Comparing Platform Groups for the Anshel-Anshel-Goldfeld Key Exchange

Reed Nelson
Department of Computer Sciences
University of Wisconsin-Madison

Michael Noguera
Department of Computer Sciences
University of Wisconsin-Madison

Carson Drury
Department of Computer Sciences
University of Wisconsin-Madison

ABSTRACT

The Anshel-Anshel-Goldfeld key exchange protocol (AAG) is a less-secure alternative to the Diffie-Hellman key exchange, based on conjugation in non-abelian groups. However where Diffie-Hellman is vulnerable to Shor's algorithm, there is no known quantum algorithm for efficiently attacking AAG. As a general protocol, AAG is compatible with any non-abelian platform group. Much theoretical research has been done on specific platform groups, but little comparing them.

In this paper we present the first generic implementation of the AAG key exchange. Our implementation is easily extensible to any group defined in SageMath, encompassing a large number of frequently-considered options. We release our program as a tool for further experimentation with new platform groups and platform-group specific attacks, and demonstrate a comparison between selected groups' resistance to naive brute-force attacks.

CCS CONCEPTS

• **Security and privacy → Public key encryption**; **Public key (asymmetric) techniques**.

1 INTRODUCTION

Cryptographic schemes play a crucial role in modern security systems, and their resistance to attack is of utmost importance. The security of the most widely used public-key algorithms, including RSA, Diffie-Hellman, and Elliptic Curve, rely on the hardness of the prime factorization problem, discrete logarithm problem, and elliptic curve discrete logaritm problem, respectively. Using Shor's algorithm [16], each of these problems are solvable in polynomial time given a sufficiently advanced quantum computer. These problems are based in number theory, and thus rely on commutative group structure. In recent years, much research has explored the domain of non-commutative cryptography.

The *Conjugacy Search Problem* (CSP) is the one-way function underlying many non-commutative cryptographic protocols [9–11, 13]. While quantum algorithms have been presented to solve many group-theoretic problems, so far no generic algorithm is known to solve the CSP in polynomial time [7].

We focus on the Anshel-Anshel-Goldfeld (AAG) key exchange protocol [1], whose security relies on a variant of the CSP. AAG is currently the most prominent non-commutative cryptographic protocol, and like many others, it can be instantiated using any non-commutative (also called *non-abelian*) group. The chosen group is the "platform" on which the rest of the protocol is constructed. Previous research has suggested a number of platform groups for AAG, but no generic AAG implementation exists, precluding an applied comparison between these platforms.

Contributions The main contributions of this paper are:

- We have created the first generic implementation of the AAG key exchange protocol that works with multiple platform groups.
- We implement a naive brute-force attack algorithm that serves as a benchmark for comparison. Researchers developing more advanced attacks can use our tool to measure their attack performance relative to this baseline.
- We provide a software platform easily extensible to additional groups defined in the SageMath library [18]. Researchers experimenting with new platform groups can use our generic implementation to perform key exchange.
- We perform a comparison between groups suggested by prior research as platforms for AAG using our naive brute-force attack.

Open-Source Release

All code and simulation data is publicly available on Github: https://github.com/reednel/aag.

2 BACKGROUND

2.1 Anshel-Anshel-Goldfeld Key Exchange

We provide here a practical explanation of an AAG exchange between Alice and Bob. For a more mathematical formulation, refer to AAG [1]. The steps of the exchange are as follows:

1. Alice and Bob agree on a platform group G. They establish that Alice is the first party and Bob the second, to resolve asymmetry in the last step.
2. **Public set (Alice)** Alice chooses an N-sized subset of G to be her public set[1], and sends it to Bob.

$$\bar{a} = (a_1, \ldots, a_N),\ a_i \in G$$

   Programatically, the data type of $\bar{a}$ is a one-dimentional set containing elements of the same type as elements of G. We also place the following restrictions on $\bar{a}$ (and $\bar{b}$): a public set cannot contain duplicate elements, nor can a public set contain elements that are inverses of other elements in the set.[2]
3. **Public set (Bob)** Bob similarly chooses a public set and sends it to Alice.

$$\bar{b} = (b_1, \ldots, b_N),\ b_i \in G$$

[1] Without loss of generality, we assume here that Alice and Bob both have public sets of size N and private keys that are products of L elements. In reality, the sizes of their sets and keys need not be equal.

