15]. In particular, it specializes in formalizing and verifying security proofs for cryptographic constructions through rigorous mathematical reasoning. The core of EasyCrypt is its Probabilistic Relational Hoare Logic (pRHL), which is used extensively through our bilateral equivalence formalization to build exact correspondences while preserving control over probabilistic reasoning [16].

While EasyCrypt’s standard reasoning strategies handle the majority of our proof obligations, certain technical steps in our bilateral reduction requires more complex approaches

2 Background

that require global program analysis and explicit management of algebraic relationships including randomness-preserving transformations and complex oracle simulations. EasyCrypt addresses these requirements through specialized reasoning techniques and transformation rules.

In our work, the formal verification process includes over 2000 lines of verified EasyCrypt code, representing a substantial mechanization of AGM-style. The formalisation not only enhances confidence in the soundness of our proof but also lays the groundwork for future developments in my future study in mechanised cryptography.

2.8.1 Core Concepts

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

Probabilistic Relational Hoare Logic (pRHL). EasyCrypt’s core aspect is probabilistic Relational Hoare Logic (pRHL), which enables reasoning about relationships between probabilistic programs [16]. In our context, pRHL allows us to establish probability equalities between different cryptographic games, such as:

$$\Pr[\text{IND_CCA1_P}(\text{ElGamal}, B).\text{main}() : \text{res}] = \Pr[\text{QDDH}(A_{\text{from_INDCCA1}}(B)).\text{main}() : \text{res}]$$

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:

```
1 rnd (fun k' => loge k' - sk0{1} * y{1})
2   (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

2.8 The EasyCrypt Verification Framework

- **State Management:** Tracking the evolution of oracle state across game executions
- **Query Validation:** Verifying that adversarial queries satisfy the required algebraic constraints

Related Work

Our work lies in the intersection of three research areas: the Algebraic Group Model in cryptography, formal verification of cryptographic proofs using EasyCrypt, and KEM Security and ElGamal. In this chapter we review relevant work in each area and position our contributions.

3.1 Formal Verification of Cryptographic Proofs using EasyCrypt

Formal verification of cryptographic security arguments has grown significantly in recent years, driven by the practical needs for high-assurance cryptography.

EasyCrypt Framework. EasyCrypt [4] is a proof assistant specialized for reasoning about cryptographic security proofs using game-based arguments. Its core logic, Probabilistic Relational Hoare Logic (pRHL) [16], enables precise reasoning about probabilistic equivalences between games.

Major applications of EasyCrypt include:

EasyCrypt has been successfully applied to verify post-quantum schemes based on lattice assumptions. Hülsing, Meijers, and Strub [17] formalized the IND-CPA security and δ -correctness of Saber’s public-key encryption scheme, establishing the properties required to transform it into an IND-CCA2 secure KEM. Their methodology introduce hand-written proofs first, then mechanizing them in EasyCrypt. In our work, we inspired from the work based on the original paper proof from paper [1], and provides the formal verification.

More recently, Almeida et al. [18] provided a complete formalization of ML-KEM (Kyber), containing IND-CCA security and correctness, indicating one of the largest Easy-

3 Related Work

Crypt formalizations. In particular, they introduce the formalization of formalization of the base Kyber PKE’s correctness and IND-CPA security, provides formalization of the Fujisaki-Okamoto transform in the Random Oracle Model and give proof that ML-KEM’s IND-CCA security and correctness as a KEM. This work represents one of most complete formalization to our knowledge, providing from high-level security theorems to verified implementations, this work indicates the capability of the easycrypt for end-to-end verification.

Additionally, Barbosa et al. [19] provide a formally verified tight security proof for SPHINCS+ (standardized by NIST as SLH-DSA), a hash-based post-quantum signature scheme. This work handles a critical gap: the original tight security proofs for SPHINCS+ contained flaws discovered by Kudinov et al. [20]. The formalization reconstructs the corrected proof of Hülsing and Kudinov [21] in a modular fashion within EasyCrypt. Notably, they formalizes a complex argument with four different security games simultaneously, a reasoning pattern not previously addressed in EasyCrypt (to the authors’ knowledge). To handle this challenge, they develop a general formal verification framework for similar multi-game arguments.

Relationship to Our Work. These three related work all show the potential and capability of using EasyCrypt to formalize and verify the paper proof, although the formalization challenges is quite similar: substantial infrastructure development (lattice/hash operations vs. algebraic representations), it require us to careful management of size constraints and index bounds, and making explicit the reasoning steps that are often left implicit in paper proofs. The EasyCrypt’s pRHL and game transformation tactics make it particularly suitable for our project.

3.2 the Algebraic Group Model in cryptography

In recent years, researchers have explored intermediate computational models that enable more tractable security arguments than the standard model while maintaining more structure than fully idealized models. The Algebraic Group Model (AGM), introduced by Fuchsbauer, Kiltz, and Loss [1], has become a influential framework for analyzing discrete-logarithm-based cryptography. 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.

Key Encapsulation and Encryption. Fuchsbauer, Kiltz, and Loss [1] establish several key results in their original AGM paper. In particular, they prove (Theorem 5.2) that the IND-CCA1 security of ElGamal-based Key Encapsulation Mechanism is equivalent to the q -Decisional Diffie-Hellman (q -DDH) assumption under the AGM. This bilateral reduction demonstrates that breaking ElGamal KEM’s security is computationally equivalent to distinguishing q -DDH distributions, *To our knowledge, our work provides*

3.3 ElGamal Encryption and Key Encapsulation Mechanism

first machine-checked formalization using EasyCrypt of this bilateral reduction (Theorem 5.2 from [1]), translating the paper proof into over 2000 lines of verified EasyCrypt code and making explicit all reasoning steps.

Digital Signatures. Fuchsbauer, Plouviez, and Seurin [22] provide a complete security analysis of Schnorr signatures within AGM, establishing security under the discrete logarithm assumption rather than requiring stronger assumptions or the ROM. This demonstrates the AGM’s utility for analyzing signature schemes compared to standard model approaches.

Advanced Protocols. Beyond encryption schemes, AGM has also been applied to more complex cryptographic protocols. In 2023 work "From Polynomial IOP and Commitments to Non-malleable zkSNARKs," Faonio et al. [23] 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 compared with standard model.

Relationship to Our Work. To my knowledge, majority of AGM security arguments exists only as paper proof, which inspired us to provide a full machine-checked formalization of a specific AGM reduction using EasyCrypt. Inspired by the paper proof in [1] and all the AGM-related work mentioned above, we explored the mechanization of AGM reasoning and make them as the foundation of our work. By comparison with our work, it also illustrates that making implicit algebraic logic of paper proofs requires extensive explicit.

3.3 ElGamal Encryption and Key Encapsulation Mechanism

ElGamal encryption [12], is a foundational public-key cryptosystem based on the Diffie-Hellman problem. Its security relies on the computational hardness of the Decisional Diffie-Hellman (DDH) assumption in cyclic groups. Understanding ElGamal’s security properties and their formal verification is significantly related to our work. The detailed explanation of ElGamal lies in section 2.3.

3.3.1 Variants and Extensions of ElGamal-Based KEMs

Building on the basic ElGamal, the related work below has developed various enhanced variants handling different security requirements:

Leakage-Resilient Constructions. Kiltz and Pietrzak [24] established one of the first leakage-resilient ElGamal constructions, proving that a CCA1-secure KEM integrated with a one-time symmetric cipher can achieve full IND-CCA security even under

3 Related Work

bounded key leakage. This work indicates that ElGamal-style constructions can provide security guarantees even when side-channel attacks impose certain information about secret keys. Based on this direction, Galindo et al. [25] further introducing BEG-KEM, a leakage-resilient variant of ElGamal KEM that strengthens resistance to side-channel attacks through masking and blinding techniques. It not only achieves provable leakage security but also preserve practical efficiency on constrained embedded device.

Tight Security in Multi-User Settings. Hashimoto et al. [26] strengthen the theoretical foundations by providing tight reduction proofs for the Twin-DH hashed ElGamal KEM in multi-user environments. Their work demonstrates that ElGamal-based KEMs can achieve security with tight bounds even when many users share the same system parameters.

Hashimoto et al. [26] strengthened the theoretical foundation of ElGamal-based KEMs by giving tight reduction proofs for the Twin-DH hashed ElGamal KEM in multi-user settings. Their work demonstrates that ElGamal-based KEMs can achieve security with tight bounds even when many users share the same system parameters.

Bandwidth-Efficient Variants. Lee et al. [27] proposed a decomposable KEM framework based on ElGamal. Their design supports continuous key agreement with lower communication costs while preserving security under the DDH assumption.

Relationship to Our Work. All in all, these works handle the practical enhancements like, leakage resilience, multi-user tightness, bandwidth efficiency, these directions are orthogonal to our formal verification of AGM-based security, which inspired us to put eyes on the ElGamal KEM as our research target. Rather than developing the scheme itself, we try to solve the foundational question of whether AGM-based proofs can be mechanically verified, illustrating the feasibility for a basic but theoretically significant construction.

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 and explained in [section 2.3](#):

Key Generation.

- 1: $\text{KeyGen}(\mathcal{G}) \rightarrow (pk, sk)$:
- 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: $\text{Enc}(pk) \rightarrow (K, C)$:
- 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: $\text{Dec}(C, sk) \rightarrow K$:
- 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:

```

1
2 (** ElGamal encryption scheme implementation **)
3 module ElGamal : Scheme = {
4   (* Key generation: sk random, pk = g^sk *)
5   proc keygen(): pk_t * sk_t = {
6     var sk;
7     sk <$ FD.dt;          (* Random secret key *)
8     return (g ^ sk, sk);  (* Public key is g^sk *)
9   }
10
11   (* Encryption: return (pk^y, g^y) where y is random *)
12   proc enc(pk:pk_t): key_t * ctxt_t = {
13     var y;
14     y <$ FD.dt;          (* Random encryption exponent *)
15     return (pk ^ y, g ^ y); (* Session key and ciphertext *)
16   }
17
18   (* Decryption: compute c^sk *)
19   proc dec(c:ctxt_t,sk:sk_t): key_t option = {
20     return Some (c ^ sk);  (* ElGamal decryption formula *)
21   }
22 }.

```

Listing 4.1: ElGamal KEM Implementation in EasyCrypt ¹

¹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L223C1-L243>

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 $\text{KEM} = (\text{KeyGen}, \text{Enc}, \text{Dec})$ and adversary \mathcal{A} , the IND-CCA1 security game proceeds as follows (detailed explained in section 2.2.2):

- 1: $(pk, sk) \leftarrow \text{KeyGen}()$ {Generate key pair}
- 2: \mathcal{A} receives pk and can query decryption oracle $\text{Dec}(sk, \cdot)$
- 3: $(K_0, C^*) \leftarrow \text{Enc}(pk)$ {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:

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

In the algebraic group model, the decryption oracle requires adversaries to provide algebraic representations for their queries, guarantee all ciphertexts can be expressed by 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 [1].

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 an intermediate step to IND-CCA2. Formal treatments and exercises with complete games, oracles and reductions can be found in standard texts and course notes [28, 29].

4.2.2 Algebraic Oracle Implementation

The key component in our formalization is the algebraic oracle that enforces AGM constraints through a two-layer design followed by the insights from paper [1]:

Interface Layer: Decryption Oracle Abstraction under AGM. We encode the phase-based constraints of the IND-CCA1 game directly. 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 scout 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 combinations relative to the known ones l for every queried group element.

```

1 (** Interface for oracles used in CCA1 security games **)
2 module type Oracles_CCA1i = {
3   proc init(sk_init : sk_t) : unit                (*
4   Initialize with secret key *)
5   proc dec(c : ctxt_t, z : exp list) : key_t option (*
6   Decryption with representation *)
7 }
8
9 (** Adversary interface for IND-CCA1 game **)
10 module type Adv_INDCCA1 (O : Oracles_CCA1i) = {
11   proc scout(pk : pk_t) : unit {O.dec}            (* Phase 1:
12   explore with queries *)
13   proc distinguish(k : key_t, c : ctxt_t) : bool {} (* Phase 2:
14   distinguish challenge *)
15 }

```

Listing 4.2: Algebraic Oracle Interface ²

Implementation Layer: Limited Oracle with Query Quota and Representation Validation. The module `O_CCA1_Limited` enforces 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 $c = \text{prodEx}(l, z)$ 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 . For invalid queries, a placeholder value `witness` is returned to avoid additional leakage.

This implementation represents the minimal principle of “queryable implies learnable”: adversaries can only incorporate new group elements that are 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.

²code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L75-L79>

²code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L170-L173>

