Converting Cryptographic Schemes from Symmetric to Asymmetric Bilinear Groups

Masayuki Abe, Jens Groth*, Miyako Ohkubo, and Takuya Tango

NTT Secure Platform Laboratories, Japan
abe.masayuki@lab.ntt.co.jp
University College London, UK
j.groth@ucl.ac.uk
Security Fundamentals Lab, NSRI, NICT, Japan
m.ohkubo@nict.go.jp
Kyoto University
tkytango@ai.soc.i.kyoto-u.ac.jp

Abstract. We propose a method to convert schemes designed over symmetric bilinear groups into schemes over asymmetric bilinear groups. The conversion assigns variables to one or both of the two source groups in asymmetric bilinear groups so that all original computations in the symmetric bilinear groups go through over asymmetric groups without having to compute isomorphisms between the source groups. Our approach is to represent dependencies among variables using a directed graph, and split it into two graphs so that variables associated to the nodes in each graph are assigned to one of the source groups. Though searching for the best split is cumbersome by hand, our graph-based approach allows us to automate the task with a simple program. With the help of the automated search, our conversion method is applied to several existing schemes including one that has been considered hard to convert.

Keywords. Conversion, Symmetric Bilinear Groups, Asymmetric Bilinear Groups

1 Introduction

It is often believed that once a scheme for a purpose is shown feasible over symmetric bilinear groupsa scheme for the same purpose should be constructable over asymmetric bilinear groups. One approach is to use different mechanisms and stronger assumptions available only in the asymmetric setting. The other, which we study in this paper, is to convert a scheme in the symmetric setting into one in the asymmetric setting keeping the original design intact.

We will given a bilinear group setting with a pairing $e : \mathbb{G} \times \tilde{\mathbb{G}} \rightarrow \mathbb{G}_T$ follow [13] in calling the symmetric setting where $\mathbb{G} = \tilde{\mathbb{G}}$ for Type-I and in the

* The research leading to these results has received funding from the the Engineering and Physical Sciences Research Council grant EP/J009520/1 and the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 307937.

asymmetric setting distinguish between Type-II, where there is an efficiently computable isomorphism $\psi : \tilde{\mathbb{G}} \to \mathbb{G}$, and Type-III where there are no efficiently computable isomorphisms between the source groups. In this paper, we focus on converting schemes over Type-I groups to Type-III groups, i.e., converting to the fully asymmetric setting, so we will in general be referring to the Type-III setting when speaking of asymmetric bilinear groups.

We argue that the benefit of conversion is threefold. First, it allows designers to focus on implementing their ideas using a simpler description in the symmetric setting without being encumbered by the deployment of group elements over two source groups. Second, it is an effective way to save schemes in the symmetric setting from cryptanalytic advances. Recent progress in solving the discrete logarithm problems over groups with small characteristics [15, 16, 14, 3] has necessitated the use of larger parameters for a major class of symmetric bilinear groups. A conversion method allows one to port these schemes to an asymmetric setting. Finally, it yields potentially more secure schemes than those dedicated to asymmetric groups because preserving the symmetric group assumptions they may remain secure even if cryptanalysis were to discover techniques to efficiently compute isomorphisms between the source groups.

There are two issues that makes conversion a non-trivial task. The first is the potential presence of symmetric pairings $e(X, X)$. A symmetric pairing can occur indirectly like $e(X, Z)$ for $Z = X \cdot Y$ and it is not necessarily easy to see if there are indirect symmetric pairings in intricate algorithms. It is in particular problematic when a function that maps a string into a source group element is used in the original scheme since it is only possible to either of the source groups at a time. The second is the security proof and the underlying assumptions. Not only should the scheme be executable, but the reduction algorithms used in the security proof must also be executable in the asymmetric setting. Furthermore, the assumptions need to be cast in the asymmetric setting as well. An instructive example is given in [18] that demonstrates how conversion without a security guarantee can yield a scheme that seems to work but is insecure.

Our Result. We propose a conversion method that turns schemes designed over symmetric bilinear groups into schemes over asymmetric bilinear groups. As our method converts not only the algorithms in a scheme but also the security proofs by black-box reduction and the underpinning assumptions, the security is preserved based on the converted assumptions. We then argue that, if the original assumptions are justified in the generic Type-I group model, the converted counterparts are justified in the generic Type-III group model. Our conversion includes schemes in the random oracle model that hash a string onto a random group element and hash group elements to a random string. We present a formal model for the class of schemes our conversion method can handle.

Our conversion method takes as input a *dependency graph* that represents computation among source group elements in the scheme to convert. This is a directed graph whose nodes correspond to group elements in the scheme. Directed edges in the graph represent dependency in such a way that the destination node is computed from the source nodes through the group operation. By splitting the dependency graph into two graphs in such a way that no dependency is lost and two nodes that represent group elements input to a pairing appear in different graphs, we obtain two dependency graphs that represents computation in the source groups of asymmetric bilinear groups. There may be nodes that appear in both graphs. Their presence is necessary for consistent computation in each source group. These nodes correspond to the symmetric group elements that need to be duplicated in both source groups when converting to an asymmetric bilinear group. The most cumbersome part of our conversion procedure is to find a splitting of the dependency graph in a way that conforms to given constraints and efficiency measures.

We present an algorithm to search for the best valid split. It is implemented with Java and applied to dependency graphs for several known cryptographic schemes originally built over symmetric bilinear groups:

- Waters' Identity-based Encryption scheme [24]. We have chosen this scheme since it has a small number of parameters so that one can manually verify the result. An interesting observation is that conversion is not possible without duplicating some group elements in the assumption.
- Boneh and Shacham's verifier-local revocation group signature scheme [7]. This scheme involves hash-onto-point functions. In [23], Smart and Vercauteren observed that converting to Type-III is not possible, and [10] introduced the scheme as an typical example that cannot be converted due to their use of hash-to-point functions and homomorphisms. We present a conversion based on assumptions that duplicates elements in the original assumptions. It does not contradict [23] as they limit themselves to the case where no element duplicates in the assumption.
- Waters' dual-system encryption scheme [25]. The purpose of converting this scheme is to compare the converted result using our conversion technique with existing manually built schemes[11, 20]. Our conversion yields a slightly less efficient scheme. An assumption that fully duplicates elements of the decision linear assumption is inevitable for this conversion.
- Tagged one-time signature scheme from [1]. By converting the scheme, we obtain the first tagged one-time signature scheme over asymmetric bilinear groups with minimal tag size. It answers the open question positively in [1].

The search algorithm runs in exponential time in the number of pairings and the number of bottom nodes that do not have outgoing edges as we will explain later. It takes a standard PC (Windows 7, Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz, 8.0GB RAM) 9 msec to convert the easiest case, Waters' IBE, and about 80 min to convert the most intricate case, Waters' Dual Encryption scheme. We summarize the result of our experimental conversions in Table 1.

It should be noted that our conversion is not tight, i.e., when our conversion algorithm fails to find a scheme in Type-III, it does not mean anything more than the fact that our conversion algorithm does not find a conversion for the scheme. In particular, it does not imply general impossibility. It is an open problem to show impossibility of conversion.

Due to space limitation, most of proofs for theorems and lemmas and all details of experiments are dropped from this version of the paper.

Scheme	Construction	Size and Delta		Assumptions
		$pp + pk$	ct or σ	
Waters' IBE	original [24]	$4 + \lambda$	2	DBDH
	conv'd(opt=pk)	$+2$	$+0$	coDBDH^{+3}
	conv'd(opt=$assm$)	$+3 + \lambda$	$+0$	coDBDH^{+2}
VLR Group Sig	original [7]	2	2	DLIN, q-SDH
	conv'd	$+1$	$+0$	coDLIN^{+5}, q-coSDH^{+q}
Dual System Enc	original [25]	12	9	DLIN, DBDH
	conv'd(opt=pk)	$+0$	$+0$	coDLIN$^{+5,+4}$, coDBDH^{+2}
	conv'd(opt=$assm$)	$+4$	$+0$	coDLIN$^{+3,+3}$, coDBDH^{+2}
	manually conv'd [19]	-3	-2	DDH1, DLIN, DBDH
		-6	-5	DDH1, DDH2v, DBDH
	new design [11]	-2	-5	SXDH
TOS	original [1]	$2k + 6$	4	SDP
	conv'd	$+5$	$+0$	coSDP^{+5}

Table 1. Size of public parameters ($pp + pk$) and ciphertext (ct) or signature (σ) in the number of source group elements including a default generator. "conv'd(opt=xxx)" denotes that the scheme is obtained by our conversion in terms of minimizing the size of assumption or public-keys. Assumption coXXX$^{+\alpha}$ denotes co-XXX assumption(s) that involve α duplicated elements. For TOS, σ includes a tag.