[2] This restriction is not part of the protocol described in AAG [1]. We have added the prohibition on inverses to make key entropy more predictable.

4. **Private key (Alice)** Alice chooses L elements at random from $\bar{a}$, and stores (at random) either it or its inverse in a tuple. Her private key A is the product of the elements in sequence. Mathematically, she computes

$$A = a_{s(1)}^{\varepsilon_1} \times a_{s(2)}^{\varepsilon_2} \times \ldots \times a_{s(L-1)}^{\varepsilon_{L-1}} \times a_{s(L)}^{\varepsilon_L}$$

where $a_i \in \bar{a}$ and $\varepsilon_i \in \{\pm 1\}$. Here A is the same type as a single element of G.

5. **Private key (Bob)** Bob does the same to derive his private key, producing

$$B = b_{t(1)}^{\delta_1} \times b_{t(2)}^{\delta_2} \times \ldots \times b_{t(L-1)}^{\delta_{L-1}} \times b_{t(L)}^{\delta_L}$$

where $b_{t(i)} \in \bar{b}$ and $\delta_i \in \{\pm 1\}$.

6. **Transition (Bob → Alice)** Alice needs to obtain the shared key $A^{-1}B^{-1}AB$ but is unable to compute $B^{-1}AB$ herself, as she does not know Bob's private key B or its inverse B^{-1}.
She could ask Bob to compute $B^{-1}a_sB$ for each a_s in her private key A, then multiply all those terms together to obtain the needed $B^{-1}AB$, but this would reveal the elements in A.
Instead, Alice asks Bob to send her $B^{-1}aB$ for all a in her public set $\bar{a}$. This is the *transition set* $a' = B^{-1}\bar{a}B$. It is a superset of the values she needs, but reveals nothing unknown about her private key. a' is an ordered one-dimensional set with entries corresponding to the elements in $\bar{a}$.

$$a' = B^{-1}\bar{a}B = (B^{-1}a_1B,\ B^{-1}a_2B,\ \ldots\ ,\ B^{-1}a_{N-1}B,\ B^{-1}a_NB)$$

Each element of the transition set, a'_i, is obtained by computing

$$\begin{aligned} B^{-1}a_iB &= B^{-1} \times a_i \times B \\ &= (b_{t(1)}^{\delta_1} \times \ldots \times b_{t(L)}^{\delta_L})^{-1} \times a_i \times (b_{t(1)}^{\delta_1} \times \ldots \times b_{t(L)}^{\delta_L}) \\ &= b_{t(1)}^{-\delta_1} \times \ldots \times b_{t(L)}^{-\delta_L} \times a_i \times b_{t(1)}^{\delta_1} \times \ldots \times b_{t(L)}^{\delta_L} \end{aligned}$$

7. **Transition (Alice → Bob)** Symmetrically, Alice computes her transition set, $b' = A^{-1}\bar{b}A$, and sends it to Bob.
8. **Shared Key (Alice)** To compute the shared key K, Alice first subsets $a' = B^{-1}\bar{a}B$, keeping only the values corresponding to those that compose her private key A. For each element retained, it is inverted if the corresponding element in A was inverted. Alice then multiplies these together to get $B^{-1}AB$. Finally, by multiplying A^{-1} on the left, Alice directly obtains $K_a = A^{-1}B^{-1}AB$, the shared key.
9. **Shared Key (Bob)** Symmetric to Alice, Bob subsets $b' = A^{-1}\bar{b}A$, keeping only the values corresponding to those composing B. Bob inverts the appropriate elements, and multiplies these together to get $A^{-1}BA$. Left-multiplying this by B^{-1} gives $K_b = B^{-1}A^{-1}BA$, the inverse of which is $A^{-1}B^{-1}AB = K_a$. Note that Bob must invert his result to reach K_a, whereas Alice performs no such step.

