Automating Fast and Secure Translations from Type-I to Type-III Pairing Schemes

Joseph A. Akinyele*
Johns Hopkins University
Baltimore, MD, USA
akinyelj@cs.jhu.edu

Christina Garman†
Johns Hopkins University
Baltimore, MD, USA
cgarman@cs.jhu.edu

Susan Hohenberger‡
Johns Hopkins University
Baltimore, MD, USA
susan@cs.jhu.edu

ABSTRACT

Pairing-based cryptography has exploded over the last decade, as this algebraic setting offers good functionality and efficiency. However, there is a huge security gap between how schemes are usually analyzed in the academic literature and how they are typically implemented. The issue at play is that there exist multiple types of pairings: Type-I called "symmetric" is typically how schemes are presented *and proven secure* in the literature, because it is simpler and the complexity assumptions can be weaker; however, Type-III called "asymmetric" is typically the most efficient choice for an implementation in terms of bandwidth and computation time.

There are two main complexities when moving from one pairing type to another. First, the change in algebraic setting invalidates the original security proof. Second, there are usually multiple (possibly thousands) of ways to translate from a Type-I to a Type-III scheme, and the "best" translation may depend on the application.

Our contribution is the design, development and evaluation of a new software tool, AutoGroup+, that automatically translates from Type-I to Type-III pairings. The output of AutoGroup+ is: (1) "secure" provided the input is "secure" and (2) optimal based on the user's efficiency constraints (excluding software and run-time errors). Prior automation work for pairings was either not guaranteed to be secure or only partially automated and impractically slow. This work addresses the pairing security gap by realizing a fast and secure translation tool.

*Supported by Zeutro, LLC and the Office of Naval Research under contract N00014-14-1-0333.

†Supported by the Office of Naval Research under contract N00014-14-1-0333.

‡Supported by the National Science Foundation CNS-1228443 and CNS-1414023; the Office of Naval Research under contract N00014-14-1-0333, and a Microsoft Faculty Fellowship.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813601.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Cryptographic controls and Verification

Keywords

Automation, Pairing-Based Cryptography, Cryptographic Compilers, Public-Key Encryption, Digital Signatures

1. INTRODUCTION

Automation is increasingly being explored as a means of assisting in the design or implementation of a cryptographic scheme. The benefits of using computer assistance include speed, accuracy, and cost.

Recently, automation for *pairing* (also called *bilinear*) cryptographic constructions (e.g., [2, 6, 7, 9]) has been under exploration. Since the seminal work of Boneh and Franklin [13], interest in pairings is strong: they have become a staple at top cryptography and security conferences, the open-source Charm library has been downloaded thousands of times worldwide and recently pairing-commercializer Voltage Security was acquired by a major US company (HP) [22].

Pairings are algebraic groups with special properties (see Section 2.1), which are often employed for their functionality and efficiency. There are different types of pairings: Type-I called "symmetric" is typically how schemes are presented *and proven secure* in the literature, because it is simpler and the complexity assumptions can be weaker; however, Type-III called "asymmetric" is typically the most efficient choice for an implementation in terms of bandwidth and computation time.

Unfortunately, translating a Type-I scheme into the Type-III scheme is complicated. First, there may be thousands of different Type-III translations of a Type-I scheme and the "best" translation may depend on the application. For instance, one translation might optimize ciphertext size while another offers the fastest decryption time. Second, each new translation requires a new proof under Type-III assumptions. Exploring and analyzing all possible translations is clearly a great burden on a human cryptographer. Indeed a small subset of manual translations of a scheme or particular set of schemes is regarded as a publishable result in its own right, e.g., [17, 18, 23].

Given this translation hurdle, common practice today is to analyze a Type-I scheme, but then use ad-hoc means to derive a Type-III translation that is unproven and possibly non-optimal. The goal of this work is to address this problem by covering new ground in cryptographic automation.

Our Contribution: The AutoGroup+ Tool. Our primary contribution is the design, development, and performance evaluation of a new publicly-available[1] tool, AutoGroup+, that automatically translates pairing schemes from Type-I to Type-III. The output of AutoGroup+ is: (1) "secure" provided the input is "secure" (see Section 3.2) and (2) optimal based on the user's efficiency constraints (see Section 3.1.5).[2] The input is a computer-readable format of the Type-I construction, metadata about its security analysis, and user-specified efficiency constraints. The output is a translated Type-III construction (in text, C++, Python, or LaTeX) with metadata about its security analysis. (See Figure 1.)

The audience for this tool is: (1) anyone wanting to implement a pairing construction, and (2) pairing construction designers. We highlight some features.

New Scheme Description Language (SDL) Database. The input to AutoGroup+ requires a computer-readable format of the Type-I construction, the Type-I complexity assumption(s), and the Type-I security proof. It was a challenge to create a means of translating human-written security proofs into SDL. We focused on a common type of proof exhibiting a certain type of black-box reduction.[3] We created a new SDL structure for representing assumptions and reductions of this type that may be of independent interest. Additionally, we did the tedious work of carefully transcribing five assumptions, eight reductions and improving the SDLs for nine popular constructions (the latter from [6]). (See the full version [4] for examples.) Once transcribed, however, these SDL files can be reused. We believe the future of cryptographic automation research will involve processing the assumptions and proofs; thus our database is made public as a testbed for future automation research.

Speed of Tool. AutoGroup+ took less than 21 seconds to process any of the test set, which included seven simple schemes (16 or less solutions), three medium schemes (256 to 512 solutions), and three complex schemes (1024 to 2048 solutions). (The preference for simple schemes was to compare with prior work.) This measures from SDL input to a C++ (or alternative) output. Speed is very important here for usage, because we anticipate that designers may iteratively use this tool like a compiler and implementors may want to try out many different efficiency optimizations.

In contrast, in CRYPTO 2014, Abe, Groth, Ohkubo and Tango [2] laid out an elegant theoretical framework for doing pairing translations in four steps. It left open the issue of whether their framework was practical to implement for a few reasons: (1) they automated only one of four steps (code not released), (2) their algorithm for this step was exponential time, and (3) they tested it on only simple and medium schemes, but their medium scheme took over 1.75 hours for *one* step. Our fully automated translation of that scheme took 6.5 seconds, which is much more in line with the "compiler"-like usage we anticipate.

We attribute our drastic efficiency improvement in part to our use of the Z3 SMT Solver. As described in Section 3, we encode the translation of the scheme, its assumption(s) and its reduction as a constraint-satisfaction problem and then use Z3 to quickly find the satisfying set of solutions.

New Results. We evaluated AutoGroup+ on 9 distinct constructions (plus 4 additional variations of one scheme), with various optimization priorities, for 48 bandwidth-optimizing translations. In Figure 4, we report the sizes compared to the symmetric case, which are significantly smaller. In Figure 5, we report on over 140 timing experiments resulting from the translations. Due both to the asymmetric setting and AutoGroup+'s optimizations, in most cases, the running times were reduced to less than 10% of the symmetric case. In the full version [4], we report on the effect that different levels of complexity have on translation time for a single scheme.

In Section 5, we compare the performance of AutoGroup+ to prior automation works, published manual translations, and translations existing as source code in the Advanced Crypto Software Collection [19] and Charm library [5]. We discovered a few things. In fourteen points of comparison with AutoGroup, AutoGroup+ matches those solutions and provides a security validation and new assumptions, adding only a few additional seconds of running time. In three points of comparison with Abe et al. [2] and subsequent personal communications [3], our translated results match.

In the five points of overlap with ACSC and Charm, we are able to confirm the security and ciphertext-size optimality of one broadcast encryption and one hierarchical identity-based encryption implementation. We are also able to confirm the security of two signature implementations, although only one is signature-size optimal. These confirmations are new results. Our tool was able to confirm the ciphertext-size optimality, but not the security of the Charm implementation of Dual System Encryption [26] (meaning it may not be secure). That implementation made changes to the keys outside the scope of the translations here or in [2, 6]. However, our tool did find a secure translation with the same ciphertext-size.

Overall, our tests show that the tool can produce high-quality solutions in just seconds, demonstrating that pairing translations can be practically and securely performed by computers.

1.1 Prior Work

The desirability of translating Type-I to Type-III pairings is well documented. First, this is an exercise that cryptographers are still actively doing by hand. In PKC 2012, Ramanna, Chatterjee and Sarkar [23] nicely translated the dual system encryption scheme of Waters [26] from the Type-I pairing setting to a number of different Type-III possibilities. Recently, Chen, Lim, Ling, Wang and Wee [17, 18] presented an elegant semi-general framework for (re-)constructing various IBE, Inner-Product Encryption and Key-Policy Functional Encryption schemes in the Type-III setting, assuming the SXDH assumption holds.[4] These works go into deeper creative detail (changing the scheme or adding assumptions) than our automator, and thus mainly get better results, but then, these works appear to have taken significant human resources. In contrast, our work offers a computerized translation as a starting point.

[1] AutoGroup+ can be downloaded at `https://github.com/jhuisi/auto-tools`.

[2] These claims regard the cryptographic transformation and exclude any software or run-time errors.

[3] The theoretical translation security results of [2] on which we will base our security are also limited to this class of proof.

[4] Informally, the SXDH assumption asserts that in a Type-III pairing group, there exist no efficient isomorphisms from $\mathbb{G}_1$ to $\mathbb{G}_2$ or from $\mathbb{G}_2$ to $\mathbb{G}_1$.