```

1 (** Limited CCA1 Oracle Implementation **)
2 module O_CCA1_Limited (S : Scheme) : Oracles_CCA1i = {
3   var sk : sk_t                                     (* Secret key *)
4   var qs : (ctxt_t * key_t option) list             (* Query history *)
5   var l : group list                                (* List of group
elements seen *)
6
7   (* Initialize oracle with secret key *)
8   proc init(sk_init : sk_t) = {
9     sk <- sk_init;
10    qs <- [];                                       (* Empty query list *)
11    l <- [];                                       (* Empty group element list *)
12  }
13
14  (* Decryption oracle with representation checking *)
15  proc dec(c : ctxt_t, z : exp list) : key_t option = {
16    var p : key_t option;
17    var invalid_query : bool;
18
19    (* Check if query is valid:
20     - Haven't exceeded q queries
21     - Ciphertext matches expected representation *)
22    invalid_query <- (q < size qs + 2  \/ c <> prodEx l z);
23
24    (* Perform actual decryption using scheme *)
25    p <@ S.dec(c, sk);
26
27    (* Update state only if query was valid *)
28    if (!invalid_query) {
29      l <- oget p :: l ;                               (* Add decrypted key to list *)
30      qs <- (c, p) :: qs;                               (* Record query *)
31    }
32
33    (* Return result or witness if invalid *)
34    return (if !invalid_query then p else witness);
35  }
36 }.

```

Listing 4.3: Limited Oracle Implementation with Validation ³

4.3 q-DDH Assumption

As we explained the background of q-DDH in the section 2.5, it's main property can be conclude as following sentence: In particular, even with access to $g^x, g^{x^2}, \dots, g^{x^q}$, it

³code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L120-L154>

4 q-DDH Correctness and IND-CCA1 Security

should be infeasible for any efficient adversary to distinguish a true Diffie-Hellman pair from a randomly generate one [30].

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:

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

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

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

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

4.3.1 EasyCrypt Formalization of q-DDH

The q-DDH game implemented in EasyCrypt in our work as follows:

```

1 (** q-DDH adversary interface **)
2 module type A_qDDH = {
3   proc guess(gtuple : group list) : bool  (* Distinguish q-DDH
4   tuple *)
5 }
6 (** q-DDH game: distinguish (g, g^x, g^{x^2}, ..., g^{x^q}, g^r,
7   g^{xr})
8   from (g, g^x, g^{x^2}, ..., g^{x^q}, g^r, g^z) where z is
9   random **)
10 module QDDH (A : A_qDDH) = {
11   proc main() : bool = {
12     var x, r, z , b_int;
13     var gtuple : group list;
14     var challenge : group;
15     var b, b' : bool;
16
17     (* Sample random values *)
18     x <$ dt;    (* Secret base *)
19     r <$ dt;    (* Random for challenge *)
20     z <$ dt;    (* Random alternative *)
21
22     b <$ {0,1}; (* Bit determining real or random *)
23
24     (* Convert boolean to integer for computation *)
25     b_int <- (if b then ZModE.zero else ZModE.one);
26
27     (* Create q-DDH tuple: g^x, g^{x^2}, ..., g^{x^q} *)

```

```

26     gtuple <- map (fun i => g^(exp x i)) (range 1 (q+1));
27
28     (* Challenge element: either g^{xr} (real) or g^{xr+z}
29     (random) *)
30     challenge <- g^((x * r) + (z * b_int));
31
32     (* Give adversary the tuple: [g^x, ..., g^{x^q}, g^r,
33     challenge] *)
34     b' <-@ A.guess(gtuple ++ [g^r] ++ [challenge]);
35
36     (* Adversary wins if they distinguish correctly *)
37     return b = b';

```

Listing 4.4: q -DDH Game Implementation ⁴

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.

Then, the key part 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 generating the challenge, and the additional random quantity z , and flips a coin to get b .

The program converts the boolean value b to an element b_int in the exponent field. 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: $g^{(x \cdot r) + (z \cdot b_int)}$. 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, represents that the adversary successfully distinguished; otherwise, returns false.

⁴code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L250-L287>

4.4 Mathematical Foundations

Our formalization builds upon a linear algebra framework that enables manipulation of algebraic representations in our entire work while preserve accurate 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 proof.

4.4.1 Core Linear Algebra Components

The heart of our approach will indicated below in EasyCrypt format for manipulating linear combinations in the exponent field while maintaining consistency with group operations.

Basic Group Operations.

```

1 (* Product of a list of group elements *)
2 op prod (elements : group list) = foldr ( * ) e elements.
3
4 (* Exponentiation: compute bases[i]^exps[i] for all i, return as
   list *)
5 op ex (bases : group list)(exps : exp list) =
6   (map (fun (i : group * exp) => i.'1 ^ i.'2) (zip bases exps)).
7
8 (* Product of exponentiation: compute product of ex(bases, exps)
   *)
9 op prodEx (bases : group list)(exps : exp list) =
10  prod (ex bases exps).

```

Listing 4.5: Basic Group Operations ⁵

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

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

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

Vector Operations in the Exponent Field.

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

⁵code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L83-L91>

$$\text{zerov} = \underbrace{[0, 0, \dots, 0]}_{q+1 \text{ elements}} \quad (4.3)$$

```

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

```

Listing 4.6: Zero Vector Definition ⁶

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

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

$$\text{addv}(a, b) = [a_1 + b_1, a_2 + b_2, \dots, a_n + b_n] \quad (4.4)$$

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

```

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

```

Listing 4.7: Vector Addition ⁷

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

Vector Multiplication (mulv) Vector multiplication serve as pointwise ultiplication of corresponding elements:

$$\text{mulv}(a, b) = [a_1 \cdot b_1, a_2 \cdot b_2, \dots, a_n \cdot b_n] \quad (4.5)$$

```

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

```

Listing 4.8: Vector Multiplication ⁸

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

⁶code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L316>

⁷code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L319>

⁸code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L322>

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

$$\text{scalev}(a, s) = [s \cdot a_1, s \cdot a_2, \dots, s \cdot a_n] \quad (4.6)$$

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

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

Listing 4.9: Scalar Multiplication ⁹

This operation enables scaling of algebraic representations, crucial for constructing linear combinations in this work.

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

$$\text{sumv}([v_1, v_2, \dots, v_k]) = v_1 + v_2 + \dots + v_k \quad (4.7)$$

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

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

Listing 4.10: Vector Sum ¹⁰

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:

$$\text{shift_trunc}([v_1, v_2, \dots, v_n]) = [0, v_1, v_2, \dots, v_q] \quad (4.8)$$

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

⁹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L325>

¹⁰code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L328>

```

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

```

Listing 4.11: Shift and Truncate Operation ¹¹

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.

4.4.2 Key Algebraic Properties and Lemmas

The correctness of our reduction relies on several fundamental algebraic properties and lemmas shown below, which are essential for our work.

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

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

Proof: Section A.1.

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

$$\left(\prod gs\right)^n = \prod(\text{map}(\lambda g \rightarrow g^n, gs))$$

Proof: See Section A.1.

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

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

Proof: See Section A.1.

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

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

Proof: See Section A.1.

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

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

¹¹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L331>

4 q -DDH Correctness and IND-CCA1 Security

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:*

$$\text{prodEx}(\text{bases}, \text{shift_trunc}(\text{exps})) = \text{prodEx}(\text{behead}(\text{bases}), \text{exps})$$

Proof: Section A.3.

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

$$\text{prodEx}(\text{bases}, []) = e \quad (4.9)$$

$$\text{prodEx}([], \text{exps}) = e \quad (4.10)$$

$$\text{prodEx}([], []) = e \quad (4.11)$$

Proof: Section A.2.

Lemma 8 (Prodex size less or equal). *When the exponent vector is longer than the base list, the product only depends on the prefix matching the base list length:*

$$\text{prodEx}(\text{bases}, \text{exps}) = \text{prodEx}(\text{bases}, \text{take}(|\text{bases}|, \text{exps})) \quad \text{when } |\text{exps}| \geq |\text{bases}| \quad (4.12)$$

Equivalently, oversized exponents beyond the base list size are ignored:

$$\prod_{i=0}^{n-1} g_i^{e_i} = \prod_{i=0}^{m-1} g_i^{e_i} \quad \text{when } |[e_0, e_1, \dots, e_m]| \geq n \text{ and } m \geq n \quad (4.13)$$

Proof: Section A.7.

Lemma 9 (Vector Addition with Empty Operands). *Vector addition with empty operands:*

$$\text{addv}([], b) = [] \quad (4.14)$$

$$\text{addv}(a, []) = [] \quad (4.15)$$

$$\text{sumv}([]) = \text{zerov} \quad (4.16)$$

Proof: Section A.4.

Lemma 10 (Size Preservation). *Vector operations preserve or predictably modify sizes:*

$$\text{size}(\text{scalev}(v, s)) = \text{size}(v) \quad (4.17)$$

$$\text{size}(\text{addv}(a, b)) = \min(\text{size}(a), \text{size}(b)) \quad (4.18)$$

$$\text{size}(\text{shift_trunc}(v)) = \min(q + 1, \text{size}(v) + 1) \quad (4.19)$$

Proof: Section A.5.

Lemma 11 (Cons Operation for prodEx). *For a group element g , group list gs , exponent e , and exponent list es :*

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

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

Proof: Section A.6.

Lemma 12 (Double prodEx Composition). *For any base list b and exponent lists $e1$, $e2$:*

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

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

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

Proof: See Section A.9.

Lemma 13 (Ex-Map-prodEx Equivalence). *For bases, exps, and constant c with matching sizes:*

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

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

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

Proof: See Section A.9.

Lemma 14 (prodEx Map with Range). *For generator g , exponent x , and range $[s, e)$:*

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

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

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

Proof: See Section A.9.

4.4.3 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 15 (Range Simplification). *For any secret key sk :*

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

Proof: See Section B.1.

Lemma 16 (Ex Cons Distribution). *or group element x , group list xs , exponent e , and exponent list es :*

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

Proof: See Section A.8.

Lemma 17 (Sumv Cons Distribution). *F to adding the first element to the sum of the remaining elements, we can get:*

$$\sum_{i=0}^n v_i = v_0 + \sum_{i=1}^n v_i$$

Proof: See Section B.2.

Lemma 18 (Ex Range Shift Property). *For secret key sk :*

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

Proof: See Section B.3.

Lemma 19 (Scalev of Nseq). *For integer n , exponents x and c :*

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

Proof: See Section B.4.

Lemma 20 (Drop-Addv Commutativity). *For vectors u, v of equal size and integer n :*

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

Proof: See Section B.5.

Lemma 21 (Addv Size Inequality). *For vectors a, b with $\text{size}(a) \leq \text{size}(b)$:*

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

Proof: See Section B.6.

Lemma 22 (Drop-Sumv Commutativity). *For non-empty list xs of uniform-sized vectors and valid index n :*

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

Proof: See Section B.7.

Lemma 23 (Sumv of Zero Vectors). *For non-negative integer n :*

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

Proof: See Section B.8.

Lemma 24 (Sumv of Singleton Zero Vectors). *For positive integer n :*

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

Proof: See Section B.9.

Lemma 25 (Zip Concatenation Distributivity). *For lists a_1, a_2, b_1, b_2 with $\text{size}(a_1) = \text{size}(b_1)$:*

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

Proof: See Section B.10.

Lemma 26 (Product Concatenation). *For group lists xs and ys :*

$$\prod(xs ++ ys) = \prod(xs) \cdot \prod(ys)$$

Proof: See Section B.11.

Lemma 27 (prodEx Split with Last Zero). *For equal-sized lists $bases$ and $exps$ where the last exponent is zero:*

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

Proof: See Section B.12.

Lemma 28 (Behead-Drop Equivalence). *For any group list $base$:*

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

Proof: See Section B.13.

Evaluation : Formalizing the Bilateral Reduction

This chapter shows the core part of our formalization and verification of our bilateral equivalence between IND-CCA1 security of ElGamal and the q-DDH assumption from the paper [1]. I will indicate both directions of the reduction with complete EasyCrypt implementations and proofs first.

5.1 Overview of bidirectional reduction

The background explanation is in the section 2.6, Our core formalization 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 $(g^x, g^{x^2}, \dots, g^{x^q}, g^r, T)$, then set the public key as $pk = g^x$.
2. **Oracle Simulation:** Then it 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 request the encryption challenge, which 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 integrate the received IND-CCA1 challenge into a q-DDH problem by generating the tuple $(g^x, g^{x^2}, \dots, g^{x^q}, C^*, K^*)$ 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 highlights 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

```

1 module (A_from_INDCCA1 (A : Adv_INDCCA1) : A_qDDH) = {
2
3   (* State variables for the reduction *)
4   var gxs : group list          (* Powers  $g^x, g^{x^2}, \dots,$ 
5    $g^{x^q}$  *)
6   var l : group list           (* List of group elements (oracle
   state) *)
7   var reps : exp list list     (* Linear representations of l in
   basis (g::gx) *)

```

Listing 5.1: $A_from_INDCCA1$ Module Structure ¹

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

gx s stores the powers $[g^x, g^{x^2}, \dots, g^{x^q}]$ extracted from the q-DDH challenge.

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;

¹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1261-L1266>

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

5.2.2 Internal Oracle Simulation

The core part 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.