Related Works. Chatterjee and Menezes [9, 10] considered conversion from schemes over Type-II groups to Type-III groups and discuss the role of the isomorphism in Type-II groups. Their conversion shares the basic idea with ours – represent a group element by a pair of source group elements and drop one of them if unnecessary. They proposed a heuristic guideline for when a scheme allows or resists conversion. Chatterjee et al. [8] discussed relations among assumptions over Type-II and Type-III groups that include ones with or without duplicated elements in a problem instance. Smart and Vercauteren [23] explored variations of Boneh and Franklin's identity-based encryption scheme [5] and Boneh, Lynn and Shacham's signature scheme [6] based on a family of BDH assumptions over Type-II groups. They investigated which variations suffice for the security proofs and how efficient the corresponding schemes are. Chen et al. [11] presented modifications of Waters' dual-system encryption scheme over Type-III groups. They obtained a more efficient scheme than the original by a careful manual conversion.

A more general work by Akinyele, Green and Hohenberger [2] introduced a powerful software system called AutoGroup whose purpose is the same as ours. It takes a specification of the target scheme written in a scheme description language and uses a satisfiability modulo theory solver [12] to find a valid deployment of elements over two source groups. AutoGroup is a powerful tool to find optimized computation in the resulting scheme conforming to one's design demands. On the other hand, and contrary to our tool, it does not say anything about the security of the resulting scheme; it is left as a subsequent task to check whether the resulting scheme is secure, to identify sufficient assumptions, and to provide convincing arguments that the assumptions are plausible. Our procedure requires manual work as well but only to examine if the original proof works over generic symmetric bilinear groups and to verify that the assumptions are plausible in the generic Type-I group model. Once this has been confirmed, the converted scheme retains its security under converted assumptions that also retain plausibility arguments in the generic group model.

2 Preliminaries

We follow standard definition and notations of symmetric and asymmetric bilinear groups. Let $\mathcal{G}$ be a group generator that takes security parameter 1^λ and outputs $(q, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_T, e, G, \tilde{G})$ where $\mathbb{G}$, $\tilde{\mathbb{G}}$, and $\mathbb{G}_T$ are groups of prime order q, $e : \mathbb{G} \times \tilde{\mathbb{G}} \to \mathbb{G}_T$ is an efficiently computable non-degenerate bilinear map, and $G, \tilde{G}$ and $G_t = e(G, \tilde{G})$ are generators of $\mathbb{G}, \tilde{\mathbb{G}}$ and $\mathbb{G}_T$ respectively. $\mathbb{G}$ and $\tilde{\mathbb{G}}$ are called source groups and $\mathbb{G}_T$ is called the target group. When $\mathbb{G} \neq \tilde{\mathbb{G}}$ and there are no efficiently computable isomorphisms between $\mathbb{G}$ and $\tilde{\mathbb{G}}$ we call it the Type-III bilinear group setting. When $\mathbb{G} = \tilde{\mathbb{G}}$ we let $G = \tilde{G}$ and call it a symmetric bilinear group or Type-I group. For simplicity, we will often use the shortened notation $\mathcal{G}_{\mathsf{sym}}$ and $(q, \mathbb{G}, \mathbb{G}_T, e, G)$ for the symmetric case, and on the other hand write $\mathcal{G}_{\mathsf{asym}}$ when emphasizing a group generator outputs a Type-III bilinear group. Hashing to $\mathbb{G}$ is doable in Type-I groups over supersingular elliptic curves. In Type-III groups over elliptic curves, it is possible to hash to $\mathbb{G}$ and $\tilde{\mathbb{G}}$ independently with different costs. We focus on source group elements in the paper and other data are mostly ignored as they are handled equally in Type-I and III settings. Notes will be given otherwise.

Throughout the paper we will work with directed graphs using the notation (X, Y) for an edge from node X to Y. For two directed graphs $\mathit{\Gamma} = (V, E)$ and $\mathit{\Gamma}' = (V', E')$ the merger operation $\mathit{\Gamma} \oplus \mathit{\Gamma}'$ is defined by a graph $(V \cup V', E \cup E')$. For a graph $\mathit{\Gamma}$ and a subgraph $\mathit{\Gamma}'$, we define $\mathit{\Gamma} \ominus \mathit{\Gamma}'$ as a graph obtained by removing nodes in $\mathit{\Gamma}'$ and edges that involves nodes in $\mathit{\Gamma}'$ from $\mathit{\Gamma}$. If a graph $\mathit{\Gamma}'$ is a subgraph of $\mathit{\Gamma}$, we write $\mathit{\Gamma}' \subseteq \mathit{\Gamma}$. For a node X in $\mathit{\Gamma}$, we define $\mathsf{Anc}(\mathit{\Gamma}, X)$ by the subgraph of $\mathit{\Gamma}$ that consists of paths reaching X. Nodes in $\mathsf{Anc}(\mathit{\Gamma}, X)$ are called ancestors of X, where we use the convention that X is an ancestor to itself.

By $(x, y) \leftarrow (A(a), B(b))$, we denote a process where two interactive algorithms A and B take inputs a and b and output x and y, respectively, as a result of their interaction.

3 Overview with Example

This section illustrates our conversion procedure for Waters' IBE [24]. We mostly use the original notation and description although the reader may want to refer to the original paper for details.

The procedure starts by looking at the original description of Water's IBE scheme over symmetric bilinear groups. It builds a *dependency graph*, that describes how each group element appearing in the scheme depends on other group elements. Later the dependency graph is used to decide which group element is computed in which source group of the asymmetric bilinear groups. The type assignment yields an instantiation of Waters IBE scheme over asymmetric bilinear groups. As we convert the reduction algorithm in the security proof and the underlying assumptions as well, the underlying security argument also translates to the asymmetric setting.

Step 1. Building a dependency graph for each algorithm. Waters' IBE scheme consists of four algorithms: Setup, Key Generation, Encryption and Decryption. The security proof consists of a reduction algorithm that uses a purported adversary in a black-box manner to break an instance of the decisional bilinear Diffie-Hellman (DBDH) problem. The first step is to build a dependency graph for each algorithm as illustrated in Fig. 2.

- The setup algorithm takes a security parameter, and outputs default generator g and random source group elements g_1, g_2, u', and $u_1, \ldots, u_n$ where $g_1 = g^{\alpha}$ for a random $\alpha \in \mathbb{Z}_q$. The dependency graph includes nodes $\mathsf{g}, \mathsf{g1}, \mathsf{g2}, \mathsf{u}'$ that correspond to g, g_1, g_2, and u', respectively. Elements $u_1, \ldots, u_n$ are represented by a single node ui in the graph. We assume that the random source group elements are generated from the default generator by using group operations. Thus the dependency graph has edges from g to every other node. The algorithm also computes a master secret key $msk = {g_2}^{\alpha}$. We thus add a node labeled by msk and have an edge from $\mathsf{g2}$ to msk. This results in graph (1) in Fig. 2.
- The key generation algorithm takes the master secret key and the public key, and computes decryption key (d_1, d_2) for an identity $v \in \{0, 1\}^n$. Let $\mathcal{V}$ be the set of indices for which the bit-string v is set to 1.

$$d_1 := msk \cdot (u' \prod_{i \in \mathcal{V}} u_i)^r \quad \text{and} \quad d_2 := g^r. \tag{1}$$

A corresponding dependency graph thus involves nodes $\mathsf{d1}$, msk, u', ui, $\mathsf{d2}$, g, and edges from $\mathsf{msk}, \mathsf{u}', \mathsf{ui}$ to $\mathsf{d1}$, and g to $\mathsf{d2}$ as illustrated in graph (2) in Fig. 2. Note that g_1 is in the public key given to the algorithm but not involved in computation. Such unused elements can be ignored and do not appear in the graphs.

- The encryption algorithm involves both group operations and pairings. It takes the public key and message M from the target group, and computes a ciphertext (C_1, C_2, C_3) as follows:
$$C_1 := e(g_1, g_2)^t \cdot M, \quad C_2 := g^t \quad \text{and} \quad C_3 := u' \prod_{i \in \mathcal{V}} u_i^t. \tag{2}$$
The pairing operation $e(g_1, g_2)$ is represented in the graph by connecting nodes g1, g2 to a pair of special nodes called *pairing nodes*, whose label looks like p1[0] and p1[1]. The trunk name p1 is unique throughout the system. A paring node indicates that its parent node corresponds to an input to the pairing operation identified by the name. C_2 and C_3 are source group elements computed from g and u', u_i, respectively, and represented in the graph in (3) in Fig. 2 accordingly. Since M and C_1 are in the target group, no corresponding nodes are included in the graph.
- The decryption algorithm computes
$$M := C_1 \; e(d_2, C_3) \; e(d_1, C_2)^{-1}. \tag{3}$$
The pairing $e(d_2, C_3)$ yields nodes d2, C3 connected to pairing nodes p2[0] and p2[1], respectively. Similarly, the pairing $e(d_1, C_2)$ yields nodes d1, C2 connected to p3[0] and p3[1], respectively. The resulting graph is (4) in Fig. 2.
- Next we consider the instance generator of DBDH that generates default generator g and random group elements A, B, and C. (Target group element Z is irrelevant and ignored here.) The graph contains nodes g, A, B, and C, and edges from g to every other node as illustrated in (5) in Fig. 2. Graphs for associated verification and random guessing algorithms are empty as they do not involve any group operations.
- Finally we consider a graph for the reduction algorithm. The whole algorithm and its analysis is intricate but group operations are only used in a few places. The reduction first takes group elements A and B from the given instance of DBDH problem and sets them to public key g_1 and g_2, respectively. The remaining parts of the public key u' and u_i are generated normally. It then simulates an individual key by
$$d_1 := g_1^{\frac{-J(v)}{F(v)}} (u' \prod_{i \in \mathcal{V}} u_i)^r \quad \text{and} \quad d_2 := g_1^{\frac{-1}{F(v)}} g^r \tag{4}$$
where we refer to [24] for $J(v)$ and $F(v)$. It is repeated for each key query and there are many d_1 and d_2 computed in the same manner. In the graph, these keys are represented by a single pair of nodes d₁ and d₂ directed from g1, u′, ui and g1, g, respectively. The reduction algorithm also creates a challenge ciphertext that includes
$$C_2 := C \quad \text{and} \quad C_3 := C^{J(v^*)}. \tag{5}$$
They are represented by nodes C2, C3, C and edges directed from C to C2 and C3. The resulting graph is (6) in Fig. 2.