The Advanced Crypto Software Collection (ACSC) [19], including the Charm library [5], contains many Type-III implementations of schemes that were published and analyzed in the Type-I format. To the best of our knowledge, there is no formal analysis of these converted schemes and thus also no guarantees that the translations are secure or optimal efficiency-wise for a user's specific application. (We remark that ACSC/Charm makes no claims that they *are* secure or optimal.) The public Github records for Charm show that it has been downloaded thousands of times; thus, it would be prudent to verify these implementations. (See our results on this in Section 5.)

In ACM CCS 2013, Akinyele, Green and Hohenberger [6] presented a publicly-available tool called AutoGroup, which offered an automated translation from Type-I to Type-III pairing schemes. This work employed sophisticated tools, such as the Z3 Satisfiability Modulo Theories (SMT) solver produced by Microsoft Research (see Section 2), to quickly find a set of *possible* assignments of elements into $\mathbb{G}_1$ or $\mathbb{G}_2$. There was not, however, any guarantee that the resulting translation remained secure. Indeed, Akinyele et al. [6] explicitly framed their results as follows: translation has two parts: (1) the search for an efficient translation, and (2) a security analysis of it. They automated the first part and left the security analysis to a human cryptographer. Since they made their source code public, we used it as a starting point and thus named our work after theirs.

While using AutoGroup is certainly faster than a completely manual approach, the lack of a security guarantee is a real drawback. At that time, there was simply no established theory on how to generalize these translations.

Fortunately, in CRYPTO 2014, Abe, Groth, Ohkubo and Tango [2] pushed the theory forward in this area. They elegantly formalized the notion that if certain dependencies from the Type-I complexity assumption(s) and the reduction in the security analysis were added to the dependencies imposed by the scheme itself, then there was a generic way to reason about the security of the translated scheme. Their main theorem, which we will later use, can informally be stated as:

Theorem 1.1 (Informal [2])**.** *Following the conversion method of [2], if the Type-I scheme is correct and secure in the generic Type-I group model, then its converted Type-III scheme is correct and secure in the generic Type-III group model.*

There are four steps in their translation: (1) build a dependency graph between the group elements for each algorithm in the construction, the complexity assumption(s) and the security reduction (In the graph, elements are nodes and a directed edge goes from g to h if h is derived from g, such as $h = g^x$.), (2) merge all graphs into a single graph, (3) split this graph into two graphs (where elements of the first graph will be assigned to $\mathbb{G}_1$ and elements of the second assigned to $\mathbb{G}_2$), and (4) derive the converted scheme.

For the four schemes tested in [2], steps (1), (2), and (4) were done by hand. The algorithm for step (3) was *exponential* in two variables[5] and the Java program to handle step (3) reported taking 1.75 hours on a medium scheme. Thus, this is a great theory advance, but it left open the question of whether the entire translation could be efficiently automated as a "real-time" tool.

[5]Their splitting algorithm runs exponentially in both the number of pairings and the bottom nodes (without outgoing edges) of the dependency graph. Thus, scalability is a real concern.

AutoGroup+ **in a Nutshell**. In short, prior work admitted a public tool that is fast, but possibly insecure [6], and a cryptographic framework that is slow, but secure [2]. Our goal was to realize the best of both worlds. Even though the implementations differed, we discovered that both works began by tracing generator to pairing dependencies, where [6] did this bottom up and [2] used a top down approach. Since both of these representations can be helpful for different optimizations, AutoGroup+ does both. It also traces these dependencies for the complexity assumptions and reductions. The pairings and hash variables in the combined dependency graph are translated into a formula and constraints, and then fed into a SMT solver. The output set is then efficiently searched for an optimal solution using the SMT solver again, then verified as a valid graph split (as formalized in [2]). Finally, if the split is valid, then a converted scheme and complexity assumption(s) are output.

2. BACKGROUND

2.1 Pairings

Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be groups of prime order p. A map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an admissible *pairing* (also called a *bilinear map*) if it satisfies the following three properties:

1. Bilinearity: for all $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$, and $a, b \in \mathbb{Z}_p$, it holds that $e(g^a, h^b) = e(g^b, h^a) = e(g, h)^{ab}$.
2. Non-degeneracy: if g and h are generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, resp., then $e(g, h)$ is a generator of $\mathbb{G}_T$.
3. Efficiency: there exists an efficient method that given any $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$, computes $e(g, h)$.

A pairing generator is an algorithm that on input a security parameter 1^λ, outputs the parameters for a pairing group $(p, g, h, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ such that p is a prime in $\Theta(2^\lambda)$, $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are groups of order p where g generates $\mathbb{G}_1$, h generates $\mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an admissible pairing.

The above pairing is called an *asymmetric* or Type-III pairing. This type of pairing is generally preferred in implementations for its efficiency. We also consider *symmetric* or Type-I pairings, where there is an efficient isomorphism $\psi : \mathbb{G}_1 \rightarrow \mathbb{G}_2$ (and vice versa) such that a symmetric map is defined as $e : \mathbb{G}_1 \times \psi(\mathbb{G}_1) \rightarrow \mathbb{G}_T$. We generally treat $\mathbb{G} = \mathbb{G}_1 = \mathbb{G}_2$ for simplicity and write $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$. These types of pairings are typically preferred for presenting constructions in the academic literature for two reasons. First, they are simpler from a presentation perspective, requiring fewer subscripts and other notations. More importantly, they are sometimes preferred because the underlying symmetric assumption on which the proof is based may be viewed as simpler or weaker than the corresponding asymmetric assumption.

We include current efficiency numbers for Type-I and Type-III groups in the full version [4] and Section 5, demonstrating the significant advantages of the latter.

2.2 Z3 Theorem Prover

Our implementation also relies on the power of the state-of-the-art Z3 SMT solver [20] developed at Microsoft Research. SMT is a generalization of boolean satisfiability (or

SAT) solving where the goal is to decide whether solutions exist to a given logical formula. The publicly available Z3 is one such tool that is highly efficient in solving constraint satisfaction problems and used in many different applications.

2.3 The SDL Toolchain

This work builds on the efforts of prior automation works [6, 7] which include several tools such as a scheme description language (or SDL), an accompanying parser for SDL, a code generator that translates SDL schemes into executable code in either C++ or Python, and a LaTeX generator for SDL descriptions. We obtained all these prior tools from the publicly-available AutoTools GitHub repository.[6] Our code and SDL database will be made public in this repository as well. The SDL for the constructions are the same in AutoGroup and AutoGroup+; the difference is that the latter also includes SDL for assumptions and security reductions. Since we used the code of AutoGroup as a starting point, we derived our tool name from it.

3. THE AUTOGROUP+ SYSTEM

As described in Section 1, AutoGroup+ is a new tool built to realize the best of both worlds from a prior tool called AutoGroup [6] (fast, but no security guarantees) and new theoretical insights [2] (secure, but exponential time and no public tool.)

3.1 How It Works

We begin with an illustration of the AutoGroup+ system in Figure 1. This system takes in the description of a symmetric (Type-I) pairing-based scheme S, together with metadata about its security and user-desired efficiency constraints, and outputs an asymmetric (Type-III) pairing-based translation S', together with metadata about its security. Informally, if S was secure, then S' will be both secure and optimal for the constraints set by the user over the space of "basic" translations.

3.1.1 Step 1: Generating Computer-Readable Inputs

AutoGroup+ operates on four inputs: an abstract description of the (1) scheme itself, (2) the complexity assumption(s) on which the scheme is based, (3) the black-box reduction in the scheme's proof of security, and (4) a set of efficiency optimization constraints specified by the user (e.g., optimize for smallest key or ciphertext size.). The abstract descriptions are all specified in a Scheme Description Language (SDL) [6, 7].

The need for SDL representations of the complexity assumptions and security reductions are new challenges for this work. To run our Section 5 tests, we had to translate the text in the published papers to the SDL format by hand. This was a time-consuming and tedious task. However, we maximize the benefit of doing this, by making these SDL files publicly available. This enables anyone to check their correctness and provides a ready-made base of test files for any future automation exercises that require this deeper scheme analysis.

One novel and curious observation we made during these experiments was that *how* group elements were derived in the symmetric group impacted the dependency graphs and therefore the asymmetric results. To say this another way, two schemes computing the exact same elements, but in different ways, could have different dependency graphs and therefore different asymmetric translations. As a toy example, suppose a scheme has $\text{PK} = (g, A = g^a, B = g^b)$ and $\text{SK} = (\text{PK}, a, b)$. Now suppose that as part of a signing algorithm, the holder of SK must compute the value $C = g^{ab}$. Suppose in Scheme 1, the signer computes this as $x = ab \bmod p$ and $C = g^x$. Suppose in Scheme 2, the signer computes this as $C = A^b$. Then in the dependency graph for Scheme 1, there is a root node g, with nodes A and C hanging off it. Whereas for the graph of Scheme 2, there is a root node g with A off it, and C off of A. The importance of these differences comes alive when we attempt to split the graph (see Step 3.1.4). Suppose there is the pairing $e(A, C)$. Then in Scheme 1, the generator g must be split, but A can be assigned to $\mathbb{G}_1$ and C to $\mathbb{G}_2$, resulting in a 4 element public key. However, in Scheme 2, the generator g *and* the element A must be split, with $A_1 \in \mathbb{G}_1$ and $A_2 \in \mathbb{G}_2$, so that one can compute $C = (A_2)^b \in \mathbb{G}_2$. This results in a 5 element public key. The general rule is that the fewer unnecessary dependencies the better. Interestingly, Abe et al. [2] sometimes *added* dependencies that did not exist in the original schemes. For instance, for the Waters 2005 IBE [25], Waters clearly states to choose g_2, u', u_i as fresh random generators, but Abe et al. explicitly "assume" that they are generated from a separate generator g. For this particular scheme, this does not impact the asymmetric translations, but in theory it could.

