INTRODUCING GRAFHEN: GROUP-BASED FULLY HOMOMORPHIC ENCRYPTION WITHOUT NOISE

PIERRE GUILLOT [1], AUGUSTE HOANG DUC [2], MICHEL KOSKAS [3], AND FLORIAN MÉHATS [4]

Ravel Technologies, Paris

Abstract. We present GRAFHEN, a new cryptographic scheme which offers Fully Homomorphic Encryption without the need for bootstrapping (or in other words, without noise). Building on the work of Nuida and others, we achieve this using encodings in groups.

The groups are represented on a machine using *rewriting systems*. In this way the *subgroup membership problem*, which an attacker would have to solve in order to break the scheme, becomes maximally hard, while performance is preserved. In fact we include a simple benchmark demonstrating that our implementation runs several orders of magnitude faster than existing standards.

We review many possible attacks against our protocol and explain how to protect the scheme in each case.

An appendix describes an expert's attempts at cracking the system using the Todd-Coxeter algorithm.

Important note: this version was updated to take into account the attack proposed by Jules Dumezy (which is the first documented, successful attack on our scheme.) The recommended parameters have been altered. The new values seem to protect GRAFHEN against the attack; this is currently under investigation.

1. Introduction

Fully Homomorphic Encryption (FHE) has emerged as a cornerstone of modern cryptography, enabling computation on encrypted data without requiring access to the plaintext. Since Gentry's seminal 2009 construction [21], FHE has promised powerful applications in privacy-preserving computation, secure outsourcing, and confidential machine learning. However, despite its theoretical potential, practical deployment has remained limited by the high computational cost associated with managing noise in ciphertexts during evaluation.

[1] pierre.guillot@raveltech.io.
[2] auguste.hoangduc@raveltech.io.
[3] michel.koskas@raveltech.io.
[4] florian.mehats@raveltech.io.

In most FHE schemes—particularly those based on lattice problems such as Learning With Errors (LWE) [34] and Ring-LWE [28]—the encryption process deliberately injects noise into ciphertexts to ensure semantic security. As homomorphic operations are performed, this noise grows, and when it exceeds a certain threshold, the ciphertext becomes undecryptable. Thus, the ability to perform arbitrary computations is tightly constrained by the need to control noise growth.

To address this, Gentry introduced the concept of bootstrapping—a technique whereby a ciphertext is homomorphically refreshed by evaluating its own decryption circuit, thereby reducing the accumulated noise. This method enables unbounded-depth computation but at a significant computational cost, as the bootstrapping step is typically the most expensive part of the pipeline.

In response, two broad strategies have emerged to mitigate the limitations of bootstrapping. The first consists in avoiding bootstrapping altogether by developing leveled FHE schemes, such as BGV [4], BFV [3, 18], and CKKS [8, 7]. These constructions allow the evaluation of circuits of limited depth without noise refreshing, relying instead on techniques like modulus switching and key switching to control noise growth. However, their expressiveness remains fundamentally bounded by the initial noise budget and the noise growth rates inherent to the underlying operations.

The second strategy embraces bootstrapping but focuses on making it faster. Notable examples include FHEW [15], TFHE [9, 10], which introduced significant algorithmic and implementation-level optimizations to drastically reduce the cost of bootstrapping. These schemes make it feasible to perform a bootstrapping operation after every gate, opening the door to low-latency homomorphic evaluation of Boolean circuits. While this represents a substantial practical improvement, the need for frequent bootstrapping still imposes a structural complexity and performance overhead that can be prohibitive in large-scale applications. Recently, this line of work has been extended to operate on large integers, rather than just bits. In particular, [5, 6] significantly broadens the scope of bit-level bootstrapping-based approaches, bringing them closer to practical applications in encrypted numerical computing.

In summary, despite more than a decade of progress, noise remains a central challenge in homomorphic encryption. Existing schemes must either limit the depth of supported computations or incur the cost of frequent bootstrapping. Bridging this gap—enabling arbitrary-depth computation without the burden of noise management—remains a key objective in the ongoing development of efficient, fully homomorphic encryption.

