Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Ross Juvik

A Memory-Tight Reduction for the Twin Hashed ElGamal

Master's thesis in Mathematical Sciences
Supervisor: Jiaxin Pan
June 2022

NTNU
Kunnskap for en bedre verden

Ross Juvik

A Memory-Tight Reduction for the Twin Hashed ElGamal

Master’s thesis in Mathematical Sciences
Supervisor: Jiaxin Pan
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Abstract

Memory efficiency has been proven to be an important complexity parameter for black-box reductions when the cryptographic primitive in question is memory sensitive. New techniques for achieving memory-tight implementations have recently been introduced, most notably the *injectively-map then prf* technique of Bhattacharyya. This technique is used for simulating random oracles in the security proof of the Cramer-Shoup version of the Hashed ElGamal scheme in the IND-CCA security game. In this thesis we take this technique and apply it to the Twin Hashed ElGamal scheme achieving a memory-tight proof with security equivalent to the CDH problem in the random oracle model with only a small efficiency penalty. Furthermore we also attempt to generalize the technique for any Key Encapsulation Mechanisms based on Hash Proof Systems, but are unable to do this for the ElGamal case. We do however in our attempt show that this scheme implemented with the ElGamal HPS is memory-tight, albeit much less efficient than the Cramer-Shoup equivalent.

Keywords: Memory-tight Reduction, Hashed ElGamal, Twin Hashed ElGamal, Hash Proof System

Sammendrag

I denne oppgaven ser vi på minnetetthet som en betingelse for sikkerhet i reduksjonsbaserte sikkerhetsbeviser. Minneeffektivitet er en viktig kompleksitetsparameter for black-box-reduksjoner når det aktuelle kryptografiske primitivet er minnesensitivt. Bhattacharyya har nylig introdusert en teknikk for å simulere tilfeldige orakler i sikkerhetsbeviset til Cramer-Shoup-versjonen av Hashed ElGamal-chifferet i IND-CCA-sikkerhetsspillet. I oppgaven bruker vi denne teknikken på Twin Hashed ElGamal-chifferet og oppnår et minnetett reduksjonsbevis med sikkerhet tilsvarende CDH-problemet i den tilfeldige orakelmodellen. Videre forsøker vi også å generalisere teknikken for alle KEM basert på avtrykksikre systemer, men klarer ikke å gjøre dette for ElGamal tilfellet. Vi klarer likevel å vise at den er minnetett.

Acknowledgments

First of all I would like to thank my supervisor Jiaxin. I really appreciate the enthusiasm you showed me whenever I had a breakthrough in my work, however small it must have seemed to you. I also would like to thank my mother Gillian. Without your steady guidance and stalwart belief in me I wouldn't even have started studying, let alone finished. I will always be grateful for you. Also you did proof read, which for the record I am grateful for that too. Finally I would like to thank my friends for keeping me adequately distracted, in particular Lesesal 395b. You have all given me many memories that I will treasure greatly, and know that some of you have become quite dear to me.

This marks the end of my time as a student and in searching for the words to describe this experience I stumbled upon the following: *Meaningful growth requires challenge and stress.* If that's the case then boy have I had a lot of meaningful growth.

Ross Juvik
Trondheim, 2022

List of Abbreviations

In order of appearance

Indistinguishability under Chosen Ciphertext Attack	IND-CCA
The Computational Diffie-Hellman	CDH
Hash Proof System	HPS
Key Encapsulation Mechanism	KEM
The Hashed ElGamal Scheme	HEG
The Discrete Logarithm	Dlog
Diffie-Hellman	DH
The Gap Diffie-Hellman	GAP-DH
Chosen Ciphertext Attack	CCA
Public Key Encryption	PKE
The Twin Hashed ElGamal	TEG
The Hash Proof System Hashed ElGamal	HPS-HEG
The Decisional Diffie-Hellman	DDH
The Strong Twin Diffie-Hellman	STDH
Pseudo Random Function	PRF
The Subset Membership	SM
Indistinguishability under Chosen Plaintext Attack	IND-CPA
Probabilistic Polynomial-Time	PPT
UnForgeable Chosen Message Attack	UF-CMA
Message AuthentiCator	MAC
Chosen Plaintext Attack	CPA
The Strong Diffie-Hellman	Strong-DH
The Oracle Diffie-Hellman	ODH

Contents

1 Introduction

Cryptographic reductions aim to be tight by transforming an adversary into an algorithm with essentially the same resources as the initial adversary. This tightness of a reduction is then the measure of all resources of the algorithm and how well they relate to those of the adversary. Typically resources such as runtime and success probability are considered. So for an adversary $\mathcal{A}$, we transform $\mathcal{A}$ into an algorithm for solving a problem, say P, similar to the scheme S. For this reduction to be probability-tight we need the following: If ε_S is the probability of $\mathcal{A}$ breaking the secure scheme S, we want $\varepsilon_P \approx \varepsilon_S$, where ε_P is the probability of the algorithm solving the problem P. We can by choosing relevant parameters and showing that these are tight on the given reduction, ensure that the scheme S is secure against the adversary $\mathcal{A}$ in a meaningful way.

In 2018 Auerbatch et al published their work on memory-tight reductions [ACFK17]. Therein they showed the importance of memory usage as a complexity parameter. A reduction that is memory loose can solve memory sensitive problems easier and quicker than the original adversary, meaning that one cannot rule out the possibility of the existence of an even more efficient adversary than first assumed. This of course leads us to conclude that the scheme in question must be further *complicated*, often by increasing the sizes of ciphertexts, in order to make sure it is secure against the initial adversary in terms of provable security through black box reductions. The hope would be to conclude that no better adversary exists for the original security parameter. In addition to shedding light on this overlooked complexity parameter Auerbatch et al also suggested some general improvements to reduce memory leakage for reductions. However useful these improvements are in simpler schemes most of the existing results are unfortunately lower bounds on memory. Indeed they also conjectured that no memory-tight reduction of the Hashed ElGamal scheme could exists. Yet in 2020 Rishiraj Bhattacharyya disproved this conjecture using what they called *injectively-map then prf* technique, [Bha20]. Almost simultaneously Ashrujit Ghoshal and Stefano Tessaro proved a claim that further supported the conjecture, showing that memory usage of all reductions of the HEG scheme must be lower bounded, [GT21]. Not very encouraging news for memory-tightness' reputation as important and worth of study. The main problem in regards to the HEG scheme is the way the cryptographic Hash function is modelled and how a reduction must simulate this in the random oracle model. It turns out that the two schemes are different in a subtle but very important way, but both are referred to as HEG. To specify the HEG scheme for which the *injectively-map then prf* idea of Bhattacharyya works, is in fact the Cramer-Shoup variant of the original HEG, as presented in [CS01]. It is also worth mentioning that Bhattacharyya's work is done over groups with pairings, while Goshal and Tessaro have shown their lower bound in the generic group model. In short, both the work of Bhattacharyya and Goshal et al are true, rather than contradict each other, they complement each other.

An important note to make in the pursuit of memory efficient adversaries is that it does not necessarily make sense to impose such efficiency measurements on all pre-existing schemes and problems. A big part of why memory often has been overlooked in the past is that it is not evident when it is necessary. Problems that can be shown to be solved faster when an algorithm has access to greater memory, are called *memory sensitive*. It is for such memory sensitive problems that it is relevant to impose security proofs with this

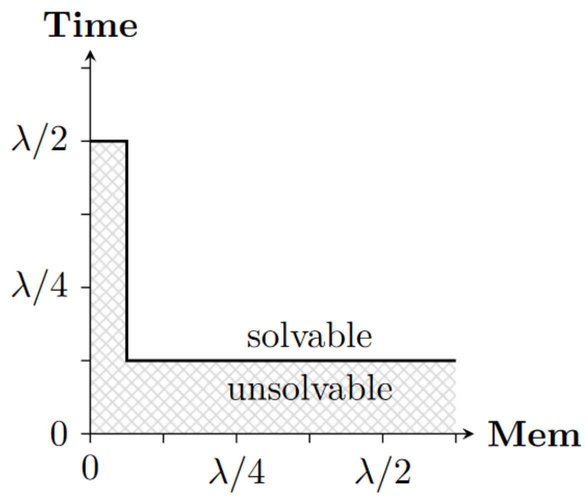

Figure 1: Graph showing impact of increased memory for an adversary against the Dlog assumption running the best known algorithms for breaking Dlog [LLMP93, Pol75]. The time and memory axes are in log scales and $\lambda = 512$. The gray area represents the time and memory combinations that we know no efficient adversaries break Dlog, while the "solvable" part we can not guarantee the non-existence of such adversaries. Original graph from [ACFK17]

additional requirement. There are many examples of memory sensitive problems, some of them are essential primitives in modern day cryptography. The Discrete Logarithm assumption, Dlog for short [Sho97], the hardness of which is an essential building block for the Diffie-Hellman assumptions, and much much more, is memory sensitive over finite prime fields. It is quite drastically so that if an algorithm $\mathcal{A}$ has expected runtime $\mathcal{O}(2^{\lambda/2})$ against the Dlog assumption with working memory of less than $\mathcal{O}(2^{\lambda/10})$, see Figure 1, it can be shown to solve the problem in time $\mathcal{O}(2^{\lambda/5})$ by only slightly increasing the working memory. Thus one can have an instance where by doubling the available memory of $\mathcal{A}$, it solves the Dlog problem in less than half the expected time. On the other hand 2-collision resistance of a Hash function is not memory sensitive, but the classical way of proving tightness for reductions in the random oracle model is to implement these Hash functions by *lazy sampling*, [BR06], leading to increased memory usage. This is unfavourable if the scheme itself is a memory sensitive problem, leading us to conclude that it would be favourable to have some idea of how one could impose a memory efficient reduction on any security problem if need be. Other problems that are memory sensitive shown in [ACFK17] to name a few are Learning Parity with Noise, a security assumption that is assumed to be post-quantum secure, using an algorithm presented in [BKW00], and factoring using [LLMP93]. It is therefore imperative to investigate the impact memory has on any cryptographic primitive and how we can improve or adjust security conclusions thereafter.

Why is it of importance to find and prove memory-tight reductions for the Hashed ElGamal scheme? First of all HEG is widely adopted in practice e.g, SECG SEC1, ISO/IEC 18033-2, IEEE 1363a and ANSI X9.63. Secondly both of the HEG schemes mentioned are in the random oracle model bounded by the $\mathsf{GAP\text{-}DH}$ assumption, [CKS08, ABR01], which in turn is bounded by the Dlog assumption. So any conclusions on the security on either one of the HEG schemes by black box reductions can't be good and meaningful if not memory-tight or at the very least taking this into account. Prior to the technique presented by Bhattacharyya, earlier reductions have not been memory-tight. The classical ElGamal scheme is known to be malleable and thus does not satisfy the notion of Chosen Ciphertext Attack, referred to as CCA. The HEG scheme is roughly explained a cryptographic scheme that deals with the problem of malleability by shifting the base scheme of ElGamal from a Public Key Encryption, or a PKE, to a Key Encapsulation Mechanism, or KEM for short. Such a KEM uses a key derivation function or a cryptographic Hash function to derive a key, rather than choosing one at random and then *padding* it with a padding scheme. If H is the cryptographic Hash function implemented in the ElGamal KEM defined in relation to some cyclic group $<g>= \mathbb{G}$, it would take as input a $Z = g^{xy}$ where x is the secret key from the asymmetric key pair (sk, pk) obtained at setup. $Y = g^y$ is the ciphertext that would be transmitted different for each encapsulation, obtained by choosing y uniform at random. The key, $K \xleftarrow{\$} \mathsf{H}(Z)$, would be the session key or *shared secret.* This differs from the regular ElGamal scheme where the session key or shared secret would just be Z itself. It is implementations of this Hash function in the random oracle model that have been the major obstacle in achieving memory-tight reductions. Note that in the Cramer-Shoup version of HEG the Hash takes in Y as an additional input before deriving a key, $K \xleftarrow{\$} \mathsf{H}(Y, Z)$. The reasoning for why this is of interest will become apparent later as we give the details of the two separate schemes.

1.1 Our Work

By applying the *injectively-map then prf* technique to the Twin Hashed ElGamal scheme, yet another version of the HEG scheme we refer to as TEG, we prove that this gives a memory-tight reduction in the $\mathsf{IND\text{-}CCA}$ security game. One of the advantages of the TEG scheme is that it achieves security under the ordinary CDH assumption in opposed to the $\mathsf{GAP\text{-}DH}$ assumption without much loss of efficiency. We also present a version of the HEG we refer to as the Hashed Proof System Hashed ElGamal, or $\mathsf{HPS\text{-}HEG}$. The purpose of this scheme was an attempt to generalize the technique of Bhattacharyya so that a reduction could simulate the private evaluation function of a given HPS and thus bounding the security of the scheme to the subset membership problem of the HPS. Unfortunately we were not able to show this in the case of ElGamal and conclude that this does not seem possible without a DDH oracle. However we did manage to prove that this also gives a memory-tight reduction in the $\mathsf{IND\text{-}CCA}$ security game, bounded by the $\mathsf{GAP\text{-}DH}$ assumption it is less efficient than the Cramer-Shoup HEG.

1.1.1 Memory-Tight Reduction for Twin Hashed ElGamal

The TEG scheme first introduced by [CKS08] is an extension of the regular HEG in that the encapsulations produce two separate Diffie-Hellman tuples both fed as input to the Key Derivation function. This means that the sizes of the key pairs of secret and public keys (sk, pk) are doubled and the complexity of the Hash is also lightly increased. Other than that the message and ciphertext spaces, that is group elements of $\mathbb{G}$, and key

```
**Key Generation** HEG
$(\mathbb{G}, g, p) \leftarrow \mathsf{Gen}(\lambda)$
$x \xleftarrow{\$} \mathbb{Z}_p^*$
$X \leftarrow g^x$
$pk \leftarrow (g, X)$
$sk \leftarrow (x)$
**return** $(pk, sk)$
```

```
**Key Generation** TEG
$(\mathbb{G}, g, p) \leftarrow \mathsf{Gen}(\lambda)$
$x_1 \xleftarrow{\$} \mathbb{Z}_p^*$
$x_2 \xleftarrow{\$} \mathbb{Z}_p^* - \{x_1\}$
$X_1 \leftarrow g^{x_1}$
$X_2 \leftarrow g^{x_2}$
$pk \leftarrow (g, X_1, X_2)$
$sk \leftarrow (x_1, x_2)$
**return** $(pk, sk)$
```

Figure 2: Secret and public key generation for the Hashed ElGamal and Twin Hashed ElGamal schemes. The oracle Gen generates a working algebraic group for this instance of the scheme with word sizes of length λ, note that the length of λ is the same in both schemes, hence also the group $\mathbb{G}$, group generator g and prime order p are the same sizes. The Key Derivation functions would then be $\mathsf{H}(Y, Z)$ and $\mathsf{H}(Y, Z_1, Z_2)$ respectively.

```
**Key Generation** HPS-HEG
$(\mathbb{G}, g, p) \leftarrow \mathsf{Gen}(\lambda)$
$g_1 \leftarrow g$
$t \xleftarrow{\$} \mathbb{Z}_p^*$
$g_2 \leftarrow g_1^t$
$a_1 \xleftarrow{\$} \mathbb{Z}_p^*$
$a_2 \xleftarrow{\$} \mathbb{Z}_p^* - \{a_1\}$
$sk \leftarrow (a_1, a_2)$
$pk \leftarrow (g_1, g_2, g_1^{a_1} \cdot g_2^{a_2})$
**return** $(pk, sk)$
```

Figure 3: The Key generation algorithm for the HPS-HEG. The oracle Gen generates a working algebraic group for the scheme with word sizes of length λ. $\mathbb{G}$ could be the same as in Figure 2. Note that given g_1 and g_2 generating new (pk, sk) pairs will not affect the fact of subset membership. It is the last part of pk we mostly will be referring to as the actual pk for computations.

spaces all stay the same size. The interesting fact of TEG being bounded by the regular Computational Diffie-Hellman assumption is due to a special trapdoor function, also introduced in [CKS08]. This trapdoor function shows that any adversary playing the CDH security game has the same advantage of breaking the CDH assumption as any adversary playing the Strong Twin Diffie-Hellman security game, STDH. In the STDH game an adversary has access to a Decisional Diffie-Hellman oracle, which returns 0 or 1 in the case a presented tuple is a Diffie-Hellman tuple. The trapdoor function allows the reduction simulating the STDH game to answer DDH oracle queries with a negligible probability of giving false information, thus perfectly simulating the STDH security game. The regular HEG is secure in the random oracle model given certain groups called Gap groups. These are groups known to be secure in the GAP-DH assumption, and are defined in this manner. An advantage of the CDH assumption is that it is a more general assumption for which there are currently many known groups thought to be secure. This simplifies setup of the scheme and makes the TEG more general than its cousin HEG. By applying the technique of Bhattacharyya we show that TEG is memory-tight in the random oracle model using specifically constructed PRFs. These PRFs use the available DDH oracle to injectively map all Hash queries so that any reduction needn't store queries or respective outputs, as opposed to in the classical lazy sampling technique.

1.1.2 Memory-Tight Reduction for the HPS Hashed ElGamal

The HPS-HEG is very similar to a PKE presented in [HK09] built on the HPS introduced in [CS98] which Hofheinz and Kiltz were able to prove is CCA in the standard model. The scheme we present uses the public and private evaluation functions of the ElGamal variant of a Hash Proof System for encapsulations and decapsulation. The interesting twist of this construction is the fact that these two functions are both deterministic. The general idea of a Hash Proof System is that given two evaluation functions and some language $\mathcal{L}$ we always have $\mathsf{Pub}(pk, X, w) = Z = \mathsf{Priv}(sk, Z)$ where Z is the outputted evaluation key when $X \in \mathcal{L}$. By modeling it as a KEM we are able to show that it is secure and memory-tight in the IND-CCA security game, in the random oracle model. This is very much related to security of the Cramer-Shoup HEG since we were not able to remove the DDH oracle requirement from the reduction. The goal was to construct a PRF using the private evaluation function of the HPS to injectively map queries, such that a more general construction could be obtained, and security bound by the SM problem of the HPS in question. A letter $X = (g_1^{w_1}, g_2^{w_2})$ of the ElGamal HPS is in the language $\mathcal{L}$ if $w_1 = w_2$. Thus any reduction against the SM problem, the problem of determining whether $X \in \mathcal{L}$, can simulate a language by generating their own set of (pk, sk), (see Figure 3). By universality of the ElGamal HPS the probability of $Z = \mathsf{Priv}(sk, X \notin \mathcal{L})$ is $1/p$ for a group $\mathbb{G}$ of order p. This means that a reduction could possibly map all H queries by checking $Z = \mathsf{Priv}(sk, X)$ and likewise generate a Z for decapsulation queries when simulating the IND-CCA security game. If at any point $Z = \mathsf{Priv}(sk, X^*)$ for the challenge value X^* in the SM problem, a reduction guesses that $X \in \mathcal{L}$ with probability $(1 - 1/p)$. The problem that we could not solve is how to extract the fact that an adversary $\mathcal{A}$ against the IND-CCA game breaks DDH if it never computes a *good* Z for X^*, a problem that boils down to proving DDH $\implies$ CDH.

1.2 Outline of Thesis

We start off by introducing in Section 2 some definitions and important preliminaries such as hardness assumptions and the definition of memory-tightness. We expect the reader to be familiar with the basics of cryptography and algebra, but include concepts that are important for the coming schemes and give general definitions of cryptographic primitives and black box reductions.

Then we introduce one of the techniques proposed by Auerbatch et al for memory efficient implementation of Hash functions following up by giving a detailed explanation on the two mentioned HEG schemes in Section 3. This will make clear why we on the one hand can have a memory-tight reduction for HEG and simultaneously on the other have a lower bound on memory.

Moving on to Section 4 we implement the technique of *injectively-map then prf* on the TEG scheme only after giving proofs for the trapdoor function and other important constructions such as the PRFs essential for the scheme.

In Section 5 we first prove that the ElGamal HPS is universal and has a SM problem bounded by the DDH assumption. Continuing we split the proof of IND-CCA into two steps. First we prove memory-tightness in the IND-CPA security game with a simple PRF replacement, then prove security in IND-CCA with a similar PRF construction as in earlier proofs. We also show briefly why we were not able to implement the private evaluation function as an injective map to the technique of Bhattacharyya.

2 Definitions and Preliminaries

If G is a set, we denote by $a \xleftarrow{\$} G$ the act of uniformly choosing a from G. Algorithms are represented as pseudo-code and are regarded as RAM's. A RAM is an abstract machine in the general class of register machines, it can indirectly address its registers, and has its instructions in a so called finite-state portion of the machine. A RAM differs from other standards like the Universal Turing Machine in that the UTM has its program in its registers as well as its data, otherwise they are equivalent. All algorithms have access to memory of word size λ and a constant number of registers which each hold one word. The symbol λ is the security parameter and will be the main efficiency measurement tool; the larger λ is, the less efficient the scheme.

Definition 2.1 (**PPT**). We say an algorithm or machine runs in **probabilistic polynomial-time** (PPT) if the time it takes to execute the algorithm is bounded by a fixed polynomial for its input size.