2.2 The Conjugacy Search Problem

The security of AAG is based on the difficulty of the Subgroup-Restricted Simultaneous Conjugacy Search Problem for the chosen platform group [17]. This problem is a variation on the easier Conjugacy Search Problem. Understanding this underlying problem will give a better intuition for the nature of AAG and the brute-force attack used later.

The **Conjugacy Search Problem** (CSP): given $g, h \in G$, find an $x \in G : x^{-1}gx = h$, if it exists.

The **Simultaneous Conjugacy Search Problem** (SCSP): given $(g_1, g_2, \ldots, g_N), (h_1, h_2, \ldots, h_N)$ for $g_i, h_i \in G$, find an $x \in G$ such that $x^{-1}g_ix = h_i$, for all $i = 1, 2, \ldots, N$, if it exists.

The **Subgroup-Restricted Simultaneous Conjugacy Search Problem** (SR-SCSP): given $g_i, h_i \in G$ and finitely generated subgroup $S \leq G$, find an $x \in S$ such that $x^{-1}g_ix = h_i$, for all $i = 1, 2, \ldots, N$, if it exists.

2.3 Attacks on AAG

Denote $\mathcal{A}_L$ as the space of Alice's private keys of length L. This contains all possible private keys of length L that Alice could choose. Given Bob's public key $b_1, \ldots, b_N$ and Alice's transition set $b'_1, \ldots, b'_N$, Eve must find $A \in \mathcal{A}_L$ such that $A^{-1}b_iA = b'_i$, for all $i = 1, 2, \ldots, N$. Without solving a variation on the *membership search problem*, Eve does not have the factors $(a_{s(1)}^{\varepsilon_1}, \ldots, a_{s(L)}^{\varepsilon_L})$ of A, so she must employ the same process to calculate B. Then Eve has $A^{-1}B^{-1}AB = K$. The naive, generic solution to this problem runs in $O(2N(2N)^L)) = O(N^L)$ time. See Algorithm 2 for specifics on our implementation.

The Length-Based Attack (LBA) [8] can *theoretically* be applied to many groups. Optimally, it recovers a private key in linear time with respect to public key length, however in practice, LBA is often less effective than more specialized attacks, and crucially, it relies on knowledge of a group-specific length function which in general is intractible to calculate. However, two of the most prominently studied groups for non-commutative protocols, the Braid group and Thompson's group, have turned out to be vulnerable to LBA [5, 6, 14, 15].

Efficient solutions to the CSP (and its variations) are, for the most part, group-specific, meaning no single attack is able to condemn the whole family of CSP-based protocols. LBA has had very limited success on the Heisenberg group [2, 12], due to the difficulty of calculating (or appoximating) the length function for the instantiation of arbitrary dimension. However there is an $O(n^5 \log^2 n)$ method [3] which claims to work more reliably. Like the Heisenberg, the Polycyclic group has demonstrated some resilience to LBA [4], yet numerous other attacks have been sprung on certain subfamilies under the Polycyclic umbrella [7].

3 METHODOLOGY/DESIGN

3.1 Threat Model

The threat model for AAG is the same as for other key exchange protocols. We consider an attacker (Eve) who can view all messages sent over a public channel, and two actors (Alice and Bob) who wish to communicate using the public channel without Eve viewing their communications. Our Eve is a *passive* attacker; she only eavesdrops on Alice and Bob's messages but does not engage in communication with either of them. The abilities of an *active* man-in-the-middle attacker may include interception, decryption, and alteration of messages en-route, but such coniderations are beyond the scope of this paper.

Furthermore, our analysis focuses only on Eve mounting a naive brute-force attack to recover Alice and Bob's private keys, in order

to directly compute the shared key $K = A^{-1}B^{-1}AB$. We choose this attack due to its generality and relative ease of implementation.

In reality, an attacker may take advantage of platform-specific knowledge to perform an informed search over the key space. We do not consider such advanced attacks here, but have intentionally built our program extensibly to enable its use for comparing these attacks. We also do not consider authenticating Alice or Bob to each other.