As Gentry emphasized in his 2022 ICM survey [22], the notion of a truly noise-free FHE—where ciphertexts remain decryptable after arbitrary computations without any need for noise control—remains largely unfulfilled. He notes that if such a scheme were to exist, it would likely look very different from today's lattice-based frameworks. Some speculative directions include building FHE from multilinear maps [2], which generalizes bilinear pairings and would allow for direct composition of encrypted functions without error accumulation. Another avenue involves using non-solvable group structures [32], where homomorphic operations might be encoded through group actions that naturally avoid noise growth. However, both approaches remain largely theoretical and face major barriers, including unresolved issues of security, functionality, or concrete efficiency.

In this paper, we build on the work of Nuida in [32], which in turn relied on pioneering work by Ostrovsky *et al.* [33]. A fundamental observation, which we describe at length in §2, is that fully homomorphic encryption is possible if we can homomorphically encrypt the elements of a carefully chosen group E. Thus the encrypted messages are elements of another group G, and the decryption map is a group homomorphism to E. The challenge is to choose the group G (together with a machine implementation) so that the resulting scheme is secure. The fundamental problem which needs to become hard for an attacker and remain easy for the user is the *subgroup membership problem*, that is, the problem of deciding whether an element $g \in G$ belongs to a given subgroup Z or not. This depends drastically on the representation of G, and is trivial for permutation groups for example.

Finding the right type of group was an open question at the end of [32], and we propose a solution, which is at the heart of our cryptosystem named GRAFHEN, for GRoup-bAsed Fully Homomorphic Encryption without Noise. The basic idea is to work with *rewriting systems* (the theory of which is summarized in §3). Given a choice of generators $x_1, \ldots, x_d$ for a group G, a rewriting system is a collection of rules of the form $u \longrightarrow v$ where u and v are words on the alphabet $\{x_1, \ldots, x_d\}$, which hold true in G, and allow us to rewrite any word and obtain a "simpler" one representing the same element of the group. Traditionally this is used to obtain canonical forms for the elements of the group; for us however, the rules are chiefly a way of obtaining words of bounded length (the rules $u \longrightarrow v$ being chosen with v shorter than u).

Concretely, the generators $x_1, \ldots, x_d$ are to remain secret, and constitute the private key. The rules are public. Users work with words and rules. They have only access to the group $\tilde{G}$ defined "by generators and relations" by the alphabet and the rules, and the subgroup membership problem is known to be hard, in fact undecidable in general, for such groups. For more about this, see the beginning of §6.

We shall review the main attacks on our scheme, and argue that with sensible choices of parameters, it becomes fully secure. We also include an appendix, written by James Mitchell who is a specialist in computational group theory (and aguably the best expert in the world on the specific task at hand), in which he describes his attempts at breaking GRAFHEN: while this can be done for tiny values of the security parameters, it becomes rapidly impossible. Thus we offer a secure protocol for fully homomorphic encryption without noise (or bootstrapping).

Here we want to address a point. GRAFHEN messages are words, which need to be shortened every now and then using the rules. We would like to insist on the differences between this rewriting and the usual bootstrapping. First, on a theoretical level, a major difference is that the bootstrapping operation to refresh the ciphers *must be performed* after a certain number of homomorphic operations were made *even on a idealized computer with infinite memory* – otherwise the messages are lost. By contrast, we only need to shorten the messages when we feel the pressure on the memory is too great – and in situations where memory usage is not critical, but speed is, we may decide not to perform the reductions for a while (and perhaps wait for the CPU to be idle to do this). In fact, the operation is one of *lossless data compression*.

Another difference is one of performance. Rewriting words does not involve arithmetic operations, only some string matchings and replacements which can be performed efficently (using automata for example); a bootstrapping may involve millions of multiplications between floating point numbers. We include a benchmark in §8 which shows that GRAFHEN is several orders of magnitude faster than OpenFHE, for example.