Step 2. Merge. Merge the graphs from Step 1 into a single graph, $\mathit{\Gamma}$, as illustrated in Fig. 3.

Step 3. Split. Split $\mathit{\Gamma}$ into two graphs $\mathit{\Gamma}_0$ and $\mathit{\Gamma}_1$ such that

- No nodes or edges are lost, i.e., merging $\mathit{\Gamma}_0$ and $\mathit{\Gamma}_1$ recovers $\mathit{\Gamma}$.
- For every pair of paring nodes, if one node is in $\mathit{\Gamma}_0$, the other node is exclusively in $\mathit{\Gamma}_1$.
- For every node X in each split graph, the ancestor subgraph of X in $\mathit{\Gamma}$ is included in the same graph.

Given 3 pairs of pairing nodes in $\mathit{\Gamma}$, there exist 2^3 valid splits that satisfy the above conditions. Select a valid split $(\mathit{\Gamma}_0, \mathit{\Gamma}_1)$ according to a criterion for ones purpose of conversion. In Fig. 4, we give a valid split that yields a minimal public key size. As shown in Table 1, another valid split is possible to minimize the assumptions. We give an algorithm that searches for the best split according to an arbitrary criteria in Section 5.4.

Step 4. Derive the converted scheme. Nodes in $\mathit{\Gamma}_0$ and $\mathit{\Gamma}_1$ correspond to elements in $\mathbb{G}$ and $\tilde{\mathbb{G}}$, respectively. Based on the assignment, one can derive the resulting Waters' IBE scheme over Type-III groups and its underlying assumption as illustrated in Fig. 1.

Remark 1. To preserve the security, it is required that the security of the cryptosystem is proven by a black-box reduction [21] and that the reduction algorithms are abstract as defined in Section 4.2.

Remark 2. In the formal model, we consider correctness as part of scheme and hence a dependency graph for correctness should be included. As nodes are given consistent names in this example, the dependency graph for correctness becomes trivial and is therefore omitted. In general, consistent names are given by *object identifiers* as explained in Section 4.2.

Remark 3. It is important to check group membership for every input. For instance, if an input X in the original scheme is converted into X and $\tilde{X}$, then the group membership testing on X in the original scheme is translated to checking the Diffie-Hellman relation $e(X, \tilde{G}) = e(G, \tilde{X})$ in the converted scheme. In the above example, the relation between g_1 and $\tilde{g_1}$ in the common parameter should be verified. Since the common parameters will be verified once for all in practice, it is not explicitly shown in Fig. 1.

4 Formal Model

4.1 Cryptographic System

We consider secure cryptographic schemes whose correctness and security are defined by game-like interactive algorithms, and the security is proven by black-box

[Waters' IBE Scheme in Type-III]

Setup$((q, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_T, e, g, \tilde{g}))$: Select $g_2 \leftarrow \mathbb{G}$, $\alpha \leftarrow \mathbb{Z}_q$. Compute $g_1 := g^\alpha$, $\tilde{g}_1 := \tilde{g}^\alpha$. $u' \leftarrow \mathbb{G}$ and a random n-length vector $U = (u_i) \in \mathbb{G}^n$, those elements are chosen at random from $\mathbb{G}$. Publish the public parameters $prm = (g, \tilde{g}, g_1, \tilde{g}_1, g_2, u', U) \in \mathbb{G}^{4+n} \times \tilde{\mathbb{G}}^2$. The master secret is $msk = g^\alpha \in \mathbb{G}$.

KeyGeneration(prm): $r \leftarrow \mathbb{Z}_q$ and a private key for identity v is

$$d_v := (d_1, \tilde{d}_2) = \left(g_2^\alpha \left(u' \prod_{i \in \mathcal{V}} u_i \right)^r, \tilde{g}^r \right) \in \mathbb{G} \times \tilde{\mathbb{G}} \tag{6}$$

Encryption(prm, M): Let a message $M \in \mathbb{G}_T$. $t \leftarrow \mathbb{Z}_q$. The ciphertext is

$$C = (C_1, \tilde{C}_2, C_3) = \left(e(g_2, \tilde{g}_1)^t M, \tilde{g}^t, \left(u' \prod_{i \in \mathcal{V}} u_i \right)^t \right) \in \mathbb{G}_T \times \tilde{\mathbb{G}} \times \mathbb{G} \tag{7}$$

Decryption(prm, d_v, C): Parse $C = (C_1, \tilde{C}_2, C_3) \in \mathbb{G}_T \times \tilde{\mathbb{G}} \times \mathbb{G}$. Calculate

$$C_1 \frac{e(C_3, \tilde{d}_2)}{e(d_1, \tilde{C}_2)} = \left(e(g_2, \tilde{g}_1)^t M \right) \frac{e((u' \prod_{i \in \mathcal{V}} u_i)^t, \tilde{g}^r)}{e(g_2^\alpha (u' \prod_{i \in \mathcal{V}} u_i)^r, \tilde{g}_t)} = M$$

[Decisional Co-BDH problem]

Given $(g, \tilde{g}, A = g^a, \tilde{A} = \tilde{g}^a, B = g^b, C = g^c, \tilde{C} = \tilde{g}^c, Z) \in \mathbb{G}^4 \times \tilde{\mathbb{G}}^3 \times \mathbb{G}_T$, where $Z = e(g, \tilde{g})^{abc + \beta r}$, $r \leftarrow \mathbb{Z}_q$ and $\beta \leftarrow \{0, 1\}$, guess β.

Fig. 1. Converted Waters' IBE and underlying hard problem.

reductions to hardness of computational or decisional problems. Formally, we formulate a secure cryptosystem $\mathit{\Pi}$ by sets of efficient algorithms $\mathit{\Pi} = \{\mathcal{F}, \mathcal{C}, \mathcal{I}, \mathcal{R}\}$ that represent the functionality, correctness, underlying problems and security reductions. Properties of these algorithms are defined in the following.

The functionality $\mathcal{F}$ is a set of algorithms $\mathcal{F} = (\mathcal{F}_1, \ldots, \mathcal{F}_t)$ where each $\mathcal{F}_i$ implements some function for the cryptosystem such as "key generation", "encryption", and so on. Correctness of $\mathcal{F}$ is defined by $\mathcal{C}$ that has black-box access to the functionalities in $\mathcal{F}$ and outputs 1 if everything works as intended.

Definition 4 (Correctness). *$\mathit{\Pi}$ is correct if* $\Pr[1 \leftarrow \mathcal{C}^{\mathcal{F}}(1^\lambda)] = 1$ *for all* λ.

A problem $\mathcal{I}$ is a triple of algorithms $\mathcal{I} = (\mathcal{I}_{gen}, \mathcal{I}_{ver}, \mathcal{I}_{guess})$, where $\mathcal{I}_{gen}$ is an instance generator that generates a problem instance, $\mathcal{I}_{ver}$ is a verification algorithm that verifies a given answer, and $\mathcal{I}_{guess}$ is a guessing algorithm that returns an answer by random guessing.

Definition 5 (Hardness of $\mathcal{I}$). *Problem $\mathcal{I}$ is hard if the advantage function*

$$\mathrm{Adv}_{\mathcal{B}}^{\mathcal{I}}(\lambda) := \Pr\left[1 \leftarrow \mathcal{I}_{ver}(x, y) \mid (x, y) \leftarrow (\mathcal{I}_{gen}(1^\lambda), \mathcal{B}(1^\lambda))\right] - \mathcal{I}_{guess}(1^\lambda)$$

is negligible in λ for any probabilistic polynomial-time adversary $\mathcal{B}$.

In this work, we consider cryptographic schemes where security is proven by an efficient algorithm called a reduction, $\mathcal{R}$, that is successful in solving problem $\mathcal{I}$ given black-box access to an adversary that successfully attacks the scheme. We define security in the form of advantage functions $\mathrm{Adv}_{\mathcal{A}}^{\mathcal{S}}(\lambda) := \Pr[1 \leftarrow \mathcal{S}^{\mathcal{A}}(1^{\lambda})]$ for algorithm $\mathcal{S}$ (which is often called a challenger), which should be negligible for any probabilistic polynomial time adversary $\mathcal{A}$.