```

1  (* Internal oracle that simulates CCA1 oracle using q-DDH
challenge *)
2  module O_Internal : Oracles_CCA1i = {
3    var sk : sk_t
4    var qs : (ctxt_t * key_t option) list
5    var challenge_c : ctxt_t
6
7    proc init(sk_init : sk_t) = {
8      sk <- sk_init;
9      qs <- [];
10
11      l <- [];
12    }
13
14    (* Core oracle: simulate decryption without knowing secret
key *)
15    proc dec(c : ctxt_t, z : exp list) : key_t option = {
16      var p : key_t option;
17      var rep_c, rep_p;
18      var invalid_query : bool;
19
20      (* Validity check: query limit and representation
consistency *)
21      invalid_query <- (q < size qs + 2  \/ c <> prodEx l z);
22
23      (* Compute representation of ciphertext in basis (g::gxs) *)
24      rep_c <- sumv (map (fun x : exp list * exp => scalev x.'1
x.'2)
25        (zip reps z));
26      (* Prepend zero for g^0 term *)
27      rep_p <- ( shift_trunc rep_c);
28
29      (* Compute corresponding group element *)
30      p <- Some (prodEx (g :: gxs) (rep_p));
31
32      (* Update state if query was valid *)
33      if (!invalid_query) {
34        reps <- rep_p :: reps ;
35        l <- oget p :: l ;          (* Add to group element

```

5 Evaluation : Formalizing the Bilateral Reduction

```

36     list *)
37     qs <- (c, p) :: qs;      (* Record query *)
38     (* Store representation *)
39
40     (* Return result *)
41     return (if invalid_query then witness else p);
42   }
43 }

```

Listing 5.2: Internal Oracle Implementation ²

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 implement 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, then it perform a validity check combining two conditions is performed, namely (i) the query limit $q < \text{size } qs + 2$ that ensures unable to exceed the allowed number of queries and (ii) the representation consistency $c \neq \text{prodEx } l \ z$ that verifies the `z` correctly represents ciphertext `c`. Then the algebraic magic happen: we compute the representation of ciphertext `c` in the basis (g, g^x, \dots, 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, \dots, 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

```

1  (* Main reduction procedure *)
2  proc guess(gtuple : group list) : bool = {
3    var c : ctxt_t;
4    var k : key_t;
5    var b' : bool;
6    var x_exp : exp;

```

²code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1268-L1310>

```

7
8      (* Parse q-DDH challenge: gtuple = [g^x, g^{x^2}, ...,
9      g^{x^q}, g^r, T] *)
10     gxs <- take q gtuple;          (* Extract [g^x, ...,
11     g^{x^q}] *)
12     c <- nth witness gtuple q;      (* Extract g^r (ciphertext)
13     *)
14     k <- nth witness gtuple (q + 1); (* Extract T (challenge
15     key) *)
16
17     (* Initialize internal oracle *)
18     O_Internal.init(witness);
19
20     (* Set initial state: adversary has seen g and g^x *)
21     l <- head witness gxs :: g :: [];
22     (* Corresponding representations in basis (g::gx) *)
23     reps <- (* g = g^1 * (g^x)^0 * ... *)
24     (zero :: one :: nseq (q-1) zero) :: [(one :: nseq q zero)];
25     (* g^x = g^0 * (g^x)^1 * ... *)
26
27
28     (* Run IND-CCA1 adversary *)
29     A(O_Internal).scout(head witness gxs);          (* Scout phase *)
30     b' <-@ A(O_Internal).distinguish(k, c);          (* Challenge phase
31     *)
32
33     return b';
34 }

```

Listing 5.3: Main Reduction Procedure ³

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 $[g^x, g^{x^2}, \dots, g^{x^q}]$ 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:

³code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1313-L1340>

5 Evaluation : Formalizing the Bilateral Reduction

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

```
1 (** Alternative adversary construction **)
2 module (B_from_qDDH (A : A_qDDH) : Adv_INDCCA1) (O :
  Oracles_CCA1i) = {
3   var gxs : group list
4
5   (* Scout phase: build up powers of x using decryption oracle *)
6   proc scout(pk:pk_t) : unit = {
7     var i : int;
8     var p : key_t option;
9     gxs <- [pk];
10    i <- 1 ;
11
12    while (i <= q-1) {
13
14      p <- O.dec(last witness gxs, ( one :: nseq i zero));
15
16      gxs <- gxs ++ [oget p];
17      i <- i + 1;
18    } }
```

Listing 5.4: `B_from_qDDH` Module Structure ⁴

Module declaration taking a q-DDH adversary `A` and implementing the IND-CCA1 adversary interface with oracle access. Then we state variable `gx`s to store the powers of x that we'll construct through oracle queries. It will then scout phase procedure where

⁴code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1345-L1361>

5.3 Backward Reduction: q -DDH to IND-CCA1

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}, \dots) and loop to construct powers $g^{x^2}, g^{x^3}, \dots, g^{x^q}$.

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 $(1, 0, 0, \dots, 0)$ "
- 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 $[g^x, g^{x^2}, \dots, g^{x^{i+1}}]$ and increase counter to build the next power.

5.3.2 Distinguish Phase

```

1  (* Convert to q-DDH challenge *)
2  proc distinguish(k: key_t, c: ctxt_t) : bool = {
3      var b' : bool;
4
5      (* Pass challenge to q-DDH adversary *)
6      b' <@ A.guess( gxs ++ [c] ++ [k] );
7      return b';
8  }
```

Listing 5.5: B_from_qDDH Distinguish Phase ⁵

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 $[g^x, g^{x^2}, \dots, g^{x^q}]$ 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.

⁵code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1366-L1372>

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:

$$\Pr[\text{IND_CCA1_P}(\text{ElGamal}, \mathcal{B}).\text{main}() : \text{res}] = \Pr[\text{QDDH}(A_from_INDCCA1(\mathcal{B})).\text{main}() : \text{res}]$$

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:

$$\Pr[\text{QDDH}(\mathcal{A}).\text{main}() : \text{res}] = \Pr[\text{IND_CCA1_P}(\text{ElGamal}, B_from_qDDH(\mathcal{A})).\text{main}() : \text{res}]$$

Both reductions are tight with no security loss.

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

```

1 (* Forward direction *)
2 lemma qDDH_Implies_INDCCA1_ElGamal &m :
3   Pr[IND_CCA1_P(ElGamal, B).main() @ &m : res] =
4   Pr[QDDH(A_from_INDCCA1(B)).main() @ &m : res].
5
6 (* Backward direction *)
7 lemma INDCCA1_ElGamal_Implies_qDDH &m :
8   Pr[QDDH(A).main() @ &m : res] =
9   Pr[IND_CCA1_P(ElGamal, B_from_qDDH(A)).main() @ &m : res].

```

Listing 5.6: Main Bilateral Equivalence Lemmas ⁶

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/blob/main/INDCCA.ec#L1378> and <https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1571>

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 $\text{rnd } (\text{fun } z \Rightarrow g^{(z + \text{sk0}\{2\} * y\{2\})})(\text{fun } k' \Rightarrow \text{loge } k' - \text{sk0}\{2\} * y\{2\})$, 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

5 Evaluation : Formalizing the Bilateral Reduction

2. **Oracle Invariant Maintenance:** The proof first introduce an 8-condition invariant for oracle simulation:

- **Query List Consistency:** $\text{IND_CCA1_P.OS.1}\{1\} = \text{A_from_INDCCA1.1}\{2\}$
- **Query Count Synchronization:** Between different oracle implementations
- **Representation-Ciphertext Mapping:** $\text{A_from_INDCCA1.1}\{2\} = \text{map } (\text{prodEx } (g :: \text{A_from_INDCCA1.gxs}\{2\})) \text{ reps}$
- **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

5.7 Case Study 1: Mechanizing Algebraic Manipulations and Self evaluation

As we mentioned before in the section 1.2, mastering easycrypt requiring substantial effort beyond reading the reference manual. To illustrate the challenges of using EasyCrypt mechanizing AGM-based proofs, I will present a case study and self experience of a key proof step.

Note: This section contains detailed implementation steps documenting the learning process. Readers may skip to Section 5.10 for the main results.

5.7.1 The Proof Goal

At a key point in the formalization of the proof of the reduction to IND-CCA1 from QDDH, we must establish that the decryption oracle simulation produces outputs consistent with the real decryption oracle. This requires proving an algebraic equality involving nested product-of-exponents operations. The goal state in EasyCrypt is:

```
1 prodEx (map (prodEx bases) reps) z ^ sk
2 = prodEx bases (shift_trunc (sumv (scalev_map (reps, z))))
```

Listing 5.7: key proof goal ⁷

where:

- **bases** is the list $[g, g^x, g^{x^2}, \dots, g^{x^q}]$ derived from the q-DDH challenge
- **reps** is the list of algebraic representations provided by the adversary for previous queries
- **z** is the representation provided for the current query
- **sk** is the secret key, corresponding to x in the q-DDH

In a paper proof, we can state: By the algebraic structure of the AGM, the nested `prodEx` operations can be flattened using distributivity. However, mechanizing this in EasyCrypt requires making explicit all intermediate steps, we specialized the proof structure below.

5.7.2 Proof Structure

The complete proof of this status needs over 200 lines of EasyCrypt code and can be divided into four major phases:

Phase 1: Key Transformation (Lines 1643–1645). The first phase applies a fundamental transformation lemma to convert nested `prodEx` operations into a single `prodEx` with combined representations.

5 Evaluation : Formalizing the Bilateral Reduction

```

1 (* Apply the fundamental lemma ex_map_prodEx:
2   prodEx(map(prodEx(bases, @$.cdot$@), reps), z)
3   = prodEx(bases, shift_trunc(sumv(scalev_map(reps, z)))) *)
4 rewrite (ex_map_prodEx _ _ (q+1) (size reps)).

```

Listing 5.8: ex_map_prodEx ⁸

This lemma encodes the mathematical principle where encoded in appendix A.9: It converts a map of `prodEx` expressions into a single `prodEx` over summed representations. Specifically, the fundamental lemma `ex_map_prodEx`, which states that:

$$\text{prodEx}(\text{map}(\text{prodEx}(\text{bases}, \cdot), \text{reps}), z) = \text{prodEx}(\text{bases}, \text{shift_trunc}(\text{sumv}(\text{scalev_map}(\text{reps}, z)))).$$

The size arguments $(q + 1)$ and (size reps) ensure that this transformation is well-defined and valid.

Phase 2: Flattening Distributivity (Lines 1628–1664). The second phase proves a key distributivity property which is formalized as:

```

1  have flat : prodEx (map (prodEx bases) reps) z
2    = prodEx bases (sumv (map (fun (x : ZModE.exp list *
3      ZModE.exp) => scalev x.'1 x.'2) (zip reps z))).
4    have reps_all_size: all (fun rep => size rep = q + 1) reps.
5    exact H1.
6    elim: reps z reps_all_size => [|rep reps IH] [|z_head
7      z_tail] reps_size //=.
8    rewrite prodEx_nil. rewrite sumv_nil.
9    rewrite /zerov. rewrite prodEx_nseq_zero. smt(gt0_q) .
10   smt().
11   rewrite prodEx_nil_l. rewrite sumv_nil. rewrite /zerov.
12   rewrite prodEx_nseq_zero. smt(gt0_q). smt().
13   rewrite prodEx_nil_r. rewrite sumv_nil. rewrite /zerov.
14   rewrite prodEx_nseq_zero. smt(gt0_q). smt().
15   rewrite prodExConsGeneral. rewrite sumv_cons. rewrite
16   prodEx_addv_distributive. rewrite /scalev.
17   rewrite size_map. have size_bases: size bases = q + 1.
18   rewrite /bases size_map size_range. smt(@List @GP gt0_q).
19   have size_rep : size rep = q + 1. smt(@G @GP). smt().
20   have sumv_size: size (sumv (map (fun (x : ZModE.exp list *
21     ZModE.exp) => scalev x.'1 x.'2)
22     (zip reps z_tail))) = q + 1.
23   case: (reps = []) => [reps_empty | reps_nonempty].
24   - rewrite reps_empty /=. rewrite zip_nil_l. rewrite
25     sumv_nil. rewrite /zerov size_nseq. smt(gt0_q).
26   - apply size_sumv.

```

⁸code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1643-L1645>

5.7 Case Study 1: Mechanizing Algebraic Manipulations and Self evaluation

```

18     move=> v /mapP [pair [pair_in ->]].
19     case: pair pair_in => [rep_elem z_elem] /= pair_in_zip.
20     rewrite size_scalev. have: rep_elem \in reps by
smt(@List). smt(@G @GP @List). rewrite sumv_size.
21     have size_bases: size bases = q + 1.
22     rewrite /bases size_map size_range.
23     smt(@List @GP gt0_q). smt().
24     congr. rewrite prodExScale1.
25     trivial. rewrite IH. smt(). trivial. rewrite flat.

```

Listing 5.9: flatten ⁹

Flatten nested `prodEx` operations using distributivity. This establishes that `prodEx` distributes over `map` and can be flattened:

$$\text{prodEx}(\text{map}(\text{prodEx}(\text{bases}, \cdot), \text{reps}), z) = \text{prodEx}(\text{bases}, \text{sumv}(\text{scalev_map}(\text{reps}, z))).$$

Left side: Apply `prodEx` to each `rep` with `bases`, then combine with `z`.

Right side: First combine `reps` with `z` using scalar multiplication, sum them, then apply `prodEx`.

This transformation is fundamental for the reduction's correctness, it proceeds by induction on the list `reps`, with extensive case analysis:

- **Base cases** (3 cases): Empty list, singleton list with empty tail, singleton list with empty head
- **Inductive case:** Non-empty list with non-empty tail

For each case, we optionally to achieve this flatten goal and rewrite:

- Rewrite using lemmas `prodExConsGeneral`(lemma11), `sumv_cons`(lemma17), `prodEx_addv_distributive`(lemma1)
- Establish size properties using `size_sumv`, `size_scalev` prove in section A.5, `size_map`
- Apply the induction hypothesis with appropriate size constraints

Phase 3: Zero Suffix Property . This phases is the most complex part of this case study, aims to establishes that the last element of the combined representation vector is zero. This property derives from the restriction that adversaries can only query the decryption oracle before the challenge phase, guarantee the $q + 1$ -th component (corresponding to the challenge ciphertext) will not appear in any query representation.

⁹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1665-L1689>

Step 3.1: Goal Statement We first state the main goal and introduce the key definition:

```

1 (* Main goal: prove the last element of exps is zero *)
2 pose exps := (sumv (map (fun (x : ZModE.exp list * ZModE.exp) =>
   scalev x.'1 x.'2)(zip reps z))).
3 have drop_last_ele : drop q exps = nseq 1 zero.
4 rewrite /exps.
5 (* exps is defined as the sum of scaled representations *)

```

Listing 5.10: Zero suffix goal statement ¹⁰

The strategy to prove this is first showing each individual element in the mapped list has a zero suffix, then showing this property is maintained after summation.

Step 3.2: Individual Zero Suffixes The core lemma proves that each element in the mapped list has a zero in its last position:

```

1 (* Prove each element has zero suffix *)
2 have all_elements_zero_suffix:
3   forall u, u ∈ map (fun (x : exp list * exp) => scalev x.'1
   x.'2) (zip reps z)
4   => drop q u = nseq 1 zero.
5 move=> u u_in_map.

```

Listing 5.11: Individual zero suffix property ¹¹

From this sub-lemma, we need to show for each element u in the mapped list, we must show $\text{drop } q \ u = [\text{zero}]$. This proceeds through following sub-steps:

Step 3.2a: Destruct the element

```

1 have: exists pair, pair @ $\in$ zip reps z /\ u = scalev pair.'1
   pair.'2.
2   smt(@List).
3 case=> [[rep z_elem]] [pair_in_zip u_def].

```

Listing 5.12: Destructing element from zip ¹²

This extracts the underlying pair (rep , z_elem) from the zip, establishing that $u = \text{scalev rep z_elem}$.

¹⁰code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1695>

¹⁰code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1713-L1714>

¹¹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1716-L1719>

¹²code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1720-L1721>

Step 3.2b: Apply hypothesis H2 Following H2 is derived from the proof goal from EasyCrypt during this stage

```

1 H2: all(fun (rep : ZModE.exp list) =>
2     drop (size A_from_INDCCA1.reps{2}) rep =
3     nseq (q + 1 - size A_from_INDCCA1.reps{2}) zero)
4 A_from_INDCCA1.reps{2}

```

Listing 5.13: H2 ¹³

It states that all representations have zero suffixes beyond the query count:

```

1 have rep_zero_suffix:
2   drop (size A_from_INDCCA1.reps{2}) rep =
3   nseq (q + 1 - size A_from_INDCCA1.reps{2}) zero.
4 move: H2. rewrite allP. move=> H2_expanded.
5 apply H2_expanded. smt(@List).

```

Listing 5.14: Applying hypothesis H2 for zero suffix ¹⁴

This establishes that `rep` has trailing zeros starting from position `size reps`.

Step 3.2c: Correspond query count to position q

We need to connect `size reps` to the position q :

```

1 have size_qs_le_q: size 0_CCA1_Limited.qs{1} <= q. smt().
2 rewrite u_def.
3 rewrite drop_scalev. simplify.
4 have rep_size: size rep = q + 1. smt(@List).
5 have drop_q_rep: drop q rep = [nth witness rep q].
6   smt(@List @G @GP gt0_q).

```

Listing 5.15: Relating query count to position q ¹⁵

The key point is `size qs` $\leq q$, so the zero suffix starting at `size reps` = `size qs` + 2 certainly includes position q .

Step 3.2d: Extract the q -th element is zero

```

1 have nth_q_zero: nth witness rep q = zero.
2 have: nth witness rep q =
3   nth witness (nseq (q + 1 - size reps) zero)

```

¹⁴code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1723-L1726>

¹⁵code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1727-L1733>

5 Evaluation : Formalizing the Bilateral Reduction

```
4      (q - size reps).  
5      smt(@List @G @GP gt0_q).  
6 smt(@List @G @GP gt0_q).
```

Listing 5.16: Code snippet ¹⁶

Using the zero-suffix property, it follows that accessing index `q` of `rep` returns zero.

Step 3.2e: Conclude by `scalev`

```
1 rewrite drop_q_rep nth_q_zero.  
2 have one_zero : [zero] = nseq 1 zero. smt(@G @GP @List).  
3 rewrite one_zero.  
4 rewrite scalev_nseq_zero. trivial.
```

Listing 5.17: Code snippet ¹⁷

Since the last element of `rep` is zero, and scalar multiplication preserves zeros (via `scalev_nseq_zero`), we can prove the last element of `u = scalev rep z_elem` is also zero.

Until here, we completes the proof that each individual element has a zero suffix.

Step 3.3: Universal Quantification We now integrate the individual zero suffix property into a universal statement:

```
1 have all_form: all (fun u => drop q u = nseq 1 zero)  
2      (map (scalev ...) (zip reps z)).  
3 apply/allP => u u_in_map.  
4 exact (all_elements_zero_suffix u u_in_map).  
5 move: all_form.  
6  
7 pose mapped_list := map (scalev ...) (zip reps z).  
8 move=> all_form.
```

Listing 5.18: Code snippet ¹⁸

This step formalizes that *every* element in `mapped_list` satisfies the zero suffix property. This is essential for the next step where we try to prove for entire list.

Step 3.4: Preservation Under Summation This step shows that `sumv` can preserves the zero suffix property.

¹⁶code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1734-L1746>

¹⁷code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1747-L1750>

¹⁸code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1760-L1768>

Step 3.4a: Empty list base case (Line 1746).

```
1 case (mapped_list = []).
2 move => map_empty.
3 rewrite map_empty sumv_nil /zerov drop_nseq.
4 smt(). smt().
```

Listing 5.19: Code snippet ¹⁹

If the list is empty, `sumv [] = zerov` and `drop q zerov` is trivially `nseq 1 zero`.

Step 3.4b: Apply `drop_sumv` lemma

Then, for non-empty lists, we use the distributivity of `drop` over `sumv`:

```
1 move => mapped_nonempty.
2 have tran : drop q (sumv mapped_list) =
3             sumv (map (drop q) mapped_list).
4 rewrite drop_sumv.
```

Listing 5.20: Code snippet ²⁰

However, `drop_sumv` requires that all elements have the same size which require us to verify this:

```
1 smt(). rewrite /mapped_list. apply/allP => u u_in_mapped.
2 have: exists pair, pair @$\in$@ zip reps z /\
3     u = scalev pair.'1 pair.'2. apply/mapP. by [].
4 case => pair [pair_in_zip u_eq].
5 rewrite u_eq /= size_scalev.
6 have pair_first_in_reps: pair.'1 @$\in$@ reps. smt(@List).
7 have: size pair.'1 = q + 1. smt(allP). by [].
8 smt(). smt(@List). rewrite tran.
```

Listing 5.21: Code snippet ²¹

This proves that each `scalev pair.'1 pair.'2` has size $q+1$ (as `scalev` preserves size, and each `pair.'1` has size $q+1$ by hypothesis H1). Where following H1 is derived from the proof goal from EasyCrypt during this stage

```
1 H1: all (fun (rep : ZModE.exp list) => size rep = q + 1)
  A_from_INDCCA1.
2     reps{2}
```

¹⁹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1771>

²⁰code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1772-L1773>

²¹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1774-L1783>

3

Listing 5.22: H1 ²²

Step 3.4c: All dropped suffixes are zero

Then we need to prove that `map (drop q) mapped_list` is a list of all zeros:

```

1 have map_all_zero: map (drop q) mapped_list =
2   nseq (size mapped_list) (nseq 1 zero).
3 apply (eq_from_nth witness).
4 rewrite size_map size_nseq.
5 case: (0 <= size mapped_list) => [ge0_size|lt0_size].
6   smt(@List). + smt(@List).
```

Listing 5.23: Code snippet ²³

We prove equality using `eq_from_nth`: two lists are equal if they have the same size, and same elements at each index.

```

1 move=> i hi.
2 rewrite size_map in hi. rewrite (nth_map witness) //.
3 rewrite nth_nseq_if.
4 case: (0 <= i < size mapped_list) => [valid_i|invalid_i].
5 have u_in_mapped: nth witness mapped_list i @$\in$@ mapped_list.
6   by apply mem_nth.
7 have u_drop_zero: drop q (nth witness mapped_list i) = nseq 1
8   zero.
9   smt(@List @GP).
10 by rewrite u_drop_zero.
11 smt().
```

Listing 5.24: Code snippet ²⁴

For each valid index i , we can use the `all_form` hypothesis to conclude that `drop q (nth witness mapped_list i) = nseq 1 zero`.

Step 3.4d: Sum of zeros is zero

```

1 rewrite map_all_zero.
2 have map_ge1: 1 <= size mapped_list by smt(@List).
3 rewrite sumv_nseq_zero_singleton. smt(). trivial.
```

Listing 5.25: Code snippet ²⁵

²³code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1784-L1791>

²⁴code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1793-L1801>

5.7 Case Study 1: Mechanizing Algebraic Manipulations and Self evaluation

Finally, we apply `sumv_nseq_zero_singleton`, which states that the summation of a list of `[zero]` vectors is `[zero]`. This finally concludes the proof that `drop q exps = nseq 1 zero`.

Phase 4: Final Simplification. The final phase uses the zero suffix property from Phase3 to simplify the expression. The state we need to prove becomes as following:

```
1 prodEx(map (fun (i : int) => g ^ exp 0_CCA1_Limited.sk{1} i)
  (range 1 (q + 2)))exps =
2 prodEx(map (fun (i : int) => g ^ exp 0_CCA1_Limited.sk{1} i)
  (range 0 (q + 1)))(shift_trunc exps)
```

Listing 5.26: Code snippet ²⁶

Step 4.1: Applying Shift-Truncate Simplification First, applying the `prodExShiftTrunce` lemma⁶, which simplifies the shifted representation:

```
1 rewrite prodExShiftTrunce.
2 rewrite size_map size_range. smt(gt0_q).
3 apply size_exps_analysis. smt().
```

Listing 5.27: Applying shift-truncate simplification ²⁷

The `prodExShiftTrunce` lemma states that shifting the representation of a vector by one position corresponds to removing the first base element. Mathematically:

$$\text{prodEx}([g_0, g_1, \dots, g_q], \text{shift_trunc}(\text{exps})) = \text{prodEx}([g_1, \dots, g_q], \text{exps})$$

Step 4.2: Proving the Last Element is Zero Before applying `prodEx_split_last_zero`, we need a proof that the q -th element of `exps` is zero:

```
1 (* Prove drop q exps is same as nth witness exps q *)
2 have last_exp_zero: nth witness exps q = zero.
3 have: drop q exps = [zero].
4   rewrite drop_last_ele.
5   have nseq_one_zero: nseq 1 zero = [zero]. smt(@List).
6   smt(@List).
```

Listing 5.28: Proving the last element is zero ²⁸

²⁵code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1801-L1802>

²⁷code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1806-L1807>

²⁸code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1811-L1814>

5 Evaluation : Formalizing the Bilateral Reduction

This prove that $\text{drop } q \text{ exps} = [\text{zero}]$ using drop_last_ele established from Phase 3, which proved $\text{drop } q \text{ exps} = \text{nseq } 1 \text{ zero}$. Then, we want to extract the specific element by the relationship between nth and drop :

```

1 move=> drop_eq.
2 have: size (drop q exps) = 1. rewrite drop_eq. smt(@List).
3 have: nth witness (drop q exps) 0 = zero. rewrite drop_eq.
  smt(@List).
4 have: nth witness exps q = nth witness (drop q exps) 0.
  rewrite nth_drop. smt(). smt(). smt().
5 smt().
6

```

Listing 5.29: Extracting nth element from drop ²⁹

Since $\text{drop } q \text{ exps}$ has size 1 where its single element is zero , by nth_drop lemma: $\text{nth witness exps } q = \text{nth witness (drop } q \text{ exps) } 0$; Therefore, $\text{nth witness exps } q = \text{zero}$

Step 4.3: Splitting Off the Zero Component With last_exp_zero established, we can apply the splitting lemma²⁷:

```

1 rewrite prodEx_split_last_zero.
2 rewrite size_map size_range.
3 smt(@G @GP @List gt0_q). smt().
4 rewrite -last_exp_zero. smt().

```

Listing 5.30: Splitting off zero component ³⁰

The $\text{prodEx_split_last_zero}$ lemma²⁷ states that if the last exponent is zero, that component contributes nothing to the product output, the detailed code format of proof for this lemma lies in the case study ^{5.8}:

$$\text{prodEx}([g_0, \dots, g_q], [e_0, \dots, e_q]) = \text{prodEx}([g_0, \dots, g_{q-1}], [e_0, \dots, e_{q-1}])$$

when $e_q = 0$.

Step 4.4: Range Reconciliation The most complex part of this phase includes reconciling range expressions. In particular, after the transformations, it becomes necessary to translate between different range representations.

Step 4.4a: Simplify arithmetic bounds

```

1 rewrite size_map size_range.

```

²⁹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1815-L1822>

³⁰code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1824-L1827>

5.7 Case Study 1: Mechanizing Algebraic Manipulations and Self evaluation

```

2 have max_qplus : (max 0 (q + 2 - 1) - 1) = q by smt(gt0_q).
3 rewrite max_qplus. rewrite -map_take.

```

Listing 5.31: Simplifying arithmetic expressions ³¹

This simplifies $(\max 0 (q + 2 - 1) - 1) = q$, which follows from $q > 0$.

Step 4.4b: Range splitting

```

1 have take_range : take q (range 1 (q + 2)) = range 1 (q+1).
2 have range_split: range 1 (q + 2) = range 1 (q + 1) ++ [q + 1].
3   smt(@List).

```

Listing 5.32: Range splitting transformation ³²

The range $[1, 2, \dots, q + 1]$ can be split as:

$$\text{range } 1 (q + 2) = [1, 2, \dots, q] ++ [q + 1]$$

Step 4.4c: Category manipulation with take and concat