We differ between deterministic algorithms, $a \leftarrow \mathcal{A}(b)$, and probabilistic algorithms, $a \xleftarrow{\$} \mathcal{A}(b)$, where on input b for the deterministic case a is fixed, for the probabilistic case the output of $\mathcal{A}$ is uniformly sampled. An algorithm $\mathcal{A}$ with oracle access to some other algorithm O, referred to as an oracle, is denoted by $\mathcal{A}^O$. Stateful oracles have states determining where they are in their pseudo-code, random coins and stored words, this is denoted st_O. This state is inaccessible for $\mathcal{A}$, even if $\mathcal{A}$ has oracle access to O. We will mainly be constructing schemes in the random oracle model [BR93].

Definition 2.2 (**Random Oracle**). An idealized function $F : \{0,1\}^\delta \longrightarrow \{0,1\}^\eta$ is said to be a **random oracle**, if for all $x \in \{0,1\}^\delta$, the output $F(x)$ is independently and uniformly distributed over $\{0,1\}^\eta$.

Algorithms, adversaries, machines, oracles and, later on, reductions are all the same abstract entities, only differing in relation to each other. Results are proven in the framework of code based games, following [BR06]. Games consist of algorithms, these algorithms are further divided into oracles, with one main game running oracle and zero or more stateful oracles. When all game playing algorithms have terminated the game outputs either 1 or 0 and terminates. If the game is implemented with some function f, this is denoted by G_f.

Definition 2.3 For an adversary $\mathcal{A}$ playing a game G, we denote the probability of the event that $\mathcal{A}$ wins the game G by

$$\Pr[\mathrm{G}^{\mathcal{A}} \to 1].$$

The $\perp$ symbol represents a general term for an undefined control sequence, and if output during a game by any oracle, the game will terminate immediately, outputting 0.

Lemma 2.4 (**The Difference lemma** [BR06]). *Let A, B, F be events defined in some probability distribution, and suppose that $A \wedge \neg F \Leftrightarrow B \wedge \neg F$. Then $|\Pr[A] - \Pr[B]| \leq \Pr[F]$.*

This is a standard and fundamental lemma for game playing proofs. We assume it is well known and omit its short and simple proof.

2.1 Primitives and Reductions

Definition 2.5 A **cryptographic primitive** $\mathcal{P}$ is a pair $(F_{\mathcal{P}}, R_{\mathcal{P}})$, where $F_{\mathcal{P}}$ is a set of computable functions $f : \{0,1\}^* \longrightarrow \{0,1\}^*$, and $R_{\mathcal{P}}$ is a relation of pairs (f, M), where M is a machine that computes the function f. Additionally we summarize attributes of a primitive.

- The set $F_{\mathcal{P}}$ is required to contain at least one function that is computable by a probabilistic polynomial-time (PPT) machine.
- The function $f \in F_{\mathcal{P}}$ is an implementation of the primitive, and the implementation is efficient if a machine M computes f in PPT.
- The relation $R_{\mathcal{P}}$ is defined in accord with the specific criteria of breaking the primitive $\mathcal{P}$, and we say that a machine M $\mathcal{P}$-breaks f if $(f, M) \in R_{\mathcal{P}}$, i.e satisfies the relation on f.
- A secure implementation of the primitive $\mathcal{P}$ is an implementation f of $\mathcal{P}$ such that there exists no machine M $\mathcal{P}$-breaking f.
- The primitive $\mathcal{P}$ exists if there exists a secure and efficient implementation f.

In other words if there exists no PPT machine M such that $(f, M) \in R_{\mathcal{P}}$, then f is a secure implementation of $\mathcal{P}$. So $\mathcal{P}$ exists if for all $f \in F_{\mathcal{P}}$ there exists at least one implementation f such that no paring containing f and a PPT machine efficiently computing f can be found in $R_{\mathcal{P}}$. The implementation f represented abstractly as a function will in specific encryption schemes represent the generation, encryption and decryption functions. The set $F_{\mathcal{P}}$ also captures structural requirements such as correctness or length preserving. For example we could have the primitive $\mathcal{P}$ represent a one way function. Then the function f or implementation would be the one way function itself, satisfying length preserving and correctness requirements normally posed upon such functions, and therefore is a structural requirement of $F_{\mathcal{P}}$. The relation $R_{\mathcal{P}}$ would then be the probability of a machine producing an inverse to the function f, where the probability need only be non-negligible for M to succeed. If we could show that the function f is efficient and no machine M satisfies the relation on f, we can conclude that the one way function f is secure, hence $\mathcal{P}$ is indeed a cryptographic primitive. Other examples of cryptographic primitives are hash functions, PRFs, ElGamal encryption scheme, and so on.

Definition 2.6 A primitive $\mathcal{P}$ exists **relative to an oracle** Π if there exists an implementation f of $\mathcal{P}$ which is computable by a PPT machine with access to Π and such that no such machine $\mathcal{P}$ breaks f.

Notation wise a PPT machine M with oracle access to some implementation f is denoted M^f and is regarded as an implementation in its own right. We call f and its related primitive a reduction in the following sense.

Definition 2.7 There exists a **Black Box Reduction** from a primitive $\mathcal{P} = (F_{\mathcal{P}}, R_{\mathcal{P}})$ to a primitive $\mathcal{Q} = (F_{\mathcal{Q}}, R_{\mathcal{Q}})$ if there exists PPT oracle machines M and N s.t:

- For every implementation $f \in F_{\mathcal{Q}}$ we have that $M^f \in F_{\mathcal{P}}$, which we call **Correctness**.

- **Security**. For every implementation $f \in F_{\mathcal{Q}}$ and every machine L, if L $\mathcal{P}$ breaks M^f, then $N^{L,f}$ $\mathcal{Q}$ breaks f.

Relations for security notions of primitives are represented as games, adversaries and the corresponding algorithms. We will be following this notion. A reduction of $\mathcal{P}$ can then be viewed as an adversary $\mathcal{B}$ against its own security game G_g for some implementation of the primitive $\mathcal{Q}$ for which it has oracle access to $\mathcal{A}$. The implementation f must be dependant on g in some fashion, and $\mathcal{A}$ is an adversary against G_f. As mentioned we will stop differentiating between adversaries, algorithms and machines.

Example 2.8 Assume that we have an $\mathsf{UF\text{-}CMA}$ (UnForgeable Chosen Message Attack) secure $\mathsf{MAC} := (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ and we wish to improve on security by hashing the message m before we run the Tag function on m. Call this new MAC (Message AuthentiCator) for $\mathsf{MAC}' := (\textsc{Gen}', \textsc{Tag}', \textsc{Ver}')$, defined in Figure 4. We view both MACs and the Hash function H as implementations of cryptographic primitives, the primitives being the abstract idea of a collision resistant hash or secure MAC. We define the security game that $\mathcal{A}$ plays as follows: An adversary $\mathcal{A}$ is allowed to query the game running oracle, we'll call it $\Pi_{\mathcal{A}}$, on valid tags for a messages m_i of their choosing, and then obtains this tag t_i. $\mathcal{A}$ wins the $\mathsf{UF\text{-}CMA}$ game if it can produce a valid message-tag pair (m, t) where $m \notin \{m_i\}_i$, i.e has not been previously queried for a corresponding tag. For the sake of simplicity the game ends once $\mathcal{A}$ presents a pair. This game is the relation defining security for our $\mathsf{UF\text{-}CMA}$ secure cryptographic primitive MAC. There are two main ways for $\mathcal{A}$ to win its game, either finding a collision on the hash H, or by breaking the underlying MAC. The security of MAC' is then strongly dependant on the security of the underlying MAC and the Hash H. Inspecting H we could construct a new adversary $\mathcal{B}$ against the collision resistance of H with oracle access to $\mathcal{A}$ such that if $\mathcal{A}$ wins its game, $\mathcal{B}$ can increase its chances of winning its own hash collision game. We call $\mathcal{B}$ a reduction of $\mathcal{A}$. We can already see that $\mathcal{B}$ satisfies the security requirement for a reduction. The correctness follows from the fact that the $\mathsf{UF\text{-}CMA}$ game uses the hash function, a reduction $\mathcal{B}$ need only simulate the remaining oracles that $\mathcal{A}$ has access to, taking on the role of $\Pi_{\mathcal{A}}$, simultaneous as it interacts with its own game running oracle $\Pi_{\mathcal{B}}$. The details of how $\mathcal{B}$ implements $\mathcal{A}$ and simulates its oracle will effect the final probability estimate for the primitive in that game. We let all such details for all relevant reductions be represented as $p(\lambda)$, where we assume for now that these details are somehow related to the security parameter. The final advantage of $\mathcal{A}$ winning its game is bounded by the advantages of the reduction $\mathcal{C}$ against $\mathsf{UF\text{-}CMA}$ of MAC and the reduction $\mathcal{B}$ against H. The last term is dependant on the specific parameters used in implementing the different games, if the term is negligible, we call the reductions tight and assuming the hash and MAC are secure in their own right,

```
Oracle $\textsc{Gen}'(\lambda)$
return $\mathsf{Gen}(\lambda)$

Oracle $\textsc{Tag}'(k, m)$
$t \leftarrow \mathsf{Tag}(k, \mathsf{H}(m))$
return $t$

Oracle $\textsc{Ver}'(k, m, t)$
$h \leftarrow \mathsf{H}(m)$
$temp \leftarrow \mathsf{Ver}(k, h, t)$
return $temp$
```

Figure 4: A $\mathsf{UF\text{-}CMA}$ secure MAC' buildt on top of an existing MAC with an addition of a hash function H. λ is the security parameter, k a key for the Tag function generated by Gen. Ver returns 1 or 0, if the message-tag pair is valid or not respectively.

we can conclude that the scheme MAC' is secure. Note one of the reductions might still be tight even if $p(\lambda)$ isn't negligible.

$$\mathsf{Adv}^{\mathsf{UF-CMA}}_{\mathcal{A},\mathsf{MAC}'} \leq \mathsf{Adv}^{\mathsf{UF-CMA}}_{\mathcal{C},\mathsf{MAC}} + \mathsf{Adv}^{\mathsf{Hash}}_{\mathcal{B}} + p(\lambda).$$

2.2 Complexity Measures

Traditionally probability and time have been the two main parameters for tightness. Indeed if a reduction is non-tight in probability, we cannot guarantee that there does not exists an adversary breaking our original scheme given the same parameters. To specify, lets say we wish to investigate whether our scheme S is secure against an arbitrary adversary $\mathcal{A}$ with some predefined success probability ε_s. If the reduction of $\mathcal{A}$ is non-tight in ε_p, P being the problem equivalent to the scheme S, we could have say $\varepsilon_p >> \varepsilon_s$ then we cannot guarantee that there does not exist some other adversary $\mathcal{B}$ with a higher success probability $\varepsilon_p \geq \varepsilon_s* > \varepsilon_s$ in breaking S. If however one could show that there does not exists any algorithm with success probability ε_p against problem P, then the conclusion is that there should not exist any $\mathcal{A}$ with $\varepsilon_s \approx \varepsilon_p$ against S. The following definitions of complexity measures of an adversary $\mathcal{A}$ can be found in [ACFK17, WMHT18].

Definition 2.9 (**Success probability**). For a primitive $\mathcal{P}$ we say that an adversary $\mathcal{A}$ $\mathcal{P}$ breaks an implementation f with some probability or wins its security game G_f if

$$\mathsf{Adv}^{\mathsf{f}}_{\mathcal{A}} := \Pr[\mathrm{G}^{\mathcal{A}}_f \to 1] = \varepsilon,$$

for ε non-negligible. If the game G_f is a bit guessing game, we define the advantage as,

$$\mathsf{Adv}^{\mathsf{f}}_{\mathcal{A}} := \left|\Pr[\mathrm{G}^{\mathcal{A}}_f \to 1] - \frac{1}{2}\right|.$$

A reasonable argument for the importance of the time parameter is that one does not necessarily need to have something be kept a secret for eternity, but just long enough so that some adversary cannot capitalize on the encrypted information. For example, a professor teaching an elementary cryptography course having a final exam at date T, could encrypt the solutions to the exam problems and hand out the encryption to the students. If the underlying scheme has been proven to be secure against some adversary $\mathcal{A}$ lower bounded by some time parameter $\varepsilon_t \geq T + L$ obtained by some sort of reduction, where L is sufficiently large, the professor can quite confidently be sure that none of the students will be able to break the security before well after the exam. Making the information useless, or at the very least unhelpful in passing the course. Of course again assuming that the resources available to the adversary in question is a good representation of the resources accessible to the students.

Definition 2.10 (**Time Complexity**). The time complexity of an adversary $\mathcal{A}$ is the number of computational steps performed by $\mathcal{A}$ in the worst case over all possible inputs of size λ. When $\mathcal{A}$ plays security game G, the time complexity is denoted **LocalTime**($\mathcal{A}$) and is the time complexity of $\mathcal{A}$ plus the number of queries $\mathcal{A}$ makes to its oracles.

2.2.1 Memory-Tightness

Memory-tightness has been up until very recently an overlooked parameter for cryptographic reductions. As discussed Auerbach et al showed that memory loose reductions can have very meaningful impacts on security, and the definitions presented here follow theirs. Some problems can be solved faster with more working memory, so if a reduction is memory loose it can break the initial assumptions of security while it seemingly is tight for both probability and time complexities. The key observation is that some problems are what we call *memory sensitive*. These types of problems can be shown to be solved faster with more working memory. In an instance of these problems where the reductions memory is neglected, one would expect to find algorithms that have quite impactful consequences on the security assumptions, while if the problem is not memory sensitive, the impact will be negligible.

Definition 2.11 (**Memory efficiency**). The memory consumption of an adversary $\mathcal{A}$ is the size, in words of length λ, of the code of $\mathcal{A}$ plus the worst-case number of registers used in memory at any step in computation, over all inputs of bit-length λ and all random choices. For an adversary $\mathcal{A}$ playing game G denote **TotalMemory**($\mathcal{A}$) to be the memory required to run G with $\mathcal{A}$. This includes, the memory needed to input and output to $\mathcal{A}$, the memory needed to input and output to each oracle in the game, and the memory for the state of each oracle. Alternatively we denote by **LocalMemory**($\mathcal{A}$) the code and memory only used by $\mathcal{A}$, this includes input and output to $\mathcal{A}$ but excludes oracles and their states.

Tying memory complexity together with tightness for traditional complexity parameters we define efficiency for reductions as the following.

Definition 2.12 (**Tightness**). Given an adversary $\mathcal{A}$ against some game G_f we call a reduction $\mathcal{B}$ against G_g, where f is dependent on g in some fashion, for **tight** or **efficient** if the following hold:

$$\mathsf{Adv}^{\mathsf{f}}_{\mathcal{A}} \approx \mathsf{Adv}^{\mathsf{g}}_{\mathcal{B}},$$

$$\mathbf{LocalTime}(\mathcal{A}) \approx \mathbf{LocalTime}(\mathcal{B}),$$

$$\mathbf{LocalMemory}(\mathcal{A}) \approx \mathbf{LocalMemory}(\mathcal{B}).$$

Where any difference between the terms is shown to be negligible over the security parameter λ.

Continuing on the example of a reduction above we present a quick example of a non-memory efficient reduction $\mathcal{C}$, but which is still probability and time tight. The reduction is shown in Figure 5.

Example 2.13 The probability that the reduction $\mathcal{C}$ forges a valid message-tag pair and the message is not found to be a hash collision with any other message, is the same as the probability of $\mathcal{A}$ outright forging a pair. Indeed if (m^*, t^*) is a valid pair for MAC, then $(\mathsf{H}(m^*), t^*)$ must also be a valid pair for MAC'. Every query made by $\mathcal{A}$ is forwarded to $\Pi_{\mathcal{C}}$, and its response returned. Only difference is $\mathcal{C}$ hashes messages before forwarding them, so if we assume one query takes one time-step, we get q_H additional time-steps. For the memory estimate we count all lines in Figure 5 that store a variable temporarily.

```
Reduction $\mathcal{C}^{\mathsf{Ver},\mathsf{Tag}}()$
01 $L = \emptyset$
02 $(m^*, t^*) \leftarrow \mathcal{A}^{\mathsf{H},\textsc{Ver}',\textsc{Tag}'}()$
03 return $(\mathsf{H}(m^*), t^*)$

Oracle $\textsc{Ver}'(m, t)$
04 $h \leftarrow \mathsf{H}(m)$
05 return $\mathsf{Ver}(m, h)$

Oracle $\textsc{Tag}'(m)$
06 $h \leftarrow \mathsf{H}(m)$
07 return $\mathsf{Tag}(h)$

Oracle $\mathsf{H}(m)$
08 if $\exists (m, h) \in L$
09     $h \leftarrow L$
10 else
11     $h \overset{\$}{\leftarrow} \{0,1\}^l$
12     $L = L \cup \{(t, h)\}$
13 return $h$
```

Figure 5: The reduction $\mathcal{C}$ from the example above against the underlying MAC, playing the role of $\Pi_{\mathcal{A}}$. $\mathcal{C}$ initiates $\mathcal{A}$ and simulates the H, $\textsc{Tag}'$ and $\textsc{Ver}'$ oracles as shown. Tag and Ver are oracles run by $\mathcal{C}$'s own game playing oracle $\Pi_{\mathcal{C}}$. L is a set for storing hash queries made by $\mathcal{A}$, which at the start of the game is empty. l denotes the size of the 'hash space', where we let $l \leq n$ for message space $\{0,1\}^n$ (the security parameter dictates the size of l; $\lambda \leq l$).

In line 02 the reduction stores $\mathcal{A}$'s output, requiring two registers, however these are already counted in the **LocalMemory**$(\mathcal{A})$ term as input and output must be counted there as per our definition. We see further that in lines 04, 06, 09 and 11, the reduction temporarily stores h. Since $\mathcal{C}$ never simulates these oracles simultaneous, it only needs one register to temporarily store h. By further inspection we see that the two registers for $\mathcal{A}$'s output can handle all of the reductions temporary storage for h, so we omit the single register counted entirely. The oracles Ver and Tag are run by $\Pi_{\mathcal{C}}$ and are not counted in the reductions local memory. The reduction stores additionally a set L for all hash queries made by $\mathcal{A}$, following the standard way of modeling a random oracle. This however is not memory efficient! For every query made by $\mathcal{A}$ the reduction checks to see if there exists a previous query and corresponding hash value. If there exists no such pair, the reduction chooses one at random and stores this new pair in L. If $\mathcal{A}$ never queries on the same input twice the reduction needs as many registers as hash queries. Which first of all is difficult to predict beforehand, but can have disastrous effects on the total amount of registers. In the case where $\mathcal{A}$ makes one query every time-step, we must have registers equal to two times the total number of times-steps used by $\mathcal{A}$ in addition to those needed to run the base code of $\mathcal{A}$. We end up with the final estimate being inefficient,

$$\mathbf{LocalMemory}(\mathcal{C}) = \mathbf{LocalMemory}(\mathcal{A}) + (2 \cdot q_{\mathsf{H}})\lambda.$$

λ is the security parameter, and as mentioned, the size of each register.

A quick note about the constant term q_{H}. When we are bounding the time complexities of reductions we often say that the constant term of q_{H} has no effect on the overall efficiency, we say that it has a negligible impact, and thus the reductions are tight. However for memory complexity, as discussed, this term is no longer a negligible constant. Why then does it have a drastically higher impact on the overall memory usage but is negligible compared to the time parameter? The main difference is that in the **LocalMemory**$(\mathcal{C})$ term q_{H} is a multiple of λ, hence it grows much quicker than q_{H} by itself when λ increases. The expected growth of q_{H} in the **LocalTime**$(\mathcal{C})$ term is upper bounded by $O(q_{\mathsf{H}})$ when

$\mathcal{A}$ is efficient, so we would get $O(q_{\mathsf{H}}) \leq O(q_{\mathsf{H}} \cdot \lambda)$. From a computer science point of view, algorithms and computing machines exist physically and although our cryptographic schemes model security abstractly, they are ultimately models for a *real world* security setting. If we are able to bound the **LocalMemory**$(\mathcal{C})$ by the minimum amount of registers needed to run **LocalMemory**$(\mathcal{A})$ and a constant number of registers, say d, then this d stays the same however much we increase the security parameter. It is much less practical to supply an algorithm an extra of q_{H} registers, when each time we increase λ the need for more storage rises.

2.3 Hardness Assumptions

The following cryptographic assumptions presented here model the base line for all our schemes to come later on. All assumptions post this section will assume that these problems are *hard*, as in no PPT algorithm can efficiently solve them. We also include some definitions for useful cryptographic primitives.

2.3.1 Public Key Encryption

Two parties, Alice and Bob, want to send encrypted messages to each other, but their communication channel is monitored by a third party, Eve. Our two parties are far enough apart so as there is no other way on agreeing on a symmetric encryption key other than using the insecure channel. Assume Eve is malicious so any attempt on agreeing on a secret key for a symmetric encryption scheme in plaintext will be picked up and the security of the system will immediately be compromised. The challenge then is how can Alice and Bob publicly agree on a secret without Eve also obtaining it? The basic idea of public key encryption or asymmetric encryption tackles exactly this. One party, say Alice, uses a key generator providing one pair of keys, a secret key *sk* and a public key *pk*. This key *pk* is used for encryption, while only the secret key *sk* can be used for decryption. That means if Alice send *pk* to Bob, effectively also sending it to Eve, only Alice will be able to decrypt any message Bob sends. Of course we assume that in this example Eve only listens and does not actively try to thwart the communication in any way. There would be no way of Alice to know whether they are receiving messages from Bob or some third active party in this particular instance. PKE's are therefore often used in addition with signature schemes and underlying symmetric encryption schemes for increased security and efficiency in dealing with more active threats.