Definition 6 (Security of $\mathit{\Pi}$). *Cryptosystem $\mathit{\Pi}$ is secure in the sense of $\mathcal{S}$ under the hardness assumption on $\mathcal{I}$ with black-box reduction $\mathcal{R}$ if for any $\mathcal{A}$ advantage $\mathrm{Adv}_{\mathcal{R}^{\mathcal{A}}}^{\mathcal{I}}(\lambda)$ is not negligible if $\mathrm{Adv}_{\mathcal{A}}^{\mathcal{S}}(\lambda)$ is not negligible.*

Though $\mathcal{C}$, $\mathcal{R}$, $\mathcal{I}$, and $\mathcal{S}$ are defined as single algorithms, they can be naturally extended to sets of algorithms. In particular, security is often proven by sequences of games and each game reduces to an individual hardness assumption.

4.2 Abstract Algorithms

Let $\tilde{A}^{\mathcal{O}}$ denote an algorithm where $\tilde{A}$ is called an abstract algorithm that computes group operations through oracle $\mathcal{O}$. Oracle $\mathcal{O}$ is called a group operation oracle and given locally to host algorithm $\tilde{A}$. It is initialized with a description of bilinear groups that is common for all algorithms in a cryptosystem. It performs generic group operations over the bilinear groups like the generic group oracle [22]. We follow the model by Maurer [17] for group operations. It forces the host algorithm be explicit in checking equality. It is useful for our purpose as we need to know which elements the host algorithm tests equality.

Oracle $\mathcal{O}$ also works as an interface for sending and receiving group elements. When $\tilde{A}^{\mathcal{O}}$ and $\tilde{B}^{\mathcal{O}}$ interact, group elements are sent and received through the oracles and all other data are exchanged directly between the host algorithms. As mentioned above, a description of bilinear groups is common to the oracles.

We also consider $\mathcal{O}$ in the random oracle model [4] to capture functions that map arbitrary input to a random source group element and that map group elements attached by arbitrary string to a random string. $\mathcal{O}$ provides these functions by interacting with random oracles $H_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ and $H_{str} : \mathbb{G}^k \times \{0,1\}^* \to \{0,1\}^{\ell}$ for some k and ℓ.

Every group element (more precisely a pointer to it) is associated with an object identifier (oid for short). It is an arbitrarily prescribed string that identifies the role of the element in a cryptosystem like "the third element of a ciphertext" or "the first element of a secret-key." The way oids are assigned to group elements is a part of an algorithm and indeed as important as computations in the algorithm. We restrict that only a constant number of distinct oids is used in a cryptosystem so that a dependency graph can be described in a constant size as we explain later. In general, oids can be arbitrarily specified. We consider a conventional case where oids are named after variables used in describing algorithms in $\mathcal{F}$ and $\mathcal{I}$. It is indeed how we did for Waters' IBE in Section 3. When there are indexed variables that grows in the security parameter like the public-key of Water's IBE, they are assigned the same oid if they are involved in the computation in the same manner. For instance, the same oid ui is given to all group elements $u_1, \dots, u_n$ in the example in Section 3. This convention for oids applies to all schemes considered in this paper.

Let $(q, \mathbb{G}, \mathbb{G}_T, e, G)$ be symmetric bilinear groups and let $G_t = e(G, G)$. We define an extended group operation oracle $\mathcal{O}$ for symmetric bilinear group of prime order q. In the following, pointers are taken sequentially from 1, and queries with unused pointers are rejected. We omit group operations and equality checking in the target group in the following description.

[Extended Group Operation Oracle $\mathcal{O}$]

- init: Initialize lists $\mathcal{L}_s$ and $\mathcal{L}_t$ with entries (pt, G) and (pt_t, G_t) respectively with fresh pointers pt and pt_t. Return (q, pt, pt_t).
- $\mathsf{gop}(pt_1, a_1, \dots, pt_k, a_k, oid)$: For every pt_i, search $\mathcal{L}_s$ for (pt_i, X_i). Compute $X := \prod {X_i}^{a_i} \in \mathbb{G}$ and store (pt, X) with fresh pt. Output pt.
- $\mathsf{pair}(pt_1, pt_2)$: Search $\mathcal{L}_s$ for (pt_1, X_1) and (pt_2, X_2). Compute $X := e(X_1, X_2)$ and store (pt, X) to $\mathcal{L}_t$ with fresh pt. Output pt.
- $\mathsf{equal}(pt_1, pt_2)$: Search $\mathcal{L}_s$ for (pt_1, X_1) and (pt_2, X_2). If $X_1 \equiv X_2 \in \mathbb{G}$ then return 1. Return 0, otherwise.
- $\mathsf{hash2g}(str, oid)$: (This query is accepted only when random oracle $H_{\mathbb{G}}$ is available.) If the same input has been queried before, return the same answer. Otherwise send (str, oid) to $H_{\mathbb{G}}$ and receive $X \in \mathbb{G}$. Store (pt, X) to $\mathcal{L}_s$ with fresh pt, and return pt.
- $\mathsf{hash2str}(pt_1, \dots, pt_k, str, oid)$: (This query is accepted only when random oracle H_{str} is available.) Search $\mathcal{L}_s$ for each $X_i \in \mathbb{G}$ that corresponds to pt_i. If $(X_1, \dots, X_k, str, oid)$ has been asked before, return the same value. Otherwise, send it to H_{str} and return the resulting string.
- $\mathsf{send}(pt, oid)$: Search $\mathcal{L}_s$ for (pt, X) and send (X, oid) to the implicitly specified destination.
- $\mathsf{receive}(oid)$: On receiving this query from the host algorithm, wait to receive (X, oid') from implicitly specified entity. Reject if $X \notin \mathbb{G}$ or $oid \neq oid'$ and continue waiting. Otherwise, store (pt, X) to $\mathcal{L}_s$ with fresh pt, and send pt to the host algorithm.

We make some remarks about object identifier oid given as input for most queries. When calling gop and $\mathsf{hash2g}$, the host algorithm assigns an object identifier to the resulting group element by specifying it with oid. The oracle does not use oid in handling gop query, but it is needed later to build a dependency graph. It is important to see that oid is included in the input to the random oracle in $\mathsf{hash2g}$. It allows the host algorithm to virtually deal with several independent random oracles indexed by oid. For $\mathsf{hash2str}$, it is assumed that every group element X_i is transformed to its canonical representation in $\mathbb{G}$ before being sent to random oracle H_{str}. In general, even if $X \equiv Y \in \mathbb{G}$ holds, hashing X and Y may yield different values. This is an important issue as we simulate a group element in Type-I groups with a pair of group elements in Type-III allowing different representations. With oid specified in the input, we can control the representation so that group elements having the same oid has the same representation.

Let $\Sigma_{\mathcal{G}_{\mathsf{sym}}}$ be the set of possible extended group operation oracles based on a group generated by $\mathcal{G}_{\mathsf{sym}}$ for all sufficiently large λ and all random coins/oracles. We say $\mathcal{O}$ is based on $\mathcal{G}_{\mathsf{sym}}$ when we refer to an extended group operation oracle $\mathcal{O}$ in $\Sigma_{\mathcal{G}_{\mathsf{sym}}}$. We now define a cryptosystem consisting of abstract algorithms. Let $\tilde{\Pi}^{\mathcal{O}}_{\mathcal{G}_{\mathsf{sym}}} := (\tilde{\mathcal{F}}^{\mathcal{O}}, \tilde{\mathcal{C}}^{\mathcal{O}}, \tilde{\mathcal{R}}^{\mathcal{O}}, \tilde{\mathcal{I}}^{\mathcal{O}})$ be a cryptosystem obtained by giving oracle $\mathcal{O}$ based on $\mathcal{G}_{\mathsf{sym}}$ to (sets of) abstract algorithms $\tilde{\mathcal{F}}$, $\tilde{\mathcal{C}}$, $\tilde{\mathcal{R}}$, and $\tilde{\mathcal{I}}$. Let also $\tilde{\mathcal{S}}$ be an abstract challenger algorithm. Let Δ_q denote all $\mathcal{G}_{\mathsf{sym}}$ that outputs q with the same probability distribution.