```

1 rewrite take_cat. rewrite size_range.
2 have case_nop : !(q < max 0 (q + 1 - 1)) by smt(gt0_q).
3 rewrite case_nop. simplify.
4 have case_not: (q - max 0 q <= 0) by smt(gt0_q).
5 rewrite case_not. simplify.
6 smt(@List).

```

Listing 5.33: Take-cat lemma application ³³

The `take_cat` lemma defined in `easyencrypt` theories, handles taking from concatenated lists:

$$\text{take } n (xs ++ ys) = \begin{cases} xs ++ \text{take } (n - |xs|) ys & \text{if } n > |xs| \\ \text{take } n xs & \text{otherwise} \end{cases}$$

It verifies we're taking exactly the first part (the full `range 1 (q+1)`).

Step 4.4d: Converting between range bases

```

1 rewrite take_range. rewrite behead_drop. rewrite -map_drop.
2 have drop_range : (drop 1 (range 0 (q + 1))) = (range 1 (q+1)).
3   smt(@List gt0_q).

```

³¹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1829-L1830>

³²code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1835-L1836>

³³code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1836-L1838>

```
4 rewrite drop_range.
```

Listing 5.34: Range base conversion via drop ³⁴

This proves the relationship between `range 0 (q+1)` and `range 1 (q+1)`:

$$\text{drop } 1 [0, 1, 2, \dots, q] = [1, 2, \dots, q]$$

The `behead` operation (removing the first element) is expressed as `drop 1`.

Step 4.5: Final Equality by Size Reasoning The proof finally concludes by establishing the final equality:

```
1 have last_eq :
2   prodEx (map (fun (i : int) => g ^ exp 0_CCA1_Limited.sk{1} i)
3   (range 1 (q + 1))) exps
4   = prodEx (map (fun (i : int) => g ^ exp 0_CCA1_Limited.sk{1} i)
5   (range 1 (q + 1)))
6   (take (size exps - 1) exps).
7 rewrite prodEx_sizele.
8 rewrite size_map size_range. rewrite size_exps_analysis.
9 smt(gt0_q @GP @ZModE).
10 rewrite size_map size_range. smt(gt0_q).
11 rewrite last_eq. trivial.
```

Listing 5.35: Final equality via `prodEx_sizele` ³⁵

The `prodEx_sizele` lemma can prove that oversized exponents are ignored:

$$\text{prodEx}(\text{bases}, \text{vec}) = \text{prodEx}(\text{bases}, \text{take}(|\text{bases}|, \text{vec}))$$

when $|\text{vec}| \geq |\text{bases}|$ where length of the `vec` greater or equal to `bases`.

5.8 Case Study 2: ProdEx Split Last Zero Lemma

This case study presents the formalization of the `prodEx_split_last_zero` lemma, a fundamental result in our linear algebra library that enables efficient handling of representations with trailing zero exponents. This lemma is crucial for the previous case study of our main proof. It is an important helper lemma we need for our main proof.

Note: This section contains detailed implementation steps documenting the learning process. Readers may skip to Section 5.10 for the main results.

³⁴code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1839-L1840>

³⁵code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1841-L1849>

5.8.1 The Lemma Statement

The lemma lies in the lemma 27, and here is the formal statement in EasyCrypt

```

1 lemma prodEx_split_last_zero (bases : group list) (exps :
  ZModE.exp list) :
2   size bases = size exps =>
3   0 < size exps =>
4   nth witness exps (size exps - 1) = zero =>
5   prodEx bases exps =
6   prodEx (take (size bases - 1) bases) (take (size exps - 1)
    exps).

```

Listing 5.36: ProdEx split last zero lemma statement ³⁶

The lemma requires three preconditions:

1. **Size equality:** `size bases = size exps` ensures lists are paired correctly
2. **Non-empty:** `0 < size exps` ensures there exists at least one element
3. **Zero last element:** `nth witness exps (size exps - 1) = zero` is the key property, which is the precondition we derived from the intermediate step of case study 1.

5.8.2 Proof Structure

The proof consists of approximately 30 lines and can be divided into four phases:

Phase 1: Splitting the Exponent List We begin by expressing `exps` as the concatenation of its prefix and the last element:

```

1 (* split exps into prefix and last element *)
2 have exps_split: exps = take (size exps - 1) exps ++ [zero].
3 rewrite -(cat_take_drop (size exps - 1)).

```

Listing 5.37: Splitting exponents into prefix and last element ³⁷

It use the following property:

$$\text{list} = \text{take } n \text{ list} ++ \text{drop } n \text{ list}$$

Next, we prove that the dropped part is exactly `[zero]`:

³⁶code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1194-L1199>

³⁷code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1204-L1205>

```

1 have drop_last: drop (size exps - 1) exps = [nth witness exps
2   (size exps - 1)].
3 smt(@List). smt(@List).

```

Listing 5.38: Proving the dropped suffix is a single zero ³⁸

Intuitively, `drop (size exps - 1) exps` removes all but the last element. For a list of size n , `drop (n-1)` results in a singleton list, which can be expressed as `[nth witness exps (n-1)]`, by hypothesis, this equals `[zero]`. Therefore, `exps = take (size exps - 1) exps ++ [zero]`.

Phase 2: Splitting the Base List Similarly, we perform the same splitting for the base list:

```

1 (* split bases into prefix and last element *)
2 have bases_split: bases = take (size bases - 1) bases ++
3   [nth witness bases (size bases - 1)].
4 rewrite -(cat_take_drop (size bases - 1)).
5 have drop_last_base: drop (size bases - 1) bases =
6   [nth witness bases (size bases - 1)].
7 smt(@List).
8 smt(@List).

```

Listing 5.39: Splitting bases into prefix and last element ³⁹

It tells:

$$\text{bases} = \text{take } (n - 1) \text{ bases} ++ [g_{n-1}]$$

where $g_{n-1} = \text{nth witness bases (size bases - 1)}$.

Phase 3: Rewriting the Goal After both splits established, we rewrite the goal to use these decompositions:

```

1 have tran : prodEx bases exps =
2   prodEx (take (size bases - 1) bases ++
3     [nth witness bases (size bases - 1)])
4     (take (size exps - 1) exps ++ [zero]).
5 smt().
6 rewrite tran.

```

Listing 5.40: Rewriting goal with split forms ⁴⁰

³⁸code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1206>

³⁹code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1210-L1212>

The goal now shows the structure we need to analyze:

```

1 prodEx (take (size bases - 1) bases ++ [nth witness bases (size
  bases - 1)])
2   (take (size exps - 1) exps ++ [zero]) =
3 prodEx (take (size bases - 1) bases) (take (size exps - 1) exps)

```

Listing 5.41: Rewriting goal with split forms ⁴¹

Phase 4: Zip Distribution and Simplification Finally, the key technical task in phase 4 is to distribute the `zip` operation across list concatenation.

Step 4.1: Zip Distribution

```

1 rewrite /prodEx /ex. rewrite -bases_split.
2 (* split zip and map *)
3 have zip_split:
4   zip bases (take (size exps - 1) exps ++ [zero]) =
5   zip (take (size bases - 1) bases) (take (size exps - 1) exps)
6   ++ zip (drop (size bases - 1) bases) [zero].

```

Listing 5.42: Distributing zip over concatenation ⁴²

This use the `zip` distributivity lemma:

$$\text{zip}(xs ++ ys, us ++ vs) = \text{zip}(xs, us) ++ \text{zip}(ys, vs)$$

where $\text{size } |xs| = |us|$ and $|ys| = |vs|$.

Then it require us verifying size constraints:

```

1 rewrite -zip_cat_distributive.
2 rewrite !size_take.
3 smt(). smt(). simplify. smt(). smt(@List).

```

Listing 5.43: Verifying size constraints for zip distribution ⁴³

Step 4.2: Splitting the Product With the `zip` split, we can now split the product:

```

1 rewrite zip_split.
2 rewrite map_cat.
3 rewrite prod_cat.

```

Listing 5.44: Splitting product over concatenation ⁴⁴

⁴⁰code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1213-L1214>

⁴²code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1215-L1219>

⁴³code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1221-L1223>

5 Evaluation : Formalizing the Bilateral Reduction

This uses standard lemmas: the first is derived from the standard easycrypt library and the second lemma 26:

- `map_cat`: $\text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys$
- `prod_cat`: $\text{prod } (xs ++ ys) = \text{prod } xs \cdot \text{prod } ys$

After these procedure, the goal becomes:

```

1
2 prod
3   (map (fun (i : group * GP.exp) => i.'1 ^ i.'2)
4     (zip (take (size bases - 1) bases) (take (size exps - 1)
5       exps))) *
6 prod
7   (map (fun (i : group * GP.exp) => i.'1 ^ i.'2)
8     (zip (drop (size bases - 1) bases) [zero])) =
9 prod
10  (map (fun (i : group * GP.exp) => i.'1 ^ i.'2)
11    (zip (take (size bases - 1) bases) (take (size exps - 1)
12      exps)))

```

Listing 5.45: Splitting product over concatenation ⁴⁵

Step 4.3: Eliminating the Zero Term Then we need to prove that the second product equals the identity using the following steps:

```

1 have : prod (map (fun (i : group * GP.exp) => i.'1 ^ i.'2)
2   (zip (drop (size bases - 1) bases) [zero])) = e.
3 have drop_single: drop (size bases - 1) bases =
4   [nth witness bases (size bases - 1)].
5   smt(@List).
6 rewrite drop_single.
7 simplify.

```

Listing 5.46: Proving the zero component equals identity ⁴⁶

After that, we need to show:

$$(\text{nth witness bases } (n - 1))^0 = e$$

By following calls:

```

1 have exp_zero: (nth witness bases (size bases - 1)) ^ zero = e.

```

⁴⁴code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1223>

⁴⁶code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1224-L1228>

5.9 Lessons Learned from case study and formalization procedure

```

2  smt(@G @GP).
3  rewrite exp_zero.
4  smt(@G @GP).

```

Listing 5.47: Applying group identity for zero exponent ⁴⁷

The SMT solver uses the group axiom: $\forall g \in G. g^0 = e$.

Step 4.4: Final Simplification With the second product equal to identity, we can say:

```

1  smt(@G @GP).

```

Listing 5.48: Final simplification using group identity ⁴⁸

This final step, the SMT solvers uses the group identity law:

$$a \cdot e = a \quad \text{for all } a \in G$$

Therefore, we can conclude the value of two side is equivalent.

5.9 Lessons Learned from case study and formalization procedure

In total, this two case studies illustrate two main parts of our work on the formal verification of the bilateral equivalence. But it can show my learning procedure during this project: attempting to prove lemmas, revealed missing definitions or lemmas, leading to refinement of the algebraic infrastructure. This iterative process, while time-consuming, led to a more robust and reusable framework, and do it repeatedly until we finished two-direction proof.

As we can learn from the case study, it is common to see operations considered "obvious" in paper proofs (e.g., "flatten nested products") require extensive formal justification, often requires dozens of proof steps than expected. In addition, even simple list operations require exhaustive case analysis. The EasyCrypt enforces completeness, preventing implicit assumptions, but it also shows the accuracy of this formalization.

5.10 Main Results

We present the first machine-checked verification of the bilateral reduction between IND-CCA1 security of ElGamal KEM and the q-DDH assumption in the Algebraic Group Model, formalizing the paper proof of Fuchsbauer, Kiltz, and Loss [1] in EasyCrypt.

⁴⁷code link :<https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec#L1230-L1231>

5.10.1 Lessons Learned from case study and formalization procedure

In total, the two case studies illustrate two main parts of our work on the formal verification of the bilateral equivalence. But it can show my learning procedure during this project: attempting to prove lemmas, revealed missing definitions or lemmas, leading to refinement of the algebraic infrastructure. This iterative process, while time-consuming, led to a more robust and reusable framework, and do it repeatedly until we finished two-direction proof.

As we can learn from the case study, it is common to see operations considered "obvious" in paper proofs (e.g., "flatten nested products") require extensive formal justification, often requires dozens of proof steps than expected. In addition, even simple list operations require exhaustive case analysis. The EasyCrypt enforces completeness, preventing implicit assumptions, but it also shows the accuracy of this formalization.

5.10.2 enhance the rigorous and the readability of algebraic reduction

Compared with A.3 of the paper [1], 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 formalization, 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. This coherence ensures that the formal proof not only eliminate the hidden flaws, but also offers a reproducible framework for future reductions and research.

5.10.3 show the power of Algebraic Group Model

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.

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, Fuchsbaauer et al. [1] 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.

5.10.4 Formalizing Algebraic Adversaries in EasyCrypt

One of the key results of our work is indicating how the special adversary model of the AGM can be captured fully using the EasyCrypt framework. The AGM is a notion that allows adversaries to perform group operations but requires they provide its algebraic representation whenever they output a new group element. In our work, the oracle enforces this restriction by checking whether $c = \text{prodEx } l \ z$ 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 establish a oracle interface that 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, ensuring 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.

Conclusion and Future work

6.1 Conclusion

This thesis presents a complete machine-checked formalization of the bilateral reduction between IND-CCA1 security of ElGamal-based KEM and the q-DDH assumption in the Algebraic Group Model, mechanizing the theoretical reduction established by Fuchsbauer, Kiltz, and Loss [1]. We have translated the high-level cryptographic arguments from their paper proof A.3 into over 2000 lines of verified EasyCrypt code, making explicit all reasoning steps that were left implicit in the original paper.

When it comes to the lessons I learned, the first is infrastructure development is crucial, A significant portion of our effort went into developing helper lemmas and infrastructure rather than directly proving the main theorem. Lemmas like `prodEx_split_last_zero`, `drop_sumv`, and `sumv_cons` were not initially available in the easycrypt but proved essential for the main proof. This teach me the significance of the infrastructure developing. In addition, it is important to make iterative proof development, it is frequent to observe initial attempts often revealed missing lemmas or inappropriate proof strategies required us to backtracking, which is an important formalization strategy within EasyCrypt

6.1.1 Scope and Limitations

What we contribute:

- A complete mechanization of the bilateral reduction from [1]
- Machine-checked correctness guarantees for both reduction directions
- A complete framework consists of useful algebraic reasoning lemmas
- Give method to future AGM formalization

What we do not contribute:

- Novel theoretical results in cryptography (the reduction is from [1])
- A general framework for all AGM proofs (our infrastructure is tailored to a specific reduction)
- Performance and efficiency improvements to EasyCrypt

6.1.2 Challenges and Effort in Mechanizing AGM-Based Proofs

While the bilateral reduction in Fuchsbauer, Kiltz, and Loss [1] spans approximately two pages in their appendix, its formalization in EasyCrypt required over 2,000 lines of verified code and several months of effort. This comparison shows the substantial gap between paper proofs and mechanical verification under the Algebraic Group Model (AGM).

To the best of our knowledge, our work represents the first attempt to formal verify AGM-based security arguments so that it almost provide no specialized support for our work. As a result, a majority of the development effort was tried to building the necessary framework for our proof, such as over thirty lemmas and several custom operators (`prodEx`, `sumv`, `scalev`, `shift_trunc`) to express and manipulate algebraic relationships between exponents and group elements. In particular, the straightforward arguments in original paper proof like “by algebraic manipulation” or “using the adversary’s representation” required divided into various intermediate lemmas to machine-checked prove.

For my personal experience, the entire developing process was both conceptually challenging and technically hard. I need to master EasyCrypt’s language and tactical system and designing a algebraic reasoning framework to support our bilateral reduction at the same time. During my developing process, I need to first understand the AGM concepts and find a efficient and accurate to encode it within EasyCrypt, then try to discover the missing lemmas necessary to make the current proof status machine-checkable, and repeatedly updating the foundational framework as the proof developed. The challenges not only highlights the difficulties of formally verify the AGM-related theorem within EasyCrypt, and also teach me the significant manual engineering required to build the foundational infrastructure for a new formal field. Nevertheless, we can say our formal verification is worth and justify our effort, that our work not only confirms correctness of the original paper proof but also provides a reusable framework for future researchers.

6.2 Future Directions

Lemma Library Refactoring Our linear algebra lemmas are currently embedded within the main proof file. A natural next step would be to refactor these into a well-documented module that could be imported by other AGM formalizations and we can make pull request to contribute to the EasyCrypt library.

Additional Reductions from [1] The paper by Fuchsbauer, Kiltz, and Loss contains several other AGM-based results that could potentially use similar formalization efforts. For instance, their proof that the algebraic Computational Diffie–Hellman (aCDH) assumption is equivalent to the standard CDH assumption might be possible to formalize in short term using the similar framework of our work. Other AGM-based results, such as the algebraic Strong Diffie–Hellman (aSH) assumption or algebraic LRSW assumptions, could be formalized using similar techniques. However, each new result might lead unique challenges, and we should not underestimate the effort required.

High-Assurance Applications and Future Directions. Modern cryptographic deployments in environments that security failures are unacceptable motivate extending formal verification to more complex AGM-based constructions. For instance, the zero-knowledge proof systems like PLONK [31], whose security analysis relies on the AGM, is a good instance of this need: PLONK has been deployed in production blockchain systems that process large quantity of dollars in financial transactions. For this kind of implementations, machine-checked security proofs provide particularly strong assurance.

While PLONK introduces substantial complexity beyond our ElGamal formalization, but our work shows that although the AGM mechanization is demanding, it is achievable. The framework we developed might be useful for future formalization efforts for these advanced constructions, and has potential to extend it into a broader cryptographic systems protecting substantial real-world value.

Hybrid Model Explorations Recent work explores computational models that combine algebraic assumptions with post-quantum hardness (e.g., Module-LWE [32] or isogeny-based systems [33]). Extending our framework to formalise hybrid arguments combining group-theoretic and lattice-based or isogeny-based reasoning presents a substantial challenge.

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. The full formal verification code is available at: <https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec>

A.1 Distributivity and Scaling Proofs

```

1 (* Distributivity: (prod gs)^n = prod (map (^n) gs) *)
2 lemma prod_exp_distributive (gs : group list) (n : exp) :
3   prod gs ^ n = prod (map (fun g => g ^ n) gs).
4 proof.
5   elim: gs => [|g gs IH] //=.
6   smt(@G @GP @List).
7   rewrite /= !prod_cons.
8   have exp_mult_dist: (g * prod gs) ^ n = g ^ n * (prod gs) ^ n.
9     smt(@G @GP).
10  rewrite exp_mult_dist. rewrite IH. smt().
11 qed.
12
13 (* Scaling lemma: prodEx with scaled exponents *)
14 lemma prodExScale1 bases exp scala :
15   prodEx bases exp ^ scala = prodEx bases (scalev exp scala).
16 proof.
17   rewrite /prodEx. rewrite /ex. rewrite !prod_exp_distributive.
18   congr.
19   rewrite /scalev. search zip. rewrite zip_mapr.
20   rewrite -!map_comp. congr. apply fun_ext => xy.
21   by case: xy => [x y] /=; rewrite /(\o) /= expM.
22 qed.

```

```

22
23 (* Alternative scaling lemma using ex *)
24 lemma prodExScale2 bases exp scale :
25   prodEx bases exp ^ scale = prodEx (ex bases (nseq (size
bases) scale)) exp.
26 proof.
27   rewrite prodExScale1. rewrite !/prodEx /ex. congr.
28   elim: bases exp => [|base_h base_t IH] [|exp_h exp_t] //=.
29   rewrite !zip_nil_l. smt().
30   rewrite !zip_nil_l. smt().
31   rewrite zip_nil_r. rewrite /scalev. rewrite zip_nil_r. smt().
32   rewrite /scalev /=. simplify.
33   have nseq_cons: nseq (1 + size base_t) scale = scale :: nseq
(size base_t) scale.
34   rewrite -nseqS. smt(size_ge0). smt(@G).
35   rewrite nseq_cons. simplify.
36   split. smt(@G @GP). rewrite IH. smt().
37 qed.
38
39 (* Addition distributivity for prodEx *)
40 lemma prod_ex_addv (base : group list) (a b : exp list) :
41   size a = size base => size b = size base =>
42   prod (ex base (addv a b)) = prod (ex base a) * prod (ex base
b).
43 proof.
44   move=> size_a size_b.
45   rewrite /addv.
46   move: a b size_a size_b.
47   elim: base.
48   move=> a b size_a size_b. rewrite /ex. rewrite !zip_nil_l.
49   by smt(@List @G).
50   move=> g gs IH a b size_a size_b.
51   case: a size_a => [|a_h a_t] size_a; first by smt(@List).
52   case: b size_b => [|b_h b_t] size_b; first by smt(@List).
53   rewrite /ex !zip_cons !map_cons !prod_cons /=.
54   rewrite prod_cons expD /ex. rewrite IH. smt(@G @GP). smt().
rewrite /ex.
55
56   pose A := prod (map (fun (i : group * GP.exp) => i.'1 ^ i.'2)
(zip gs a_t)).
57   pose B := prod (map (fun (i : group * GP.exp) => i.'1 ^ i.'2)
(zip gs b_t)).
58   pose X := g ^ a_h.
59   pose Y := g ^ b_h.
60   rewrite mulcA. rewrite mulcA. congr.
61
62   have com : Y * A = A * Y. rewrite mulcC. smt().
63   rewrite -mulcA. rewrite com.
64   smt(@G @GP).

```

```

65 qed.
66
67 (* ProdEx distributes over vector addition *)
68 lemma prodEx_addv_distributive (bases : group list) (a b : exp
list) :
69   size a = size bases => size b = size bases =>
70   prodEx bases (addv a b) = prodEx bases a * prodEx bases b.
71 proof.
72   move => ha hb.
73   rewrite /prodEx prod_ex_addv. smt(). smt(). smt().
74 qed.

```

Listing A.1: Proof of Product Exponentiation Distributivity

A.2 Zero Vector and Identity Proofs

```

1  (* ProdEx with all zero exponents is identity *)
2  lemma prodEx_nseq_zero (bases : group list) (n : int) :
3    0 <= n =>
4    prodEx bases (nseq n zero) = e.
5  proof.
6    move=> ge0_n.
7    rewrite /prodEx /ex.
8    elim: bases n ge0_n => [!g gs IH] n ge0_n.
9    - (* bases = [] *)
10     by rewrite zip_nil_l map_nil prod_nil.
11    - (* bases = g :: gs *)
12     case: (n = 0) => [n_zero | n_pos].
13     + (* n = 0 *)
14       rewrite n_zero nseq0.
15       by rewrite zip_nil_r map_nil prod_nil.
16     + (* n > 0 *)
17     have n_gt0: 0 < n by smt().
18     have nseq_cons: nseq n zero = zero :: nseq (n-1) zero.
19     rewrite -nseqS. smt(). simplify. trivial.
20     rewrite nseq_cons zip_cons map_cons prod_cons /= exp0 IH.
21     smt(). smt(@G @GP).
22 qed.
23
24 (* ProdEx with empty exponent list is identity *)
25 lemma prodEx_nil_r (bases : group list) :
26   prodEx bases [] = e.
27 proof.
28   rewrite /prodEx /ex.
29   rewrite zip_nil_r.
30   rewrite map_nil.
31   rewrite prod_nil.

```

```

32   trivial.
33 qed.
34
35 lemma prodEx_nil_1 (exps : exp list) :
36   prodEx [] exps = e.
37 proof.
38   rewrite /prodEx /ex.
39   rewrite zip_nil_1.
40   rewrite map_nil.
41   rewrite prod_nil.
42   trivial.
43 qed.
44
45 lemma prodEx_nil :
46   prodEx [] [] = e.
47 proof.
48   by rewrite prodEx_nil_1.
49 qed.

```

Listing A.2: Proof of Zero Vector Properties

A.3 Shift-Truncate Operation Proof

```

1  (* Shift-truncate lemma: key property for oracle simulation *)
2  lemma prodExShiftTrunc bases exps :
3    size bases = q + 1 =>
4    size exps = q + 1 =>
5    drop q exps = nseq 1 zero =>
6    prodEx bases (shift_trunc exps) = prodEx (behead bases)
7    exps.
8  proof.
9    move=> size_bases size_exps last_zero.
10   rewrite /shift_trunc.
11   have take_eq : (take (q+1) (zero :: exps)) = zero :: (take
12   q exps).
13   smt(@List gt0_q).
14   rewrite take_eq.
15   have exps_split: exps = take q exps ++ [zero].
16   rewrite -(cat_take_drop q).
17   have h : take q (take q exps ++ drop q exps) = take q exps.
18   have size_take_q : size(take q exps) = q by smt(@List
19   gt0_q).
20   rewrite take_size_cat. smt(). smt(). rewrite h.
21   rewrite last_zero. smt(@List @G @GP).
22   rewrite exps_split.
23   have size_take_q : size(take q exps) = q by smt(@List
24   gt0_q).

```

```

21     rewrite take_size_cat. smt().
22     have base_cons : (head witness bases) :: (behead bases) =
bases.
23     have : bases <> []. smt().
24     apply head_behead.
25     have h : (prodEx bases (zero :: take q exps)) =
26     prodEx (head witness bases :: behead bases) (zero :: take q
exps).
27     smt().
28     rewrite h.
29     rewrite prodExCons.
30     pose oversize := (take q exps ++ [zero]).
31     pose a := (behead bases).
32     rewrite (prodEx_sizele a oversize). smt().
33     have size_a: size a = q by smt(@G @List @GP gt0_q).
34     rewrite size_a. rewrite /oversize.
35     rewrite take_size_cat. smt(@List @G gt0_q). smt().
36 qed.

```

Listing A.3: Proof of Shift-Truncate Property

A.4 Vector Addition Properties

```

1  (* Vector addition with empty left operand *)
2  lemma addv_nil_l (b : exp list) :
3    addv [] b = [].
4  proof.
5    rewrite /addv.
6    rewrite zip_nil_l.
7    rewrite map_nil.
8    trivial.
9  qed.
10
11 (* Vector addition with empty right operand *)
12 lemma addv_nil_r (a : exp list) :
13   addv a [] = [].
14 proof.
15   rewrite /addv.
16   rewrite zip_nil_r.
17   rewrite map_nil.
18   trivial.
19 qed.
20
21 (* Vector addition with empty operand *)
22 lemma addv_empty (a b : exp list) :
23   a = [] \ / b = [] =>
24   addv a b = [].

```

```

25 proof.
26   case => [a_empty | b_empty].
27   - by rewrite a_empty addv_nil_l.
28   - by rewrite b_empty addv_nil_r.
29 qed.
30
31 (* Sum of empty vector list is zero vector *)
32 lemma sumv_nil :
33   sumv [] = zerov.
34 proof.
35   rewrite /sumv. simplify. smt().
36 qed.
37
38 (* sumv concat lemma*)
39 lemma sumv_cons (v : exp list) (vs : exp list list) :
40   sumv (v :: vs) = addv v (sumv vs).
41 proof.
42   rewrite /sumv. simplify. smt().
43 qed.

```

Listing A.4: Proof of Vector Addition Properties

A.5 Size Preservation Proofs