It is worth noting that existing key exchange protocols like Diffie-Hellman are theorized to be more secure than AAG under this same threat model. The main novelty of AAG in this situation is its presumed resistance to quantum attack, although its practical security is undermined by group-specific weaknesses like those that this paper attempts to analyze.

3.2 Implementation

We have created three Python programs: `aag.py`, `attack.py`, and `compare.py`. These handle the key exchange, brute force attack, and performance measurements respectively.

To handle group operations, we use the SageMath library [18], a Computer Algebra System that exposes a Python interface. Sage contains its own Python interpreter bundled with its math functions; the Sage library can only be used through this interpreter. As a result, our program is run through Sage as well. For example, `compare.py` is invoked from the terminal as `$ sage -python compare.py`, rather than being directly passed to the system Python.

Each of our programs expose functions useful for experimenting with the AAG protocol. The operation of each is detailed in this section.

3.2.1 Key Exchange. In `aag.py`, we implement the key exchange protocol. We define a generic `AAGExchangeObject` class, which represents the information held by a single party in the key exchange. The exchange object exposes functions to generate public and private keys, as well as to compute the transition set. Private keys are held internally and scoped privately such that accessing them from outside the class will throw an error, however no effort is made to actually secure them against an attacker with access to the computer.

An example usage of the `AAGExchangeObject` is shown in Algorithm 1. In the first step, a Sage BraidGroup object is created. The object is passed to the `AAGExchangeObject` constructor, and its type (in this case BraidGroup) is specified as the type parameter. Here we initialize Alice and Bob this way.

Next, Alice and Bob are instructed to choose public sets and private keys (`generatePublicKey(`N`)` and `generatePrivateKey(`L`)`). These methods take the public set size N and private key size L as parameters. The influence of each of these parameters is analyzed in the evaluation section of this paper.

To exchange keys, we call `deriveSharedKey()` for both Alice and Bob, giving them references to each other as parameters. To resolve the asymmetry at this step, where Bob must invert the value he calculates so that it agrees with Alice's, we use a boolean parameter where Alice enters True and Bob enters False. At this step, Alice also asks Bob for his transition set by calling the internal method `bob.transition(alice)`. This is abstracted away by deriveSharedKey so is not explicit in Algorithm 1, but the method is publically exposed if needed. For normal usage it does not need to be called by the user. The values returned by each call to `deriveSharedKey` are K_a and $(K_b)^{-1} = K_a$ respectively. This concludes the example of a key exchange.

Algorithm 1: Performing a key exchange using `aag.py`

```
from sage.groups.braid import BraidGroup
from aag import AAGExchangeObject

bg ← BraidGroup(5)  /* instantiate group object */
alice ← AAGExchangeObject[BraidGroup](bg)
bob ← AAGExchangeObject[BraidGroup](bg)

/* Choose public sets and private keys.       */
alice.generatePublicKey(length = 7)
bob.generatePublicKey(length = 7)
alice.generatePrivateKey(length = 5)
bob.generatePrivateKey(length = 5)

/* Shared keys. Alice: True, Bob: False       */
/* Transition sets are handled automatically. */
aliceSharedKey = alice.deriveSharedKey(True, bob)
bobSharedKey = bob.deriveSharedKey(False, alice)

assert aliceSharedKey = bobSharedKey
```

Finally, we also provide a test oracle invoked as `alice.oracle(bob)`, which accesses the private `bob._privateKey` field to directly compute the shared key $A^{-1}B^{-1}AB$. Of course, accessing this field is not allowed during the key exchange; this is only done in the test oracle.

3.2.2 Getting Random Elements. A critical component of any AAG exchange is the ability to get an element at random from the platform group. The implementation of this function varies from group to group in the Sage library, and for some groups, this function becomes unusably slow at sufficiently high cardinalities. For this reason, we wrote group-specific `getRandomElement()` functions when necessary, for the groups we analyzed, which circumvent Sage's behavior. These functions are used automatically when defined, otherwise Sage's random element is used as a fallback.