Definition 2.14 A **Public Key Encryption** is three efficiently computable algorithms, $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$, $c \xleftarrow{\$} \mathsf{E}(pk, m)$, and $m \leftarrow \mathsf{D}(sk, c)$ defined over a message space $\mathcal{M}$ a ciphertext space $\mathcal{C}$ and key sapce $\mathcal{K}$. We additionally impose the following,

- $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$ is a probabilistic algorithm, *sk* is called the secret key, and *pk* is called the public key.
- $c \xleftarrow{\$} \mathsf{E}(pk, m)$ is a probabilistic algorithm, where m is a message and c is the corresponding ciphertext.
- $m \leftarrow \mathsf{D}(sk, c)$ is a deterministic algorithm decrypting a ciphertext c back to its original message m.

- We require correctness in the sense that a key pair must decrypt c to the original plain-text m, $c = \mathsf{E}(pk, \mathsf{m})$.

As Alice may publicize their public key for anyone to obtain it must be hard to compute sk from pk. We require that for any pk there must be many possible choices of sk, each of which are likely to be valid pairs. Once the keys have been generated and distributed, Bob may send as many messages to Alice as they desire, with no loss of security. In fact anyone who has pk may send messages to Alice.
All provable security hinges on certain hardness assumptions. Any cryptographic primitive is, as mentioned earlier, defined in relation to a relation between an implementation and an adversary. These relations reflect the type of security issues we are modelling for, and might slightly vary from scheme to scheme. However there are some main assumptions that are relevant for almost all Public Key Encryptions, these are called Chosen Planitext Attack and Chosen Ciphertext Attack, CPA and CCA respectively.

Definition 2.15 (Chosen Ciphertext Attack). Let $\mathsf{Gen}(\lambda), \mathsf{E}(pk, m)$ and $\mathsf{D}(sk, c)$ be a PKE defined over a message space $\mathcal{M}$, ciphertext space $\mathcal{C}$ and a key space $\mathcal{K}$. Let $\mathcal{A}$ be an adversary playing the Chosen Ciphertext Attack game defined as follows, where $b \xleftarrow{\$} \{0,1\}$ is chosen uniform at random by the game running oracle otherwise knows as the challenger.

- The challenger computes $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$, and sends pk to $\mathcal{A}$.
- $\mathcal{A}$ chooses or computes a sequence of queries, which can be of two types
  - Encryption queries: $\mathcal{A}$ chooses message pairs of equal length (m_{i0}, m_{i1}), sends the sequence to the challenger. We denote $\mathcal{A}$'s total number of encryption queries as q_E. The challenger then computes $c_i \xleftarrow{\$} \mathsf{E}(pk, m_{ib})$, and sends each c_i back to $\mathcal{A}$.
  - Decryption queries: The adversary can choose any ciphertext $\hat{c}_j$ not already known to be an encryption of any earlier encryption queries and sends this to the challenger. The challenger then returns $m_j \leftarrow \mathsf{D}(sk, \hat{c}_j)$ to $\mathcal{A}$. We denote the total amount of decryption queries as q_D.
- $\mathcal{A}$ ends the game by returning a bit $\hat{b} \leftarrow \{0,1\}$. $\mathcal{A}$ wins the game if $\hat{b} = b$.

Further we let $\mathcal{A}$ be adaptive in that they may send one query at a time and then use each answer to compute their next query and they may decrypt or encrypt in any order of their choosing. The adversaries advantage is defined to be

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CCA}} := \left| \Pr[\mathrm{G}_{\mathsf{CCA}}^{\mathcal{A}} \to 1] - \frac{1}{2} \right|,$$

where $\Pr[\mathrm{G}_{\mathsf{CCA}}^{\mathcal{A}} \to 1]$ is the probability of $\mathcal{A}$ guessing the bit b correctly.

A CPA secure PKE is defined in relation to an almost identical security game as shown above. In the CPA security game we remove $\mathcal{A}$'s ability to ask for decryption queries, otherwise the games are identical. By further inspection we see then that if a PKE is CCA secure, then it also CPA. Therefore we will for the most part only be interested in if our schemes are CCA secure, knowing that the rest follows from this.

2.3.2 Key Encapsulations Mechanisms

Key Encapsulation Mechanisms, or KEM for short, are extensions of the PKE primitives. They are often used to establish a secure symmetric key between parties in very much the same way as PKE's. A downside to PKE's is that their relatively large keys can only encrypt relatively small plaintexts, thus by increasing the size of the plaintext, the overall efficiency is reduced. The solution of increasing the symmetric key size can have rough consequences for the PKE key sizes, not to mention the symmetric encryption scheme itself. Another approach is to take a small plaintext and *pad* it with a *padding* scheme so that the resulting ciphertext is much larger, and thus safer to transmit. The receiver then uses its asymmetric secret key to *de-pad* the ciphertext back into the smaller plaintext. The downside to this approach is that when it comes to proving security of padding schemes, most proofs are lacking. Traditionally for a padding scheme to be useful it has to be easily reversible, meaning if an adversary gets their hands on the padded plaintext M they essentially gain access to the original plaintext k. The solution and idea of a KEM is to randomly generate or choose an element x and by using a Key Derivation Function or a Cryptographic Hash, obtain a key k. The ciphertext-key pair (c, k) is derived from x, but only c is transmitted as it is the actual encrypted message. In this case if an adversary somehow obtains k learns nothing of the original plaintext x, and thus gains no knowledge of the Key Derivation Function, making KEM's more useful for purposes of security proofs. This approach eliminates the need for padding altogether. A KEM uses the same type of asymmetric key exchange as a PKE. The initiating party uses a public key to run the KEM: The algorithm generates a pair (c, k), we say c is the *encapsulation* of k, transmits c to the receiving party which *decapsulates* c back into k with their secret key and the Key Derivation Function. If this k is a symmetric key, the two parties have successfully established a more efficient means to private communication.

Definition 2.16 A **Key Encapsulation Mechanism** consists of three efficiently computable algorithms $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$, $(c, K) \xleftarrow{\$} \mathsf{Encap}(pk)$ and $(K) \leftarrow \mathsf{Decap}(sk, \mathsf{c})$, defined over some non-empty key space $\mathcal{K}$ and ciphertext space $\mathcal{C}$.

- $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$ is a probabilistic algorithm, sk is the secret key, and pk is the public key.
- $(c, K) \xleftarrow{\$} \mathsf{Encap}(pk)$ is the probabilistic encapsulation algorithm outputting a ciphertext key pair. We say that c is the encapsulation of K.
- $(K) \leftarrow \mathsf{Decap}(sk, \mathsf{c})$ is the deterministic decapsulation algorithm outputting the encapsulated key K from corresponding encapsulation c.
- We call the KEM δ-correct if $\Pr[\mathsf{Decap}(sk, c) \neq K | (sk, pk) \leftarrow \mathsf{Gen}(\lambda); (c, K) = \mathsf{Encap}(pk)] \leq \delta$.

Security of a KEM is defined by a Indistinguishable Chosen Ciphertext Attack security game ($\mathsf{IND\text{-}CCA}$), see Figure 6.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CCA}} := \left| \Pr[\mathrm{G}_{\mathsf{IND\text{-}CCA}}^{\mathcal{A}} \to 1] - \frac{1}{2} \right|.$$

```
Game(IND-CCA)
$b \xleftarrow{\$} \{0,1\}$
$(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$
$(c^*, K_0^*) \xleftarrow{\$} \mathsf{Encap}(pk)$
$K_1^* \xleftarrow{\$} \mathcal{K}$
$\hat{b} \leftarrow \mathcal{A}^{\textsc{Decap}}(c^*, K_b^*)$
If $b = \hat{b}$
    return 1
else
    return 0
```

```
Oracle Decap(c)
if $c = c^*$
    return $\perp$
$K \leftarrow \mathsf{Decap}(sk, \mathsf{c})$
return $K$
```

Figure 6: The IND-CCA game for Key Encapsulation Mechanisms. $\mathcal{A}$ wins if it can efficiently distinguish between a uniformly random chosen key K_1 and the generated key K_0 output by the encapsulation algorithm. c^* will in both cases be the encapsulation of K_0.

2.3.3 Diffie-Hellman Assumptions

All the following Diffie-Hellman assumptions are based on the Discrete Logarithm assumption, or Dlog for short. The Dlog assumption roughly states that given g^x it is hard for any efficient PPT algorithm to factor out x in a reasonable amount of time [Sho97]. The security of the ElGamal scheme variants we shall be introducing later are all directly bounded by the following Diffie-Hellman assumptions.

Definition 2.17 (**Computational Diffie-Hellman**). Let $\mathbb{G}$ be a cyclic group with generator g and prime order p, let $\mathbb{Z}_p^*$ be the multiplicative group of integers defined in relation to p. Given g, g^x and g^y for some unknown $(x, y) \xleftarrow{\$} \mathbb{Z}_p^{*2}$ the CDH-assumption states that the advantage of any efficient adversary $\mathcal{A}$ in producing g^z where $z = x \cdot y$ is negligible.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CDH}} := \Pr[\mathrm{G}_{\mathsf{CDH}}^{\mathcal{A}} \to 1].$$

Definition 2.18 (**Decisional Diffie-Hellman**). Let $\mathbb{G}$ be a cyclic group with generator g and prime order p, let $\mathbb{Z}_p^*$ be the multiplicative group of integers defined in relation to p. Given g, g^x, g^y and g^z for some unknown $(x, y) \xleftarrow{\$} \mathbb{Z}_p^{*2}$ and unknown $z \in \mathbb{Z}_p^*$ the DDH-assumption states that the advantage of an efficient adversary $\mathcal{A}$ in determining whether $z = x \cdot y$ or $z \xleftarrow{\$} \mathbb{Z}_p^*$ is negligible. In the security game the challenger chooses $b \xleftarrow{\$} \{0, 1\}$ where $b = 0$ implies $z = x \cdot y$ and $b = 1$ implies $z \xleftarrow{\$} \mathbb{Z}_p^*$.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{DDH}} := \left| \Pr[\mathrm{G}_{\mathsf{DDH}}^{\mathcal{A}} \to 1] - \frac{1}{2} \right|.$$

We refer to the tuple (g^x, g^y, g^z) as a Diffie-Hellman tuple, or DH-tuple for short, when $z = x \cdot y$. Define the DDH oracle as an efficient algorithm answering queries on the form (g^x, g^y, g^z) with 1 if the tuple is indeed a DH-tuple, and 0 otherwise. The following assumption allows an adversary access to this oracle during the security game.

Definition 2.19 (**Gap-Diffie-Hellman**). Let $\mathbb{G}$ be a cyclic group with generator g and prime order p, let $\mathbb{Z}_p^*$ be the multiplicative group of integers defined in relation to p. The

Gap-DH assumption states that an adversary $\mathcal{A}$ against the CDH-assumption with access to a DDH oracle has negligible advantage in breaking the CDH-assumption.

$$\mathsf{Adv}_{\mathcal{A}}^{\text{Gap-DH}} := \Pr[\mathrm{G}_{\mathsf{CDH}}^{\mathcal{A}^{\mathsf{DDH}}} \to 1].$$

The Gap-DH assumption does not hold in general, in fact it only holds for certain groups. We call the groups for which the Gap-DH is hard for Gap-groups. Their construction is defined by the Gap-DH, as in if the assumption holds, the group $\mathbb{G}$ is a Gap-group. The $\mathsf{GAP\text{-}DH}$ assumption is very closely related to the $\mathsf{Strong\text{-}DH}$, but the DDH oracles are slightly different. In the $\mathsf{GAP\text{-}DH}$ assumption, an adversary can chose freely all parts of the tuples presented to the DDH oracle. In the $\mathsf{Strong\text{-}DH}$ assumption the first element $X = g^x$ is always fixed. This is also the case for the STDH assumption.

Definition 2.20 (**Oracle Diffie-Hellman**). Let $\mathcal{A}$ be an adversary playing a game similar to that of the DDH game. Assume we have a similar group construction. For a bit $b \xleftarrow{\$} \{0,1\}$ the game plays as follows:

- The challenger draws $(x, y) \xleftarrow{\$} \mathbb{Z}_p^{*2}$, computes group elements (g^x, g^y). Then using a cryptographic Hash function either computes:
  - if $b = 0$: $K_0 \leftarrow \mathsf{H}(g^{xy})$
  - if $b = 1$: $K_1 \xleftarrow{\$} \mathcal{K}$

  The challenger sends K_b, g^x and g^y to the adversary.

- $\mathcal{A}$ is given access to an oracle answering queries on the form $\mathsf{H}_y(g^u) := \mathsf{H}(g^{uy})$, where for a fixed y and input g^u group element returns $\mathsf{H}(g^{uy})$. The oracle is restricted not to answer queries of the challenge value g^x. $\mathcal{A}$ may issue as many queries as they like, we denote the total number of queries as q_{H_y}.

- The game terminates when $\mathcal{A}$ outputs a bit $\hat{b}$. If $\hat{b} = b$ $\mathcal{A}$ wins the game.

The ODH assumption states that $\mathcal{A}$'s advantage is negligible,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{ODH}} := \left| \Pr[\mathrm{G}_{\mathsf{ODH}}^{\mathcal{A}} \to 1] - \frac{1}{2} \right|.$$

This Oracle Diffie-Hellman assumption is essential for the security of the Hashed ElGamal scheme, HEG, under the standard model. In the random oracle model this assumption is implied by the $\mathsf{GAP\text{-}DH}$ assumption, and the security game is slightly modified so that $\mathcal{A}$, in addition to H_y, also gains access to the random oracle H and can query it on any group element of their choosing. We shall see later that this implementation is the main point of interest when proving memory-tightness of the HEG scheme. In fact it is the main reason for the impossibility conjecture given in [ACFK17].

Definition 2.21 (**Strong Twin Diffie-Hellman**). Let $\mathbb{G}$ be a cyclic group with generator g and prime order p, let $\mathbb{Z}_p^*$ be the multiplicative group of integers defined in relation to p. Given g, g^{x_1}, g^{x_2}, g^y for unknown $(x_1, x_2, y) \xleftarrow{\$} \mathbb{Z}_p^{*3}$ and access to a DDH oracle, the STDH assumption states that the advantage of any efficient adversary $\mathcal{A}$ in producing g^{z_1} and g^{z_2} where $z_1 = x_1 \cdot y$ and $z_2 = x_2 \cdot y$ is negligible.

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{STDH}} := \Pr[\mathrm{G}_{\mathsf{STDH}}^{\mathcal{A}} \to 1].$$

2.3.4 Pseudo Random Functions

Pseudo Random Functions, or PRF's, are deterministic functions that simulate randomness. In addition to taking on input from the function domain, say $\mathcal{X}$, the PRF also takes a key value chosen uniform at random from some defined key space $\mathcal{K}$. Formally we have $F : \mathcal{K} \times \mathcal{X} \longrightarrow \mathcal{Y}$, for a PRF F.

Definition 2.22 (**Pseudo Random Function**). Let $\mathsf{Func}[\mathcal{X}, \mathcal{Y}]$ be the set of all functions $f : \mathcal{X} \longrightarrow \mathcal{Y}$. For a PRF F defined over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$ and an efficient adversary $\mathcal{A}$ we define the security game of F as follows. At the start of the game, the game running oracle or challenger, chooses a bit $b \xleftarrow{\$} \{0, 1\}$.

- The challenger chooses a function as follows:
  - if $b = 0$: $k \xleftarrow{\$} \mathcal{K}$, $f \leftarrow F(k, .)$.
  - if $b = 1$: $f \xleftarrow{\$} \mathsf{Func}[\mathcal{X}, \mathcal{Y}]$.
- $\mathcal{A}$ submits a sequence of queries to the challenger on the form of elements from $\mathcal{X}$. We let $\mathcal{A}$ be adaptive and denote the total number of queries as q. The challenger answers each query with $y \leftarrow f(x)$.
- The adversary ends the game by sending a bit $\hat{b} \leftarrow \{0, 1\}$ to the challenger. $\mathcal{A}$ wins the game if $\hat{b} = b$.

We say that a PRF is secure if no $\mathcal{A}$ can efficiently distinguish the PRF from an arbitrary function, i.e if $\mathcal{A}$'s advantage is negligible,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{PRF}} := \left| \Pr[\mathrm{G}_{\mathsf{PRF}}^{\mathcal{A}} \to 1] - \frac{1}{2} \right|.$$

2.3.5 Hash Proof Systems

Let $\mathcal{L} \subseteq \mathcal{X}$ for some space $\mathcal{X}$. We call $\mathcal{L}$ a *language* if there exists some function $R(x, w)$ that takes in a element x from $\mathcal{X}$ and a *witness* w such that $R(x, w) = 1$ if $x \in \mathcal{L}$. We call R a verifying function. The witness space will be defined in conjunction with R and the relation which determines the structure of $\mathcal{L}$.

Definition 2.23 (**Hash Proof System**). A Hash Proof System consists of three efficiently computable algorithms $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$, $z \leftarrow \mathsf{Pub}(pk, x, w)$ and $z \leftarrow \mathsf{Priv}(sk, x)$. The HPS is defined over a language $\mathcal{L}$ with a verifying function R and witness space $\mathcal{W}$.

- $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$ is a probabilistic key generator providing a secret-public key pair.
- $z \leftarrow \mathsf{Pub}(pk, x, w)$ is a deterministic public evaluation function that on given input $x \in \mathcal{X}$, witness w and public key pk returns a $z \in \mathcal{Z}$.
- $z \leftarrow \mathsf{Priv}(sk, x)$ is a private evaluation function only accessible with the secret key sk. This function is also deterministic and outputs a key $z \in \mathcal{Z}$.
- For correctness we require that $\mathsf{Pub}(pk, x, w) = z = \mathsf{Priv}(sk, x)$ for all $x \in \mathcal{L}$ and public-secret key pairs (sk, pk). There are no restrictions or requirements for any $x \notin \mathcal{L}$, also denoted $x \in \mathcal{X}/\mathcal{L}$.

When it comes to Hash functions we call a seeded Hash function H_s, with uniform random seed s, for universal if for all x, $\mathsf{H}_s(x)$ is uniform random. An example of a seeded universal Hash function is $\mathsf{H}_s(x) = s \oplus x$. Not to be confused with a universal Hash function we have a similar requirement for universality of a HPS. The Pub and Priv functions are deterministic, and by correctness we know pk defines the action of Priv for all $x \in \mathcal{L}$. So the best we can do for universality's sake is impose a similar notion as for the regular hash function on all inputs $x \notin \mathcal{L}$ to the HPS.

Definition 2.24 (**Universality of** HPS). We call a HPS universal or sound if for all $x \notin \mathcal{L}$, i.e $x \in \mathcal{X}/\mathcal{L}$, and for all $z \in \mathcal{Z}$ the following holds:

$$\Pr[\mathsf{Priv}(sk, x) = z] = \frac{1}{|\mathcal{Z}|}.$$

In other words the distributions of $(pk, \mathsf{Priv}(sk, x))$ and (pk, z) are the same for $z \xleftarrow{\$} \mathcal{Z}$ and $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\lambda)$.

If we know that both Priv and Pub are deterministic, and defined by pk, why would imposing the requirement of universality on elements outside the language be of any use? This is where this final, but very important security assumption for HPS comes in, the Subset Membership problem, SM-problem. The SM-problem states that it is hard to distinguish between an element of the language and an element outside of the language. We formalize in the following security game.

Definition 2.25 (**Subset Membership**). Let $\mathcal{L}$ be a language and let $\mathsf{Sample}_{\mathcal{X}/\mathcal{L}}$ and $\mathsf{Sample}_{\mathcal{L}}$ be two efficiently computable probabilistic functions, returning an element x from outside or inside the language $\mathcal{L}$ respectively. Let $\mathcal{A}$ be a PPT algorithm playing the SM-problem security game defined as follows. The challenger chooses $b \xleftarrow{\$} \{0, 1\}$ at the start of the game.

- The challenger chooses x^* in the following fashion:
  - if $b = 0$: $x_0^* \xleftarrow{\$} \mathsf{Sample}_{\mathcal{L}}$, $(x \xleftarrow{\$} \mathcal{L})$, along with w, where w is a witness to the fact that $x \in \mathcal{L}$.
  - if $b = 1$: $x_1^* \xleftarrow{\$} \mathsf{Sample}_{\mathcal{X}/\mathcal{L}}$, $(x \xleftarrow{\$} \mathcal{X}/\mathcal{L})$.

  The challenger sends x_b^* to $\mathcal{A}$.

- The adversary can query the challenger on elements x either outside the language or inside the language. The challenger replies with $\mathsf{Sample}_{\mathcal{X}/\mathcal{L}}$ and $\mathsf{Sample}_{\mathcal{L}}$ as requested.
- The game terminates after $\mathcal{A}$ outputs a bit $\hat{b} \leftarrow \{0,1\}$, winning if $\hat{b} = b$.

We require that $|\mathcal{X}/\mathcal{L}| \geq 2^{\lambda}$ and say the SM-problem is hard if $\mathcal{A}$'s advantage is negligible,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{SM}} := \left|\Pr[\mathrm{G}_{\mathsf{SM}}^{\mathcal{A}} \to 1] - \frac{1}{2}\right|.$$

Now, a universal HPS with a hard SM-problem is very useful indeed. The fact that an adversary cannot efficiently distinguish between $x \in \mathcal{L}$ and $x^{'} \in \mathcal{X}/\mathcal{L}$ means that from their point of view, the functions Priv and Pub might as well be probabilistic with uniform probability space. This motivated our attempt in using the HPS for generalizing the *injectively-map then prf* method of memory-tight reductions introduced by Bhattacharyya.