```

1 (* Size of scaled vector equals original size *)
2 lemma size_scalev (v : exp list) (s : exp) :
3   size (scalev v s) = size v.
4 proof.
5   rewrite /scalev.
6   by rewrite size_map.
7 qed.
8
9 (* Size of sum of vectors with uniform size *)
10 lemma size_sumv (l : exp list list) :
11   (forall v, v \in l => size v = q + 1) =>
12   size (sumv l) = q + 1.
13 proof.
14   elim: l => [|v vs IH] uniform_size.
15   - rewrite sumv_nil /zerov.
16     by rewrite size_nseq ler_maxr // gt0_q.
17   - rewrite sumv_cons.
18     have size_v: size v = q + 1.
19       by apply uniform_size; rewrite in_cons.
20     have size_sumv_vs: size (sumv vs) = q + 1.
21       apply IH => w w_in_vs.
22     by apply uniform_size; rewrite in_cons w_in_vs orbT.
23     rewrite /addv size_map size_zip.

```

```

24   by rewrite size_v size_sumv_vs minrr.
25 qed.

```

Listing A.5: Proof of Size Preservation Properties

A.6 prodEx cons Proofs

```

1  (* Cons with zero exponent lemma *)
2  lemma prodExCons bs e b :
3    prodEx (b :: bs) (zero :: e) = prodEx bs e.
4  proof.
5    rewrite /prodEx /ex. rewrite zip_cons. rewrite map_cons.
6    rewrite prod_cons.
7    rewrite /= exp0. smt(@G @GP @List).
8  qed.
9  (* General cons lemma for prodEx *)
10 lemma prodExConsGeneral (g : group) (gs : group list) (e : exp)
11   (es : exp list) :
12   prodEx (g :: gs) (e :: es) = g ^ e * prodEx gs es.
13 proof.
14   rewrite /prodEx /ex.
15   rewrite zip_cons map_cons prod_cons.
16   trivial.
17 qed.

```

Listing A.6: Proof of Size Preservation Properties

A.7 prodEx sizele Proofs

```

1  (* ProdEx with oversized exponent list can be truncated *)
2  lemma prodEx_sizele a b :
3    size a < size b => prodEx a b = prodEx a (take (size a) b).
4  proof.
5    move => size_neq.
6    rewrite /prodEx /ex.
7    congr. congr.
8    elim: a b size_neq => [|a_h a_t IH] [|b_h b_t] // = size_lt.
9    have h : !(1 + size a_t <= 0) by smt(size_ge0). rewrite h.
10   simplify. rewrite IH. smt(). smt().
11 qed.

```

Listing A.7: Proof of Size Preservation Properties

A.8 sumv cons Proofs

```

1 lemma sumv_cons (v : exp list) (vs : exp list list) :
2   sumv (v :: vs) = addv v (sumv vs).
3 proof.
4   rewrite /sumv. simplify. smt().
5 qed.

```

Listing A.8: Proof of Size Preservation Properties

A.9 prodEx map Proofs

```

1 (* Double prodEx composition *)
2 lemma prodExEx b e1 e2 :
3   prodEx (ex b e1) e2 = prodEx b (mulv e1 e2).
4 proof.
5   rewrite /prodEx /ex /mulv. rewrite zip_mapl. rewrite zip_mapr.
6   rewrite -!map_comp. congr. rewrite /(\o) /=.
7   elim: b e1 e2 => [!g_h g_t IH] [!e1_h e1_t] [!e2_h e2_t] //.
8   split. smt(@G @GP). by apply IH.
9 qed.
10
11
12
13 (* Ex with map and constant scaling *)
14 lemma ex_map_prodEx bases exps size_bases size_exps c :
15   size_bases = size bases => size_exps = size exps =>
16   ex (map (prodEx bases) exps) (nseq size_exps c) =
17   map (prodEx (ex bases (nseq size_bases c))) (exps).
18 proof.
19   move => h h0. apply (eq_from_nth g). smt(@List).
20   move => i hi. rewrite /ex.
21   rewrite (nth_map (witness, witness) g). smt(@List size_ge0).
22   rewrite nth_zip. smt(size_ge0 @List). simplify.
23   rewrite (nth_map witness). smt(size_ge0 @List).
24   rewrite nth_nseq. smt(size_ge0 @List).
25
26   have inner_ex:
27     map (fun (i0 : group * GP.exp) => i0.'1 ^ i0.'2) (zip bases
28       (nseq size_bases c)) =
29     ex bases (nseq size_bases c).
30     rewrite /ex //.
31   rewrite inner_ex.
32   rewrite (nth_map (witness ) g). smt(size_ge0 @List).
33   rewrite h.
34   rewrite -prodExScale2. smt().

```

```

34 qed.
35
36 (* Map exponentiation with range *)
37 lemma prodExMap g (x : exp) (s e : int) :
38   map (fun i => g ^ exp x i) (range s e) =
39   ex (nseq (e-s) g) (map (fun i => exp x i) (range s e)).
40 proof.
41   apply (eq_from_nth g). smt(@List).
42   move => i hi. rewrite (nth_map 0). smt(@List).
43   move => @/ex. rewrite (nth_map (g,zero)). smt(@List).
44   simplify. smt(@List).
45 qed.

```

Listing A.9: Proof of Size Preservation Properties

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. The full formal verification code is available at: <https://github.com/GuSheldom/INDCCA-QDDH/blob/main/INDCCA.ec>

B.1 Range Simplification Proofs

```

1 (* Simplification: prepending g to powers equals mapping from 0 *)
2 lemma q0_simp (sk : exp) :
3   (g :: map (fun (i : int) => g ^ exp sk (i))(range 1 (q+1
4 ))) =
5   map (fun (i : int) => g ^ exp sk (i))(range 0 (q + 1)).
6 proof.
7   have range_split: range 0 (q + 1) = 0 :: range 1 (q + 1).
8   smt(@ G @GP @List gt0_q). rewrite range_split map_cons.
9   congr. smt(@G @GP).
10  qed.

```

Listing B.1: Proof of Range Simplification

B.2 Ex Cons Distribution Proofs

```

1 (* Ex operation distributes over cons structure *)
2 lemma ex_cons_general:

```

```

3 forall (x : group) (xs : group list) (e : ZModE.exp) (es :
  ZModE.exp list),
4   ex (x :: xs) (e :: es) = (x ^ e) :: ex xs es.
5 proof.
6   move => x xs e es .
7   rewrite /ex /=. trivial.
8 qed.

```

Listing B.2: Proof of Ex Cons Distribution

B.3 Ex Range Shift Property Proofs

```

1 (* Ex with range shift property *)
2 lemma ex_range_shift (sk : exp) :
3   ex (map (fun (i : int) => g ^ exp sk i) (range 0 (q + 1)))
  (nseq (q + 1) sk) =
4   map (fun (i : int) => g ^ exp sk i) (range 1 (q + 2)).
5
6 proof.
7   rewrite /ex.
8   have Hlen: size (range 0 (q + 1)) = size (nseq (q + 1) sk) by
  rewrite size_range size_nseq.
9   rewrite zip_map1 -map_comp /(\o).
10  apply (eq_from_nth witness).
11  rewrite size_map size_zip size_range size_nseq.
12  smt(@List @G @GP).
13  move=> i hi.
14  rewrite !(nth_map witness) //. smt(@List). smt(@List @G @GP).
15  case (0 <= i < q + 1) => [valid_i|]; last smt(@List).
16  simplify.
17  have zip_nth: nth witness (zip (range 0 (q + 1)) (nseq (q + 1)
  sk)) i = (i, sk).
18  have range_i: nth witness (range 0 (q + 1)) i = i.
19  rewrite nth_range //.
20  have nseq_i: nth witness (nseq (q + 1) sk) i = sk.
21  rewrite nth_nseq_if valid_i. smt().
22  (* Zip structure property: nth element of zip equals pair of
  nth elements *)
23  have zip_structure: forall j, 0 <= j < min (size (range 0 (q +
  1))) (size (nseq (q + 1) sk)) =>
24    nth witness (zip (range 0 (q + 1)) (nseq (q + 1) sk)) j =
25    (nth witness (range 0 (q + 1)) j, nth witness (nseq (q + 1) sk)
  j).
26  move=> j hj. smt(@List).
27  rewrite zip_structure. smt(@List @G @GP).
28  rewrite range_i nseq_i. smt().
29  rewrite zip_nth. rewrite /= nth_range //.

```

```

30   simplify. rewrite -expM. congr.
31   smt(@G @GP @ZModE).
32 qed.

```

Listing B.3: Proof of Ex Range Shift Property

B.4 Scalev of Nseq Proofs

```

1  (** Scalev of nseq **)
2  lemma scalev_nseq (n : int) (x c : ZModE.exp) :
3    scalev (nseq n x) c = nseq n (x * c).
4  proof.
5    rewrite /scalev.
6    by rewrite map_nseq.
7  qed.
8
9  (* Scalev of nseq 0 *)
10 lemma scalev_nseq_zero (n : int) (c : ZModE.exp) :
11   scalev (nseq n zero) c = nseq n zero.
12 proof.
13   rewrite scalev_nseq. smt(@ZModE).
14 qed.

```

Listing B.4: Proof of Scalev of Nseq

B.5 Drop-Addv Commutativity Proofs

```

1  (** Drop commutes with addv **)
2  lemma drop_addv (n : int) (u v : ZModE.exp list) :
3    size u = size v => drop n (addv u v) = addv (drop n u) (drop n
4    v).
5    proof.
6      move => size_eq.
7      elim: u v n size_eq => [|u_head u_tail IH] [|v_head v_tail] n
8      //=. smt(). smt().
9      move=> size_eq.
10     case: (n <= 0) => [le0_n|ltzNge gt0_n]. smt(). smt().
11 qed.

```

Listing B.5: Proof of Drop-Addv Commutativity

B.6 Addv Size Inequality Proofs

```

1 (* Addv with size inequality a <= b*)
2 lemma addv_neq a b :
3   size a <= size b => addv a b = addv a (take (size a) b).
4   proof.
5     move=> size_lt.
6     rewrite /addv.
7     congr. elim: a b size_lt => [|x xs IH] [|y ys] // = size_lt.
8   smt().
9 qed.

```

Listing B.6: Proof of Addv Size Inequality

B.7 Drop-Sumv Commutativity Proofs

```

1 (* Drop commutes with sumv: dropping elements from sum equals sum
2   of dropped elements
3   This is a key lemma for vector operations in the reduction,
4   showing that
5   drop and sumv operations can be interchanged under certain
6   conditions. *)
7 lemma drop_sumv (n : int) (xs : ZModE.exp list list) :
8   xs <> [] => (* xs is non-empty *)
9   all (fun v => size v = q + 1) xs => (* all vectors have
10   uniform size q+1 *)
11   0 <= n <= q => (* drop index is valid
12   *)
13   drop n (sumv xs) = sumv (map (drop n) xs).
14 proof.
15   move=> xs_nonempty all_size_eq [ge0_n len_q].
16   (* Establish that xs has at least one element *)
17   have xs_largeT : 1 <= size xs by smt(@List).
18   move : xs_largeT.
19   (* Induction on the list xs *)
20   elim: xs xs_nonempty all_size_eq => [|v vs IH] xs_nonempty
21   all_size.
22   - (* Base case: xs = [] - contradiction with non-empty
23     assumption *)
24     by [].
25   (* Inductive case: xs = v :: vs *)
26   have v_size: size v = q + 1 by smt(@List ).
27   (* v has size q+1 *)
28   have vs_all_size: all (fun u => size u = q + 1) vs by
29   smt(@List). (* vs elements have size q+1 *)
30
31   (* Rewrite using sumv_cons and map_cons *)
32   rewrite sumv_cons map_cons sumv_cons.

```

```

24  (* Key step: drop commutes with addv since both operands have
    same size *)
25  rewrite drop_addv. rewrite size_sumv. smt(@List @ZModE ).
    smt().
26  move=> size_ge1.
27  have : 0<= size vs. smt(@List). move => size_ge0.
28  (* Case analysis on whether vs is empty or not
    Case 1 : vs has at least 1 elements *)
29  case: (1 <= size vs ) => [vs_ge1|vs_eq0]. rewrite IH.
30  smt(). smt(). smt(). trivial.
31  (* Case 2: vs is empty *)
32  have vs_empty : vs = [] by smt(@List). rewrite vs_empty map_nil.
33  (* When vs is empty, sumv vs = zerov *)
34  rewrite sumv_nil. rewrite /zerov.
35  (* Use addv_neq to handle size mismatch between drop n v and
    zerov *)
36  rewrite (addv_neq (drop n v) (nseq (q + 1) zero)).
37  rewrite size_drop. smt(). rewrite v_size size_nseq. have rev :
    0 < q + 1 - n by smt().
38  rewrite !StdOrder.IntOrder.ler_maxr. smt(). smt(). smt().
39  (* Establish size equality for the final step *)
40  have : size (drop n v) = q + 1 - n. rewrite size_drop. smt().
    rewrite v_size. smt(@GP @G).
41  move => size_dd. rewrite size_dd. congr.
42  (* Final simplification using drop_nseq_eq_take *)
43  rewrite drop_nseq_eq_take. smt(). trivial.
44  qed.

```

Listing B.7: Proof of Drop-Sumv Commutativity

B.8 Sumv of Zero Vectors Proofs