3.2.3 Attack. As already discussed, an inherent difficulty with analyzing the security of AAG is that the range of potential platform groups is immense, and even for a given platform, an attack's effectiveness may be highly dependent on the the precise instantiation chosen. By implementing a completely generic brute-force attack, we are able to judge any instantiation of any platform using the same tool, and to the same standard. The brute-force algorithm used is specified in Algorithm 2. It performs the attack described previously in Section 2.3, solving SR-SCSP twice to directly compute both A and B.

Algorithm 2: A brute force solution to SR-SCSP

Function `bruteforce`$(\overline{x}', L, \overline{y}, X^{-1}\overline{y}X)$**:**
- **for** ***each*** x ***in*** $\overline{x}'^{L}$ **do**
  - $g \leftarrow x_1 \cdot x_2 \cdot \ldots \cdot x_{2N}$
  - bool conj $\leftarrow$ TRUE
  - **for** ***each*** $\overline{y}_i$ ***in*** $\overline{y}$ **do**
    - **if** $g^{-1} \cdot \overline{y}_i \cdot g \neq X^{-1}\overline{y}_i X$ **then**
      - conj $\leftarrow$ FALSE
      - break
    - **end**
  - **end**
  - **if** *conj* **then**
    - break
  - **end**
- **end**
- **return** g

Function `main`$(\overline{a}, \overline{b}, L, A^{-1}\overline{b}A, B^{-1}\overline{a}B)$**:**
- $\overline{a}' \leftarrow \overline{a} \cup \overline{a}^{-1}$
- $A \leftarrow$ bruteforce$(\overline{a}', L, \overline{b}, A^{-1}\overline{b}A)$
- $\overline{b}' \leftarrow \overline{b} \cup \overline{b}^{-1}$
- $B \leftarrow$ bruteforce$(\overline{b}', L, \overline{a}, B^{-1}\overline{a}B)$
- $K \leftarrow A^{-1}B^{-1}AB$
- **return** K

4 EVALUATION

4.1 Results

4.1.1 Evidence of Correctness. We assert that the key values themselves are correct by verifying obtained keys against those calculated by the test oracle mentioned in Section 3.2.1.

Figure 1 provides visual confirmation that our AAG implementation and brute-force attack behave as expected, with generally exponential increase in guesses required to find Alice and Bob's shared key with respect to private keys of L factors. This trend is true of four distinct groups: The Symmetric group S_{16}, the Rubik's Cube group, the Heisenberg Group H_5, and the Braid group B_3.

4.1.2 Relative Lengths. Preliminary analysis showed that the average proportion of guesses made by the brute-force algorithm before it found a key *decreases* as we increase the number of private key factors (L), for a fixed public key size (N). To investigate this, we hold fixed the size of the search space, and vary the values (namely, L and N) that determine this size. The naive search space is size $(2N)^L$, so we must choose $N_1 \neq N_2$ and $L_1 \neq L_2$ satisfying the equality $(2N_1)^{L_1} = (2N_2)^{L_2}$. In Figure 2, we use $(2 \cdot 2)^{2L_2} = (2 \cdot 8)^{L_2}$, for $L_2 = 2, 3, \ldots, 7$.

The results from this test corroborate the initial observation: on brute-force, the average number of iterations required is much (and more consistently) higher when N is bigger than L, even for equally sized search spaces. Curves in the lower graph in Figure 2 showing the larger public set size are all right-leaning. Compare this to the curves in the upper graph showing the smaller public set size (with larger secret keys). The upper graph curves are less agressively right-leaning, meaning that it is easier to brute force longer secret keys chosen from small public sets than it is to brute force short secret keys chosen from larger sets.

Looking solely at the growth of the naive search space would suggest that Alice and Bob, constrained by a finite cost per exchange, can maximize their security by using very long private keys. After all, private key size is exponentially related to the number of permutations, whereas public set size only reflects the base of the exponent. On the contrary, this result indicates that security is actually favorable when public set size exceeds the number of private key factors. Running an identical test using the Rubik's Cube group as platform yields the same confirmation.