Our experiments did not add any dependencies. We note that in this step, a human is not being tasked with any job but simple transcription of the input into a language the computer can understand.

System Limitations and Allowable Inputs.

This system shares some of the same limitations as prior works [2, 6]. First, this is a junk-in-makes-junk-out system. AutoGroup+ assumes that the security reduction is correct, the complexity assumptions are true, and that the SDL was typed in correctly. If any of these turn out to be false, the output cannot be depended on. Fortunately, we can mitigate these risks as follows. The correctness of the security reductions might be verified automatically using a number of tools, such as EasyCrypt [24], but this likely requires further research. The pairing-based assumptions may be sanity-checked in the generic group model using the recently developed tool by Barthe et al. [9] from CRYPTO 2014. Finally, the SDL transcriptions can be verified in the usual crowd-based manner which we encourage by making them public.

Second, the system does not accept all possible schemes that might appear in the literature. AutoGroup+ supports only prime-order symmetric pairing schemes with a "standard" reduction analysis[7]. It can support most non-interactive assumptions. It can also support dynamic (also called q-based) assumptions, where the size of the assumption may grow depending on the usage of the scheme. It can also sup-

[6]Project link: `https://github.com/jhuisi/auto-tools`

[7]We refer the reader to Abe et al. [2] for a formal definition of the allowed reductions. Roughly, we mean an analysis where there is an efficient algorithm called a *reduction* that is successful in solving the hard problem (underlying the complexity assumption) given black-box access to an adversary that successfully attacks the scheme.

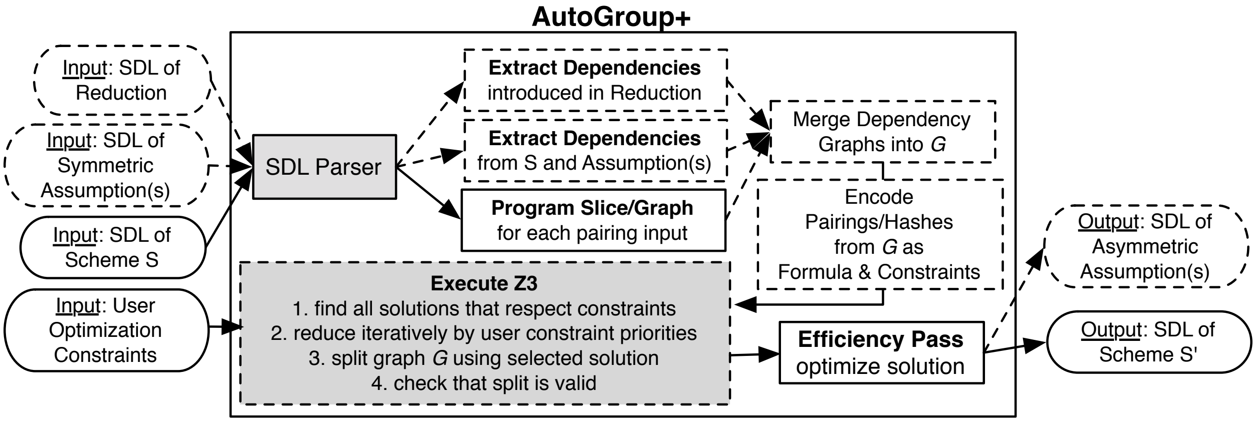


Figure 1: A high-level presentation of the AutoGroup+ tool. Components that are new or improved, over AutoGroup, are included with dashed lines. Both AutoGroup+ and AutoGroup use external tools Z3, SDL Parser and Code generator (omitted from the figure).

port interactive (also called oracle-based) assumptions such as the LRSW assumption behind the popular Camenisch-Lysyanskaya [16] pairing-based signatures.

Third, how the scheme hashes into pairing groups also may disqualify it from being translated. We now give an example of how to alter the Setup algorithm of the Waters 2005 IBE scheme [25], so that AutoGroup+ cannot translate it. (Indeed, it is not clear to us if a translation even exists.) In the original Setup algorithm, the master authority chooses a generator $g \in \mathbb{G}$ at random. Then public parameter g_1 is derived from g, while parameters $g_2, u_0, \ldots, u_n \in \mathbb{G}$ are chosen independently at random. Instead, suppose we treat the hash function $H : \{0,1\}^* \rightarrow \mathbb{G}$ as a random oracle. Let generator $g \in \mathbb{G}$ be computed as $g = H(ID)$, where ID is some string describing the master authority. Then g_1 is derived from g as before, but we set $g_2 = g^r, u_0 = g^{r_0}, \ldots, u_n = g^{r_n}$ for random $r, r_0, \ldots, r_n \in \mathbb{Z}_p$ (where p is the order of $\mathbb{G}$). It is easy to see that the public parameters have the same distribution as before (assuming the random oracle model); all we have changed is *how* the master authority samples these parameters. Thus, this variant of the Waters IBE remains secure in the symmetric setting, and yet it is not clear how to translate it to the asymmetric setting. We return to this example in Section 5.

These limitations also appear in the theoretical work of Abe et al. [2], and fortunately, these issues seem relatively rare and did not come up for any of the schemes we tested (except our hand-made counterexample). As in [2, 6], we note that if AutoGroup+ cannot produce a translation, it does not imply that a translation does not exist. A characterization of untranslatable schemes is an open problem.

3.1.2 Step 2: Extracting Algebraic Dependencies

Once AutoGroup+ has parsed all its input files, it begins processing them to graph the algebraic dependencies between source group elements in a scheme, assumption and reduction. All source group elements are nodes in the graph and a directed edge exists if there is a direct dependency between two elements. E.g., if $h = g^x$, then h is derived from g and we place an edge from g and h.

AutoGroup+ extracts the dependency graphs *automatically from the SDL* for each input file and builds a distinct graph from the SDL representations and metadata. AutoGroup+ defines two new procedures that programmatically extract the dependency graph for the assumption(s) as well as the reduction(s) (see Section 4 for an example). Then, AutoGroup+ reuses logic from AutoGroup to programmatically build the graph of the scheme by tracking the generators in the setup algorithm and by tracing backward from each pairing in the scheme. It merges the program slice (or trace) extracted for each pairing input into one dependency graph for the scheme. The resulting graphs are the same as those produced by Abe et al. [2] (except where we reduced dependencies by computing elements more directly as discussed in the last step.)

The work of Abe et al. [2] required a human to build (and later merge) these dependency graphs and the graphs were constructed starting from the common generators downward. The AutoGroup work of Akinyele et al. [6] automatically derived these graphs *for the scheme only* from the SDL description of the scheme. They did not consider the assumptions or reduction dependencies. Indeed, AutoGroup only graphed the dependencies as a traceback from the pairings, whereas AutoGroup+ also adds a top-down analysis from the assumption down to the pairings for the security logic.

3.1.3 Step 3: Merge Dependency Graphs

After extracting the dependencies, AutoGroup+ has a set of distinct graphs: one graph that represents dependencies from the setup, key generation, encryption/signature and decryption/verification algorithms, as well as a graph for each complexity assumption and one or more graphs for the reduction. These graphs are then systematically merged together using the metadata provided with the SDL inputs. The metadata includes a reduction map which relates the names of source group elements in the reduction to those in the assumption. We require this map to understand which nodes represent the same group element (across the scheme, assumption and reduction) to simplify merging into a single node. See the example in Section 4. AutoGroup+ programmatically checks the type information in the reduction map across all SDL inputs to ensure correctness during the merge.

3.1.4 Step 4: Assign Variables using the SMT Solver

This is the most complex step in the automation. In the symmetric setting, all group elements in the scheme were in $\mathbb{G}$. To move to the asymmetric setting, we must assign elements to either $\mathbb{G}_1$ or $\mathbb{G}_2$ in such a way that the dependencies between elements are not violated (*e.g.*, if $h = g^x$, then both g, h must be in the same group) and such that for all variables a, b, if we have a pairing between them $e(a, b)$, then a and b must be in distinct source groups (*e.g.*, $a \in \mathbb{G}_1$ and $b \in \mathbb{G}_2$ or vice versa). Such an assignment may not be feasible (see such an example in Section 3.1.1) or it may require that one or more variables in the symmetric scheme be duplicated in the asymmetric scheme with one assigned to $\mathbb{G}_1$ and another to $\mathbb{G}_2$. E.g., in the symmetric setting if $g \in \mathbb{G}$, $a = g^x$ and $b = g^y$ and these elements are paired as $e(a, b)$, then in the asymmetric setting, g will be split into $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, where $a = g_1^x$ and $b = g_2^y$, so that one can compute $e(a, b)$.

To *efficiently* make these variable assignments, AutoGroup+ follows the approach of AutoGroup in that it uses a powerful Z3 Satisfiability Modulo Theories (SMT) solver produced by Microsoft Research (see Section 2) to compute the set of all possible splits (*i.e.*, all possible variable assignment combinations) and then later identifies the best one. Z3 takes as input a logical formula and determines whether valid variable assignments exist that evaluate that formula to *true*. Similar to AutoGroup, AutoGroup+ expresses the pairing equations as a logical formula of conjunctions and inequality operations over binary variables. For example, $e(a, b) \cdot e(c, d)$ is translated to the logical formula $P1[0] \neq P1[1] \wedge P2[0] \neq P2[1]$ where $P1[0]$ is a reference to a, $P1[1]$ to b, and so on. AutoGroup+ simply follows the pairing identifier convention established by Abe et al. [2].