Definition 7 (Correct and Secure Abstract Cryptosystem). *A set of abstract algorithms $\tilde{\Pi} = (\tilde{\mathcal{F}}, \tilde{\mathcal{C}}, \tilde{\mathcal{R}}, \tilde{\mathcal{I}})$ is an abstract cryptosystem with respect to Δ_q and it correct and secure in the sense of $\tilde{\mathcal{S}}$ if, for any $\mathcal{G}_{\mathsf{sym}} \in \Delta_q$ and $\mathcal{O} \in \Sigma_{\mathcal{G}_{\mathsf{sym}}}$, $\tilde{\Pi}^{\mathcal{O}}_{\mathcal{G}_{\mathsf{sym}}} := (\tilde{\mathcal{F}}^{\mathcal{O}}, \tilde{\mathcal{C}}^{\mathcal{O}}, \tilde{\mathcal{R}}^{\mathcal{O}}, \tilde{\mathcal{I}}^{\mathcal{O}})$ is a cryptosystem that is correct and secure in the sense of $\tilde{\mathcal{S}}^{\mathcal{O}}$.*

5 Conversion Using Dependency Graph

5.1 Simulating Group Operation Oracle

Let $(q, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_T, e, G, \tilde{G})$ be asymmetric bilinear groups generated by an asymmetric group generator $\mathcal{G}_{\mathsf{asym}}$. Let $\phi : \mathbb{G} \to \tilde{\mathbb{G}}$ be an (inefficient) isomorphism between the source groups. We use three representations, $(\mathbb{G}, -)$, $(-, \tilde{\mathbb{G}})$, and $(\mathbb{G}, \tilde{\mathbb{G}})$, for a source group element and say that they are of type left, right, and both, respectively. By type we denote $\{\mathsf{left}, \mathsf{right}, \mathsf{both}\}$. We say that type t is *covered* by t' if $t' = \mathsf{both}$ or $t' = t$, and denote by $t \subseteq t'$. If two types cover at least in one way, we say that they are *compatible*. We design operations so that they can be performed efficiently over two compatible elements. Let $\mathbb{G}'$ denote $(\mathbb{G} \cup \{\perp\}) \times (\tilde{\mathbb{G}} \cup \{\perp\}) \setminus (\perp, \perp)$ where $\perp$ represents absence of data. Let $\mathcal{J} : \mathbb{G}' \to \mathsf{type}$ be a function that takes an element of $\mathbb{G}'$ as input and outputs its type. Let $\mathsf{matchtype} : \mathbb{G}' \times \mathsf{type} \to \mathbb{G}'$ be a subroutine that takes $(X, \tilde{X}) \in \mathbb{G}'$ and $t \in \mathsf{type}$ as input, and remodeled $(X, \tilde{X})$ so that $\mathcal{J}(X, \tilde{X}) = t$ holds. It is done by computing $\tilde{X} := \phi(X)$ or $X := \phi^{-1}(\tilde{X})$ if necessary, and setting $\perp$ to X or $\tilde{X}$ if either is unnecessary.

Types are assigned by an algorithm $\mathcal{D}$ called a *deployment*. It is an algorithm that takes an object identifier as input and outputs a type to assign to the identifier. Based on the asymmetric bilinear groups and deployment $\mathcal{D}$, we construct an oracle $\mathcal{O}^*$ that simulates a symmetric group operation oracle.

[Simulated Group Operation Oracle $\mathcal{O}^*$]

- init: Initialize lists $\mathcal{L}_s$ and $\mathcal{L}_t$ with entries $(pt, G, \tilde{G})$ and (pt_t, G_t) respectively with fresh pointers pt and pt_t. Return (q, pt, pt_t).
- $\mathsf{gop}(pt_1, a_1, \ldots, pt_k, a_k, oid)$: For every pt_i, search $\mathcal{L}_s$ for $(pt_i, X_i, \tilde{X}_i)$. For every i where $\mathcal{J}(X_i, \tilde{X}_i)$ does not cover $t := \mathcal{D}(oid)$, call $\mathsf{matchtype}(X_i, \tilde{X}_i, t)$. Then compute $X := \prod X_i^{a_i}$ for $t = \mathsf{left}$, or $\tilde{X} := \prod \tilde{X}_i^{a_i}$ for $t = \mathsf{right}$, or both for $t = \mathsf{both}$. Set $\perp$ to X or $\tilde{X}$ if either is not computed. Store $(pt, X, \tilde{X})$ with fresh pt, and return pt.

- $\mathsf{pair}(pt_1, pt_2)$: Search $\mathcal{L}_s$ for $(pt_1, X_1, \tilde{X}_1)$ and $(pt_2, X_2, \tilde{X}_2)$. If both $\mathcal{J}(X_1, \tilde{X}_1)$ and $\mathcal{J}(X_2, \tilde{X}_2)$ are left or right, then use ϕ or ϕ^{-1} to compute an element on the missing side. Then compute $Z := e(X_1, \tilde{X}_2)$ or $e(X_2, \tilde{X}_1)$ whichever possible. Store (pt, Z) to $\mathcal{L}_t$ with fresh pointer pt, and output pt.
- $\mathsf{equal}(pt_1, pt_2)$: Search $\mathcal{L}_s$ for $(pt_1, X_1, \tilde{X}_1)$ and $(pt_2, X_2, \tilde{X}_2)$. If $\mathcal{J}(X_1, \tilde{X}_1)$ and $\mathcal{J}(X_2, \tilde{X}_2)$ are incompatible, then compute either $\tilde{X}_2 := \phi(X_2)$ or $X_2 := \phi^{-1}(\tilde{X}_2)$ whichever missing. Then if $X_1 \equiv X_2 \in \mathbb{G}$ or $\tilde{X}_1 \equiv \tilde{X}_2 \in \tilde{\mathbb{G}}$, return 1. Return 0, otherwise.
- $\mathsf{hash2g}(str, oid)$: (This query is accepted only when random oracles $H_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ and $H_{\tilde{\mathbb{G}}} : \{0,1\}^* \to \tilde{\mathbb{G}}$ are available.) Compute $t \leftarrow \mathcal{D}(oid)$ and pick fresh pt. If (str, oid) has been queried before and answered with $(pt', X, \tilde{X}) \in \mathcal{L}_s$, store $(pt, X, \tilde{X})$ to $\mathcal{L}_s$ and return pt. Otherwise query (str, oid) to random oracle $H_{\mathbb{G}}$ for $t = \mathsf{right}$ or $H_{\tilde{\mathbb{G}}}$ for $t = \mathsf{left}$. If $t = \mathsf{both}$, query (str, oid) to $H_{\mathbb{G}}$ and use ϕ to get the corresponding element in $\tilde{\mathbb{G}}$. Store the result with pt to $\mathcal{L}_s$. Then return pt.
- $\mathsf{hash2str}(pt_1, \dots, pt_k, str, oid)$: (This query is accepted only when random oracle $H_{str} : (\mathbb{G}')^k \times \{0,1\}^* \to \{0,1\}^{\mathrm{poly}(\lambda)}$ is available.) Search $\mathcal{L}_s$ for each $(pt_i, X_i, \tilde{X}_i)$. Let $oid := (oid_i, \dots, oid_k)$. For every i where $\mathcal{J}(X_i, \tilde{X}_i)$ does not cover $t_i := \mathcal{D}(oid_i)$, perform $\mathsf{matchtype}(X_i, \tilde{X}_i, t_i)$. Then, if $(X_1, \tilde{X}_1, \dots, X_k, \tilde{X}_k, str, oid)$ has been queried before, return the same value. Otherwise, send it to random oracle H_{str}, and receive a string, str'. Then return str'.
- $\mathsf{send}(pt, oid)$: Search $\mathcal{L}_s$ for $(pt, X, \tilde{X})$. Compute $t \leftarrow \mathcal{D}(oid)$. If $\mathcal{J}(X, \tilde{X}) \neq t$, call $\mathsf{matchtype}(X, \tilde{X}, t)$. Output $((X, \tilde{X}), oid)$.
- $\mathsf{receive}(oid)$: On receiving this query from the host algorithm, wait for receiving $((X, \tilde{X}), oid')$ from outside. Ignore if $(X, \tilde{X}) \notin \mathbb{G}'$ or $oid' \neq oid$ or $\mathcal{J}(X, \tilde{X}) \neq \mathcal{D}(oid)$, and continue waiting. Otherwise, pick fresh pt, store $(pt, X, \tilde{X})$ to $\mathcal{L}_s$, and send pt to the host algorithm.

Observe that there are some cases where oracle $\mathcal{O}^*$ performs inefficient computations ϕ or ϕ^{-1}. Nevertheless, it is not hard to inspect that $\mathcal{O}^*$ perfectly simulates the extended symmetric group operation oracle.

For abstract cryptosystem $\tilde{\Pi}$, let $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{\mathsf{asym}}, \mathcal{D}}$ denote a cryptosystem where oracle $\mathcal{O}^*$ based on asymmetric groups generated by $\mathcal{G}_{\mathsf{asym}}$ and deployment $\mathcal{D}$. Let Δ'_q denote the set of $\mathcal{G}_{\mathsf{asym}}$ that outputs q with the same distribution as those in Δ_q. We claim that if $\tilde{\Pi}^{\mathcal{O}}_{\mathcal{G}_{\mathsf{sym}}}$ is correct and secure, then so is $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{\mathsf{asym}}, \mathcal{D}}$ for any $\mathcal{G}_{\mathsf{sym}} \in \Delta_q$, $\mathcal{G}_{\mathsf{asym}} \in \Delta'_q$, and $\mathcal{D}$ that stops with an output for any input. Nevertheless, the notion of secure Π requires algorithms to run efficiently. For a while, we assume that ϕ and ϕ^{-1} can be computed efficiently and state the following.