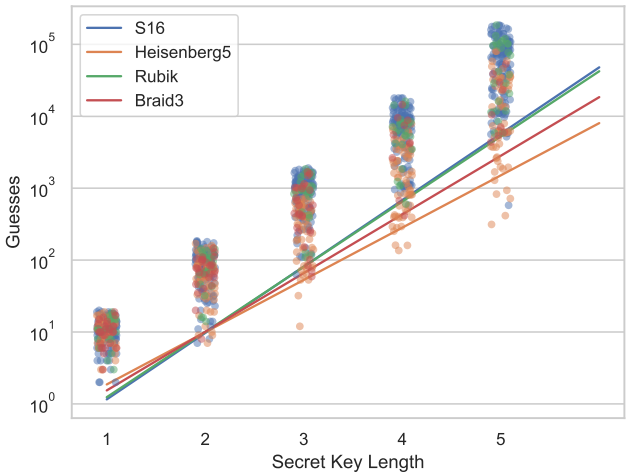

Figure 1: Comparison between groups with respect to number of guesses needed to obtain the shared key using the brute-force algorithm. Each group's performance is measured for various private key lengths, with public set size fixed at 5.

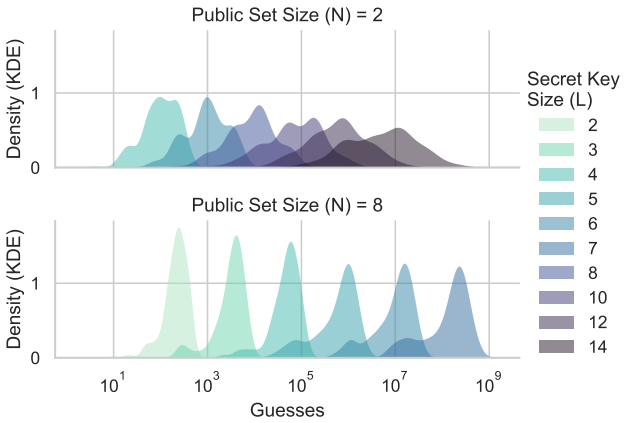

Figure 2: Density of guesses needed to obtain the shared key using the brute-force algorithm, against private keys of L factors. Search space is comparable between the upper and lower graphs, which show different public set sizes N (see Section 4.1.2). Platform: S_{16}; 100 samples per curve.

We explain this result by analogy. Consider searching an undirected graph representing the key space of a specific group. It is possible that multiple permutations of group elements multiply to reach the same value, so some nodes in the graph may be more connected than others. The graph exists in N dimensions, as we have ammended the AAG protocol[3], prohibiting public sets from containing any two elements in the same dimension.

In this situation, private key size is analogous to search depth, and public set size relates to the breadth that must be explored. Maximizing one of these factors while holding the other small produces a less than maximal search boundary. If some nodes in the graph are much more connected than others, increasing search depth without increasing breadth only serves to increase the number of times that you visit these highly connected nodes. The presence of these over-represented nodes explains the behavior in Figure 2. Likewise, having breadth without depth only makes a certain border set of the nodes possible as destinations, limiting key entropy.

The topology of this graph for a given N and L depends on the structure of the chosen platform group, as well as the constituent elements of the public set and private key.

5 CONCLUSION

We have presented a generic implementation of the AAG key exchange protocol that works with any platform group defined in SageMath, and a baseline brute-force attack that should be universally applicable. We have performed a basic comparison between platform groups using small key sizes, and discovered that public set size is more important than expected. We believe that this tool has utility for future cryptographic research investigating the security of the AAG protocol under different platform groups as well as implementing novel attacks against such groups.

Future Work

The scope of this work is limited to a simple analysis of naive brute force attacks using small key sizes. Key sizes considered here are much smaller than those that would be used cryptographically; studying feasable key lengths would require more processing power than we had access to.

Notes on Optimality and Security All results presented here rely on SageMath's implementations of group-theoretic operations. Any extra time taken due to suboptimal implementation of such operations can be factored out as it will be present in all analysis of attacks against said group.

This implementation is proposed for research use only, and of course, is not intended a real-world cryptographic setting. In addition to the practical suboptimalities, no effort is made to make the implementation secure.