One major difference between AutoGroup+ and AutoGroup is that the former's dependency graphs include dependencies based on the assumptions and reductions. The formula is derived from the pairings that occur in the graph (from the construction, reduction and assumption(s)) with a conjunction joining each pairing piece, plus extra constraints added for variables that cannot be duplicated (regarding hashing). This formula is then fed into the solver. The solver returns a set of 0 or 1 assignments for each variable. We then apply each solution to the merged dependency graph to generate the split (variables assigned to 0 on one side and the rest on the other).

3.1.5 Step 5: Search for Optimal Solution

There are often many (possibly thousands) of ways to translate a symmetric scheme into an asymmetric one; thus, we can end up with many feasible graph splits. Indeed, the output of the SMT solver in the last step is a *set* of assignments of the variables. In this step, we again use the SMT solver to deduce which assignment from this set is "best". AutoGroup+ allows selection of assignments based on a number of user-specified optimization constraints. For public-key encryption, the user can choose to minimize the public-key, assumption, secret key and/or ciphertext size. Similarly for signature schemes, the user can mimize the public-key parameters, assumption, and/or the signature size.

To select an optimal assignment, AutoGroup+ encodes these user requirements as parameters of some *objective function.* We then call the solver a second time with this objective function set to rank/narrow the given solutions to one. Depending on the optimization goal, the objective function can be specified in one of two ways. If reducing public-key size or the assumption, then we are concerned with minimizing the duplication of source group elements. As such, we first specify an EvalGraph function that the solver uses to compute the splits for each element in the public key or assumption: $\mathsf{EvalGraph}(A_j, B, G) = S$, where $A_j = a_1, \ldots, a_n$ represents pairing input variable assignments for the j-th solution (each a_i variable is either $0 = \mathbb{G}_1$ or $1 = \mathbb{G}_2$), $B = b_1, \ldots, b_m$ represents the source group elements to minimize either in the assumption or public-key, and G represents the merged dependency graph.

Our search algorithm first applies the EvalGraph function to determine how the b_i values are assigned for each solution. Once the b_i values are assigned, we then compute $S = s_1, \ldots, s_m$ where each s_i corresponds to one of three values for each b_i assignment. That is, let a w_1 value denote a $\mathbb{G}_1$ only assignment, w_2 is $\mathbb{G}_2$ only, and $w_3 = w_1 + w_2$ is both a $\mathbb{G}_1$ and $\mathbb{G}_2$ assignment (or simply a split). We then set w_1 and w_2 to the group size of $\mathbb{G}_1$ and $\mathbb{G}_2$ for Type-III pairing curves (*e.g.*, BN256). Each solution is ranked in terms of splits and the total size of group elements in B. Our search returns the j-th solution that results in the fewest splits in B with the smallest overall size S_j. This overall size breaks ties between multiple solutions with the same number of splits.

$$\min_{j \in |A|} (\mathsf{CountSplits}(S_j), \sum_{i=1}^{m} S_{j,i}) \tag{1}$$

For the other optimization options (*i.e.*, secret-key, ciphertext, etc), we can reuse the objective function specified by AutoGroup as is:

$$\min_{j \in |A|} F(A_j, C, w_1, w_2) = \sum_{i=1}^{n} ((1 - a_i) \cdot w_1 + a_i \cdot w_2) \cdot c_i \tag{2}$$

where the A_j represents the j-th solution as before, $C = \{c_1, \ldots, c_n\}$ represent some *cost* associated with each a_i variable reference, and w_1 and w_2 correspond to *weights* (for different Type-III pairing curves) over groups $\mathbb{G}_1$ and $\mathbb{G}_2$. By encoding these cost values, it is feasible to create different weight functions that adhere to the user specified constraints. Once these functions are specified correctly, we minimize it across the set of assignments and return the solution that yields the lowest value. Thus, the combination of equations 1 and 2 yield all the possible ways a current user can optimize a given symmetric scheme. Further optimizations are future work.

Once the "best" solution is found, we have a CheckValidSplit procedure that verifies that the conditions (1) and (2) of a "valid split" hold as defined in Definition 3.1. If this solution satisfies these conditions, we are done. If not, we simply test the next best solution, because the solver caches all solutions and we record metadata about each solution in terms of efficiency and security.

3.1.6 Step 6: Evaluate and Process the Solution

Once a split is chosen, AutoGroup+ must reconstruct SDL for the asymmetric scheme *and assumption(s).* It reuses the functionality provided by AutoGroup to construct the SDL as dictated by the split.[8] To output the new asymmetric assumptions, AutoGroup+ follows the logic of Abe et al. [2] (although they did not implement this step) and implements a new procedure that uses the graph split to reconstruct the asymmetric assumption(s). For each element in the asymmetric assumption, we learn the new assignments of the elements using the graph split and mechanically generate the asymmetric assumption SDL. Finally, we rely on existing tools [6, 7] to translate the new asymmetric SDL representation into executable code for C++ or Python, or LaTeX.

3.2 Analysis of AutoGroup+

We analyze AutoGroup+'s security and optimizations.

Security. At a high-level, the Abe et al. [2] security argument works as follows. In the Type-I setting, we treat $\mathbb{G}_1 = \mathbb{G}_2$ because there are efficient isomorphisms between these two groups. However, suppose we work in the generic Type-I group model, where elements are a black box and to compute this isomorphism, a party must utilize an oracle $\mathcal{O}$. Next, consider moving to a Type-III group, where every element (for which the discrete logarithm is known with respect to the base generators) is duplicated; that is, for $h = g^x \in \mathbb{G}$, we have $h_1 = g_1^x \in \mathbb{G}_1$ and $h_2 = g_2^x \in \mathbb{G}_2$. Then in the generic Type-III group model, we can simulate having efficiently computable isomorphisms between these groups by exposing an oracle $\mathcal{O}'$ that on input $d_1 \in \mathbb{G}_1$ outputs $d_2 \in \mathbb{G}_2$ (or vice versa). In essence, by exposing the "corresponding" group element (through the oracle in the Type-III setting), we "allow" all necessary isomorphism computations for the scheme itself to operate, however, at the same time, we can argue that any adversary that breaks this scheme (with these elements exposed) can be turned into an attacker against the Type-I scheme, where these isomorphisms are natively computable. The resulting theorem was summarized in Theorem 1.1: namely, the Type-III conversion will be secure in the generic group model, if one follows the conversion criteria in [2] and the Type-I input was secure in the generic group model.

Thus, we must argue that the AutoGroup+ implementation satisfies the criteria in [2]. The dependency graphs are created and merged according to the same algorithm. (AutoGroup+ tracks some additional information on the side for optimization purposes.) What is required is that the splitting of the merged dependency graph satisfies Abe et al.'s notion of a "valid split."

Definition 3.1 (Valid Split [2])**.** *Let $\Gamma = (V, E)$ be a dependency graph for $\Pi = (\mathsf{S}, \mathsf{R}, \mathsf{A})$, a tuple representing a scheme, reduction and assumption(s) that are in the set covered by the [2] translation. Let $P = (p_1[0], \ldots, p_n[1]) \subset V$ 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* NoDup $\subseteq V$ *if the following hold:*

1. *merging Γ_0 and Γ_1 recovers Γ,*
2. *for each $i \in \{0, 1\}$ and every $X \in V_i \backslash P$, the ancestor subgraph of X in Γ is included in Γ_i.*
3. *for each $i \in \{1, \ldots, n_p\}$ pairing 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. NoDup *is a list of nodes that cannot be assigned to both V_0 and V_1.*

[8]We further perform an efficiency check on the final scheme as previously done in AutoGroup.

In terms of AutoGroup+ security, conditions (1) and (2) are satisfied in the search procedure (step 5). That is, before we admit a split, we do these simple tests. Condition (3) is satisfied by the SMT solver with the logical formula encoding of pairing nodes (step 4). Condition (4) is also satisfied by the SMT solver (step 4). We encode the output of hashes as constraints over the logical formula; specifically, we ask the solver to find splits that keep hashes in $\mathbb{G}_1$. This is the only place we differ slightly. Abe et al. allow $\mathbb{G}_1$ or $\mathbb{G}_2$ assignment for hashes but not both. Our approach prioritizes solutions that preserve efficiency but we could give the user the option of relaxing this to match Abe et al. The translation back to SDL is fairly straightforward from the split.

Optimizations. In terms of optimality over the set of solutions admitted by the "valid split" method, AutoGroup+ finds the "best" one by searching over the entire set. It does this efficiently by turning the user-specified optimizations into the appropriate objective function and passing this function into the SMT solver. Our experiments in Section 5 provide evidence that the tool is, indeed, finding the optimal solutions over the space of valid translations.

As discussed in Section 1.1, we do not rule out the existence of even better solutions that employ insights outside of this method (such as altering the construction or adding "stronger" assumptions, such as SXDH.)

4. AN EXAMPLE WITH BB-HIBE

In this section, we illustrate each phase of the AutoGroup+ implementation described in Section 3 by showing the step-by-step translation of the Boneh-Boyen hierarchical identity-based encryption [10] (or BB HIBE) scheme. We begin by recalling the scheme: an efficient HIBE scheme (with $\ell = 2$) [11, §4.1] that is selective identity secure based on the standard Decisional Bilinear-Diffie Hellman (DBDH) assumption.