Lemma 8. *If $\tilde{\Pi}$ is a cryptosystem with respect to Δ_q, and it is correct and secure in the sense of $\tilde{\mathcal{S}}$, then for any $\mathcal{G}_{asym} \in \Delta'_q$ and for any $\mathcal{D}$, cryptosystem $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{asym}, \mathcal{D}}$ is correct and secure in the sense of $\tilde{\mathcal{S}}^{\mathcal{O}^*}$ if $\mathcal{O}^*$ computes ϕ and ϕ^{-1} efficiently.*

Proof. (sketch) Correctness can be assured by observing that views of abstract algorithms with $\mathcal{O}^*$ and $\mathcal{O}$ are identical. Regarding security, we show that if there exists $\mathcal{G}_{\mathsf{sym}} \in \Delta_q$ and $\mathcal{A}$ that successfully attacks $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{\mathsf{asym}},\mathcal{D}}$ in the sense of $\tilde{\mathcal{S}}^{\mathcal{O}^*}$, then there exists adversary $\mathcal{B}$ that is successful in attacking $\tilde{\Pi}^{\mathcal{O}}_{\mathcal{G}_{\mathsf{sym}}}$ in the sense of $\tilde{\mathcal{S}}^{\mathcal{O}}$ for $\mathcal{G}_{\mathsf{sym}}$. We construct $\mathcal{G}_{\mathsf{sym}}$ using $\mathcal{G}_{\mathsf{asym}}$ by representing a group element with a pair of source group elements of asymmetric groups. We then construct $\mathcal{B}$ by using $\mathcal{A}$. On receiving a group element, $\mathcal{B}$ invokes $\mathcal{D}$ and run matchtype to remodel the input for $\mathcal{A}$. Outgoing group elements from $\mathcal{A}$ are also remodeled by applying matchtype. Non-group elements are sent and received intact. We argue that if $\mathrm{Adv}^{\tilde{\mathcal{S}}^{\mathcal{O}^*}}_{\mathcal{A}}(\lambda)$ is not negligible then so is $\mathrm{Adv}^{\tilde{\mathcal{S}}^{\mathcal{O}}}_{\mathcal{B}}(\lambda)$ since the view of $\tilde{\mathcal{S}}$ is identical. Since $\mathcal{O}^*$ computes ϕ and ϕ^{-1} in matchtype efficiently by hypothesis, all algorithms are efficient here. □

In reality, however, ϕ and ϕ^{-1} are inefficient in Type-III groups and hence $\mathcal{O}^*$ is inefficient in general. Nevertheless, there may exist $\tilde{\Pi}$ and $\mathcal{D}$ where $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{\mathsf{asym}},\mathcal{D}}$ never performs the inefficient computation. For such $\tilde{\Pi}$ and $\mathcal{D}$, $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{\mathsf{asym}},\mathcal{D}}$ is correct and secure. Accordingly, the task of conversion is now reduced to find efficient $\mathcal{D}$ that never have $\mathcal{O}^*$ compute either ϕ or ϕ^{-1}. It is the main issue we address in the rest of this section.

We proceed to argue whether the assumption deduced from the converted problem is plausible or not. Consider group operation oracle $\mathcal{O}$ based on $\mathcal{G}_{\mathsf{sym}}$. Let $\mathcal{I}_{\mathsf{sym}} = (\tilde{\mathcal{I}}^{\mathcal{O}}_{gen}, \tilde{\mathcal{I}}^{\mathcal{O}}_{ver}, \tilde{\mathcal{I}}^{\mathcal{O}}_{guess})$ be a problem defined over symmetric bilinear groups. Similarly, let $\mathcal{O}^*$ be a group operation oracle based on $\mathcal{G}_{\mathsf{asym}}$ and $\mathcal{D}$. Let $\mathcal{I}_{\mathsf{asym}} = (\tilde{\mathcal{I}}^{\mathcal{O}^*}_{gen}, \tilde{\mathcal{I}}^{\mathcal{O}^*}_{ver}, \tilde{\mathcal{I}}^{\mathcal{O}^*}_{guess})$ be a problem defined over asymmetric bilinear groups. Let $\{\mathcal{D}\}$ be a set of $\mathcal{D}$ that makes $\mathcal{I}_{\mathsf{asym}}$ efficient. By $\{\mathcal{I}_{\mathsf{asym}}\}$ we denote the family of problems obtained by defining $\mathcal{I}_{\mathsf{asym}}$ for each $\mathcal{D} \in \{\mathcal{D}\}$. We call $\{\mathcal{I}_{\mathsf{asym}}\}$ a family of co*-problems.

Some restrictions apply to $\mathcal{I}_{\mathsf{sym}}$. We only consider $\mathcal{I}_{\mathsf{sym}}$ that verifies membership of all incoming group elements, and do not use hash functions hash2g and hash2str. Furthermore, the default generator is included in every problem instance. Then the following holds.

Theorem 9 (Generic Hardness of co*-Problem Family). *If there exists $(q_s, \tilde{q}_s, q_t, q_p, \epsilon, \tau)$-successful generic adversary for $\mathcal{I}_{\mathsf{asym}} \in \{\mathcal{I}_{\mathsf{asym}}\}$, then there exists an $(q_s + \tilde{q}_s, q_t, q_p, \epsilon, \tau')$-successful generic adversary against $\mathcal{I}_{\mathsf{sym}}$ where τ' is $\tau + O(q_s + \tilde{q}_s)$.*

5.2 Dependency Graphs

We begin by defining a dependency graph for an abstract algorithm. Let $\mathcal{O}$ be a group operation oracle for some group order q. Let $\tilde{\mathcal{A}}$ be an abstract algorithm and $\tilde{\mathcal{A}}^{\mathcal{O}}(in)$ be its execution on input *in*. (When $\tilde{\mathcal{A}}$ is interactive, *in* also represents inputs obtained through interaction.) We define a dependency graph for $\tilde{\mathcal{A}}$ through the interaction between $\tilde{\mathcal{A}}$ and $\mathcal{O}$ as follows. It also defines a list NoDup called a ban list that includes oids for group elements generated by hash2g.

[Dependency Graph for Abstract Algorithm $\tilde{\mathcal{A}}$]

1. Initialize Γ to an empty directed graph. Also initialize lists L_{pair}, L_{eq}, and NoDup to be empty.
2. Pick $\mathcal{O}$ for order q, and select *in* from appropriate domain. Initialize L_{pt} to empty. Run $\tilde{\mathcal{A}}$. For each query from $\tilde{\mathcal{A}}$ to $\mathcal{O}$, do as follows.
   - $\mathsf{gop}(pt_1, a_1, \ldots, pt_k, a_k, oid) \rightarrow pt$: Add a node labeled by oid to Γ and record (pt, oid) to L_{pt}. Then, for every node oid_i that corresponds to pt_i, add an edge (oid_i, oid) to Γ.
   - $\mathsf{pair}(pt_1, pt_2)$: Find (pt_1, oid_1) and (pt_2, oid_2) from L_{pt}. If $(oid_1, oid_2) \in L_{pair}$ in any order, do nothing. Otherwise, pick two nodes with unique labels, say $p[0]$ and $p[1]$, and add them to Γ. Add edges $(oid_1, p[0])$ and $(oid_2, p[1])$ to Γ as well. Add (oid_1, oid_2) to L_{pair}.
   - $\mathsf{equal}(pt_1, pt_2)$: Let oid_1 and oid_2 be nodes for pt_1 and pt_2, respectively. If $oid_1 = oid_2$, or $(oid_1, oid_2) \in L_{eq}$ in any order, do nothing. Otherwise, add a node with a unique label, say E, and edges (oid_1, E) and (oid_2, E) to Γ. Add (oid_1, oid_2) to L_{eq}.
   - $\mathsf{hash2g}(str, oid) \rightarrow pt$: Add a node oid to Γ and add (pt, oid) to L_{pt}. Store oid to NoDup if not yet stored.
   - $\mathsf{hash2str}(pt_1, \ldots, pt_k, str, oid)$: Let $oid = (oid_i, \ldots, oid_k)$. For every oid'_i stored with pt_i in L_{pt}, if $oid_i \neq oid'_i$, then add node oid_i and edge (oid'_i, oid_i) to Γ. (This means that the element identified by pt_i was originally associated to object identifier oid'_i but now regarded as oid_i by the host algorithm.)
   - $\mathsf{send}(pt, oid)$: For oid' stored with pt in L_{pt}, if $oid \neq oid'$, then add node oid' and edge (oid', oid).
   - $\mathsf{receive}(oid) \rightarrow pt$: Add node oid to Γ. Then record (pt, oid) to L_{pt}.

   In the above, adding nodes and edges are done if they do not exist in Γ. Also skip adding self-directing edges.
3. Go back to step 2 and repeat the above for all q and *in*.

The above algorithm defines how to construct a dependency graph for $\tilde{\mathcal{A}}$. In fact, repeating for all q and *in* as instructed in the last step is infeasible in reality. We nevertheless expect that Γ is finite size for certain $\tilde{\mathcal{A}}$. It is particularity the case when $\tilde{\mathcal{A}}$ behaves independently of the security parameter, or nodes related to the security parameter are indexed and given the same object identifier as we see for public key u_i in the example in Section 3. In the real world, building a dependency graph for algorithm $\tilde{\mathcal{A}}$ needs to look into $\tilde{\mathcal{A}}$ rather than treating $\tilde{\mathcal{A}}$ as black-box as above.