REFERENCES

[1] Iris Anshel, Michael Anshel, and Dorian Goldfeld. 2001. An Algebraic Method For Public-Key Cryptography. *Mathematical Research Letters* 6 (01 2001). https://doi.org/10.4310/MRL.1999.v6.n3.a3

[2] Sébastien Blachère. 2003. Word distance on the discrete Heisenberg group. *Colloquium Mathematicae* 95, 1 (2003), 21–36. http://eudml.org/doc/285003

[3] Kenneth R. Blaney and Andrey Nikolaev. 2016. A PTIME solution to the restricted conjugacy problem in generalized Heisenberg groups. *Groups Complexity Cryptology* 8, 1 (2016), 69–74. https://doi.org/10.1515/gcc-2016-0003

[4] David Garber, Delaram Kahrobaei, and Ha T. Lam. 2013. Length-based attacks in polycyclic groups. *Journal of Mathematical Cryptology* 9 (2013), 33 – 43.

[5] D. Garber, S. Kaplan, M. Teicher, B. Tsaban, and U. Vishne. 2002. Length-based conjugacy search in the Braid group. (2002). https://doi.org/10.48550/ARXIV.MATH/0209267

[6] David Garber, Shmuel Kaplan, Mina Teicher, Boaz Tsaban, and Uzi Vishne. 2005. Probabilistic solutions of equations in the braid group. *Advances in Applied Mathematics* 35, 3 (sep 2005), 323–334. https://doi.org/10.1016/j.aam.2005.03.002

[7] Jonathan Gryak and Delaram Kahrobaei. 2016. The Status of Polycyclic Group-Based Cryptography: A Survey and Open Problems. (2016). https://doi.org/10.48550/ARXIV.1607.05819

[8] James Hughes and Allen Tannenbaum. 2003. Length-Based Attacks for Certain Group Based Encryption Rewriting Systems. (07 2003).

[9] Delaram Kahrobaei and Michael Anshel. 2009. Decision and Search in Non-Abelian Cramer-Shoup Public Key Cryptosystem. *Groups, Complexity, Cryptology* 1 (10 2009), 217–225. https://doi.org/10.1515/GCC.2009.217

[10] Delaram Kahrobaei and Bilal Khan. 2006. A Non-Commutative Generalization of ElGamal Key Exchange using Polycyclic Groups.

[11] Delaram Kahrobaei and Charalambos Koupparis. 2012. Non-commutative Digital Signatures. *Groups, Complexity, Cryptology* 4 (10 2012). https://doi.org/10.1515/gcc-2012-0019

[12] Delaram Kahrobaei and Ha Lam. 2014. Heisenberg Groups as Platform for the AAG Key-Exchange Protocol. *Proceedings - International Conference on Network Protocols, ICNP* (03 2014). https://doi.org/10.1109/ICNP.2014.105

[13] Ki Ko, Sang Jin Lee, Jung Cheon, Jae Han, Ju-Sung Kang, and Choonsik Park. 2000. New Public-Key Cryptosystem Using Braid Groups. *Advances in Cryptology - CRYPTO 2000, LNCS 1880*, 166–183. https://doi.org/10.1007/3-540-44598-6_10

[14] Alex D. Myasnikov and Alexander Ushakov. 2007. Length Based Attack and Braid Groups: Cryptanalysis of Anshel-Anshel-Goldfeld Key Exchange Protocol. In *International Conference on Theory and Practice of Public Key Cryptography*.

[15] Dima Ruinskiy, Adi Shamir, and Boaz Tsaban. 2006. Length-based cryptanalysis: The case of Thompson's Group. *Journal of Mathematical Cryptology* 1 (07 2006). https://doi.org/10.1515/jmc.2007.018

[16] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 124–134. https://doi.org/10.1109/SFCS.1994.365700

[17] Vladimir Shpilrain and Alexander Ushakov. 2004. The Conjugacy Search Problem in Public Key Cryptography: Unnecessary and Insufficient. *Applicable Algebra in Engineering, Communication and Computing* 17 (12 2004). https://doi.org/10.1007/s00200-006-0009-6

[18] The Sage Developers. 2022. *SageMath, the Sage Mathematics Software System (Version 9.7)*. `https://www.sagemath.org`.

[3] See Section 2.1 and related footnote.