3 Achieving Memory Efficiency

According to the current literature problems are memory sensitive if there are algorithms known to break these problem faster given more memory. t collision resistance of a Hash function H for $t = 2$ is known not to be memory sensitive, but for $t \geq 3$ more memory is more good. The algorithm currently referenced as the best example of this fact is a $t \geq 3$ collision finder, which can be run by numerous parallel processors with only added constant memory for each processor. In [JL09] Joux and Lucks present this collision finding algorithm and show that the algorithm can find a 3-collision in $N^{1-\alpha}$ time steps given memory of N^{α} where $\alpha \leq 1/3$ and N being the cardinality of a finite image for the hash in question. The time efficiency gained by using more memory is capped at $\alpha \leq 1/3$, but when t increases, so does the potential for finding collisions faster with parallel processors. This algorithm is shown to solve a t-collision problem in time $N^{(t-1)/t-s}$ with N^s processors each with access to memory of $N^{(t-2)/t-s}$. This emphasizes the importance for memory-tight Hash implementations when we are investigating security of schemes dependant on multi collision resistance. The Hashed ElGamal scheme however does not assume the need of any t-collision resistance Hash for $t \geq 3$, but this scheme is still memory sensitive. As mentioned in the introduction, it is the memory sensitivity of the Dlog assumption that makes HEG memory sensitive. Even though it is not directly the security of the Hash function that is susceptible when memory is neglected, it is often implementations of Hash functions, as we saw in Figure 5, that might lead to greater memory usage and as a consequence weaker overall security. It opens up for adversaries to potentially exploit other more memory sensitive assumptions that build up a cryptographic scheme. An important obstacle to overcome in obtaining memory-tight reductions is then reducing the impact of the Hash function implementation. One solution presented in [ACFK17], although not good enough by itself for solving the questions of memory for HEG, is to swap out the Hash altogether with a PRF.

3.1 PRF in the Random Oracle Model

Imagine that a reduction $\mathcal{B}$ is simulating a Hash function for some adversary $\mathcal{A}$ while $\mathcal{B}$ is playing its own security game. Each Hash query made by the adversary $\mathcal{A}$ to $\mathcal{B}$ must be answered in such a way that if $\mathcal{A}$ ever queries on the same input twice, the same output is returned back to $\mathcal{A}$. In the Random Oracle model this is done by choosing an output at random from the output space and storing it in some set or table, say L. Then $\mathcal{B}$ can easily check the table L for outputs before answering future queries to make sure that the reduction's Hash simulation is consistent. This technique for modeling random oracles is commonly referred to as *lazy sampling*. The table L grows linearly with the number of Hash queries made by $\mathcal{A}$. It is a source of huge increases to memory usage and is therefore subject for alternative implementations. There are some pitfalls to be avoided when dealing with these alternative implementations, the first of which is loss of consistency. Lets say that in order to reduce the impact on memory by the storage table L, it is removed entirely without any further modifications. This solves the need for extra memory but opens up for consistency issues. Neglecting the table L altogether will mean that each time $\mathcal{A}$ queries the Hash function they will get a different answer. Even if $\mathcal{A}$ queries on the same input twice $\mathcal{B}$ has no way of checking what $\mathcal{A}$ has previously queried. This means that H is no longer well defined and will not be consistent enough for $\mathcal{A}$ to use effectively. One possible solution to this, as Auerbach et al present, could be to replace the random Hash with a PRF, see Figure 7. Then we would have regained the well definiteness of the simulated Hash and decreased the need for q_{H} extra registers down to one single register for holding the PRF key. If the PRF is secure, i.e it is indistinguishable from a random function, then this is a sufficient replacement. $\mathcal{A}$ will only be able to tell the difference with advantage of $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{PRF}}$, which would then be negligible. This is an easy replacement and will work for many simple schemes using simple implementations of Hash functions. However things get a bit more complicated when inspecting the reductions of the HEG scheme.

```
Random Oracle H(x)
    if $\mathsf{H}(x) \in L$
        return $\mathsf{H}(x)$
    else
        $\mathsf{H}(x) \xleftarrow{\$} \mathcal{Y}$
        $L = L \cup \mathsf{H}(x)$
        return $\mathsf{H}(x)$

Random Oracle PRF_k(x)
    $y \leftarrow F(k, x)$
    return $y$
```

Figure 7: Two implementations of a random oracle, on the left the standard *lazy sampling* technique, on the right a memory efficient implementation with a PRF $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ with key k from key space $\mathcal{K}$. Both oracles have input and output space $\mathcal{X}$ and $\mathcal{Y}$ respectively.

3.2 Reductions in the Security Proof of Hashed ElGamal

The Hashed ElGamal scheme, HEG for short, is a KEM made up of three efficiently computable algorithms GEN, ENCAP and DECAP as shown in Figure 8.

```
Oracle GEN(λ)
    $(\mathbb{G}, g, p) \leftarrow \mathsf{Gen}(\lambda)$
    $x \xleftarrow{\$} \mathbb{Z}_p^*$
    $X \leftarrow g^x$
    $pk \leftarrow (g, X)$
    $sk \leftarrow (x)$
    return $(pk, sk)$

Oracle ENCAP(pk)
    $(g, X) \leftarrow pk$
    $y \xleftarrow{\$} \mathbb{Z}_p^*$
    $Y \leftarrow g^y$
    $Z \leftarrow X^y$
    $K \leftarrow \mathsf{H}(Z)$
    return $(Y, K)$

Oracle DECAP(sk, Y)
    $x \leftarrow sk$
    $Z \leftarrow Y^x$
    $K \leftarrow \mathsf{H}(Z)$
    return $K$
```

Figure 8: Hashed ElGamal Key Encapsulation scheme, denoted HEG, as presented in [GT21]. In this version the Hash function $\mathsf{H} : \mathbb{G} \longrightarrow \mathcal{K}$ only takes in one group element, X^y.

In the standard oracle model the original security proof of IND-CCA for HEG is shown secure under the Oracle Diffie-Hellman assumption, ODH [ABR01]. In the random oracle model it turns out this assumption is implied from the GAP-DH assumption. The proof given in the random oracle model has a reduction $\mathcal{B}$ trying to break the GAP-DH assumption using the ODH adversary $\mathcal{A}$. $\mathcal{B}$ has access to a DDH oracle, but needs to simulate the $\mathcal{A}$'s oracle H_y without having access to y. The security game from Definition 2.20 with the minor modification mentioned is simulated by $\mathcal{B}$ in Figure 9. $\mathcal{B}$ wins if it can output $Z = g^{xy}$. In this game $\mathcal{B}$ needs to answer queries to the random oracle H itself as well as to H_y, remember that $\mathsf{H}_y(X) := \mathsf{H}(X^y)$. Queries to H_y and H are simulated in the following way:

- $\mathsf{H}_y(X)$: $\mathcal{B}$ first checks whether $X = g^x$, if so it returns $\perp$. Otherwise it continues and checks if any previous query on X was made, if so it returns the same response. If this is not the case it checks with its DDH oracle whether (X, g^y, Z) is a DH-tuple with any of the previous queries Z to H. If it turns out this is the case for some Z it outputs $\mathsf{H}(Z)$ and stores this response with X for future queries. If none of the previous checks are true, the reduction chooses a K uniform at random from the image of H, $\mathcal{B}$ stores this response together with X, and finally outputs K.

- $\mathsf{H}(Z)$: $\mathcal{B}$ first checks with the DDH oracle whether (g^x, g^y, Z) is a DH-tuple, if this is true $\mathcal{B}$ updates *temp* to be Z. Otherwise it does the same as before, only now checking with the DDH oracle on inputs X to H_y stored in L_{H_y} instead of in L_{H}.

By using the DDH oracle in this way $\mathcal{B}$ makes sure there are no discrepancies in the consistency of the Hash outputs. If $\mathcal{B}$ where not to check the current query if a DH-tuple with any other previous query exists, the adversary $\mathcal{A}$ could easily see that for $Z = X^y$, $\mathsf{H}_y(X) = \mathsf{H}(X^y) = \mathsf{H}(Z)$ might not be consistent. Especially since the Hash responses are uniform at random due to the random oracle model. This is not memory efficient. The reduction needs extra storage of $(q_{\mathsf{H}_y} + q_{\mathsf{H}})\lambda$ additional registers on top of the memory needed to run $\mathcal{A}$. Furthermore since all Diffie-Hellman assumptions naturally rely on the hardness of the Dlog assumption, and we know that Dlog is memory sensitive it is clear that current security assumptions of HEG are less meaningful than previously assumed. This emboldens the requirement for memory efficient implementations of the HEG. Again it is the random oracle Hash function causing headaches. We have already covered the consistency problems caused if we were to remove the tables L_{H_y} and l_{H}. Ideally we would want to solve both the memory usage and the consistency problem, but why not just swap out the Hash function with a PRF as before?

Lets say we swapped out the Hash function H with a PRF F. Now if the adversary queries H on input Z the reduction would simply run $F(k, Z)$ and return this answer. If $\mathcal{A}$ queries on Z again, $\mathcal{B}$ gets the same value as before by running $F(k, Z)$ a second time without the need for storing the output. So far so good. The security game simulated by the reduction needs to simulate both H and H_y as described above. Therefore when $\mathcal{B}$ gets a Hash query on Z it would also have to check whether any previous query to H_y, say X, is a DH-tuple with Z. The problem is that the reduction has no way of retrieving previous Hash queries since it has no table to store them in! We are therefore still faced with the same consistency problem as before. Without corresponding queries the DDH oracle becomes useless in helping $\mathcal{B}$ distinguish between DH-tuples, except in the case where $Z = g^{xy}$, but this does not help $\mathcal{B}$ in simulating the security game needed to obtain this Z. If $\mathcal{B}$ could somehow check themselves if $Z = X^y$ it still would not be able to tell if $\mathcal{A}$ had queried on Z or X earlier. Even so the security game becomes pointless since we have assumed y to be secret and unknown to $\mathcal{B}$. It is this particular construction of the Hash function in the $\mathsf{GAP\text{-}DH\text{-}ODH}$ security game that led Auerbach et al in [ACFK17] to conjecture that no memory-tight reduction for HEG as shown above exists. With this scheme as presented Figure 8 together with the general group setting Ghoshal and Tessaro were able to prove this conjecture [GT21]. However, with one minor modification to this scheme we shall see that there does in fact exist a memory-tight reduction for HEG, in particular the Cramer-Shoup version, and because of the modification it does not contradict other works mentioned.

```
**Reduction** $\mathcal{B}^{\mathsf{DDH}}(g^x, g^y)$
$K \xleftarrow{\$} \mathcal{K}$
$temp \xleftarrow{\$} \mathbb{G}$
$L_{\mathsf{H}_y} \leftarrow \emptyset$
$L_{\mathsf{H}} \leftarrow \emptyset$
$\hat{b} \leftarrow \mathcal{A}^{\mathsf{H}_v,\mathsf{H}}(K, g^x, g^y)$
**return** $temp$

**Oracle** $\mathsf{H}_y(X)$
**if** $X = g^x$
   **return** $\perp$
**else**
   **if** $\exists X \in L_{\mathsf{H}_y}$
      $(X, K) \leftarrow L_{\mathsf{H}_y}$
      **return** $K$
   **elif** $\exists Z \in L_{\mathsf{H}}$ **s.t** $1 \leftarrow \mathsf{DDH}(X, g^y, Z)$
      $(Z, K) \leftarrow L_{\mathsf{H}}$
      $L_{\mathsf{H}_y} = L_{\mathsf{H}_y} \cup (X, K)$
      **return** $K$
   **else**
      $K \xleftarrow{\$} \mathcal{K}$
      $L_{\mathsf{H}_y} = L_{\mathsf{H}_y} \cup (X, K)$
      **return** $K$

**Oracle** $\mathsf{H}(Z)$
**if** $1 \leftarrow \mathsf{DDH}(g^x, g^y, Z)$
   $temp \leftarrow Z$
**if** $\exists Z \in L_{\mathsf{H}}$
   $(Z, K) \leftarrow L_{\mathsf{H}}$
   **return** $K$
**elif** $\exists X \in L_{\mathsf{H}_v}$ **s.t** $1 \leftarrow \mathsf{DDH}(X, g^y, Z)$
   $(X, K) \leftarrow L_{\mathsf{H}_y}$
   $L_{\mathsf{H}} = L_{\mathsf{H}} \cup (Z, K)$
   **return** $K$
**else**
   $K \xleftarrow{\$} \mathcal{K}$
   $L_{\mathsf{H}} = L_{\mathsf{H}} \cup (Z, K)$
   **return** $K$
```

Figure 9: $\mathcal{B}$ an adversary playing the GAP-DH security game utilizing an adversary $\mathcal{A}$ playing the ODH security game as defined in Definition 2.19 and Definition 2.20 respectively. $\mathcal{B}$ has access to its own DDH oracle answering queries on the form $(X, Y, Z) \leftarrow \mathbb{G}^3$ returning 1 if and only if (X, Y, Z) is a DH-tuple. $\mathcal{K}$ is the image of the Hash function H. L_{H_y} and L_{H} are sets for storing Hash queries and Hash outputs initially empty. *temp* is the final output of $\mathcal{B}$ and is updated if $\mathcal{A}$ ever queries H on $Z = X^y$, in which case $\mathcal{B}$ wins its security game. The H_y function is defined to be $\mathsf{H}_y(X) := \mathsf{H}(X^y)$ for a Hash function H and any input $X \in \mathbb{G}$ except if $X = g^x$, in which case it outputs $\perp$.

3.3 Injectively Map then PRF

The Cramer-Shoup variant of HEG shown in Figure 10 has a minor modification to the cryptographic Hash function, this scheme and the original proof of security was first presented in [CS01]. Instead of only taking in the shifted public key X^y, the Hash in addition takes the encapsulation Y as input. This results in a slightly less efficient version of HEG, however the scheme in the random oracle model is shown to be bounded by the GAP-DH assumption. This scheme also gives a possible reduction a solution to the problem of supplying pairs of (Y, Z) to the DDH oracle discussed in the section above. The reduction from HEG against the GAP-DH security game still needs to maintain consistency in Hash and Decapsulation queries. The original reduction needs three storage tables to maintain consistency, see Figure 11.

The first table L_1 stores all *full* Hash queries made by $\mathcal{A}$ regardless of whether the pairs are DH-tuples or not. In this section we call (Y, Z) a DH-pair w.r.t $X = g^x$ if (X, Y, Z) is a DH-tuple. We will refer to the query as full if for any pair (Y, Z) there exists a response K that the reduction has output in the past. The second table L_2 saves pairs that are known to be DH-tuples with respect to g^x. This table is used for decapsulation

```
Oracle GEN(λ)
    $(\mathbb{G}, g, p) \leftarrow \mathsf{Gen}(\lambda)$
    $x \xleftarrow{\$} \mathbb{Z}_p^*$
    $X \leftarrow g^x$
    $pk \leftarrow (g, X)$
    $sk \leftarrow (x)$
    return $(pk, sk)$

Oracle ENCAP(pk)
    $(g, X) \leftarrow pk$
    $y \xleftarrow{\$} \mathbb{Z}_p^*$
    $Y \leftarrow g^y$
    $Z \leftarrow X^y$
    $K \leftarrow \mathsf{H}(Y, Z)$
    return $(Y, K)$

Oracle DECAP(sk, Y)
    $x \leftarrow sk$
    $Z \leftarrow Y^x$
    $K \leftarrow \mathsf{H}(Y, Z)$
    return $K$
```

Figure 10: The Cramer-Shoup variant of the Hashed ElGamal Key Encapsulation scheme as presented in [Bha20]. Identical to the first version presented, however with one subtle difference in the input of the Hash function $\mathsf{H} : \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$. In addition to X^y the Hash function also takes as input the encapsulation Y.

queries as the reduction on input Y checks to see if there exists any Z for which (Y, Z) is a DH-pair and outputs the corresponding K stored in L_1. If there are no known Z that have been queried for earlier, then there would be no pair (Y, Z) for which a key value was previously given and the reduction does not cause any consistency problems by outputting a randomly sampled K for this query. If there does exists such a Z then it will already have been queried by the adversary in a Hash query, thus stored in L_1. L_3 stores singular decapsulation queries Y and the corresponding key value K that was returned for that query. The reduction $\mathcal{B}$ simulates the Hash and Decapsulation oracles in the following way.

- $\mathsf{H}(Y, Z)$: On receiving input (Y, Z) from $\mathcal{A}$ the reduction first checks L_1 to see if they have answered this particular query in the past, if so it outputs the same K. Then it checks to see if (Y, Z) is a DH-pair. If this is the case $\mathcal{B}$ checks to see if $Y = g^y$, and stores Z in *temp* accordingly. In either case the pair is stored in L_2 as a DH-pair. The reduction continues checking in L_3 if any previous decapsulation query on Y was answered, if so $\mathcal{B}$ outputs the same K and updates L_1 with this new full pair. If no earlier decapsulation query on Y was stored, the reduction chooses a K uniform at random, updates L_1 and outputs K. This last part is identical to the rest of the procedure if it turns out (Y, Z) was not a DH-pair.

- DECAP(Y): $\mathcal{B}$ first checks and aborts if $Y = g^y$, otherwise it does the following. It first checks L_2 to see if there exists a known DH-pair containing Y, if this is the case it retrieves K from L_1 and outputs this. If not it checks to see if an earlier decapsulation query on Y was answered and outputs K accordingly. If none of the above are true $\mathcal{B}$ chooses K uniform at random, stores (Y, K) in L_3 and outputs the key K.

By implementing these three tables this way the reduction $\mathcal{B}$ is able to maintain consistency when $\mathcal{A}$'s queries are DH-pairs. Since as before if (Y, Z) is a DH-pair w.r.t $X = g^x$ we would have $\mathsf{H}(Y, Z) = \mathsf{H}(Y, X^y)$. As we have mentioned several times now this implementation of lazy sampling is not memory efficient. A memory efficient reduction must somehow get rid of these three tables L_1, L_2 and L_3, but still maintain consistency. In order to tackle this problem we construct a special PRF $\tilde{F}$. For the proof of efficiency and the fact that the construction is also a PRF we refer to the original construction found in [Bha20] or our similar construction for which a proof can be found in Lemma 4.5.

```
Reduction $\mathcal{B}^{\mathsf{DDH}}(g^x, g^y)$
$K \xleftarrow{\$} \mathcal{K}$
$temp \xleftarrow{\$} \mathbb{G}$
$L_1 = \emptyset$
$L_2 = \emptyset$
$L_3 = \emptyset$
$\hat{b} \leftarrow \mathcal{A}^{\mathsf{H},\mathsf{Decap}}(K, g^x, g^y)$
return $temp$

Oracle Decap$(Y)$
if $Y = g^y$
  return $\perp$
else
  if $\exists Z$ s.t $(Y, Z) \in L_2$
    $(Y, Z, K) \leftarrow L_1$
    return $K$
  elif $Y \in L_3$
    $(Y, K) \leftarrow L_3$
    return $K$
  else
    $K \xleftarrow{\$} \mathcal{K}$
    $L_3 = L_3 \cup (Y, K)$
    return $K$
```

```
Oracle $\mathsf{H}(Y, Z)$
if $(Y, Z, K) \in L_1$
  $(Y, Z, K) \leftarrow L_1$
  return $K$
if $\mathsf{DDH}(g^x, Y, Z) = 1$
  if $Y = g^y$
    $temp \leftarrow Z$
  $L_2 = L_2 \cup (Y, Z)$
  if $(Y, K) \in L_3$
    $(Y, K) \leftarrow L_3$
    $L_1 = L_1 \cup (Y, Z, K)$
    return $K$
  else
    $K \xleftarrow{\$} \mathcal{K}$
    $L_1 = L_1 \cup (Y, Z, K)$
    return $K$
else
  $K \xleftarrow{\$} \mathcal{K}$
  $L_1 = L_1 \cup (Y, Z, K)$
  return $K$
```

Figure 11: The reduction $\mathcal{B}$ playing the GAP-DH security game, $\mathcal{A}$ is an adversary against the IND-CCA security game of the Cramer-Shoup variant of HEG. This security game is in the random oracle model, hence any output K is initially chosen uniformly at random from the key space $\mathcal{K}$. *temp* is the final output of $\mathcal{B}$, holding a guess for $Z = g^{xy}$ at the start of the game. L_1, L_2 and L_3 are storage tables empty at start. L_1 contains all answered queries of *full* pairs. L_2 contains DH-tuples in respect to g^x from earlier queries. L_3 is a set of decapsulation queries of Y that do not have any corresponding Z yet. DDH is the decisional Diffie-Hellman oracle accessible to $\mathcal{B}$. This game is very similar to the ODH security game, however H is ever so slightly different.

Construction 3.1 ([Bha20]). *Let* DDH *be the decisional Diffie-Hellman oracle defined over a cyclic group $\mathbb{G}$ with generator g and prime order p and which for inputs $X, Y, Z \in \mathbb{G}$ returns* 1 *if the input is a DH-Tuple, and* 0 *otherwise. For fixed X we define the function $\tilde{F}_X : \{0,1\}^\lambda \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ as*

$$\tilde{F}_X(k, Y, Z) = \begin{cases} F(k, 0, Y, Z) & \textit{\textbf{for}}\ \mathsf{DDH}(X, Y, Z) = 0 \\ F(k, 1, Y, g) & \textit{\textbf{for}}\ \mathsf{DDH}(X, Y, Z) = 1 \end{cases}$$

Define $F : \{0,1\}^\lambda \times \{0,1\} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ as a PRF.