This scheme consists of four algorithms: Setup, KeyGen, Encrypt and Decrypt. The Setup algorithm takes as input a security parameter and defines public keys (ID) of depth ℓ as vectors of elements in $\mathbb{Z}_p^\ell$. We define $\ell = 2$, thus the identity is comprised of $\mathsf{ID} = (ID_1, ID_2) \in \mathbb{Z}_p^2$. The algorithm generates system parameters as follows. First, select a random generator $g \in \mathbb{G}$, a random $\alpha \in \mathbb{Z}_p$, and sets $g_1 = g^\alpha$. Then, pick random $h_1, h_2, g_2 \in \mathbb{G}$. Set the master public parameters $params = (g, g_1, g_2, h_1, h_2)$ and the master secret key $msk = {g_2}^\alpha$.
The KeyGen algorithm takes as input an $\mathsf{ID} = (ID_1, ID_2) \in {\mathbb{Z}_p}^2$, picks random $r_1, r_2 \in Z_p$ and outputs:

$$d_1 = {g_2}^\alpha \cdot ({g_1}^{ID_1} \cdot h_1)^{r_1} \cdot ({g_1}^{ID_2} \cdot h_2)^{r_2}, d_2 = g^{r_1}, d_3 = g^{r_2}$$

and the algorithm outputs $d_{ID} = (d_1, d_2, d_3)$
The Encrypt algorithm takes as input the public parameters $params$, an identity ID and a message $M \in \mathbb{G}_T$. To encrypt the message M under the public key $\mathsf{ID} = (ID_1, ID_2)$, picks a random $s \in \mathbb{Z}_p$ and computes:

$$C = (e(g_1, g_2)^s \cdot M, g^s, ({g_1}^{ID_1} \cdot h_1)^s, ({g_1}^{ID_2} \cdot h_2)^s)$$

and the algorithm outputs $C = (C_1, C_2, C_3, C_4)$.

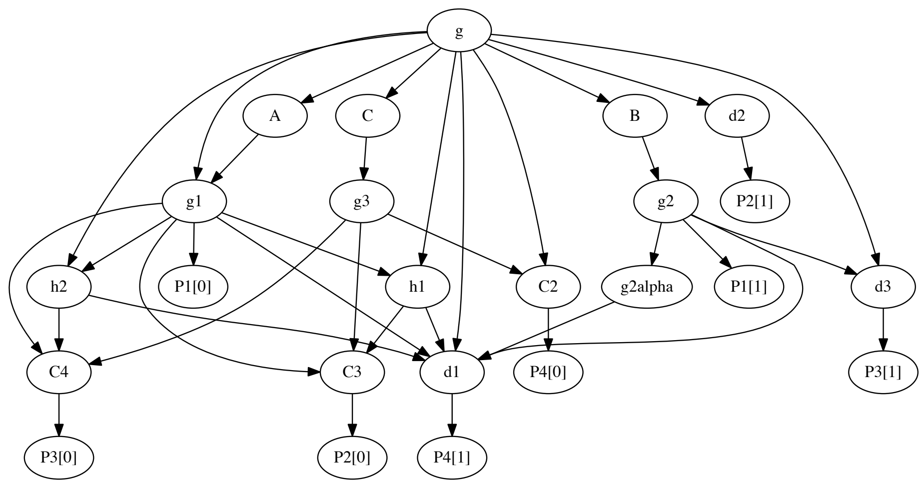


Figure 2: The merged dependency graph for the assumption, reduction to DBDH, and the BB HIBE scheme. This graph was generated by AutoGroup+.

The Decrypt algorithm takes as input a private key $d_{ID} = (d_1, d_2, d_3)$ and a ciphertext C and computes M as:

$$M = C_1 \cdot \frac{e(C_3, d_2) \cdot e(C_4, d_3)}{e(C_2, d_1)}$$

The scheme is based on the DBDH assumption.

Assumption 1 (Decisional Bilinear Diffie-Hellman)**.** *Let g generate group $\mathbb{G}$ of prime order $p \in \Theta(2^\lambda)$ with mapping $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$. For all p.p.t. adversaries $\mathcal{A}$, the following probability is negligible in λ:*

$$|\tfrac{1}{2} - \Pr[a, b, c \leftarrow \mathbb{Z}_p, z \leftarrow \{0, 1\}, A = g^a,$$
$$B = g^b, C = g^c, T_0 = e(g, g)^{abc}, T_1 \leftarrow \mathbb{G}_T;$$
$$z' \leftarrow \mathcal{A}(g, A, B, C, T_z) : z = z']|.$$

Step 1: Generating SDL Inputs.

In order for AutoGroup+ to perform the translation, we first begin by transcribing the scheme, reduction and the DBDH assumption into SDL. We provide the SDL description of the above scheme, reduction and assumption in [4]. The SDL descriptions closely and concisely follow the paper counterpart. This design is on purpose as to reduce the burden of transcribing these constructions for AutoGroup+ users. Indeed, in our experience the most time consuming and tedious part is in specifying the reductions accurately.

Step 2: Extracting the Dependencies.

Once the SDLs have been generated along with the metadata and the user's desired optimization goal, the user can proceed with executing AutoGroup+ to begin deriving the dependency graphs for each input file. AutoGroup+ programmatically extracts the dependencies from the SDL descriptions starting with the assumption(s), then the reduction(s) and finally, the scheme. The dependency graph diagrams for BB HIBE [11, §4.1] are included in the full version [4]. Note that these diagrams were generated automatically by our tool; we believe this feature provides more transparency to make it easier for humans to verify that the software is operating correctly. In "naming" the nodes of our dependency graphs, we closely follow the naming conventions that the user employed in the SDL, thus supporting the quick and easy verification.

Step 3: Merge the Graphs.

In Figure 2, we show the third step in AutoGroup+ which is to merge the multiple dependency graphs (assumption, reduction and scheme graphs) into one single graph.

Step 4: Assignment of Variables.

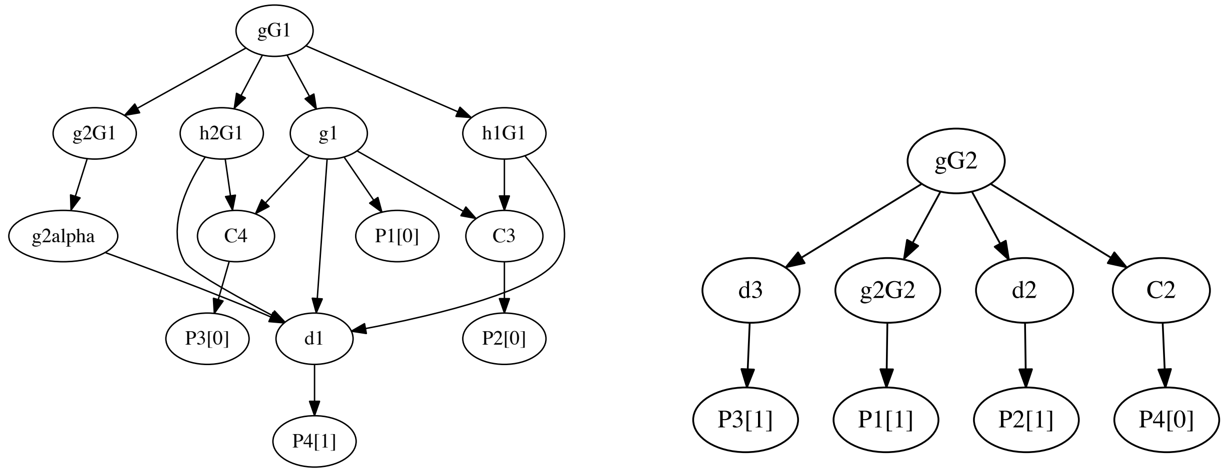
With the merged graph, we encode the pairing equations as a logical formula as in AutoGroup but also encode certain group elements in the dependency graph as additional constraints to the solver (with optimization requirements):

$$P1[0] \neq P1[1] \wedge P2[0] \neq P2[1] \wedge$$
$$P3[0] \neq P3[1] \wedge P4[0] \neq P4[1]$$

Recall that pairing identifiers (*e.g.*, $P2[0]$, $P2[1]$) are unique references which refer to pairing inputs from the scheme (*e.g.*, $e(C_3, d_2)$).

Step 5: Search for an Optimal Solution.

In our BB HIBE example, the goal is to minimize the number of splits in the master public parameters $params$, so this requires specifying the following parameters of the EvalGraph function. Let $B = \{g, g_1, g_2, h_1, h_2\}$ be the set of elements in the public parameters we wish to minimize and let G be an encoding of the merged dependency graph shown in Figure 2. As reflected in Table 4, the solver identifies 16 possible solutions for the BB HIBE scheme and computes the following on each solution as $S_j = \mathsf{EvalGraph}(A_j, B, G)$ where A_j is the j-th set of possible variable assignments. Recall that EvalGraph simply applies a given solution to G and records how elements of B are assigned. From the set S, the solver finds an assignment that has the fewest number



(a) Showing $\mathbb{G}_1$ elements in the scheme

(b) Showing $\mathbb{G}_2$ elements in the scheme

Figure 3: The dependency graphs for the asymmetric translation of BB HIBE scheme only (with PK optimization).

of duplicated public key elements with the smallest overall size. Based on this criteria, the solver returned a optimal solution in the fifth step which consisted of 2 splits (i.e., two duplicated elements). The new public key elements are assigned as $B' = \{g, \tilde{g}, g_1, g_2, \tilde{g_2}, h_1, h_2\} \in \mathbb{G}_1^5 \times \mathbb{G}_2^2$. This constitutes only an addition of 2 group elements in $\mathbb{G}_2$.