```

1  (** Sum of n copies of zero vector equals zero vector
2     This lemma establishes that summing any number of zero
3     vectors gives the zero vector,
4     which is fundamental for vector arithmetic in the reduction.
5     **)
6
7  lemma sumv_nseq_zerov (n : int) :
8    0 <= n =>
9    sumv (nseq n zerov) = zerov.
10 proof.
11   elim/natind: n => [n le0_n|n ge0_n IH].
12   - (* Base case: n <= 0 *)
13     move=> _ .
14     (* When n <= 0, nseq gives empty list, sumv of empty list is
15        zerov *)

```

```

13     rewrite nseq0_le // sumv_nil //.
14
15     (* Inductive case: n >= 0 *)
16     move=> ge0_n1.
17     rewrite nseqS // sumv_cons. rewrite IH //.
18     have addv_zero_identity: addv zerov zerov = zerov.
19     rewrite /addv /zerov.
20     (* Prove equality by showing each element is equal *)
21     apply (eq_from_nth zero).
22     - rewrite !size_map !size_zip !size_nseq. smt().
23     - move=> i hi.
24       rewrite size_map size_zip !size_nseq in hi.
25       have hi_simplified: 0 <= i < q + 1. smt(). rewrite (nth_map
(zero, zero)) //.
26       rewrite size_zip !size_nseq. smt().
27       (* Each element of zip is (zero, zero), and zero + zero =
zero *)
28       rewrite nth_zip //. rewrite /=.
29       rewrite !nth_nseq_if. simplify. smt(@GP @ZModE @G).
30       (* Apply the zero identity to complete the proof *)
31       rewrite addv_zero_identity. smt().
32 qed.

```

Listing B.8: Proof of Sumv of Zero Vectors

B.9 Sumv of Singleton Zero Vectors Proofs

```

1  (** Sum of n singleton zero vectors equals one singleton zero
2  vector
3  This lemma shows that summing any positive number of
4  singleton zero vectors
5  [nseq 1 zero] results in a single singleton zero vector. This
6  is important
7  for handling uniform-sized vector operations in the
8  reduction. **)
9  lemma sumv_nseq_zero_singleton (n : int) :
10     1 <= n =>
11     sumv (nseq n (nseq 1 zero)) = nseq 1 zero.
12 proof.
13   move=> ge1_n.
14   (* Induction on n *)
15   elim/natind: n ge1_n => [m le0_m | m ge0_m IH] ge1_n.
16   - (* Base case: m <= 0 *)
17     (* Contradiction: we have 1 <= n and n = m, but m <= 0 *)
18     smt().
19   - (* Inductive case: m >= 0 *)

```

```

17  case: (m = 0) => [m_eq_0|m_neq_0].
18
19  + (* Case m = 0, so n = 1 *)
20  rewrite m_eq_0. rewrite nseqS. smt(). rewrite nseq0. rewrite
sumv_cons sumv_nil. rewrite /addv.
21  rewrite /zerov. have : (zip (nseq 1 zero) (nseq (q + 1) zero))
= (zip (nseq 1 zero) (nseq (1) zero)).
22  rewrite nseqS. smt(gt0_q). rewrite nseq1. rewrite zip_cons.
rewrite zip_nil_1. smt().
23
24  move => H. rewrite H. apply (eq_from_nth zero). rewrite size_map
size_zip !size_nseq.
25
26  smt(). move=> i hi.
27  rewrite size_map size_zip !size_nseq in hi.
28  (* hi: 0 <= i < 1, i = 0 *)
29  have i_eq_0: i = 0 by smt().
30  rewrite i_eq_0. rewrite (nth_map (zero, zero)) //.
31  + (* prove index valid *)
32  rewrite size_zip !size_nseq. smt(). rewrite !nth_zip //.
33  rewrite /= !nth_nseq_if /=.
34  smt(@ZModE). have gel_m: 1 <= m by smt(). rewrite nseqS //.
35
36
37
38  rewrite sumv_cons. rewrite IH. smt(). rewrite /addv.
39  apply (eq_from_nth zero).
40  rewrite !size_map !size_zip !size_nseq.
41  smt().
42  move=> i hi.
43  rewrite size_map size_zip !size_nseq in hi.
44  have i_eq_0: i = 0 by smt().
45  rewrite i_eq_0.
46  rewrite (nth_map (zero, zero)) //.
47  + rewrite size_zip !size_nseq. smt().
48
49  rewrite nth_zip //. rewrite /= !nth_nseq_if /=.
50  smt(@ZModE).
51  qed.

```

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

B.10 Zip Concatenation Distributivity Proofs

```

1 lemma zip_cat_distributive (a1 a2 : 'a list) (b1 b2 : 'b list) :
2   size a1 = size b1 =>
3   zip (a1 ++ a2) (b1 ++ b2) = zip a1 b1 ++ zip a2 b2.

```

```

4 proof.
5   move=> size_eq.
6   apply (eq_from_nth witness).
7
8   - (* size equal *)
9     rewrite size_zip size_cat. smt(@G @GP @List).
10  - (* equal each element *)
11    move=> i hi.
12    rewrite size_zip size_cat in hi.
13
14    (* discuss the locate of i *)
15    case: (i < size a1) => [i_lt_a1|i_ge_a1]. rewrite nth_cat.
16    - smt(@List). smt(@List).
17
18 qed.

```

Listing B.10: Proof of Zip Concatenation Distributivity

B.11 Product Concatenation Proofs

```

1 lemma prod_cat (xs ys : group list) :
2   prod (xs ++ ys) = prod xs * prod ys.
3 proof.
4   elim: xs => [|x xs IH|.
5     - (* Base case: xs = [] *)
6       rewrite cat0s.
7       (* prod ([] ++ ys) = prod ys *)
8       (* prod [] * prod ys = 1 * prod ys = prod ys *)
9       rewrite prod_nil.
10      smt(@G @GP).
11    - (* Inductive case: xs = x :: xs *)
12      rewrite cat_cons.
13      (* prod ((x :: xs) ++ ys) = prod (x :: (xs ++ ys)) *)
14      rewrite prod_cons.
15      (* = x * prod (xs ++ ys) *)
16      rewrite IH.
17      (* = x * (prod xs * prod ys) *)
18      rewrite prod_cons.
19      (* prod (x :: xs) * prod ys = (x * prod xs) * prod ys *)
20      algebra.
21 qed.

```

Listing B.11: Proof of Product Concatenation

B.12 prodEx Split with Last Zero Proofs

```

1  (** Product split with last zero exponent
2      This lemma shows that when the last exponent in a list is
3      zero, the product
4      can be computed by ignoring the last base-exponent pair. This
5      is crucial for
6      oracle simulation where we need to handle cases with trailing
7      zero exponents.
8
9      Intuitively: prodEx([g1, g2, ..., gn], [a1, a2, ..., an-1, 0])
10     = prodEx([g1, g2, ..., gn-1], [a1, a2, ..., an-1])
11
12     This property allows the reduction to efficiently handle
13     representations
14     that have zero coefficients in the last position. **)
15 lemma prodEx_split_last_zero (bases : group list) (exps :
16   ZModE.exp list) :
17   size bases = size exps =>
18   0 <size exps =>
19   nth witness exps (size exps - 1) = zero =>
20   prodEx bases exps =
21   prodEx (take (size bases - 1) bases) (take (size exps - 1)
22     exps).
23 proof.
24   move=> size_eq size_gt0 last_zero.
25
26   (* split exps into prefix and last element*)
27   have exps_split: exps = take (size exps - 1) exps ++ [zero].
28   rewrite -(cat_take_drop (size exps - 1)).
29   have drop_last: drop (size exps - 1) exps = [nth witness exps
30     (size exps - 1)]. smt(@List). smt(@List).
31
32   (* split bases into prefix and last element *)
33   have bases_split: bases = take (size bases - 1) bases ++ [nth
34     witness bases (size bases - 1)].
35   rewrite -(cat_take_drop (size bases - 1)).
36   have drop_last_base: drop (size bases - 1) bases = [nth witness
37     bases (size bases - 1)]. smt(@List). smt(@List).
38   have tran : prodEx bases exps = prodEx (take (size bases - 1)
39     bases ++
40       [nth witness bases (size bases - 1)]) (take (size
41     exps - 1) exps ++ [zero]). smt(). rewrite tran.
42   rewrite /prodEx /ex. rewrite -bases_split.
43   (* split zip and map *)
44   have zip_split: zip bases (take (size exps - 1) exps ++ [zero])
45   =

```

```

35         zip (take (size bases - 1) bases) (take (size exps
- 1) exps)
36         ++ zip (drop (size bases - 1) bases) [zero].
37
38     rewrite -zip_cat_distributive. rewrite !size_take.
39
40     smt(). smt(). simplify. smt(). smt(@List). rewrite zip_split.
rewrite map_cat. rewrite prod_cat.
41     have : prod (map (fun (i : group * GP.exp) => i.'1 ^ i.'2)
42         (zip (drop (size bases - 1) bases) [zero])) = e.
43     have drop_single: drop (size bases - 1) bases = [nth witness
bases (size bases - 1)].
44     smt(@List). rewrite drop_single.
45     simplify. search exp.
46
47     have exp_zero: (nth witness bases (size bases - 1)) ^ zero = e.
smt(@G @GP).
48     rewrite exp_zero. smt(@G @GP). smt(@G @GP).
49
50 qed.

```

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

B.13 Behead-Drop Equivalence Proofs

```

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

```

Listing B.13: Proof of Behead-Drop Equivalence

B.14 Advv Nth Distribution Proofs

```

1 (* distributive of nth and advv*)
2 lemma advv_nth (a b : ZModE.exp list) (pos : int) :
3     size a = size b =>
4     0 <= pos < size a =>

```

B.14 Addv Nth Distribution Proofs

```
5   nth witness (addv a b) pos = nth witness a pos + nth witness b
   pos.
6 proof.
7   move=> size_eq pos_valid.
8   rewrite /addv. rewrite(nth_map witness witness). smt(@List).
   smt( @List).
9 qed.
```

Listing B.14: Proof of Addv Nth Distribution

Bibliography

- [1] 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 1, 2, 3, 4, 5, 8, 9, 10, 12, 13, 17, 18, 19, 23, 37, 63, 64, 67, 68, and 69.]
- [2] 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.]
- [3] Manuel Fersch. *The provable security of elgamal-type signature schemes*. PhD thesis, Dissertation, Bochum, Ruhr-Universität Bochum, 2018, 2018. [Cited on page 2.]
- [4] 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 and 17.]
- [5] 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 3 and 8.]
- [6] Tatsuaki Okamoto. Authenticated key exchange and key encapsulation in the standard model. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 474–484. Springer, 2007. [Cited on page 7.]
- [7] 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 pages 7 and 10.]

Bibliography

- [8] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Annual international cryptology conference*, pages 232–249. Springer, 1993. [Cited on page 7.]
- [9] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 256–266. Springer, 1997. [Cited on page 8.]
- [10] 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 9.]
- [11] Steven D Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012. [Cited on page 9.]
- [12] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. doi: 10.1109/TIT.1985.1057074. [Cited on pages 10 and 19.]
- [13] 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 10.]
- [14] Per Bjesse. What is formal verification? *ACM Sigda Newsletter*, 35(24):1–es, 2005. [Cited on page 13.]
- [15] Manuel Barbosa, Andreas Hülsing, Matthias Meijers, and Peter Schwabe. Formal verification of post-quantum cryptography. NIST PQC Standardization Conference, 2024. URL <https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/meijers-formal-verification-pqc2021.pdf>. Presentation slides. [Cited on page 13.]
- [16] 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 13, 14, and 17.]
- [17] Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Formal verification of saber’s public-key encryption scheme in easycrypt. In *Annual International Cryptology Conference*, pages 622–653. Springer, 2022. [Cited on page 17.]
- [18] João Bártolo Almeida, Sergio A. Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Victor Laporte, Jean-Christophe Lécenet, 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 page 17.]
- [19] Manuel Barbosa, François Dupressoir, Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. A tight security proof for SPHINCS⁺, formally verified. Cryptology ePrint Archive, Paper 2024/910, 2024. URL <https://eprint.iacr.org/2024/910>. [Cited on page 18.]
- [20] Andreas Hülsing. Sphincs. 2020. [Cited on page 18.]
- [21] Mikhail Kudinov, Andreas Hülsing, Eyal Ronen, and Eylon Yogev. SPHINCS+c: Compressing SPHINCS+ with (almost) no cost. Cryptology ePrint Archive, Paper 2022/778, 2022. URL <https://eprint.iacr.org/2022/778>. [Cited on page 18.]
- [22] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed elgamal encryption in the algebraic group model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 63–95. Springer, 2020. [Cited on page 19.]
- [23] Alessandro Faonio, Daniele Fiore, Markulf Kohlweiss, Luca Russo, and Michał Zająć. 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 19.]
- [24] 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 19.]
- [25] 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 20.]
- [26] 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 20.]
- [27] 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 20.]

Bibliography

- [28] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014. ISBN 1466570261. [Cited on page 23.]
- [29] 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 23.]
- [30] 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 26.]
- [31] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. URL <https://eprint.iacr.org/2019/953>. [Cited on page 69.]
- [32] 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 69.]
- [33] Luca De Feo. Mathematics of isogeny based cryptography. *arXiv preprint arXiv:1711.04062*, 2017. [Cited on page 69.]