This PRF will act as our replacement for the random Hash H. For a fixed X, which the reduction defines to be their challenge value g^x and a PRF key k chosen at random from some PRF key space, $\mathcal{B}$ can easily answer all decapsulation and Hash queries without storing them for consistency. By the nature of the Diffie-Hellman attribute of tuples in cyclic groups we are guaranteed that exactly one DH-tuple exists for any pair of (Y, Z) w.r.t X. This means that when answering decapsulation queries $\mathcal{B}$, not having access to a comparable Z, simply replies with $F(k, 1, Y, g)$ as detailed in Construction 3.1. In the

```
Reduction $\mathcal{B}^{\mathsf{DDH}}(g^x, g^y)$
$k \xleftarrow{\$} \{0,1\}^\lambda$
$K \xleftarrow{\$} \mathcal{K}$
$temp \xleftarrow{\$} \mathbb{G}$
$\hat{b} \leftarrow \mathcal{A}^{\mathsf{H},\mathsf{Decap}}(K, g^x, g^y)$
return $temp$

Oracle Decap($Y$)
if $Y = g^y$
    return $\perp$
$K \leftarrow F(k, 1, Y, g)$
return $K$

Oracle H($Y, Z$)
if DDH($g^x, Y, Z$) = 1
    if $Y = g^y$
        $temp \leftarrow Z$
    $K \leftarrow F(k, 1, Y, g)$
    return $K$
else
    $K \leftarrow F(k, 0, Y, Z)$
    return $K$
```

Figure 12: A memory-tight reduction $\mathcal{B}$ playing the GAP-DH security game using $\mathcal{A}$ an adversary against the IND-CCA of the Cramer-Shoup variant of HEG. A key feature of any DH-tuple is that for X fixed there is only one $Z \in \mathbb{G}$ such that (X, Y, Z) is a DH-tuple. This characteristic is what we refer to as the *injectively-map* part of Construction 3.1.

actual implementation of the scheme the game playing oracle $\Pi_{\mathcal{A}}$ has the secret key x, and can always supply $\tilde{F}$ with a DH-pair (Y, Y^x), the output is the same. If the adversary $\mathcal{A}$ were to later query the Hash on a DH-tuple pair (Y, Z), $\mathcal{B}$ checks with its DDH oracle to verify this, and can simply reply with the same $F(k, 1, Y, g)$ knowing that there exists no other possible Z for which this can be the case. Indeed during such a Hash query the reduction also checks to see if $Y = g^y$ and stores Z in *temp* if this is the case, ensuring it can win its own security game. In the case where the Hash query pair (Y, Z) is not a DH-pair, the reduction returns $F(k, 0, Y, Z)$ ensuring consistency by the nature of the PRF if $\mathcal{A}$ where to query on the same input again. There will never be a collision of DH-tuples and non DH-tuples resulting again from the nature of PRFs in general. The reduction is shown in Figure 12, as we can see the three tables have successfully been removed and $\mathcal{B}$ perfectly simulates $\Pi_{\mathcal{A}}$. The reduction is therefore tight, in particular memory-tight, and the conjecture in [ACFK17] is refuted!

At first glance the work done by Bhattacharyya and Ghoshal et al seem to contradict each other. This is not the case. The main difference lies in their assumed base scheme of HEG. The impact of the minor detail in the implementation of the Hash function opened up for the memory efficient implementation of the special PRF constructed. In addition to that the base schemes differ ever so slightly they both have different assumptions of the base algebraic group $\mathbb{G}$. In [GT21] they assume that $\mathbb{G}$ is modeled in the general group model, this roughly means that a third party oracle does all group related calculations for an adversary i.e the binary operation is not known to $\mathcal{A}$. Their works are therefore not contradicting but rather complementary of each other. We continue to build upon the work of Bhattacharyya to build a memory efficient reduction of the Twin Hashed ElGamal scheme, referred to as TEG.

4 Memory-Tightness of Hashed Twin ElGamal

Both of the HEG's mentioned so far are bounded by the hardness of the GAP-DH assumption in the random oracle model. The GAP-DH assumption formalized in Definition 2.19 roughly states that solving the CDH problem with access to a DDH oracle is hard given certain cyclic groups, called Gap-groups. The motivation for the presented Hashed Twin Diffie-Hellman ElGamal scheme, or TEG first presented in [CKS08], is to eliminate the dependency on these types of groups altogether. By removing this dependency we achieve a more general scheme and by using the *injectively-map then prf* technique of [Bha20] prove that this scheme is also memory efficient. The loss in efficiency by essentially doubling the complexity of the Hash inputs is manageable compared to the need of implementing Gap-groups. In fact the plaintext and ciphertext lengths stay the same as in the original Cramer-Shoup version of HEG. We show that the TEG scheme is secure up to the regular CDH assumption even with a DDH oracle. The following lemmas give us the tools we need to construct and prove security of the TEG scheme in the random oracle model.

4.1 Required Tools

First we present the main motivation and core principle for the construction of TEG. The formal definition of the STDH assumption can be found in Definition 2.21.

Lemma 4.1 (**Ordinary DH ⇔ Strong Twin DH**, [CKS08]). *Let $\mathbb{G}$ be a cyclic group generated by g and with prime order p. Let $\mathcal{A}$ be an adversary against the ordinary Diffie-Hellman assumption, also known as the Computational Diffie-Hellman assumption. Further let $\mathcal{B}$ be an adversary against the Twin Diffie-Hellman assumption with access to a Decisional Diffie-Hellman oracle. If one adversary solves its respective problem, the other solves its own problem with the same advantage. i.e*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CDH}} = \mathsf{Adv}_{\mathcal{B}}^{\mathsf{STDH}}.$$

Before we present the proof we must familiarize ourselves with an important trapdoor test. In proving Lemma 4.1 we present two reductions each using the other to solve its own problem. If we can show that they both perfectly simulate each others security game, oracles and all, without any loss of consistency then they are equivalent. The reduction solving the STDH assumption can easily run the CDH adversary and complete its own challenge with the same advantage. The tricky part however arises when the reduction playing the original CDH game wants to utilize the STDH adversary, as the STDH adversary has access to a decisional Diffie-Hellman oracle which needs to be simulated. This by the DDH assumption is no trivial task. However the following trapdoor test is the tool needed to overcome this obstacle and simulate the DDH oracle without knowing the secret values x_1, x_2 or y, all with a negligible probability of *guessing* wrong.

Lemma 4.2 (**DH Trapdoor test**, [CKS08]). *Let $\mathbb{G}$ be a cyclic group generated by g with prime order p. Assume $(v, w, \tilde{w})$ are random group elements given by some function of u, where u is a group element. Let σ and τ be uniform over $\mathbb{Z}_p$ and define $\tilde{u}$ as below,*

$$(\sigma, \tau) \xleftarrow{\$} \mathbb{Z}_p^2,\ \ \tilde{u} = g^{\sigma} u^{\tau},\ \ (v, w, \tilde{w}) \leftarrow f(u).$$

Given the above the following hold,

(i) $\tilde{u}$ is uniformly distributed over $\mathbb{G}$ and is also independent from u.

(ii) Let S be the event that (u, v, w) and $(\tilde{u}, v, \tilde{w})$ are both DH-tuples, and let T be the event that $\tilde{w} = v^{\sigma} w^{\tau}$. Then,

(a) $\Pr[S \wedge \neg T] = 0$.

(b) $\Pr[\neg S \wedge T] = 1/p$.

Proof. (i) Let $r \in \mathbb{Z}_p$ s.t $g^r = u$, then $\tilde{u} = g^{\sigma} \cdot g^{r\tau} = g^{\sigma + r\tau}$ and since $\tilde{u} \in \mathbb{G}$ there exists $s \in \mathbb{Z}_p$ s.t $s = \sigma + r\tau$. For fixed r, s is determined by the random variables σ and τ, both of which are uniform over $\mathbb{Z}_p$. Thus since $\tilde{u}$ is determined by s, it is also uniformly distributed over $\mathbb{G}$ and independent from u.

(ii) As above let $\tilde{u} = g^s = g^{\sigma + r\tau}$ for r fixed.

(a) If S is true, then we must have that $w = v^r$ and $\tilde{w} = v^s$, but $s = \sigma + r\tau$ and implies that $\tilde{w} = v^s = v^{\sigma + r\tau} = v^{\sigma} w^{\tau}$. Clearly $\Pr[S \wedge \neg T] = 0$ holds.

(b) By $\tilde{w} = v^{\sigma} w^{\tau} = v^{s - r\tau} w^{\tau} = v^s \cdot v^{-r\tau} w^{\tau}$ rewrite $\tilde{w} = v^{\sigma} w^{\tau}$ as

$$\tilde{w} v^{-s} = (v^{-r} w)^{\tau}. \tag{1}$$

First look at the case when (u, v, w) is DH, but $(\tilde{u}, v, \tilde{w})$ is not. Then the right hand side of (1) equals $(v^{-r+r})^{\tau} = 1^{\tau}$, but this implies that $\tilde{w} = v^s$ which is a contradiction to the fact that $(\tilde{u}, v, \tilde{w})$ is not a DH-tuple, hence this case cannot happen. Now consider the opposite case, when $(\tilde{u}, v, \tilde{w})$ is DH. Then the left hand side of (1) evaluates to 1 giving

$$1 = (v^{-r} w)^{\tau}. \tag{2}$$

v^{-r} and w are both elements in the cyclic group $\mathbb{G}$ so the part of (2) inside the parenthesis is also an element of $\mathbb{G}$. Since $\mathbb{G}$ is cyclic and of prime order there exists for all $h \in \mathbb{G}$ a $n \in \mathbb{Z}_p$ s.t $h^n = \tilde{h} \in \mathbb{G}$ for any fixed $\tilde{h}$. Therefore the probability of $\neg S \wedge T$ in this case amounts to the probability of $\tau = n$ where $1 = (v^{-r} w)^n$. As τ is uniform over $\mathbb{Z}_p$, which is the space of all choices for n, the probability of hitting a fixed element is $1/p$. The same argument applies when both tuples are not DH.

□

This trapdoor test essentially gives the reduction $\mathcal{A}$ in the following security game a way to simulate $\mathcal{B}$'s DDH oracle. The probability of $\mathcal{A}$ returning a false verification of a DH-tuple is negligible and thus has a negligible impact on the simulation of the security game, even though the secret values are unknown. We formalize in the following proof.

*Proof. (**CDH** $\Leftrightarrow$ **ST-DH**).* Let $\mathcal{A}$ and $\mathcal{B}$ be as defined in Lemma 4.1. First of all, if the CDH problem is easy, then the Strong-Twin Diffie-Hellman problem is easy too. $\mathcal{B}$ can simply utilize $\mathcal{A}$ to compute two sets of Z for (X_1, Y) and (X_2, Y) respectively, and returns the outputs (Z_1, Z_2). Assuming that the X_1 and X_2 are independent from each other the advantage is less than or equal to $\mathcal{A}$'s advantage. For the next part of the proof we utilize Lemma 4.2 to build the algorithm for the reduction $\mathcal{A}$ against the CDH assumption. Upon receiving the challenge values (X, Y), $\mathcal{A}$ uniformly chooses two variables $(\sigma, \tau) \xleftarrow{\$} \mathbb{Z}_p^2$, defines u as $X = X_1$, X_2 as $\tilde{u} = g^\sigma u^\tau$ and initiates $\mathcal{B}$ on (X_1, X_2, Y). $\mathcal{A}$ answers DDH queries on the form (Y, Z_1, Z_2) from $\mathcal{B}$ by checking if $Z_2 = Y^\sigma Z_1^\tau$, and returns 1 if this is the case 0 otherwise, making a mistake with probability at most $1/p$, which is negligible. Finally when $\mathcal{B}$ outputs $(\tilde{Z}_1, \tilde{Z}_2)$ the reduction checks if $\tilde{Z}_2 = Y^\sigma \tilde{Z}_1^\tau$ and outputs $\tilde{Z}_1$. If the equality does not hold it outputs a guess. $\mathcal{A}$ outputs the same $\tilde{Z}_1$ as $\mathcal{B}$ therefore the advantage of $\mathcal{A}$ is the same as the advantage of $\mathcal{B}$ unless if it outputs a guess, at which point it has a probability of $1/p$ to succeed which is again negligible. We conclude,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CDH}} = \mathsf{Adv}_{\mathcal{B}}^{\mathsf{STDH}}.$$

See Figure 13 for reference. □

```
Reduction $\mathcal{A}(X, Y)$
01 $\sigma \xleftarrow{\$} \mathbb{Z}_p$
02 $\tau \xleftarrow{\$} \mathbb{Z}_p$
03 $X_1 \leftarrow X$
04 $X_2 \leftarrow g^\sigma \cdot X_1^\tau$
05 $(\tilde{Z}_1, \tilde{Z}_2) \leftarrow \mathcal{B}^{\mathrm{DDH}}(X_1, X_2, Y)$
06 if $\tilde{Z}_2 = Y^\sigma \tilde{Z}_1^\tau$
07   return $\tilde{Z}_1$
08 else
09   $Z^* \xleftarrow{\$} \mathbb{G}$
10   return $Z^*$

Oracle DDH$(Y, Z_1, Z_2)$
11 if $Z_2 = Y^\sigma Z_1^\tau$
12   return 1
13 else
14   return 0
```

Figure 13: $\mathcal{A}$ is an adversary trying to break the Computational Diffie-Hellman assumption, $\mathcal{B}$ is an adversary against the Strong Twin Diffie-Hellman assumption. DDH is $\mathcal{B}$'s decisional Diffe-Hellman oracle simulated by $\mathcal{A}$. The variables (X_1, X_2, Y, Z_1, Z_2) are respectively the variables $(u, \tilde{u}, v, w, \tilde{w})$ from the Lemma 4.2, and by its proof this reduction has the same advantage as $\mathcal{B}$.

We have in some detail discussed how switching a Hash function with a PRF can improve memory efficiency, and briefly mentioned that if the underlying PRF is secure, then the difference is negligible. We formally prove these claims as they are essential for the main theorem in this section.

Lemma 4.3 ([Bha20]). *For all adversaries $\mathcal{A}$ playing game* G *with a random oracle* H*, denoted* G_{H}*, there exists an adversary $\mathcal{B}$ playing the PRF game against F such that in game* G_F*, where the random oracle is replaced by a PRF F, it holds that*

$$\left|\Pr\left[\mathrm{G}_{\mathsf{H}}^{\mathcal{A}} \to 1\right] - \Pr\left[\mathrm{G}_{F}^{\mathcal{A}} \to 1\right]\right| \leq \mathsf{Adv}_{\mathcal{B},F}^{\mathsf{PRF}}.$$

$$\textbf{LocalTime}(\mathcal{B}) = \textbf{LocalTime}(\mathcal{A}) + \textbf{LocalTime}(\mathrm{G}) + q_{\mathsf{H}}.$$

$$\textbf{LocalMemory}(\mathcal{B}) = \textbf{LocalMemory}(\mathcal{A}) + \textbf{LocalMemory}(\mathrm{G}).$$

q_H denotes the maximum number of queries made to H by $\mathcal{A}$ during the game.

Proof. The only change in $\mathrm{G}_F^{\mathcal{A}}$ is the replacement of H, thus the first equation follows directly from the difference lemma, Lemma 2.4, where $\mathsf{Adv}_{\mathcal{B},F}^{\mathsf{PRF}}$ represents the advantage $\mathcal{B}$ has winning the PRF game against F. Time efficiency of $\mathcal{B}$ is the time it takes to run the algorithm $\mathcal{A}$, the time it takes to simulate the game G and, assuming any query made by $\mathcal{A}$ takes one time unit, the amount of queries made to H by $\mathcal{A}$, denoted q_{H}. The reduction does not run the hash oracle, rather it queries its own oracle F and relays the answer back to $\mathcal{A}$. The memory of the reduction $\mathcal{B}$ includes by definition the memory needed in running $\mathcal{A}$ plus any extra registers needed to run the code of the original game G and variables needed to simulate it. □

The final tool needed before the presentation of the TEG scheme and completion of its security proof is a construction of the function $\tilde{F}$. This PRF embodies the *injectively map then prf* technique discussed earlier, and is essential for memory-tightness. This construction is a simple extension of the one given for the HEG in [Bha20], thus the following construction, Lemma 4.5 and proof follows theirs closely.

Construction 4.4 *Let* DDH *be the decisional Diffie-Hellman oracle defined over a cyclic group $\mathbb{G}$ with generator g and prime order p and which for inputs $X, Y, Z \in \mathbb{G}$ returns* 1 *if the input is a DH-Tuple, and* 0 *otherwise. For fixed X_1 and X_2 we define the function $\tilde{F}_{X_1,X_2} : \{0,1\}^\lambda \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ as*

$$\tilde{F}_{X_1,X_2}(k, Y, Z_1, Z_2) = \begin{cases} F(k, 0, Y, Z_1, Z_2) & \textbf{\textit{for}} \quad \mathsf{DDH}_{X_1,X_2}(Y, Z_1, Z_2) = 0 \\ F(k, 1, Y, g, g) & \textbf{\textit{for}} \quad \mathsf{DDH}_{X_1,X_2}(Y, Z_1, Z_2) = 1 \end{cases}$$

Define $F : \{0,1\}^\lambda \times \{0,1\} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ as a PRF. The proof of the following lemma ensures us that $\tilde{F}$ is also a PRF, and that a reduction against F is memory efficient.

```
Reduction $\mathcal{B}^{F,\mathsf{DDH}}(g,p)$      Oracle $\tilde{\mathsf{F}}(Y, Z_1, Z_2)$
01 $x_1 \xleftarrow{\$} \mathbb{Z}_p^*$                07 $par \leftarrow \psi_{X_1,X_2}(Y, Z_1, Z_2)$
02 $x_2 \xleftarrow{\$} \mathbb{Z}_p^* - \{x_1\}$      08 $K \leftarrow \mathsf{F}(par)$
03 $X_1 \leftarrow g^{x_1}$                 09 return $K$
04 $X_2 \leftarrow g^{x_2}$
05 $\hat{b} \leftarrow \mathcal{A}^{\tilde{\mathsf{F}}}(X_1, X_2)$
06 return $\hat{b}$
```

Figure 14: The reduction $\mathcal{B}$ against a PRF F. F is an oracle answering queries by $\mathcal{B}$ and $\tilde{\mathsf{F}}$ is $\mathcal{A}'s$ oracle run by $\mathcal{B}$. The function ψ is run by $\mathcal{B}$ but during its run $\mathcal{B}$ queries its own DDH oracle not run directly by $\mathcal{B}$.

Lemma 4.5 *If F is a PRF, then $\tilde{F}$ is also a PRF. Furthermore for every adversary $\mathcal{A}$ against $\tilde{F}$ there exists an adversary $\mathcal{B}$ against F such that the following hold:*

$$\mathsf{Adv}_{\mathcal{B},F}^{\mathsf{PRF}} = \mathsf{Adv}_{\mathcal{A},\tilde{F}}^{\mathsf{PRF}}$$

$$\textbf{LocalTime}(\mathcal{B}) = \textbf{LocalTime}(\mathcal{A})) + q$$

$$\textbf{LocalMemory}(\mathcal{B}) \leq \textbf{LocalMemory}(\mathcal{A}) + 11\lambda.$$

q is the number of queries made by $\mathcal{A}$ and λ denotes the size of the registers.