Step 6: Assignment of Variables.
In the last step, AutoGroup+ splits the graph as dictated by the optimal solution found by the solver. The resulting graphs for $\mathbb{G}_1$ and $\mathbb{G}_2$ assignments for the BB HIBE scheme are shown in Figure 3. AutoGroup+ programmatically converts the split graph into an asymmetric translation for the scheme and assumption. We improve on code from AutoGroup to do the former translation and write a new module to do the latter. See the full version for the graph split of co-DBDH and resulting SDL files [4]. As mentioned before, there is a publicly-available tool (see Section 2.3) for automatically turning this SDL into C++, Python or LaTeX.

5. EXPERIMENTAL EVALUATION

We tested AutoGroup+ on 9 schemes, with 3-4 optimization options and 4 different levels of BB HIBE, for 48 total translations.[9] Figure 4 summarizes the translation times and resulting scheme sizes.[10] To demonstrate the improvement in running times due to both the asymmetric setting and AutoGroup+'s optimizations, Figure 5 includes over 140 timing experiments, showing drastic improvements. In the full version [4], we summarize the effect of scheme complexity on AutoGroup+ conversion time by varying the complexity of BB HIBE. We note that even given a more complex scheme than attempted by any other tool, AutoGroup+ still provides fast conversion times.

System Configuration. All of our benchmarks were executed on a standard workstation that has a 2.20GHz quad-core Intel Core i7-2720QM processor with 8GB RAM running Ubuntu 11.04 LTS, Linux Kernel version 2.6.38-16-generic (x86-64-bit architecture). Our measurements only use a single core of the Intel processor for consistency. The AutoGroup+ implementation utilizes the same building blocks as AutoGroup which include the MIRACL library (v5.5.4) and/or RELIC cryptographic toolkit [8], Charm v0.43 [5] in C++ or Python code, and the Z3 SMT solver (v4.3.2).

Limitations. In Section 3.1.1, we provide an example of a scheme which falls into a category of things that Abe et al. warned about and on which AutoGroup gets confused. AutoGroup tries to power through and split the hash output (which it cannot really do because the discrete log is unknown), so while it eventually outputs some SDL, this SDL is not a proper translation. Unlike AutoGroup, AutoGroup+ includes logic to output a warning when processing such inputs and continues trying to translate the scheme. If the verification check of a valid split fails (*e.g.*, due to hash split), then AutoGroup+ identifies the split as invalid and attempts checking the next best solution. If there are no such solutions, AutoGroup+ outputs no solution.

5.1 Comparison with ACSC/Charm

Our experiments have five schemes in common with public implementations in the Advanced Crypto Software Collection [19] and Charm [5]. Where we have matches, our new results confirm the security and optimality of those (unproven) implemented translations.

For Waters 2009 [26], we compare with the Charm implementation by Fan Zhang. For our PK-size optimization, our

[9]Currently the tool does not support the assumption minimization option for schemes with more than one assumption. This is future work, although we would like to explore how valuable assumption minimization is to tool users.
[10]We only give details for two variations of BB HIBE because the results are similar for all levels.

	Conversion	Number of Group Elements				Assumption	Num.
	Time	Public Key	Secret Key	Ciphertext	Assumption		Solutions
ID-Based Enc.							
BB04 HIBE [10, §4] Symmetric (l = 2)	-	$\mathbb{G}^5$	$\mathbb{G}^3$	$\mathbb{G}^3 \times \mathbb{G}_T$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	592 ms	$\mathbb{G}_1^5 \times \mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2^2$	$\mathbb{G}_1^2 \times \mathbb{G}_2 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
Asymmetric [Min. SK]	641 ms	$\mathbb{G}_1^5 \times \mathbb{G}_2^4$	$\mathbb{G}_1^3$	$\mathbb{G}_2^3 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
Asymmetric [Min. CT]	626 ms	$\mathbb{G}_1^4 \times \mathbb{G}_2^5$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
Asymmetric [Min. Assump]	582 ms	$\mathbb{G}_1^4 \times \mathbb{G}_2^5$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
BB04 HIBE [10, §4] Symmetric (l = 9)	-	$\mathbb{G}^{12}$	$\mathbb{G}^{10}$	$\mathbb{G}^{10} \times \mathbb{G}_T$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	20629 ms	$\mathbb{G}_1^{12} \times \mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2^9$	$\mathbb{G}_1^9 \times \mathbb{G}_2 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
Asymmetric [Min. SK]	15714 ms	$\mathbb{G}_1^{12} \times \mathbb{G}_2^{11}$	$\mathbb{G}_1^{10}$	$\mathbb{G}_2^{10} \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
Asymmetric [Min. CT]	15690 ms	$\mathbb{G}_1^{11} \times \mathbb{G}_2^{12}$	$\mathbb{G}_2^{10}$	$\mathbb{G}_1^{10} \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
Asymmetric [Min. Assump]	20904 ms	$\mathbb{G}_1^{11} \times \mathbb{G}_2^{12}$	$\mathbb{G}_2^{10}$	$\mathbb{G}_1^{10} \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
GENTRY06 [21, §3.1] Symmetric	-	$\mathbb{G}^3$	$\mathbb{Z}_p \times \mathbb{G}$	$\mathbb{G} \times \mathbb{G}_T^2$	$\mathbb{G}^{3+q} \times \mathbb{G}_T$	trunc. dec. q-ABDHE	
Asymmetric [Min. PK]	669 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^2$	$\mathbb{Z}_p \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{3+q} \times \mathbb{G}_2^{2+q} \times \mathbb{G}_T$		4
Asymmetric [Min. SK]	718 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^3$	$\mathbb{Z}_p \times \mathbb{G}_1$	$\mathbb{G}_2 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{2+q} \times \mathbb{G}_2^{3+q} \times \mathbb{G}_T$		4
Asymmetric [Min. CT]	723 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^2$	$\mathbb{Z}_p \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{3+q} \times \mathbb{G}_2^{2+q} \times \mathbb{G}_T$		4
Asymmetric [Min. Assump]	676 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^2$	$\mathbb{Z}_p \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{3+q} \times \mathbb{G}_2^{1+q} \times \mathbb{G}_T$		4
WATERS05 [25, §4] Symmetric	-	$\mathbb{G}^{4+n}$	$\mathbb{G}^2$	$\mathbb{G}^2 \times \mathbb{G}_T$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	725 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. SK]	770 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^{3+n}$	$\mathbb{G}_1^2$	$\mathbb{G}_2^2 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. CT]	767 ms	$\mathbb{G}_1^{3+n} \times \mathbb{G}_2^{4+n}$	$\mathbb{G}_2^2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. Assump]	716 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^{3+n}$	$\mathbb{G}_1^2$	$\mathbb{G}_2^2 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
WATERS09 (DSE) [26, §3.1] Symmetric	-	$\mathbb{G}^{13} \times \mathbb{G}_T$	$\mathbb{G}^8 \times \mathbb{Z}_p$	$\mathbb{Z}_p \times \mathbb{G}^9 \times \mathbb{G}_T$	$(\mathbb{G}^4 \times \mathbb{G}_T), (\mathbb{G}^6), (\mathbb{G}^6)$	DBDH, DLIN, DLIN	
Asymmetric [Min. PK]	6217 ms	$\mathbb{G}_1^{10} \times \mathbb{G}_2^4 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^4 \times \mathbb{Z}_p$	$\mathbb{G}_1^5 \times \mathbb{G}_2^4 \times \mathbb{G}_T$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Asymmetric [Min. SK]	5871 ms	$\mathbb{G}_1^7 \times \mathbb{G}_2^{13} \times \mathbb{G}_T$	$\mathbb{G}_1^8 \times \mathbb{Z}_p$	$\mathbb{G}_2^9 \times \mathbb{G}_T$	$(\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Asymmetric [Min. CT]	5858 ms	$\mathbb{G}_1^{13} \times \mathbb{G}_2^7 \times \mathbb{G}_T$	$\mathbb{G}_2^8 \times \mathbb{Z}_p$	$\mathbb{G}_1^9 \times \mathbb{G}_T$	$(\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Asymmetric [Min. Assump]	6228 ms	$\mathbb{G}_1^{12} \times \mathbb{G}_2^5 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^5 \times \mathbb{Z}_p$	$\mathbb{G}_1^6 \times \mathbb{G}_2^3 \times \mathbb{G}_T$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^2 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Broadcast Encryption							
BGW05 [14, §3.1] Symmetric (n users)	-	$\mathbb{G}^{2n+1}$	$\mathbb{G}$	$\mathbb{G}^3$	$\mathbb{G}^{2l+1} \times \mathbb{G}_T$	decision l-BDHE	
Asymmetric [Min. PK]	530 ms	$\mathbb{G}_1^{2n+1} \times \mathbb{G}_2^{2n}$	$\mathbb{G}_2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Asymmetric [Min. SK]	601 ms	$\mathbb{G}_1^{2n} \times \mathbb{G}_2^{2n+1}$	$\mathbb{G}_1$	$\mathbb{G}_2^2 \times G_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Asymmetric [Min. CT]	587 ms	$\mathbb{G}_1^{2n+1} \times \mathbb{G}_2^{2n}$	$\mathbb{G}_2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Asymmetric [Min. Assump]	544 ms	$\mathbb{G}_1^{2n+1} \times \mathbb{G}_2^{2n}$	$\mathbb{G}_2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Signature							
ACDKNO [1, §5.3] Symmetric	-	$\mathbb{G}^{15}$	$\mathbb{G}^2$	$\mathbb{G}^8$	$(\mathbb{G}^4), (\mathbb{G}^6), (\mathbb{G}^6)$	CDH, DLIN, DLIN	
Asymmetric [Min. PK]	18216 ms	$\mathbb{G}_1^{14} \times \mathbb{G}_2^5$	$\mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2^7$	$(\mathbb{G}_1^2 \times \mathbb{G}_2^4)$, $(\mathbb{G}_1^2 \times \mathbb{G}_2^6)$, $(\mathbb{G}_1^2 \times \mathbb{G}_2^6)$		1024
Asymmetric [Min. Sig]	14689 ms	$\mathbb{G}_1^6 \times \mathbb{G}_2^{14}$	$\mathbb{G}_1^2$	$\mathbb{G}_1^8$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^2)$, $(\mathbb{G}_1^6 \times \mathbb{G}_2^2)$, $(\mathbb{G}_1^6 \times \mathbb{G}_2^2)$		1024
Asymmetric [Min. Assump]	18135 ms	$\mathbb{G}_1^5 \times \mathbb{G}_2^{14}$	$\mathbb{G}_1^2$	$\mathbb{G}_1^7 \times \mathbb{G}_2$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^2)$, $(\mathbb{G}_1^6 \times \mathbb{G}_2^2)$, $(\mathbb{G}_1^6 \times \mathbb{G}_2^2)$		1024
BLS [15, §2.2] Symmetric	-	$\mathbb{G}^2$	$\mathbb{Z}_p^*$	$\mathbb{G}$	$\mathbb{G}^4$	CDH	
Asymmetric [Min. PK]	515 ms	$\mathbb{G}_2^2$	$\mathbb{Z}_p^*$	$\mathbb{G}_1$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3)$, $(\mathbb{G}_1^3 \times \mathbb{G}_2^3)$, $(\mathbb{G}_1^3 \times \mathbb{G}_2^3)$		2
Asymmetric [Min. Sig]	556 ms	$\mathbb{G}_2^2$	$\mathbb{Z}_p^*$	$\mathbb{G}_1$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3)$, $(\mathbb{G}_1^3 \times \mathbb{G}_2^3)$, $(\mathbb{G}_1^3 \times \mathbb{G}_2^3)$		2
Asymmetric [Min. Assump]	517 ms	$\mathbb{G}_2^2$	$\mathbb{Z}_p^*$	$\mathbb{G}_1$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3)$, $(\mathbb{G}_1^3 \times \mathbb{G}_2^2)$, $(\mathbb{G}_1^3 \times \mathbb{G}_2^3)$		2
CL04 [16, §3.1] Symmetric	-	$\mathbb{G}^3$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}^3$	$\mathbb{G}^3$	LRSW	
Asymmetric [Min. PK]	278 ms	$\mathbb{G}_1^3 \times \mathbb{G}_2$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3$		2
Asymmetric [Min. Sig]	328 ms	$\mathbb{G}_1 \times \mathbb{G}_2^3$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}_1^3$	$\mathbb{G}_2^3$		2
Asymmetric [Min. Assump]	275 ms	$\mathbb{G}_1^3 \times \mathbb{G}_2$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3$		2
WATERS05 [25, §7] Symmetric	-	$\mathbb{G}^{4+n}$	$\mathbb{G}$	$\mathbb{G}^2$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	724 ms	$\mathbb{G}_1^3 \times \mathbb{G}_2$	$\mathbb{G}_2^3$	$\mathbb{G}_1 \times \mathbb{G}_2$	$\mathbb{G}_1^4 \times \mathbb{G}_2^2 \times \mathbb{G}_T$		8
Asymmetric [Min. Sig]	721 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^{3+n}$	$\mathbb{G}_1$	$\mathbb{G}_1^2$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. Assump]	755 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^2$	$\mathbb{G}_1$	$\mathbb{G}_1 \times \mathbb{G}_2$	$\mathbb{G}_1^4 \times \mathbb{G}_2^2 \times \mathbb{G}_T$		8