The nodes added in pair are called *pairing nodes.* A node is called a *regular node* if it is not either of the above.

Next we define a dependency graph for a cryptosystem, $\tilde{\Pi} = (\tilde{\mathcal{F}}, \tilde{\mathcal{C}}, \tilde{\mathcal{R}}, \tilde{\mathcal{I}})$. Basically it is a graph obtained by merging dependency graphs for all algorithms in $\tilde{\Pi}$. Yet we need to work on some details for formality as shown below.

[Dependency Graph for Cryptosystem $\tilde{\Pi}$]

1. Build a dependency graph and list NoDup for every algorithm in $\tilde{\Pi}$. It is assumed that pairing nodes, equality nodes, and local nodes are given globally unique names.

2. Merge all the graphs and NoDup obtained in the previous step.
3. If two nodes are connected to more than one pair of pairing nodes, remove all but one pair of the pairing nodes. Do the same for equality nodes.
4. Output the resulting graph and NoDup.

Let $\mathcal{C}_{\mathsf{const}}$ be a class of abstract cryptosystems that has a constant-size dependency graph in the security parameter. In the rest of the paper, we focus on cryptosystems in $\mathcal{C}_{\mathsf{const}}$.

5.3 Deployment Algorithm

Given a dependency graph Γ for $\tilde{\Pi} \in \mathcal{C}_{\mathsf{const}}$, we construct a deployment $\mathcal{D}$. Recall that if the target algorithm performs $\mathsf{hash2g}(str, oid)$, then the result should be a group element in either $\mathbb{G}$ or $\tilde{\mathbb{G}}$ but not both simultaneously. Thus $\mathcal{D}$ must not return both for such oid. To deal with similar demand from practice that some nodes should stay in either group, we use a ban list, NoDup, which specifies oids that must not be assigned to both.

We consider splitting a dependency graph into two graphs so that each graph represents nodes and computations in $\mathbb{G}$ or $\tilde{\mathbb{G}}$. The split must meet the conditions defined below.

Definition 10 (Valid Split). *Let $\Gamma = (V, E)$ be a dependency graph for $\tilde{\Pi} \in \mathcal{C}_{\mathsf{const}}$. Let $P = (p_1[0], \ldots, p_{n_p}[1]) \subset V$ be pairing nodes. A pair of graphs $\Gamma_0 = (V_0, E_0)$ and $\Gamma_1 = (V_1, E_1)$ is a valid split of Γ with respect to $\mathsf{NoDup} \subseteq V$ if:*

1. *merging Γ_0 and Γ_1 recovers Γ,*
2. *for each $i \in \{0, 1\}$ and every $X \in V_i \setminus P$ the subgraph $\mathsf{Anc}(\Gamma, X)$ is in Γ_i,*
3. *for each $i \in \{1, \ldots, n_p\}$ paring nodes $p_i[0]$ and $p_i[1]$ are separately included in V_0 and V_1.*
4. *No node in $V_0 \cap V_1$ is included in NoDup.*

We then construct a deployment $\mathcal{D}$ based on a valid split as follows:

[Deployment Algorithm $\mathcal{D} : \mathsf{oid} \to \mathsf{type}$]
Given object identifier oid as input, return left or right or both if a node labeled as oid is included in Γ_0 or Γ_1 or both, respectively.

Lemma 11. *If there exists a valid split with respect to $\tilde{\Pi} \in \mathcal{C}_{\mathsf{const}}$ and NoDup, then, for oracle $\mathcal{O}^*$ based on $\mathcal{G}_{\mathsf{asym}}$ and $\mathcal{D}$ with a valid split, $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{\mathsf{asym}}, \mathcal{D}}$ is efficient.*

The above lemma can be proved by observing that for each case that $\tilde{\Pi}^{\mathcal{O}^*}_{\mathcal{G}_{\mathsf{asym}}, \mathcal{D}}$ computes inefficient isomorphisms in gop, pair, equal, $\mathsf{hash2g}$, and $\mathsf{hash2str}$, the split $\mathcal{D}$ is based on must be invalid.

5.4 How to Find the Best Valid Split

Let $\varGamma = (V, E)$ be a dependency graph for $\tilde{\varPi}$ where V consists of regular nodes $\{X_1, \ldots, X_k\}$ and pairing nodes $P := \{p_1[0], \ldots, p_{n_p}[1]\}$. We construct an algorithm $\mathsf{FindSplit}$ that finds all valid splits.

[**Algorithm:** $\mathsf{FindSplit}(\varGamma,\ \mathsf{NoDup})$]

```
1. Initialize $L$ to empty.
2. Set $B \subseteq R$ so that every $B_i \in B$ has no outgoing edges. For each $X_i \in R$ do:
   – if $X_i$ is not in $\mathsf{Anc}(\varGamma, B_i)$ for all $B_i \in B \cup P$, then:
      • For every $X_j \in B$ that is in $\mathsf{Anc}(\varGamma, X_i)$, do $B := B \setminus \{X_j\}$.
      • $B := B \cup \{X_i\}$.
3. Repeat the following for $\ell = 0, \ldots, 2^{n_p+n_b} - 1$ where $n_b := |B|$.
   (a) Set $\varGamma_0 = (V_0, E_0), \varGamma_1 = (V_1, E_1)$ be empty graphs.
   (b) For $i = 1, \ldots, n_p$, do $\varGamma_0 \leftarrow \varGamma_0 \oplus \mathsf{Anc}(\varGamma, p_i[\mathsf{bit}_i(\ell)])$ and $\varGamma_1 \leftarrow \varGamma_1 \oplus \mathsf{Anc}(\varGamma, p_i[1 - \mathsf{bit}_i(\ell)])$.
   (c) For $j = 1, \ldots, n_b$ and $i = \mathsf{bit}_{n_p+j}(\ell)$, do $\varGamma_i \leftarrow \varGamma_i \oplus \mathsf{Anc}(\varGamma, B_j)$.
   (d) Append $(\varGamma_0, \varGamma_1)$ to $L$ if $V_0 \cap V_1 \cap \mathsf{NoDup}$.
4. Output $L$.
```

Lemma 12. *List L includes all valid split of $\varGamma$.*

Proof. (sketch) We verify that every $(\varGamma_0, \varGamma_1)$ in L satisfies the conditions in Definition 10. First we show that $\varGamma_0 \oplus \varGamma_1 = \varGamma$. Observe that $\mathsf{FindSplit}$ is deterministic and only the order of elements in R may impact the result through construction of B. Consider B obtained from R, and B' from permutation of R. Suppose that $X \in B$ and $X \notin B'$ happens. Then there exits $Y \in B'$ that has path from X to Y. We can argue that such Y cannot exist in B without contradicting to the presence of X in B. Similarly, as Y is not in B, there exists node, say Z, in B that has path from Y to Z. If Z is not identical to X, there exists path from X to Z that contradicts to the presence of X in B. Thus, we have $X = Z$. This means that X and Y are on a circle and thus $\mathsf{Anc}(\varGamma, X) = \mathsf{Anc}(\varGamma, Y)$. Thus procedures in further steps are not affected whichever B or B' are used. Then observe that, from step (b) and (c), we have $\varGamma_0 \oplus \varGamma_1 = \mathsf{Anc}(\varGamma, p_1[1]) \oplus \mathsf{Anc}(\varGamma, p_1[0]) \oplus \cdots \oplus \mathsf{Anc}(\varGamma, p_1[0]) \oplus \mathsf{Anc}(\varGamma, B_1) \oplus \cdots \oplus \mathsf{Anc}(\varGamma, B_{n_b}) = \varGamma$. Next, the second condition is met since, in step 3-(b) and (c), every node is included in $\varGamma_i$ together with their ancestor subgraphs. By the property of $\mathsf{Anc}()$, the subgraph contains a subgraph of every node in it. The third condition is assured since in step 3-(b) every pair of pairing nodes are merged to $\varGamma_0$ and $\varGamma_1$ separately. Finally, the constraint by NoDup is met due to step 3-(d) which forces the exactly the same constraint as in the fourth condition. ∎