Proof. For fixed $X_1, X_2 \in \mathbb{G}$ define the following function $\psi_{X_1,X_2} : \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \{0,1\} \times \mathbb{G}, \times \mathbb{G} \times \mathbb{G}$ as

$$\psi_{X_1,X_2}(Y, Z_1, Z_2) = \begin{cases} (0, Y, Z_1, Z_2) & \textbf{for} \quad \mathsf{DDH}_{X_1,X_2}(Y, Z_1, Z_2) = 0 \\ (1, Y, 0^\lambda, 0^\lambda) & \textbf{for} \quad \mathsf{DDH}_{X_1,X_2}(Y, Z_1, Z_2) = 1 \end{cases}$$

ψ is an injective function. To see this we only need inspect the DDH oracle. A tuple $(X = g^x, Y = g^y, Z)$ is a DH-tuple if and only if $Z = g^{xy}$. Thus if (X, Y, Z') is also a DH-tuple, we must have that $Z' = g^{xy} = Z$. Now, we have that $\tilde{F} = F \circ \psi$, $\tilde{F}(\mathsf{par}) = F(\psi(\mathsf{par}))$, and since ψ is injective and F is a PRF, $\tilde{F}$ is also a PRF. As for how the reduction works see Figure 14 for reference. $\mathcal{B}$ initializes the PRF game by first choosing two elements $x_1, x_2 \in \mathbb{Z}_p$ uniformly random and storing these as the secret key *sk*. Then $\mathcal{B}$ computes the public key $pk = (X_1, X_2)$, initializes $\mathcal{A}$ and gives it the public key *pk*. When $\mathcal{B}$ receives a query from $\mathcal{A}$ it checks whether both of the tuples are DH-tuples, and forwards this to it's own oracle F. Any answer from F is sent directly to $\mathcal{A}$. When $\mathcal{A}$ returns a bit $\hat{b}$, $\mathcal{B}$ returns the same bit and terminates. This perfectly simulates the PRF game of $\tilde{F}$ and we see that

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F} = \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{A},\tilde{F}}.$$

We assume every query made by $\mathcal{A}$ takes one time unit, and that $\mathcal{A}$ makes at most q queries to its oracle simulated by $\mathcal{B}$. Thus we get that

$$\textbf{LocalTime}(\mathcal{B}) = \textbf{LocalTime}(\mathcal{A}) + q.$$

Finally the memory required by $\mathcal{B}$ is bounded by the local memory of the algorithm $\mathcal{A}$ and registers for storing all variables presented in the reduction. We count the input to $\mathcal{B}$ and all lines, for the oracle query (Y, Z_1, Z_2) to $\tilde{F}$ three registers are needed, plus the memory for temporarily storing *par* and K. totaling at thirteen additional registers and one bit, see Figure 14. Two register and one bit is included in $\mathcal{A}$'s local memory, The worst case memory of the reduction is then

$$\textbf{LocalMemory}(\mathcal{B}) \leq \textbf{LocalMemory}(\mathcal{A}) + 11\lambda.$$

Since the maximum size of each *word* used is determined by the security parameter λ we define the size of any one register to be at least one λ. As we only need a small constant of extra memory for the reduction $\mathcal{B}$ we achieve memory efficiency. □

The upper bound for memory shown here is a rather large increase of memory usage from the original scheme presented in [Bha20]. Even though this is expected by the nature of the construction, it could be more strictly bounded. However it is not clear from the calculations presented by Bhattacharyya what and when certain variables are stored, when registers are already included in other memory terms, or when registers can be reused during a run of an algorithm. In this paper we write a reasonable overestimate that is much clearer to interpret, and will describe in detail which registers are counted and which ones we sweep under other memory terms. Because of this it is expected that we count certain registers more than once, or miss out on reusable registers and thus lose some efficiency. As we still only need a constant of extra memory, the memory-tightness conclusions stay the same.

4.2 Proof of Memory-Tightness

The TEG scheme is presented in Figure 15. The proof follows the one given for the Cramer-Shoup variant of HEG in [Bha20] closely. We have left out the time complexities of the reductions as not to distract from the main result of memory-tightness. We present our first main Theorem.

```
Oracle GEN(λ)
01 $(\mathbb{G}, g, p) \leftarrow \mathsf{Gen}(\lambda)$
02 $x_1 \xleftarrow{\$} \mathbb{Z}_p^*$
03 $x_2 \xleftarrow{\$} \mathbb{Z}_p^* - \{x_1\}$
04 $X_1 \leftarrow g^{x_1}$
05 $X_2 \leftarrow g^{x_2}$
06 $pk \leftarrow (g, X_1, X_2)$
07 $sk \leftarrow (x_1, x_2)$
08 return $(pk, sk)$

Oracle ENCAP(pk)
09 $(g, X_1, X_2) \leftarrow pk$
10 $y \xleftarrow{\$} \mathbb{Z}_p^*$
11 $Y \leftarrow g^y$
12 $Z_1 \leftarrow X_1^y$
13 $Z_2 \leftarrow X_2^y$
14 $K \leftarrow \mathsf{H}(Y, Z_1, Z_2)$
15 return $(Y, K)$

Oracle DECAP(sk, Y)
16 $(x_1, x_2) \leftarrow sk$
17 $Z_1 \leftarrow Y^{x_1}$
18 $Z_2 \leftarrow Y^{x_2}$
19 $K \leftarrow \mathsf{H}(Y, Z_1, Z_2)$
20 return $K$
```

Figure 15: Hashed Twin ElGamal Key Encapsulation scheme, denoted TEG. $\mathsf{H} : \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ is a hash function.

Theorem 4.6 *Let g be the generator of a cyclic group $\mathbb{G}$ of prime order p. Let* DDH *be the Decisional Diffie-Hellman oracle on $\mathbb{G}$, and let $F : \{0,1\}^\lambda \times \{0,1\} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ be a PRF. Let $\mathcal{A}$ be an adversary against the Hashed Twin ElGamal Key Encapsulation scheme in the* IND-CCA *game with security parameter λ. Suppose $\mathcal{A}$ makes q_H hash queries and q_D decapsulation queries. Then, in the random oracle model, there exists adversaries $\mathcal{B}$ against the PRF F and $\mathcal{C}$ against the Computational Diffie-Hellman assumption such that*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CCA}} \leq \mathsf{Adv}_{\mathcal{B},F}^{\mathsf{PRF}} + \mathsf{Adv}_{\mathcal{C}}^{\mathsf{CDH}}.$$

Additionally we get the following memory bounds on the reductions,

$$\mathbf{LocalMemory}(\mathcal{B}_F) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(\mathsf{Gen}) + 27\lambda + 1,$$

$$\mathbf{LocalMemory}(\mathcal{C}_{\mathsf{CDH}}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(F) + 14\lambda + 1.$$

The reductions are in other words memory-tight.

Proof. The sequences of games is presented in Figure 16.

Game G_0. This is the IND-CCA game and we define the advantage of an adversary $\mathcal{A}$ as

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CCA}} = \left| \Pr[\mathsf{G}_0^{\mathcal{A}} \to 1] - \frac{1}{2} \right|. \tag{1}$$

Game G_1. In game G_1 we choose both K_0^* and K_1^* uniform at random from the key space $\mathcal{K}$. Hence from the adversary's point of view the games are identical and we have

$$\Pr[\mathsf{G}_0^{\mathcal{A}} \to 1] = \Pr[\mathsf{G}_1^{\mathcal{A}} \to 1]. \tag{2}$$

If the adversary somehow could detect the change it would also be able, in the original game, to decide when K_b is random or real, with the same accuracy as in this game. So in either case the advantage is the same.

```
Games $G_0$-$G_4$
01 $b \xleftarrow{\$} \{0,1\}$
02 $k \xleftarrow{\$} \{0,1\}^\lambda$                      // $G_2$-$G_4$
03 $(pk, sk) \xleftarrow{\$} \textsc{Gen}(\lambda)$
04 $y^* \xleftarrow{\$} \mathbb{Z}_p^*$
05 $Y^* \leftarrow g^{y^*}$
06 $Z_1^* \leftarrow X_1^{y^*}$
07 $Z_2^* \leftarrow X_2^{y^*}$
08 $K_0^* \leftarrow \mathsf{H}(Y^*, Z_1^*, Z_2^*)$           // $G_0$
09 $K_0^* \xleftarrow{\$} \mathcal{K}$                        // $G_1$-$G_4$
10 $K_1^* \xleftarrow{\$} \mathcal{K}$
11 $b' \leftarrow \mathcal{A}^{\textsc{Decap},\mathsf{H}}(pk, Y^*, K_b^*)$
12 if $b = b'$
13     return 1
14 else
15     return 0

Oracle $\textsc{Decap}_{sk}(Y)$
16 if $Y = Y^*$
17     return $\perp$
18 $Z_1 \leftarrow Y^{x_1}$                                  // $G_0$-$G_2$
19 $Z_2 \leftarrow Y^{x_2}$                                  // $G_0$-$G_2$
20 $K \leftarrow \mathsf{H}(Y, Z_1, Z_2)$                    // $G_0$-$G_1$
21 $K \leftarrow \tilde{F}_{X_1,X_2}(k, Y, Z_1, Z_2)$        // $G_2$
22 $K \leftarrow F(k, 1, Y, g, g)$                           // $G_3$-$G_4$
23 return $K$
```

```
Oracle $\mathsf{H}(Y, Z_1, Z_2)$
24 if $\mathsf{H}(Y, Z_1, Z_2)$ undefined                                  // $G_0$
25     $\mathsf{H}(Y, Z_1, Z_2) \xleftarrow{\$} \mathcal{K}$                // $G_0$
26 return $\mathsf{H}(Y, Z_1, Z_2)$                                        // $G_0$
27 if $Z_1 = Y^{x_1} \wedge Z_2 = Y^{x_2} \wedge Y = Y^*$                  // $G_1$-$G_4$
28     return $K_0^*$                                                      // $G_1$-$G_3$
29     $\textsc{Flag} = 1$                                                 // $G_4$
30     Abort                                                               // $G_4$
31 elif $Z_1 = Y^{x_1} \wedge Z_2 = Y^{x_2}$                               // $G_3$-$G_4$
32     $K \leftarrow F(k, 1, Y, g, g)$                                     // $G_3$-$G_4$
33 else                                                                    // $G_1$-$G_4$
34     if $\mathsf{H}(Y, Z_1, Z_2)$ undefined                              // $G_1$
35         $\mathsf{H}(Y, Z_1, Z_2) \xleftarrow{\$} \mathcal{K}$            // $G_1$
36     return $\mathsf{H}(Y, Z_1, Z_2)$                                    // $G_1$
37     $K \leftarrow \tilde{F}_{X_1,X_2}(k, Y, Z_1, Z_2)$                  // $G_2$
38     $K \leftarrow F(k, 0, Y, Z_1, Z_2)$                                 // $G_3$-$G_4$
39 return $K$                                                              // $G_2$-$G_4$
```

Figure 16: IND-CCA game of TEG. H is the encapsulation oracle accessible to $\mathcal{A}$. $\tilde{F}$ and F are PRFs only accessible to $\mathcal{A}$ through the encapsulation oracle. The notation of $//\mathrm{G}_i$ denotes which part of the pseudo code that is active in each game, if no such tag exists that part is always active.

Game G_2. Replace the hash H with the PRF $\tilde{F}$, described in Construction 4.4, for encapsulation and decapsulation queries made by $\mathcal{A}$. This function is otherwise inaccessible to $\mathcal{A}$. By the difference lemma:

$$\left|\Pr[\mathsf{G}_1^{\mathcal{A}} \to 1] - \Pr[\mathsf{G}_2^{\mathcal{A}} \to 1]\right| \leq \mathsf{Adv}_{\mathcal{B},\tilde{F}}^{\mathsf{PRF}}. \tag{3}$$

Game G_3. Following the construction the PRF $\tilde{F}$ is replaced by the PRF F. The lines 18 and 19 in Figure 16 ensure that the decapsulation oracle always returns $\tilde{F}_{X_1,X_2}(Y, Z_1, Z_2) = F(k, 1, Y, g, g)$ during G_2. Hence from the adversary's point of view the game is unchanged:

$$\Pr[\mathsf{G}_2^{\mathcal{A}} \to 1] = \Pr[\mathsf{G}_3^{\mathcal{A}} \to 1]. \tag{4}$$

Game G_4. Line 09 and 10 in Figure 16 ensure both K_b^* are uniform random and since the bit b is also uniform random the adversary's advantage can be no better:

$$\Pr[\mathsf{G}_4^{\mathcal{A}} \to 1] = \frac{1}{2}. \tag{5}$$

```
**Reduction** $\mathcal{C}^{\mathsf{DDH}}(g, X_1, X_2, Y^*)$
$pk \leftarrow (g, X_1, X_2)$
$k \xleftarrow{\$} \{0,1\}^\lambda$
$K^* \xleftarrow{\$} \mathcal{K}$
$temp \xleftarrow{\$} \mathbb{G}^2$
$b' \leftarrow \mathcal{A}^{\mathrm{H},\textsc{Decap}}(pk, Y^*, K^*)$
return $temp$

**Oracle** $\textsc{Decap}(Y)$
if $Y = Y^*$
    return $\perp$
$K \leftarrow F(k, 1, Y, g, g)$
return $K$

**Oracle** $\mathrm{H}(Y, Z_1, Z_2)$
if $\mathsf{DDH}_{X_1,X_2}(Y, Z_1, Z_2) = 1$
    if $Y = Y^*$
        $temp \leftarrow (Z_1, Z_2)$
    else
        $K \leftarrow F(k, 1, Y, g, g)$
else
    $K \leftarrow F(k, 0, Y, Z_1, Z_2)$
return $K$
```

Figure 17: The reduction $\mathcal{C}$ against STDH, with access to its own DDH oracle, simulates oracles $\textsc{Decap}$ and H as shown, rewriting the *temp* variable in case $\mathcal{A}$ queries on a Diffie-Hellman tuple pair. In the case where no such pair is obtained the reduction chooses two random group elements and outputs these.

This is true because in this final version we abort the game if $\mathcal{A}$ queries the encapsulation oracle on valid Diffie-Hellman tuples $Y^{x_1} = Z_1$ and $Y^{x_2} = Z_2$ when $Y = Y^*$. Now the adversary cannot compute $\mathsf{H}(Y^*, Z_1^*, Z_2^*)$, and will never know if K_b^* is truly random or not. When $Y = Y^*$ the decapsulation oracle returns $\perp$, ensuring that $\mathcal{A}$ cannot obtain information from trivially decapsulating Y^*. We apply the difference lemma again and get that

$$\left|\Pr[\mathsf{G}_3^{\mathcal{A}} \to 1] - \Pr[\mathsf{G}_4^{\mathcal{A}} \to 1]\right| \leq \Pr[\textsc{Flag} = 1]. \tag{6}$$

$\textsc{Flag} = 1$ is set when the adversary queries the encapsulation oracle on (Y, Z_1, Z_2) so that the tuples (X_1, Y, Z_1), (X_2, Y, Z_2) are valid Diffie-Hellman tuples. This means that it has successfully broken the Strong Twin Diffie-Hellman assumption, and is by construction of G_4, its only chance to win. To see this we construct the reduction $\mathcal{C}$ against the Strong Twin Diffie-Hellman assumption, see Figure 17. The reduction $\mathcal{C}$ is initialized on challenge values (X_1, X_2, Y^*), it defines the public key as $pk = (g, X_1, X_2)$, chooses a PRF key k, picks K^* uniformly random from the key space, and initializes $\mathcal{A}$. On receiving encapsulation queries from $\mathcal{A}$, $\mathcal{C}$ first checks with its own DDH oracle if the tuples are DH. It then checks if these tuples are valid tuples for the challenge variables $(Y^* = Y)$, if this is the case, $\mathcal{C}$ returns the corresponding $Z's$ and wins its game. If not, it computes K using the prf F and returns K to $\mathcal{A}$. By the end of the game if no true DH pair on the challenge variables is obtained, the reduction outputs two randomly chosen group elements. Applying Lemma 4.1 we get the following bound

$$\Pr[\textsc{Flag} = 1] = \mathsf{Adv}_{\mathcal{C}}^{\mathsf{STDH}} = \mathsf{Adv}_{\mathcal{C}}^{\mathsf{CDH}}. \tag{7}$$

By summarizing all the equations, (1),(2),(3),(4),(5),(6),(7), and by Lemma 4.5, we conclude that:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CCA}} \leq \mathsf{Adv}_{\mathcal{B},F}^{\mathsf{PRF}} + \mathsf{Adv}_{\mathcal{C}}^{\mathsf{CDH}}. \tag{8}$$

Memory-Tightness. Finally we show the memory usage of the reductions $\mathcal{B}$ and $\mathcal{C}$. By

Lemma 4.5 we have

$$\mathbf{LocalMemory}(\mathcal{B}_F) \leq \mathbf{LocalMemory}(\mathcal{B}_{\tilde{F}}) + 11\lambda.$$

By Lemma 4.3

$$\mathbf{LocalMemory}(\mathcal{B}_{\tilde{F}}) = \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(\mathsf{G}_2).$$

For G_2 we count the needed registers to store sk and pk in line 03 in $\mathbf{LocalMemory}(\textsc{Gen})$. We count 7 registers for variables in lines 02, 04, 05, 06, 07, 09, 10, in Figure 16, plus one bit for b. Line 11 is included in $\mathbf{LocalMemory}(A)$. For decapsulation and H queries we need four additional registers. Summing up so far we get 11 registers and one bit. The key generation algorithm from Figure 15 is needed in the setup of the game, and must be included, totaling at $\mathbf{LocalMemory}(\mathsf{Gen}) + 5\lambda$, the needed registers for storing the public and secret key are as mentioned absorbed into this term. Our estimate is then,

$$\mathbf{LocalMemory}(\mathsf{G}_2) \leq \mathbf{LocalMemory}(\mathsf{Gen}) + 16\lambda + 1.$$

The three registers and one bit needed in line 11 of G_2 we count in the local memory of $\mathcal{A}$, already accounted for. Thus adding up we get the $\mathcal{B}_F$'s memory efficiency to be,

$$\mathbf{LocalMemory}(\mathcal{B}_F) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(\mathsf{Gen}) + 27\lambda + 1.$$

Lastly we calculate the memory efficiency of $\mathcal{C}$ against the CDH assumption. From the reduction in Figure 13 we count 6 registers for initialization and one bit for answering the DDH oracle simulated by the reduction, additionally we count two registers for storing Z_1 and Z_2 but we include these registers in the local memory of the adversary against the Strong Twin Diffie-Hellman assumption counted in line 5. Thus

$$\mathbf{LocalMemory}(\mathcal{C}_{\mathsf{CDH}}) \leq \mathbf{LocalMemory}(\mathcal{C}_{\mathsf{STDH}}) + 6\lambda + 1.$$

To calculate $\mathcal{C}_{\mathsf{CDH}}$ we inspect $\mathcal{C}_{\mathsf{STDH}}$. Referring to Figure 17 we count 8 registers, the bit b' and 3 registers needed to initialize $\mathcal{A}$ included in $\mathcal{A}$'s local memory, and gives us the following estimate

$$\mathbf{LocalMemory}(\mathcal{C}_{\mathsf{STDH}}) \leq \mathbf{LocalMemory}(A) + \mathbf{LocalMemory}(F) + 8\lambda.$$

Adding up we get our final estimate

$$\mathbf{LocalMemory}(\mathcal{C}_{\mathsf{CDH}}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(F) + 14\lambda + 1.$$

This completes the proof. □

5 Memory-Tightness of HPS Hashed ElGamal

The motivation for this section is to see if we can replace the *injectively map* part of Bhattacharyya's technique with a more general construction such as the Hash Proof System. Hash Proof Systems, or HPS, have deterministic encryption algorithms. This is of interest to us since this means that for every element X in the language $\mathcal{L}$ of the HPS there is exactly one output Z, in other words it is an injective function when restricted on $\mathcal{L}$, (Definition 2.23). In Bhattacharyya's technique it was precisely the one to one relation between elements Y in the group $\mathbb{G}$ and the corresponding Diffie-Hellman element Z in the DH-tuple (X, Y, Z) for a fixed X that made their construction possible. There happens to exist an ElGamal variant of a HPS, so we naturally start with a KEM constructed from this system, we will refer to it as the Hash Proof System Hashed ElGamal, or HPS-HEG for short, see Figure 18. It is almost identical to the PKE presented in [HK09], and the underlying HPS is a simplification of the scheme presented in [CS98].

```
Oracle Gen_HPS($\lambda$)
    $(\mathbb{G}, g, p) \leftarrow \mathsf{Gen}(\lambda)$
    $g_1 \leftarrow g$
    $t \xleftarrow{\$} \mathbb{Z}_p^*$
    $g_2 \leftarrow g_1^t$
    $a_1 \xleftarrow{\$} \mathbb{Z}_p^*$
    $a_2 \xleftarrow{\$} \mathbb{Z}_p^* - \{a_1\}$
    $sk \leftarrow (a_1, a_2)$
    $pk \leftarrow (g_1, g_2, g_1^{a_1} \cdot g_2^{a_2})$
    return $(pk, sk)$
```

```
Oracle Encap($pk$)
    $w \xleftarrow{\$} \mathbb{Z}_p^*$
    $X \leftarrow (g_1^w, g_2^w)$
    $Z \leftarrow \mathsf{Pub}(pk, X, w)$
    $K \xleftarrow{\$} \mathsf{H}(X, Z)$
    return $(X, K)$

Oracle Pub($pk, X, w$)
    $g_1^{a_1} \cdot g_2^{a_2} \leftarrow pk$
    $Z \leftarrow (g_1^{a_1} \cdot g_2^{a_2})^w$
    return $Z$
```

```
Oracle Decap($sk, X$)
    $Z \leftarrow \mathsf{Priv}(sk, X)$
    $K \xleftarrow{\$} \mathsf{H}(X, Z)$
    return $K$

Oracle Priv($sk, X$)
    $(a_1, a_2) \leftarrow sk$
    $(X_1, X_2) \leftarrow X$
    $Z \leftarrow (X_1^{a_1} \cdot X_2^{a_2})$
    return $Z$
```

Figure 18: A KEM based on a ElGamal variant of a HPS, we denote by HPS-HEG. $\mathsf{H} : \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ is a Hash function. The group elements g_1 and g_2 are included in $Par = (\mathbb{G}, g_1, g_2, p)$ given as public information after the HPS generator has generated the key pair (pk, sk). When referring to the public key pk, we will mostly be referring to $h = g_1^{a_1} \cdot g_2^{a_2}$.

5.1 CPA proof of HPS Hashed ElGamal

As a starting point for future discussion, we prove that under the IND-CPA security game the HPS-HEG is memory-tight by simply replacing the cryptographic Hash with a PRF in the random oracle model. Because of Lemma 4.3 and the simplicity of the IND-CPA game we can easily do this without the need for any special constructions or *tricks*. The HPS used in our construction of the scheme in Figure 18, has a hard SM problem and fulfills the definition of universality stated in Definition 2.25 and Definition 2.24 respectively. Although we will not need the hardness assumption of SM to formally prove Theorem 5.2 we still state and prove both universality and the SM assumption before proceeding.