Figure 4: A summary of the experimental evaluations of AutoGroup+ on a variety of schemes and optimization options. For the symmetric baseline with curve SS1536, elements in $\mathbb{G}$ are 1536 bits and $\mathbb{G}_T$ are 3072 bits. For the asymmetric translations with BN256, elements in $\mathbb{G}_1$ are 256 bits, $\mathbb{G}_2$ are 1024 bits, and $\mathbb{G}_T$ are 3072 bits. For BGW05, the private key size is listed for a single user.

translation is 3 elements shorter (we split only g, whereas they split g, w, u, h.) For our ciphertext-size optimization, it looks the closest to theirs, but they do not match. Both translations have short ciphertexts leaving all base elements in $\mathbb{G}_1$. However, the Charm translation appears to have shifted some elements from the public key to the secret key and dropped some elements from the master secret key (e.g., we split v and include both in the MSK, because that is the naive way to do it, but they use the v split for $\mathbb{G}_1$ only in the Setup and then drop it from the MSK.) While we cannot confirm the security of this implementation using our tool, the tool did produce a translation with the same ciphertext-size that is secure.

For BGW 2005 [14], we compared with the C implementation on the ACSC website by Matt Steiner and Ben Lynn. Indeed, our translations that minimize the public parameters or ciphertext size are the same, and the same as their manual translation. We confirm security and PP/ciphertext-size optimality.

For BB HIBE [10], Charm has a full HIBE implementation. We tested it for a minimum of 2 levels, but their implementation matches ours for ciphertext minimization, except that they add a precomputed pairing (element in $\mathbb{G}_T$) to the public key so that it does not have to be done per encryption. This impacts only efficiency. We confirm security and ciphertext-size optimality.

For CL [16], we can confirm that the Charm implementation is secure and public-key-size optimal. However, in the more likely event that one wants to minimize signature size, AutoGroup+ found a translation with a shorter signature.

For BLS [15], our translations also match. This is a simple case with only two translation options.

Charm [5] also includes variants of the Waters encryption and signature schemes [25] from 2005, but we translated the original schemes (as did [2, 6]), so our translations are not directly comparable to these Charm variants.

5.2 Comparison with Abe et al.

Abe et al. [2] tested their method on two encryption schemes: Waters 2005 [25] and Waters 2009 (Dual System Encryption) [26]. They looked at minimizing the size of the public key and the Type-III assumption. We conjecture that practitioners would be more interested in minimizing ciphertext or private key size, so our summary also includes those optimizations.

	Time•			
	Setup	**Keygen**	**Encrypt/ Sign**	**Decrypt/ Verify**
ID-Based Enc.				
BB04 HIBE [10, §4] Symmetric (SS1536) (l = 2)	346.47 ms	84.75 ms	118.64 ms	133.48 ms
Asymmetric (BN256) [Min. PK]	5.09 ms	4.79 ms	12.92 ms	21.36 ms
Asymmetric (BN256) [Min. SK]	8.15 ms	2.95 ms	14.95 ms	21.32 ms
Asymmetric (BN256) [Min. CT]	9.84 ms	6.23 ms	12.38 ms	21.22 ms
Asymmetric (BN256) [Min. Assump]	9.08 ms	7.30 ms	12.27 ms	21.64 ms
BB04 HIBE [10, §4] Symmetric (SS1536) (l = 9)	892.69 ms	283.11 ms	217.39 ms	446.10 ms
Asymmetric (BN256) [Min. PK]	9.25 ms	17.64 ms	17.10 ms	70.84 ms
Asymmetric (BN256) [Min. SK]	20.53 ms	11.14 ms	24.36 ms	71.45 ms
Asymmetric (BN256) [Min. CT]	21.60 ms	27.02 ms	16.48 ms	72.03 ms
Asymmetric (BN256) [Min. Assump]	21.68 ms	31.96 ms	16.77 ms	70.48 ms
GENTRY06 [21, §3.1] Symmetric (SS1536)	172.30 ms	28.23 ms	137.79 ms	48.42 ms
Asymmetric (BN256) [Min. PK]	2.88 ms	2.47 ms	21.08 ms	10.01 ms
Asymmetric (BN256) [Min. SK]	4.22 ms	1.18 ms	22.46 ms	9.96 ms
Asymmetric (BN256) [Min. CT]	2.93 ms	2.53 ms	21.02 ms	10.02 ms
Asymmetric (BN256) [Min. Assump]	2.88 ms	2.53 ms	21.10 ms	10.09 ms
WATERS05 [25, §4] Symmetric (SS1536)	908.94 ms	29.78 ms	78.08 ms	111.76 ms
Asymmetric (BN256) [Min. PK]	10.31 ms	2.04 ms	11.98 ms	14.23 ms
Asymmetric (BN256) [Min. SK]	24.11 ms	1.37 ms	13.68 ms	14.11 ms
Asymmetric (BN256) [Min. CT]	25.39 ms	3.67 ms	11.25 ms	14.23 ms
Asymmetric (BN256) [Min. Assump]	23.81 ms	1.36 ms	13.71 ms	14.38 ms
WATERS09 (DSE) [26, §3.1] Symmetric (SS1536)	755.50 ms	195.27 ms	212.88 ms	414.79 ms
Asymmetric (BN256) [Min. PK]	23.13 ms	9.71 ms	13.70 ms	66.45 ms
Asymmetric (BN256) [Min. SK]	36.83 ms	7.07 ms	20.08 ms	66.42 ms
Asymmetric (BN256) [Min. CT]	34.41 ms	14.82 ms	11.08 ms	66.92 ms
Asymmetric (BN256) [Min. Assump]	29.90 ms	11.09 ms	13.03 ms	66.92 ms
Broadcast Encryption				
BGW05 [14, §3.1] Symmetric (SS1536) ($n = 10$)	376.84 ms	140.27 ms	86.96 ms	68.65 ms
Asymmetric (BN256) [Min. PK]	55.29 ms	13.98 ms	11.457 ms	6.13 ms
Asymmetric (BN256) [Min. SK]	38.45 ms	5.82 ms	12.49 ms	8.122 ms
Asymmetric (BN256) [Min. CT]	37.75 ms	12.32 ms	11.18 ms	6.27 ms
Asymmetric (BN256) [Min. Assump]	37.74 ms	12.31 ms	11.186 ms	6.12 ms
Signature				
ACDKNO [1, §5.3] Symmetric (SS1536)	395.23 ms	497.04 ms	275.99 ms	937.14 ms
Asymmetric (BN256) [Min. PK]	9.05 ms	17.19 ms	15.27 ms	147.62 ms
Asymmetric (BN256) [Min. Sig]	8.31 ms	22.65 ms	14.33 ms	152.60 ms
Asymmetric (BN256) [Min. Assump]	8.43 ms	22.23 ms	13.94 ms	147.77 ms
BLS [15, §] Symmetric (SS1536)	-	93.20 ms	92.61 ms	167.73 ms
Asymmetric (BN256) [Min. PK]	-	2.99 ms	0.74 ms	14.20 ms
Asymmetric (BN256) [Min. Sig]	-	3.00 ms	0.75 ms	14.20 ms
Asymmetric (BN256) [Min. Assump]	-	3.03 ms	0.69 ms	14.18 ms
CL04 [16, §3.1] (SS1536)	-	464.7 ms	178.18 ms	973.48 ms
Asymmetric (BN256) [Min. PK]	-	9.27 ms	15.12 ms	121.61 ms
Asymmetric (BN256) [Min. Sig]	-	14.54 ms	7.38 ms	119.16 ms
Asymmetric (BN256) [Min. Assump]	-	11.53 ms	15.32 ms	124.19 ms
WATERS05 [25, §7] (SS1536)	-	720.75 ms	29.72 ms	135.00 ms
Asymmetric (BN256) [Min. PK]	-	10.42 ms	2.02 ms	21.44 ms
Asymmetric (BN256) [Min. Sig]	-	25.60 ms	1.43 ms	23.13 ms
Asymmetric (BN256) [Min. Assump]	-	10.18 ms	2.01 ms	21.42 ms