References

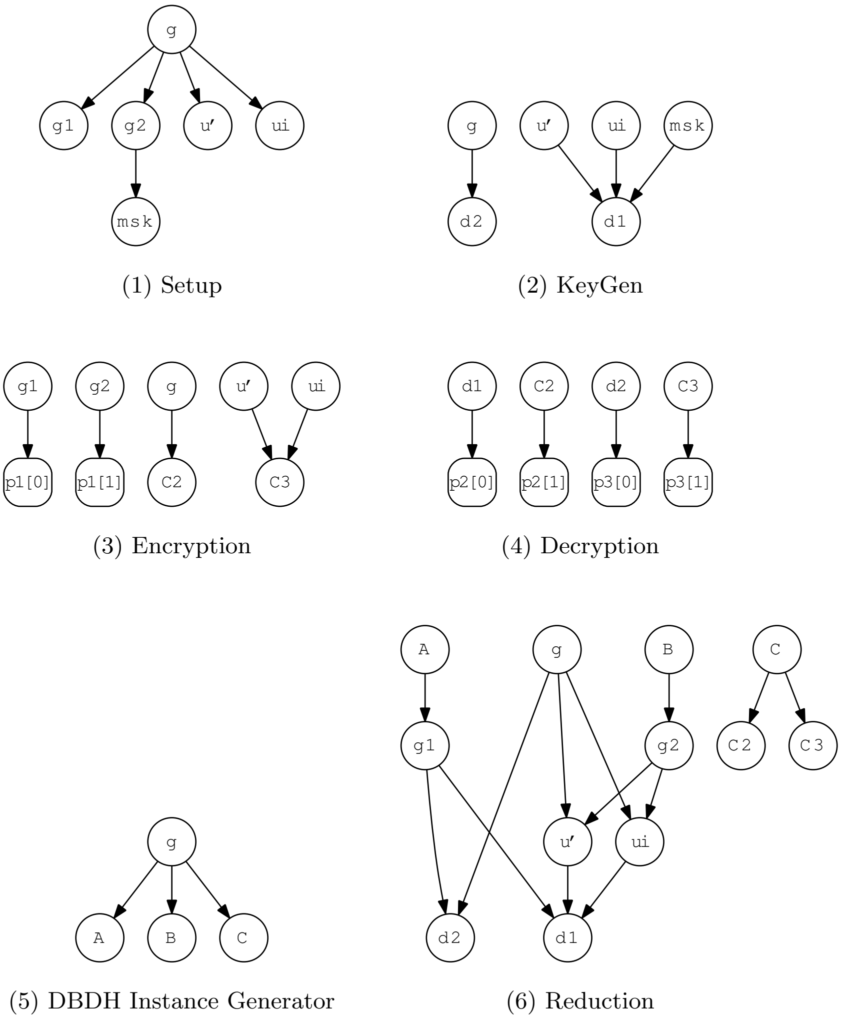
1. M. Abe, B. David, M. Kohlweiss, R. Nishimaki, and M. Ohkubo. Tagged one-time signatures: Tight security and optimal tag size. *PKC 2013*, *LNCS* 7778, pp. 312–331, 2013.

2. J. A. Akinyele, M. Green, and S. Hohenberger. Using SMT solvers to automate design tasks for encryption and signature schemes. *ACM CCS 2013*, pp. 399–410, 2013.
3. R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé. A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. IACR ePrint Archive, 2013/400, 2013.
4. M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. *ACM CCS 1993*, pp. 62–73, 1993.
5. D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *CRYPTO 2001*, *LNCS* 2139, pp. 213–229, 2001.
6. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *ASIACRYPT 2001*, *LNCS* 2248, pp. 514–532, 2001.
7. D. Boneh and H. Shacham. Group signatures with verifier-local revocation. *ACM CCS 2004*, pp. 168–177, 2004.
8. S. Chatterjee, D. Hankerson, E. Knapp, and A. Menezes. Comparing two pairing-based aggregate signature schemes. *DCC 2010*, 55(2-3):141–167, 2010.
9. S. Chatterjee and A. Menezes. On cryptographic protocols employing asymmetric pairings - the role of psi revisited. IACR ePrint Archive, 2009/480, 2009.
10. S. Chatterjee and A. Menezes. On cryptographic protocols employing asymmetric pairings - the role of revisited. *Discrete Applied Math.*, 159(13):1311–1322, 2011.
11. J. Chen, H. W. Lim, S. Ling, H. Wang, and H. Wee. Shorter IBE and signatures via asymmetric pairings. *Pairing 2012*, *LNCS* 7708, pp. 122–140, 2013.
12. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS'08/ETAPS'08*, pp. 337–340, 2008.
13. S. D. Galbraith, K. G. Paterson, and N. P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.
14. F. Göloglu, R. Granger, G. McGuire, and J. Zumbrägel. On the function field sieve and the impact of higher splitting probabilities: Application to discrete logarithms in $f_{2^{1971}}$. IACR ePrint Archive, 2013/074, 2013.
15. A. Joux. Faster index calculus for the medium prime case application to 1175-bit and 1425-bit finite fields. *EUROCRYPT 2013*, *LNCS* 7881, pp. 177–193, 2013.
16. A. Joux. A new index calculus algorithm with complexity l(1/4+o(1)) in very small characteristic. IACR ePrint Archive, 2013/095, 2013.
17. U. M. Maurer. Abstract models of computation in cryptography. *Cryptography and Coding, 10th IMA International Conference*, *LNCS* 3796, pp. 1–12, 2005.
18. A. Menezes. Asymmetric pairings. Invited Talk in ECC 2009, 2009. available from `http://math.ucalgary.ca/sites/ecc.math.ucalgary.ca/files/u5/Menezes_ECC2009.pdf`.
19. S. C. Ramanna, S. Chatterjee, and P. Sarkar. Variants of waters' dual-system primitives using asymmetric pairings. IACR ePrint Archive, 2012/024, 2012.
20. S. C. Ramanna, S. Chatterjee, and P. Sarkar. Variants of waters' dual system primitives using asymmetric pairings - (extended abstract). *PKC 2012*, *LNCS* 7293, pp. 298–315, 2012.
21. O. Reingold, L. Trevisan, and S. P. Vadhan. Notions of reducibility between cryptographic primitives. *TCC 2004*, *LNCS* 2951, pp. 1–20, 2004.
22. V. Shoup. Lower bounds for discrete logarithms and related problems. *EUROCRYPT 1997*, *LNCS* 1233, pp. 256–266, 1997.
23. N. P. Smart and F. Vercauteren. On computable isomorphisms in efficient asymmetric pairing-based systems. *Discrete Applied Mathematics*, 155(4):538–547, 2007.

24. B. Waters. Efficient identity-based encryption without random oracles. *EUROCRYPT 2005*, *LNCS* 3494, pp. 114–127, 2005.
25. B. Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. *CRYPTO 2009*, *LNCS* 5677, pp. 619–636, 2009.

Appendix

A Dependency Graphs for Waters' IBE



(1) Setup

(2) KeyGen

(3) Encryption

(4) Decryption

(5) DBDH Instance Generator

(6) Reduction

Fig. 2. Dependency graph for each algorithm in Waters' IBE scheme.

.

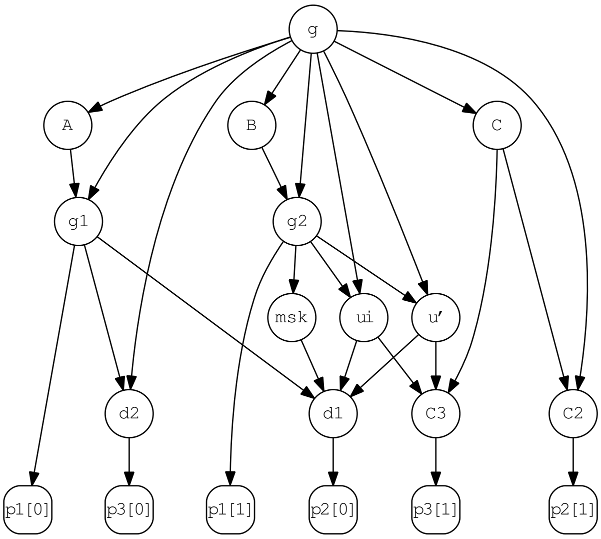
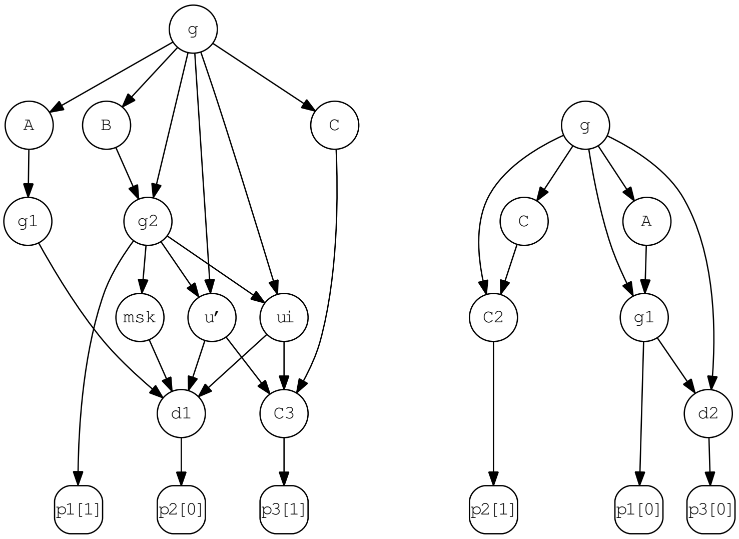


Fig. 3. Dependency graph for Waters' IBE scheme obtained by merging all graphs for individual algorithms.

.



(1) Dependency in $\mathbb{G}$ (2) Dependency in $\tilde{\mathbb{G}}$

Fig. 4. A valid split for minimum public key. Nodes in graph (1) and (2) represent group elements in $\mathbb{G}$ and $\tilde{\mathbb{G}}$, respectively.

.