Lemma 5.1 ([CS98]). *The ElGamal variant of the* HPS *presented in Figure 18 is a universal* HPS *with* SM *problem bounded by the* DDH *assumption as shown,*

$$\mathsf{Adv}_{\mathcal{B}}^{\mathsf{SM}} = \mathsf{Adv}_{\mathcal{A}}^{\mathsf{DDH}}.$$

The memory used by the reduction against the DDH *assumption totals at,*

$$\mathbf{LocalMemory}(\mathcal{A}) \leq \mathbf{LocalMemory}(\mathcal{B}) + 4\lambda.$$

Proof. For correctness of the HPS see that when $X \in \mathcal{L}$, we must have $\mathsf{Pub}(pk, X, w) = Z = \mathsf{Priv}(sk, X)$ by,

$$pk^w = g_1^{a_1 w} \cdot g_2^{a_2 w} = g_1^{w a_1} \cdot g_2^{w a_2} = X_1^{a_1} \cdot X_2^{a_2}.$$

When $X \in \mathcal{X}/\mathcal{L}$, thus $w \neq w^{'}$, we have universality. To see this fix $X = (g_1^w, g_2^{w^{'}}) \in \mathcal{X}/\mathcal{L}$, we show that the distribution of $(pk, \mathsf{Priv}(sk, X))$ is that of two independent and random group elements. Take the function $f : \mathbb{Z}_p^2 \to \mathbb{G}^2$,

$$f(a_1, a_2) = (pk, Z) = (g_1^{a_1} \cdot g_2^{a_2}, X_1^{a_1} \cdot X_2^{a_2}).$$

This function is injective, and thus the output is completely determined by the input, which are two independent and random group elements. We prove the above by inspecting the function $\tilde{f} : \mathbb{Z}_p^2 \to \mathbb{Z}_p^2$,

$$\tilde{f}(a_1, a_2) = \log_{g_1}(f(a_1, a_2)) = (a_1 + a_2 t, w a_1 + w^{'} a_2 t).$$

$\tilde{f}$ is a linear map can be represented by a linear transformation,

$$\tilde{f}(a_1, a_2) = M \begin{pmatrix} a_1 & a_2 \end{pmatrix}, \text{ with } M = \begin{pmatrix} 1 & t \\ w & tw^{'} \end{pmatrix}.$$

The determinant of M is

$$\det(M) = tw^{'} - tw = t(w^{'} - w) \neq 0.$$

Hence there exists an inverse to M and $\tilde{f}$ is injective and therefore f is also. Finally we bound the SM problem, see Figure 19. First see that $\mathsf{Adv}_{\mathcal{B}}^{\mathsf{SM}} \leq \mathsf{Adv}_{\mathcal{A}}^{\mathsf{DDH}}$. By the left hand side of Figure 19 this game perfectly simulates the DDH security game. Since X^* is a tuple of two, and g_2 is a known group element to all parties in the SM security game, $\mathcal{B}$ can easily construct a tuple of three which if in the case it is a DDH-tuple will mean that $X^* \in \mathcal{L}$. In fact if it is not a DDH-tuple, then $X^* \notin \mathcal{L}$. By Definition 2.18 $\mathcal{B}$ need not simulate any oracles, thus the reduction simulates the DDH game perfectly. In the opposite case $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{DDH}} \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{SM}}$ see the right hand side of Figure 19. Again the reduction can easily construct a challenge value X^* by the same arguments as before, this time simulating a language $\mathcal{L}$ with Y. The Sample oracles can easily be simulated, but are redundant as they don't actually give any information that $\mathcal{B}$ does not know or cannot obtain themselves assuming $\mathcal{A}$ includes $g_2 = Y$ in *Par*. We conclude that

$$\mathsf{Adv}_{\mathcal{B}}^{\mathsf{SM}} = \mathsf{Adv}_{\mathcal{A}}^{\mathsf{DDH}}.$$

Memory-tightness. The reduction $\mathcal{A}$ needs registers to store lines 06, 07 worth 3λ and a single bit for line 08. In addition it needs enough memory to run $\mathcal{B}$, this includes the input and output to $\mathcal{B}$, which covers two of the three registers needed to simulate the Sample oracles and the single bit. Hence

$$\mathbf{LocalMemory}(\mathcal{A}) \leq \mathbf{LocalMemory}(\mathcal{B}) + 4\lambda.$$

□

```
**Reduction** $\mathcal{B}(X^*)$
$X \leftarrow X_1^*$
$Y \leftarrow g_2$
$Z \leftarrow X_2^*$
$\hat{b} \leftarrow \mathcal{A}(X, Y, Z)$
**return** $\hat{b}$
```

```
**Reduction** $\mathcal{A}(X, Y, Z)$
$g_2 \leftarrow Y$
$X^* \leftarrow (X, Z)$
$\hat{b} \leftarrow \mathcal{B}^{\mathsf{Sample}_{\mathcal{L}}, \mathsf{Sample}_{\mathcal{X}/\mathcal{L}}}(X^*)$
**return** $\hat{b}$
```

```
**Oracle** $\mathsf{Sample}_{\mathcal{L}}()$
$w \xleftarrow{\$} \mathbb{Z}_p^*$
$X_{\mathcal{L}} \leftarrow (g^w, Y^w)$
**return** $(X_{\mathcal{L}}, w)$

**Oracle** $\mathsf{Sample}_{\mathcal{X}/\mathcal{L}}()$
$w \xleftarrow{\$} \mathbb{Z}_p^*$
$w' \xleftarrow{\$} \mathbb{Z}_p^*/w$
$X_{\mathcal{X}/\mathcal{L}} \leftarrow (g^w, Y^{w'})$
**return** $X_{\mathcal{X}/\mathcal{L}}$
```

Figure 19: On the left hand side is the reduction $\mathcal{B}$ against the SM problem utilizing an adversary $\mathcal{A}$ playing the DDH security game as defined in Definition 2.18. $X^* = (g_1^w, g_2^{w'})$, where $X_1^* = g_1^w$ and $X_2^* = g_2^{w'}$. The oracles $\mathsf{Sample}_{\mathcal{L}}$ and $\mathsf{Sample}_{\mathcal{X}/\mathcal{L}}$ are oracles that $\mathcal{B}$ has access to, but they are not needed and hence omitted here. $\mathcal{B}$ wins if it guesses the bit b correctly, where $b = 0$ means that $X^* \in \mathcal{L}$. On the right hand side we have $\mathcal{A}$ playing the DDH security game using $\mathcal{B}$. In this game $\mathcal{A}$ has to simulate a language, thus we included how $\mathcal{A}$ easily can do this by simulating the Sample oracles.

The matter of generalizing the *injectively map* with a private evaluation function seems to be unachievable for the HPS-HEG scheme. The main problem is that in order to construct a well defined reduction we must extract from $\mathcal{A}$ information for solving the SM assumption, but $\mathcal{A}$ is not trying to break the DDH assumption when playing the IND-CPA game, or IND-CCA game for HPS-HEG. We discuss this in more detail after we first present our second main Theorem, where we must use the GAP-DH assumption.

Theorem 5.2 *Let g be a generator of a cyclic group $\mathbb{G}$ of prime order p. Let $F : \{0,1\}^\lambda \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \to \mathcal{K}$ be a* PRF. *Let $\mathcal{A}$ be an adversary against the* HPS-HEG *in the* IND-CPA *game with security parameter λ. If $\mathcal{A}$ makes q_{H} Hash queries, then in the random oracle model, there exists an adversary $\mathcal{B}$ against the* PRF F *and an adversary $\mathcal{C}$ against the* GAP-DH *assumption such that*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}} \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{PRF}} + \mathsf{Adv}_{\mathcal{C}}^{\mathsf{GAP\text{-}DH}} + \frac{q_{\mathsf{H}}}{p}.$$

Additionally we get the following memory bounds on the reductions,

$$\mathbf{LocalMemory}(\mathcal{B}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(\mathsf{Gen}) + 15\lambda + 1,$$

$$\mathbf{LocalMemory}(\mathcal{C}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(F) + 3\lambda.$$

Proof. The sequence of games is presented in Figure 20 and follows the same pattern as the proof for the TEG scheme.

Game G_0. We start of with the definition of $\mathcal{A}$'s advantage in the IND-CPA security game,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}} := \left| \Pr[\mathsf{G}_0^{\mathcal{A}} \to 1] - \frac{1}{2} \right|. \tag{1}$$

```
Games G_0-G_3
01 $b \xleftarrow{\$} \{0,1\}$
02 $k \xleftarrow{\$} \{0,1\}^\lambda$                // G_2-G_3
03 $(pk, sk) \xleftarrow{\$} \textsc{Gen}_{\text{HPS}}(\lambda)$
04 $w^* \xleftarrow{\$} \mathbb{Z}_p^*$
05 $X^* \leftarrow (g_1^{w^*}, g_2^{w^*})$
06 $Z^* \leftarrow \mathsf{Pub}(pk, X^*, w^*)$        // G_0
07 $K_0^* \xleftarrow{\$} \mathsf{H}(X^*, Z^*)$         // G_0
08 $K_0^* \xleftarrow{\$} \mathcal{K}$                 // G_1-G_3
09 $K_1^* \xleftarrow{\$} \mathcal{K}$
10 $\hat{b} \leftarrow \mathcal{A}^{\mathsf{H}}(pk, X^*, K_b^*)$
11 if $\hat{b} = b$
12    return 1
13 else
14    return 0
```

```
Oracle H(X, Z)
15 if $\mathsf{H}(X, Z)$ undefined                       // G_0
16    $\mathsf{H}(X, Z) \xleftarrow{\$} \mathcal{K}$                  // G_0
17 return $\mathsf{H}(X, Z)$                             // G_0
18 if $Z = \mathsf{Priv}(sk, X) \wedge X = X^*$          // G_1-G_3
19    return $K_0^*$                                     // G_1-G_2
20    $\textsc{Flag} = 1$                                // G_3
21    Abort                                              // G_3
22 else                                                  // G_1-G_3
23    if $\mathsf{H}(X, Z)$ undefined                    // G_1
24       $\mathsf{H}(X, Z) \xleftarrow{\$} \mathcal{K}$               // G_1
25    return $\mathsf{H}(X, Z)$                          // G_1
26    $K \leftarrow F(k, X, Z)$                          // G_2-G_3
27 return $K$                                            // G_2-G_3
```

Figure 20: IND-CPA game of HPS-HEG presented in Figure 18. $\textsc{Gen}_{\text{HPS}}$ is the HPS key generator, H is a cryptographic Hash function, $X = (g_1^w, g_2^{w'})$ where $X \in \mathcal{L}$ if $w = w'$ for a language $\mathcal{L}$. Priv and Pub are the evaluation algorithms of the HPS both deterministic. F is a PRF.

Game G_1. Choose $K_0 \xleftarrow{\$} \mathcal{K}$ as in line 10 in Figure 20. Then the games G_0 and G_1 are indistinguishable assuming H is a proper collision resistant cryptographic Hash function.

$$\Pr[\mathsf{G}_0^{\mathcal{A}} \to 1] = \Pr[\mathsf{G}_1^{\mathcal{A}} \to 1]. \tag{2}$$

Game G_2. In this step we swap out the Hash function with a PRF F. By Lemma 4.3,

$$\left|\Pr[\mathsf{G}_1^{\mathcal{A}} \to 1] - \Pr[\mathsf{G}_2^{\mathcal{A}} \to 1]\right| \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{PRF}}. \tag{3}$$

Game G_3. In the third and final step we add the lines 20 and 21, aborting if an adversary ever queries on a valid pair (X, Z) where $X = X^*$. Thus winning game G_3 is strictly bounded by the choice of the bit b, which is uniformly random over $\{0, 1\}$,

$$\Pr[\mathsf{G}_3^{\mathcal{A}} \to 1] = \frac{1}{2}. \tag{4}$$

Continuing the difference lemma gives,

$$\left|\Pr[\mathsf{G}_3^{\mathcal{A}} \to 1] - \Pr[\mathsf{G}_3^{\mathcal{A}} \to 1]\right| \leq \Pr[\textsc{Flag} \to 1]. \tag{5}$$

In the case of bounding $\Pr[\textsc{Flag} \to 1]$ we inspect the correctness of Priv and Pub. Since $X^* \in \mathcal{L}$ always is the case in G_3 there is no way for $\mathcal{A}$ to obtain $Z = \mathsf{Priv}(sk, X^*)$ other than to randomly guess Z or by computing $Z = pk^{w^*}$. $\mathcal{A}$ has access to $g^{a_1+ta_2}$ and g^{w^*}, so in computing Z for $X^* \in \mathcal{L}$ they would have to solve the following,

$$Z = pk^{w^*} = g_1^{(a_1+ta_2)w^*}.$$

This is equivalent to the CDH problem for $y = a_1 + ta_2$ and $x = w^*$. However in order to construct a well defined CDH adversary using only $\mathcal{A}$ and staying memory-tight, we construct a reduction in Figure 21 against the GAP-DH security game. $\mathcal{C}$ Simulates a language by choosing a t such that it can generate a challenge value $X^* = (g_1^x, (g^x)^t)$ for $g_1 = g$ and $g_2 = g^t$. The reduction then proceeds to generate a public key $pk = g_1^y \cdot (g^y)^t$ with unknown $a_1 = y = a_2$ as a simulated secret key. $\mathcal{C}$ does not need to know the actual secret key $sk = (y, y)$ as they can query their DDH oracle as shown to check whether,

$$\mathsf{Pub}(pk, x) = pk^x = g^{yx+tyx} = Z = g^{xy+txy} = X_1^y \cdot X_2^y = \mathsf{Priv}((y, y), X^*).$$

For if the DDH oracle returns 1 on input (X_1, pk, Z) we must have $Z = pk^x$. The reduction, knowing t and thus the inverse to $(t+1)$, can easily obtain g^{xy} by the following calculation,

$$Z^{(t+1)^{-1}} = (g^{yx+tyx})^{(t+1)^{-1}} = (g^{yx})^{\frac{(t+1)}{(t+1)}} = g^{xy}.$$

$\mathcal{C}$ returns this g^{xy} and wins its game with the same probability of $\mathcal{A}$ triggering line 20 in Figure 20. Notice that in line 11 of Figure 21, the reduction $\mathcal{C}$ does not need $X \in \mathcal{L}$, only that $X_1 = g^x$, but the probability of $\mathcal{C}$ obtaining the correct Z when $X \notin \mathcal{L}$ is q_{H}/p. We add the last summand to account for this,

$$\Pr[\textsc{Flag} \to 1] = \mathsf{Adv}_{\mathcal{C}}^{\mathsf{GAP\text{-}DH}} + \frac{q_{\mathsf{H}}}{p}. \tag{6}$$

Our final bound is therefore,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}} \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{PRF}} + \mathsf{Adv}_{\mathcal{C}}^{\mathsf{GAP\text{-}DH}} + \frac{q_{\mathsf{H}}}{p}.$$

```
Reduction $\mathcal{C}^{\mathsf{DDH}}(g, g^x, g^y)$
01 $k \xleftarrow{\$} \{0,1\}^\lambda$
02 $t \xleftarrow{\$} \mathbb{Z}_p^*$
03 $(t+1)^{-1} \leftarrow \mathbb{Z}_p^*$
04 $X^* \leftarrow (g^x, g^{xt})$
05 $pk \leftarrow (g^y \cdot g^{yt})$
06 $K^* \xleftarrow{\$} \mathcal{K}$
07 $temp \xleftarrow{\$} \mathbb{G}$
08 $\hat{b} \leftarrow \mathcal{A}^{\mathsf{H}}(pk, X^*, K^*)$
09 return $temp$

Oracle $\mathsf{H}(X, Z)$
10 $(X_1, X_2) \leftarrow X$
11 if $\mathsf{DDH}(X_1, pk, Z) = 1$
12    if $X_1 = g^x$
13       $temp \leftarrow Z^{(t+1)^{-1}}$
14 $K \leftarrow F(k, X, Z)$
15 return $K$
```

Figure 21: $\mathcal{C}_{\mathsf{GAP\text{-}DH}}$ playing the GAP-DH security game, simulating the IND-CPA game for $\mathcal{A}$. F is a PRF and the output of $\mathcal{C}$ is *temp* a group element of $\mathbb{G}$. Publicly known elements, $\mathbb{G}$, g, g_2, p, g^x, g^{xt} and pk. The input X_1 to the DDH oracle is the first part of $X = (X_1, X_2)$ and may vary.

Memory-Tightness. By Lemma 4.3 we get,

$$\textbf{LocalMemory}(\mathcal{B}) \leq \textbf{LocalMemory}(\mathcal{A}) + \textbf{LocalMemory}(\mathsf{G}_2).$$

To bound **LocalMemory**(G_2) we first start counting all needed registers. Firstly we need enough memory to store the bit in line 01, one register to store the PRF key k in line 02, five registers to store the public and secret key from line 03, five more registers for lines 04 through 10, counting two register for K_b, line 11 is included in $\mathcal{A}$'s local memory.

On top of that we also count four extra register for answering Hash queries in line 28, that totals in fifteen registers and two bits. In addition to these registers we must have enough memory to run $\textsc{Gen}_{\text{HPS}}$, (see Figure 18), totaling at $\mathbf{LocalMemory}(\mathsf{Gen}) + 5\lambda$ this includes the registers needed to store the key pair (pk, sk). The oracle Gen is the group generating algorithm which outputs the group $\mathbb{G}$, the generator g and the group order p. Summarizing we get,

$$\mathbf{LocalMemory}(\mathsf{G}_2) \leq \mathbf{LocalMemory}(\mathsf{Gen}) + 15\lambda + 1,$$

implying,

$$\mathbf{LocalMemory}(\mathcal{B}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(\mathsf{Gen}) + 15\lambda + 1.$$

Finally we bound $\mathbf{LocalMemory}(C)$ playing the GAP-DH security game. The memory needed for the reduction from the IND-CPA security game to the GAP-DH game is counted in Figure 21. We count eight registers through lines 01 to 07 and one bit for line 08. To answer Hash queries four registers are needed, but five registers are included in the memory term of the PRF F. Additionally four registers plus one bit is counted in $\mathbf{LocalMemory}(A)$. This totals then at,

$$\mathbf{LocalMemory}(\mathcal{C}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(F) + 3\lambda.$$

□

5.2 CCA proof of HPS Hashed ElGamal

In [CS98] the presented simplified scheme as a PKE is only CPA secure. This largely stems for the fact that if an adversary obtains $(pk, Z = \mathsf{Priv}(sk, X \notin \mathcal{L}))$ they can determine the parts of pk, therefore $Z' = \mathsf{Priv}(sk, X' \notin \mathcal{L})$ can no longer be universal if $\mathcal{A}$ knows of a pair as described above. It then is a natural first step before we try and prove IND-CCA security of HPS-HEG to see if the scheme suffers from the same weaknesses. In the HPS-HEG as in the Cramer-Shoup version of the regular HEG, we obtain the key K by using a cryptographic Hash function. This function takes as input the encapsulations X which are elements in the HPS's language, and the output of the evaluation functions Z. As we have shown in Lemma 5.1 we have correctness, ensuring that if $X \in \mathcal{L}$ we also obtain $Z = \mathsf{Priv}(sk, X)$ and thus $K = \mathsf{H}(X, Z)$. An important difference in the HPS-HEG scheme than that of the one presented in [CS98] is that during the encapsulation and decapsulation algorithms Z is never given to the adversary. To specify $\mathcal{A}$ can obtain any Z they like, but the fact whether any of those particular Z's are equal to $\mathsf{Priv}(pk, X \notin \mathcal{L})$ is never disclosed to $\mathcal{A}$. We present three scenarios.

- **$\mathcal{A}$ obtains K from the corresponding encapsulation $X \in \mathcal{L}$.** If $\mathcal{A}$ has a key K, for which they know the encapsulation X, they still won't be able to obtain the corresponding Z unless they constructed X themselves at which point they have the witness w and can compute $Z = pk^w$, or break the cryptographic Hash security assumptions. If they did not construct X the difficulty of obtaining the correct Z is equivalent to solving the CDH problem:

$$X = (g_1^w, g_2^w = g_1^{tw}),\ Z = pk^w = g_1^{(a_1+ta_2)w}.$$

  Neither will help them obtain a pair $(X, Z = \mathsf{Priv}(sk, X))$ for $X \notin \mathcal{L}$.

- **$\mathcal{A}$ obtains K from the corresponding encapsulation $X \notin \mathcal{L}$.** If they obtain a key K from the corresponding encapsulation $X \notin \mathcal{L}$ they still have to break the security of the one-way Hash function in order to obtain Z, assuming they do not know the secret key sk. Another way they could obtain Z is in the security game of $\mathsf{IND\text{-}CPA}$, $\mathcal{A}$ can query for encapsulations $K_i \leftarrow \mathsf{H}(X, Z_i)$, if any of the $K_i = K$ then $\mathcal{A}$ knows that $Z_i = \mathsf{Priv}(sk, X)$. The chance of this happening is somewhat dependant on how we model the Hash and replacement PRF, but for the security game mentioned universality gives us the probability i/p.