•Average time measured over 100 test runs and the standard deviation in all test runs were within ±1% of the average.

Figure 5: A summary of the running times of the AutoGroup+ translations using curve BN256 as compared to the running times using the roughly security-equivalent symmetric curve SS1536 in MIRACL. The asymmetric setting plus AutoGroup+'s optimizations cut the running times by one or two orders of magnitude.

For Waters 2005, AutoGroup+ found the same construction as their semi-automated method. As remarked in Section 3.1.1, their dependency graph for this scheme included some unnecessary dependencies. Waters [25] clearly states to choose g_2, u', u_i as fresh random generators, but Abe et al. explicitly "assume" that they are generated from a common generator g. From a functionality and security standpoint *of the Type-I scheme*, this distinction certainly does not matter. However, it does change the intermediate dependency graphs, which could in some cases affect the output (though it does not in this situation). Both their partial automation and our full one of Waters 2005 took under one second.

For Waters 2009, AutoGroup+ first appeared to find a PK-optimized construction with one less group element than the PK-optimized construction of Abe et al. [2]. However, subsequent discussions [3] determined that this was merely the product of a different counting method; the numbers reported in this work are the correct ones for both AutoGroup+ and the Abe et al. method.

In the original work [2], no schemes with interactive assumptions were reported on. In subsequent communications [3], Abe et al. demonstrated a translation for the Camenisch-Lysyanskaya signatures [16] based on the interactive LSRW assumption. We derived the SDL files for the scheme, assumption and proof and ran it through AutoGroup+. The results matched.

Drawing and merging the dependency graphs by hand is tedious and becomes infeasible for a complex scheme like [1]. In addition, the Abe et al. graph splitting program took 1.75 hours for Waters09, whereas our tool handled everything in 6.5 seconds. Thus, we find that it is considerably easier and faster to transcribe the SDL and use AutoGroup+.

5.3 Comparison with AutoGroup

The AutoGroup tool [6] was used as the starting point for our implementation, hence the name of AutoGroup+. Our 48 translation experiments overlap with AutoGroup in 14 points (seven schemes in common and they do fewer optimizations). For these 14, the tools found the same constructions. However, a major difference is that with AutoGroup+, we have security guarantees. This required us to write new SDL descriptions for all the assumptions and proofs involved.

Indeed, one crucial question was how the security logic would increase translation times. We focused our effort on leveraging an SMT Solver to help handle this security logic, which kept the running times of AutoGroup+ within a few seconds of AutoGroup.

In addition to the security logic we added, we also found that the public key optimization flag for encryption was not implemented. Because we wanted to compare our results with [2], we implemented it.

AutoGroup was tested on one signature scheme omitted here. Boneh-Boyen [12] has a nested proof structure that falls outside of the black box reductions considered here.

5.4 Comparison with manual translations

The Dual System Encryption scheme of Waters [26] has a few manual translations with a security analysis. Ramanna, Chatterjee and Sarkar [23] provide a variety of translations, one with the smallest public parameter/key size, at the cost of introducing some mild complexity assumptions. Similarly, Chen, Lim, Ling, Wang and Wee [17] presented a translation introducing the SXDH assumption, which achieved the shortest ciphertext size. These results are superior to those derived by AutoGroup+ and [2, 6], but it is not yet clear how to generalize and systematize the human creativity used.

6. CONCLUSIONS

Automation is the future for many cryptographic design tasks. This work successfully demonstrates automating a complex translation of a scheme from one algebraic setting to another. There was a demonstrated need for such a compiler both for pairing designers and implementors. Its realization combined and improved on contributions from the systems [6] and theory [2] communities. The result is a practical tool, AutoGroup+, that enables secure pairing translations for everyone.

7. ACKNOWLEDGMENTS

The authors thank Masayuki Abe, Jens Groth, Miyako Ohkubo, Takeya Tango for very helpful discussions regarding this work and their prior CRYPTO 2014 work.

8. REFERENCES

[1] M. Abe, M. Chase, B. David, M. Kohlweiss, R. Nishimaki, and M. Ohkubo. Constant-size structure-preserving signatures: Generic constructions and simple assumptions. Cryptology ePrint Archive, Report 2012/285, 2012.

[2] M. Abe, J. Groth, M. Ohkubo, and T. Tango. Converting cryptographic schemes from symmetric to asymmetric bilinear groups. In *CRYPTO*, pages 241–260, 2014.

[3] M. Abe, J. Groth, M. Ohkubo, and T. Tango, 2015. Private communications.

[4] J. A. Akinyele, C. Garman, and S. Hohenberger. Automating Fast and Secure Translations from Type-I to Type-III Pairing Schemes, 2015. Full version at Cryptology ePrint Archive Report 2015/290.

[5] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013. `http://www.charm-crypto.com/Main.html`.

[6] J. A. Akinyele, M. Green, and S. Hohenberger. Using SMT solvers to automate design tasks for encryption and signature schemes. In *ACM CCS*, pages 399–410, 2013.

[7] J. A. Akinyele, M. Green, S. Hohenberger, and M. W. Pagano. Machine-generated algorithms, proofs and software for the batch verification of digital signature schemes. In *ACM CCS*, pages 474–487, 2012.

[8] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient Library for Cryptography. `http://code.google.com/p/relic-toolkit/`.

[9] G. Barthe, E. Fagerholm, D. Fiore, J. C. Mitchell, A. Scedrov, and B. Schmidt. Automated analysis of cryptographic assumptions in generic group models. In *CRYPTO 2014*, pages 95–112, 2014.

[10] D. Boneh and X. Boyen. Efficient selective-id secure identity-based encryption without random oracles. In *EUROCRYPT*, volume 3027, pages 223–238. 2004.

[11] D. Boneh and X. Boyen. Efficient selective-id secure identity based encryption without random oracles. Cryptology ePrint Archive, Report 2004/172, 2004.

[12] D. Boneh and X. Boyen. Short signatures without random oracles. In *EUROCRYPT*, volume 3027, pages 382–400, 2004.

[13] D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, pages 213–229, 2001.

[14] D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *CRYPTO*, pages 258–275, 2005.

[15] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *J. of Cryptology*, 17(4):297–319, 2004.

[16] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, volume 3152, pages 56–72, 2004.

[17] J. Chen, H. W. Lim, S. Ling, H. Wang, and H. Wee. Shorter IBE and signatures via asymmetric pairings. In *Pairing*, pages 122–140. Springer, 2013.

[18] J. Chen, H. W. Lim, S. Ling, H. Wang, and H. Wee. Shorter identity-based encryption via asymmetric pairings. *Des. Codes Cryptography*, 73(3):911–947, 2014.

[19] A. Contributors. Advanced crypto software collection. `http://hms.isi.jhu.edu/acsc/`.

[20] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of Software*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[21] C. Gentry. Practical identity-based encryption without random oracles. In *EUROCRYPT*, pages 445–464, 2006.

[22] S. Krishnamurthy. HP to Acquire Voltage Security to Expand Data Encryption Security Solutions for Cloud and Big Data. `http://www.voltage.com/blog/releases`, February 9, 2015.

[23] S. C. Ramanna, S. Chatterjee, and P. Sarkar. Variants of Waters' dual system primitives using asymmetric pairings - (extended abstract). In *PKC '12*, pages 298–315, 2012.

[24] E. P. Team. Easycrypt: Computer-aided cryptographic proofs. `https://www.easycrypt.info/trac/`.

[25] B. Waters. Efficient identity-based encryption without random oracles. In *EUROCRYPT '05*, volume 3494 of LNCS, pages 320–329. Springer, 2005.

[26] B. Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In *CRYPTO*, pages 619–636, 2009.