- **$\mathcal{A}$ obtains $Z = \mathsf{Priv}(sk, X \notin \mathcal{L})$ by constructing a pair (X, Z).** $\mathcal{A}$ can construct as many pairs of (X, Z) as they like, if they construct $X \in \mathcal{L}$ they can easily obtain Z as mentioned. If we assume $\mathcal{A}$ does not know $sk = (a_1, a_2)$ but has managed to get their hands on a pair $(X, Z = \mathsf{Priv}(sk, X))$ for X outside the language they can quite simply calculate $g_1^{a_1}$ and $g_2^{a_2}$ if we presume they know w and w' for $Z = g_1^{a_1 w} \cdot g_2^{a_2 w'}$. It is clear that if $\mathcal{A}$ obtains this information we no longer have universality, for they now know all combinations of (X, Z) pairs for $X \notin \mathcal{L}$ by easily computing $Z = g_1^{a_1 w} \cdot g_2^{a_2 w'}$. The bright side is that in order for $\mathcal{A}$ to obtain such a pair in the first place they need to break the first instance of universality outright, which is done with probability $1/p$ for any single pair.

5.2.1 Constructing a PRF

Before formally proving the $\mathsf{IND\text{-}CCA}$ of $\mathsf{HPS\text{-}HEG}$ we have to decide how to model the Hash replacement, the PRF F. Starting with the familiar construction of Construction 5.4 we show that this is adequate in the $\mathsf{IND\text{-}CCA}$ proof. As was discussed in Section 3 one of the problems for a reduction playing the $\mathsf{GAP\text{-}DH}$ security game was answering decapsulation oracles consistently when only $Y = g^y$ was given. The split PRF $\tilde{F}$ was constructed so that the reduction did not need any corresponding input, and because of the nature of the DH-tuples there was no chance of inconsistencies. A reduction against the SM problem can always generate a key pair $(sk = (a_1, a_2), pk = (g_1^{a_1} \cdot g_2^{a_2}))$ such that when an adversary $\mathcal{A}$ playing the $\mathsf{IND\text{-}CCA}$ security game queries the decapsulation oracle on X, the reduction can always compute a corresponding $Z = X_1^{a_1} \cdot X_2^{a_2}$ for consistency. This Z is never revealed to $\mathcal{A}$ since it will only be used as input for the Hash, for encapsulation queries $\mathcal{A}$ must supply their own Z. A possible construction could be to remove the DDH oracle altogether and replace it with the private evaluation algorithm Priv, and use this to present a tight adversary against the SM problem of the HPS and bound the $\mathsf{HPS\text{-}HEG}$ by the DDH assumption. This would be favourable in trying to construct a generalization of Bhattacharyya's approach using general HPS as reductions need only simulate the Priv evaluation function as discussed. However none of the possible reductions we constructed were able to extract from $\mathcal{A}$, playing the $\mathsf{IND\text{-}CCA}$ security game, the fact that they could tell the difference of elements in $\mathcal{L}$ and $\mathcal{X}/\mathcal{L}$. We specify in Remark 5.3. Furthermore in the regular CDH security game a reduction still needs a DDH oracle to even check if $Z = \mathsf{Priv}(sk, X)$ since the actual secret key would be $sk = (y, y)$, which of course is unknown.

Remark 5.3 Lets assume $\mathcal{A}$ is looking for a pair $(X, Z = \mathsf{Priv}(sk, X \notin \mathcal{L}))$. It has access to a decapsulation and encapsulation oracle as shown in Figure 22. $\mathcal{A}$ first queries the

Decap oracle on an $X \notin \mathcal{L}$ of their choosing. They then proceed to ask H for keys related to the encapsulation X, choosing each Z freely. Notice that if $\textsc{Flag} = 1$ then $\mathcal{A}$ obtains the key $K = F(k, X, Z)$. By universality this happens with probability,

$$\Pr[\textsc{Flag} \to 1] \leq \frac{q_{\mathsf{H}}}{p}.$$

We now disregard any assumptions about $\mathcal{A}$ and inspect $\mathcal{C}$'s chance of solving the SM problem as shown. If $\textsc{Flag} = 1$ then that must mean $X^* \in \mathcal{L}$ with probability $1 - q_{\mathsf{H}}/p$. In this case $\mathcal{C}$ wins its game with negligible probability of error. The problem arises when $\textsc{Flag} = 1$ never is triggered. Although $\textsc{Flag} = 1$ implies $X^* \in \mathcal{L}$ with negligible probability of error, the opposite is not necessarily true. If this were the case then we would have that $\mathcal{A}$ generates $Z = pk^{w^*}$ without knowing w^*, t, or the secret key sk. This is exactly the probability of breaking the CDH assumption, and there is no proof showing that breaking $\mathsf{DDH} \implies$ breaking CDH, there are in fact groups for which this has been dis-proven. We assume that $\mathcal{A}$ does not instantly obtain w^* by recognizing that $X \in \mathcal{L}$.

```
Reduction $\mathcal{C}(X^*)$
$k \xleftarrow{\$} \{0,1\}^\lambda$
$a_1 \xleftarrow{\$} \mathbb{Z}_p^*$
$a_2 \xleftarrow{\$} \mathbb{Z}_p^* - \{a_1\}$
$sk \leftarrow (a_1, a_2)$
$pk \leftarrow g_1^{a_1} \cdot g_2^{a_2}$
$K \xleftarrow{\$} \mathcal{K}$
$\hat{b} \leftarrow \mathcal{A}^{\mathsf{H},\mathsf{Decap}}(pk, X^*, K)$
if $\textsc{Flag} = 1$
    return $0$
else
    return $1$

Oracle $\mathsf{H}(X, Z)$
if $Z = \mathsf{Priv}(sk, X)$
    if $X = X^*$
        $\textsc{Flag} = 1$
$K \leftarrow F(k, X, Z)$
return $K$

Oracle $\mathsf{Decap}(X)$
if $X = X^*$
    return $\perp$
$Z \leftarrow \mathsf{Priv}(sk, X)$
$K \leftarrow F(k, X, Z)$
return $K$
```

Figure 22: The reduction $\mathcal{C}$ playing the SM security game running $\mathcal{A}$, an adversary playing the $\mathsf{IND\text{-}CCA}$ game against $\mathsf{HPS\text{-}HEG}$. By adding an extra step in the security game of Theorem 5.2 right after switching to the PRF in G_2; we choose X^* from outside the language and if we could show that $\mathcal{C}$ is a tight reduction, we could conclude that security of $\mathsf{HPS\text{-}HEG}$ is bounded by the DDH assumption instead of the $\mathsf{GAP\text{-}DH}$ assumption by Lemma 5.1. The probability that G_4 aborts in the modified version of Figure 20 would simply be q_{H}/p in this case. This proof of security does not seem possible by Remark 5.3.

Construction 5.4 *Let* $(\textsc{Gen}_{\textsc{hps}}, \mathsf{Pub}, \mathsf{Priv})$ *be a* HPS *defined as in Figure 18 with public key pk and language* $\mathcal{L}$ *over a cyclic group* $\mathbb{G}$ *with prime order p and generator g. Let* DDH *be an oracle answering queries on tuples replying whether they are DH-tuples or not. Define the function* $\tilde{F} : \{0,1\}^\lambda \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ *as,*

$$\tilde{F}(k, X, Z) = \begin{cases} F(k, 0, X, Z) & \textbf{\textit{for}} \;\; \mathsf{DDH}(X_1, pk, Z) = 0 \\ F(k, 1, X, g) & \textbf{\textit{for}} \;\; \mathsf{DDH}(X_1, pk, Z) = 1 \end{cases}$$

Define $F : \{0,1\}^\lambda \times \{0,1\} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathcal{K}$ *as a PRF.*

Lemma 5.5 *If F is a PRF, then $\tilde{F}$ is also a PRF. Furthermore for every adversary $\mathcal{A}$ against $\tilde{F}$ there exists an adversary $\mathcal{B}$ against F such that the following hold:*

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F} = \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{A},\tilde{F}} + \frac{q}{p}$$

$$\mathbf{LocalMemory}(\mathcal{B}) \leq \mathbf{LocalMemory}(\mathcal{A}) + 5\lambda.$$

q is the number of queries made by $\mathcal{A}$.

Proof. Follows by Lemma 4.5 and the similar proof in [Bha20]. The sumand q/p accounts for the error when $Z = \mathsf{Priv}(sk, X \notin \mathcal{L})$. For memory-tightness we count: additionally to needing $\mathbf{LocalMemory}(\mathcal{A})$ the reduction needs to store g_2, t, g, $sk = (a_1, a_2)$, $pk = g_1^{a_1} \cdot g_2^{a_2}$ and p. The bit output by $\mathcal{B}$ including one register for g_2 and one register for pk is included in $\mathbf{LocalMemory}(\mathcal{A})$. All registers needed to answer $\tilde{F}$ queries are counted in the PRF F not run by $\mathcal{B}$. $\square$

5.2.2 Proof of CCA security

Theorem 5.2 along with the discussion above gives a good starting point for proving memory-tightness in the IND-CCA security game. Even though we unfortunately could not generalize the technique of Bhattacharyya in this instance we still state and prove memory-tightness of the HPS-HEG in our third and final main Theorem.

Theorem 5.6 *Let g be a generator of a cyclic group $\mathbb{G}$ of prime order p. Let $F : \{0,1\}^\lambda \times \{0,1\} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \to \mathcal{K}$ be a* PRF. *Let $\mathcal{A}$ be an adversary against the* HPS-HEG *in the* IND-CCA *game with security parameter λ. If $\mathcal{A}$ makes q_{H} Hash queries and q_{D} decapsulation queries in the random oracle model, there then exists an adversary $\mathcal{B}$ against the* PRF *F and an adversary $\mathcal{C}$ against the* GAP-DH *assumption such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A}} \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F} + \mathsf{Adv}^{\mathsf{GAP\text{-}DH}}_{\mathcal{C}} + 2 \cdot \frac{q_{\mathsf{H}}}{p}.$$

Additionally we get the following memory bounds on the reductions,

$$\mathbf{LocalMemory}(\mathcal{B}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(\mathsf{Gen}) + 20\lambda + 1,$$

$$\mathbf{LocalMemory}(\mathcal{C}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(F) + 6\lambda + 1.$$

Proof. The sequence of games is almost identical to the one presented Figure 20 with the addition of a decapsulation oracle as presented in Figure 23, and an extra game step switching from $\tilde{F}$ to F as in Figure 16 of Section 4.
In G_2 we have

$$\left|\Pr[\mathrm{G}_1^{\mathcal{A}} \to 1] - \Pr[\mathrm{G}_2^{\mathcal{A}} \to 1]\right| \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{B},\tilde{F}}, \tag{1}$$

and thus by Lemma 5.5 the left hand side of (1) $\leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{B},F} + q_{\mathsf{H}}/p$.

In G_3 we use the PRF F directly, adding similar lines as 31, 32, 38 in Figure 16 to H only now checking with the private evaluation function as in Figure 20. As the only way for $\mathcal{A}$ to

```
Oracle Decap($X$)
01 if $X = X^*$
02   return $\perp$
03 $Z \leftarrow \mathsf{Priv}(sk, X)$  // $\mathrm{G}_0$-$\mathrm{G}_2$
04 $K \leftarrow \mathsf{H}(X, Z)$      // $\mathrm{G}_0$-$\mathrm{G}_1$
05 $K \leftarrow \tilde{F}(k, X, Z)$      // $\mathrm{G}_2$
06 $K \leftarrow F(k, 1, X, g)$     // $\mathrm{G}_3$
07 return $K$
```

Figure 23: The $\mathsf{IND\text{-}CCA}$ security game decapsulation oracle, Decap. The oracle rejects any attempts to decapsulate the challenge ciphertext X^*, and always outputs consistently in accord with H since the input (X, Z) to the key derivation functions in line 04 and 05, is unique up to each X by the secret key sk. In G_3 the adversary only obtains the decapsulated key K from H queries if they present a pair $(X, Z = \mathsf{Priv}(sk, X))$, same as before, thus the difference from G_2 to G_3 is unnoticeable to $\mathcal{A}$.

obtain the decapsulated key K from queries to H is to present a pair $(X, Z = \mathsf{Priv}(sk, X))$ we see this is exactly as in G_2. Thus the games are identical,

$$\Pr[\mathrm{G}_2^{\mathcal{A}} \to 1] = \Pr[\mathrm{G}_3 \to 1]. \tag{2}$$

In the final step of G_4 we get as before,

$$\Pr[\mathrm{G}_4^{\mathcal{A}} \to 1] = \frac{1}{2}, \tag{3}$$

and

$$\left|\Pr[\mathrm{G}_3^{\mathcal{A}} \to 1] - \Pr[\mathrm{G}_4^{\mathcal{A}} \to 1]\right| \leq \Pr[\textsc{Flag} \to 1]. \tag{4}$$

We construct a reduction against the $\mathsf{GAP\text{-}DH}$ assumption using Construction 5.4. Notice that multiple queries to the decapsulation oracle does not help $\mathcal{A}$ in breaking universality as for all $X \notin \mathcal{L}$ we have $\Pr[Z = \mathsf{Priv}(sk, X)] = 1/p$ so having multiple *key-non-letter-pairs* (K, X) and trying to find a Z for any of these, still amounts to be as hard as just trying to find one Z for one pair (K, X) because $K_i \neq \mathsf{H}(X_i, Z_j)$ does not mean $K_i \neq \mathsf{H}(X_l, Z_j))$ for $i \neq l$. By (1), (2), (3), (4), Figure 24 and Theorem 5.2 we conclude that,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}} \leq \mathsf{Adv}_{\mathcal{B},F}^{\mathsf{PRF}} + \mathsf{Adv}_{\mathcal{C}}^{\mathsf{GAP\text{-}DH}} + 2 \cdot \frac{q_{\mathsf{H}}}{p}.$$

Memory-Tightness. From Lemma 4.3 and Lemma 5.5 we get the following,

$$\mathbf{LocalMemory}(\mathcal{B}_F) \leq \mathbf{LocalMemory}(\mathcal{B}_{\tilde{F}}) + 5\lambda,$$

$$\mathbf{LocalMemory}(\mathcal{B}_{\tilde{F}}) \leq \mathbf{LocalMemory}(\mathcal{A}) + \mathbf{LocalMemory}(\mathrm{G}_2).$$

By Theorem 5.2 and the fact that the registers covering Hash queries in G_2 are adequate to cover any decapsulation queries to Decap, we arrive at the same memory bound for $\mathbf{LocalMemory}(\mathrm{G}_2)$,

$$\mathbf{LocalMemory}(\mathrm{G}_2) \leq \mathbf{LocalMemory}(\mathsf{Gen}) + 15\lambda + 1.$$

```
**Reduction** $\mathcal{C}^{\mathsf{DDH}}(g, g^x, g^y)$
08 $k \xleftarrow{\$} \{0,1\}^\lambda$
09 $t \xleftarrow{\$} \mathbb{Z}_p^*$
10 $(t+1)^{-1} \leftarrow \mathbb{Z}_p^*$
11 $X^* \leftarrow (g^x, g^{xt})$
12 $pk \leftarrow (g^y \cdot g^{yt})$
13 $a_1 \xleftarrow{\$} \mathbb{Z}_p^*$
14 $a_2 \xleftarrow{\$} \mathbb{Z}_p^* - \{a_1\}$
15 $sk \leftarrow (a_1, a_2)$
16 $K \xleftarrow{\$} \mathcal{K}$
17 $temp \xleftarrow{\$} \mathbb{G}$
18 $\hat{b} \leftarrow \mathcal{A}^{\mathsf{H},\mathsf{Decap}}(pk, X^*, K)$
19 **return** $temp$
```

```
**Oracle** $\mathsf{H}(X, Z)$
20 $(X_1, X_2) \leftarrow X$
21 **if** $\mathsf{DDH}(X_1, pk, Z) = 1$
22    **if** $X_1 = g^x$
23       $temp \leftarrow Z^{(t+1)^{-1}}$
24    **if** $\mathsf{DDH}(X_1, g^t, X_2) = 1$
25       $K \leftarrow F(k, 1, X, g)$
26 **else**
27    $K \leftarrow F(k, 0, X, Z)$
28 **return** $K$
```

```
**Oracle** $\mathsf{Decap}(X)$
29 **if** $X = X^*$
30    **return** $\perp$
31 $(X_1, X_2) \leftarrow X$
32 **if** $\mathsf{DDH}(X_1, g^t, X_2) = 1$
33    $K \leftarrow F(k, 1, X, g)$
34 **else**
35    $Z \leftarrow \mathsf{Priv}(sk, X)$
36    $K \leftarrow F(k, 0, X, Z)$
37 **return** $K$
```

Figure 24: The reduction $\mathcal{C}$ playing the GAP-DH security game running $\mathcal{A}$, an adversary playing the IND-CCA game against HPS-HEG. The reduction is very similar to the one presented in Figure 21 but with an additional Decap oracle, split PRF as shown and a pair (a_1, a_2) not used anywhere but to generate inputs to F when $X \notin \mathcal{L}$. The actual secret key is $sk = (y, y)$ unknown to $\mathcal{C}$. During the call to the DDH oracle in line 31, the reduction checks whether $X \in \mathcal{L}$, for a fixed g_2 this is true if and only if $X \in \mathcal{L}$, again the DDH oracle is not restricted on any input per the GAP-DH assumption and X_1 is not necessarily g^x. The reduction simulates the IND-CCA game perfectly.

The memory bound for **LocalMemory**$(\mathcal{B}_F)$ is therefore,

$$\textbf{LocalMemory}(\mathcal{B}_F) \le \textbf{LocalMemory}(\mathcal{A}) + \textbf{LocalMemory}(\mathsf{Gen}) + 20\lambda + 1.$$

Inspecting Figure 24 we see that the reduction needs an addition of two registers to store the simulated $sk = (a_1, a_2)$ otherwise it is identical to Figure 21, with the registers covering queries to H also covering decapsulation queries,

$$\textbf{LocalMemory}(\mathcal{C}) \le \textbf{LocalMemory}(\mathcal{A}) + \textbf{LocalMemory}(F) + 6\lambda + 1.$$

□

6 Concluding Remarks

As a starting point for our work we presented the two different schemes commonly referred to as the Hashed ElGamal. In detail we discussed how the implementation of the cryptograhic Hash function in the random oracle model opened up for the main technique of Bhattacharyya. Knowing the details it was then a simple matter to understand why the work of Ghoshal and Tessaro did not contradict this technique. We proceeded and used this for proving memory-tightness of the schemes following Section 3, the first of which being our main result the Twin Hashed ElGamal in Theorem 4.6. We were able to prove that this scheme eliminates the need for so called GAP groups under the GAP-DH assumption and with a small increase to complexity achieved security under the regular CDH assumption. The small increases to complexity were strictly contained to the Hash function, the PRF following this and the public and secret key sizes. Otherwise the message, ciphertext and key spaces all stayed the same as for the original HEG.

We also presented a modification of the original Hashed ElGamal scheme in Theorem 5.6 implemented with a Hash Proof System hoping that the deterministic evaluation functions of the HPS could be a substitute for the Diffie-Hellman oracle in the construction of the PRF as in Figure 22. Unfortunately we were not able to show that this could be done in the specific instance of the HPS-HEG, but we did end up proving that the reductions were still memory-tight. As for the usefulness of the HPS-HEG we have to conclude that it is less secure and more complex than the regular Cramer-Shoup variant of the HEG and that the proof of memory-tightness is more or less identical to the original proof given by Bhattacharyya. Compared with the TEG, it is certainly not the best of the two substitutes as far as we can tell.

For future projects it might still be interesting to see if Key Encapsulation Mechanisms implemented with other Hash Proof Systems, for example a HPS based on quadratic residuosity [Pai99], can use the technique of Bhattacharyya to achieve memory-tightness. Other interesting projects could be to investigate how to improve, if possible, memory-tight conclusions on primitives built upon for example Learning Parity with Noise.

References

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 143–158, San Francisco, CA, USA, April 8–12, 2001. Springer, Heidelberg, Germany.

[ACFK17] Benedikt Auerbach, David Cash, Manuel Fersch, and Eike Kiltz. Memory-tight reductions. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 101–132, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[Bha20] Rishiraj Bhattacharyya. Memory-tight reductions for practical key encapsulation mechanisms. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 12110 of *Lecture Notes in Computer Science*, pages 249–278, Edinburgh, UK, May 4–7, 2020. Springer, Heidelberg, Germany.

[BKW00] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *CoRR*, cs.LG/0010022, 2000.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[CKS08] David Cash, Eike Kiltz, and Victor Shoup. The twin Diffie-Hellman problem and applications. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 127–145, Istanbul, Turkey, April 13–17, 2008. Springer, Heidelberg, Germany.

[CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[CS01] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack.

Cryptology ePrint Archive, Report 2001/108, 2001. https://ia.cr/2001/108.

[GT21] Ashrujit Ghoshal and Stefano Tessaro. On the memory-tightness of hashed ElGamal. Cryptology ePrint Archive, Report 2021/448, 2021. https://eprint.iacr.org/2021/448.

[HK09] Dennis Hofheinz and Eike Kiltz. The group of signed quadratic residues and applications. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 637–653, Santa Barbara, CA, USA, August 16–20, 2009. Springer, Heidelberg, Germany.

[JL09] Antoine Joux and Stefan Lucks. Improved generic algorithms for 3-collisions. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 347–363, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

[LLMP93] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard. The number field sieve. In Arjen K. Lenstra and Hendrik W. Lenstra, editors, *The development of the number field sieve*, pages 11–42, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany.

[Pol75] J. M. Pollard. A monte carlo method for factorization. *BIT*, 15(3):331–334, sep 1975.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.

[WMHT18] Yuyu Wang, Takahiro Matsuda, Goichiro Hanaoka, and Keisuke Tanaka. Memory lower bounds of reductions revisited. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 61–90, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

NTNU

Kunnskap for en bedre verden