Acknowledgements. We are grateful to Florent Hivert and James Mitchell for many discussions on questions of security. James Mitchell has also written Appendix B, in which he reports on his attempts to break the scheme we propose here. We are also indebted to Jean-Baptise Rouquier for his careful proofreading and many suggestions.

In addition, we extend our thanks to Koji Nuida for his encouraging words about an early version of the manuscript, and for spotting a typo. Finally, we have benefited from the advice of David Naccache, and we would like to express our gratitude.

Some conventions. When working with permutations, we rely on some convenient conventions. We would like to see the symmetric group S_n as a subgroup of S_{n+1}. In order to achieve this, and following the custom of certain computer algebra packages such as GAP[1], we define

[1] https://www.gap-system.org

a *permutation* to be a bijection of $\mathbb{Z}$ which is the identity outside of a finite set, and S_n is the group of permutations which are the identity outside of $\{1, \ldots, n\}$. Thus $S_n \subset S_{n+1}$ as promised. An element such as (15) (which can be also written $(1, 5)$ for clarity) belongs to S_n for all $n \geq 5$. We note that permutations, such as we have defined them, are best written in cycle notation.

We also want to identify $S_n \times S_m$ as a subgroup of S_{n+m}. This we do silently. That is, we may informally write $S_n \times S_m$ to mean the group of all products ab with $a \in S_n$ and b a permutation which is the identity outside of $\{n+1, \ldots, n+m\}$. Of course this is only used in situations where no confusion can arise.

We also point out, but this will only be of consequence (perhaps) for the diligent reader who tries to reproduce all of our computations, that when a and b are permutations, we write ab for $b \circ a$ – that is, first apply a, then apply b. This is again in order to follow GAP.

Contents

1. Introduction 1
2. Homomorphic encodings in groups 6
2.1. Goals & strategy – from two operations to one 6
2.2. Homomorphic encodings – hands-on approach 9
2.3. Homomorphic encodings – some theory 10
2.4. Return to the general discussion 12
3. Rewriting systems 13
3.1. First definitions 13
3.2. The Knuth-Bendix algorithm 16
3.3. The Froidure-Pin algorithm 17
3.4. The use of rewriting systems for homomorphic encryption; pseudo-boundedness 18
4. The protocol 19
4.1. The complete protocol (private key version) 19
4.2. A toy example 20
4.3. Public-key variant 21
5. Interlude: summary of notation 22
6. Security evidence: analysis 23
6.1. Brute-force attack 24
6.2. Todd-Coxeter 27
6.3. Reducing the number of generators, first method 27
6.4. Reducing the number of generators, second method 28
6.5. On the number of ciphers; attack by random reduction 28
6.6. Using relations between ciphers 29
7. Security evidence: enhancements 30
7.1. Admissible rules 30
7.2. Semidirect products 31

7.3. Automorphisms 34
8. Recommended parameters & a benchmark 35
Appendix A. A complete example 37
A.1. Initial choices 37
A.2. Producing elements 38
A.3. Performing the homomorphic operations 40
A.4. Decryption 41
A.5. Computing the rules 41
Appendix B. Computational attacks, by James Mitchell (University of St-Andrews) 42
Appendix C. A failed attack on GRAFHEN 46
C.1. GRAFHEN does not use confluent systems 46
C.2. Mathematical mistakes 47
C.3. Comments on an updated version 48
C.4. Experimentation 50
C.5. Additional experimentation 52
C.6. Summary 53
C.7. Epilogue 53
References 54

2. Homomorphic encodings in groups

In this section we describe the strategy laid out in [32], itself building on [33]. We explain how the construction of a scheme capable of encrypting a group element and to perform the group operation homomorphically leads to a fully homomorphic encryption scheme. For this one needs to pick what we call a *homomorphic encoding*, and we explore their basic theory.

The notation introduced in this section remains in place throughout the paper.

2.1. Goals & strategy – from two operations to one.

Let $\mathbb{K}$ be a commutative ring. Many of our examples in the paper will deal with $\mathbb{K} = \mathbb{F}_2 = \{0, 1\}$, the field with two elements, but some basic principles hold more generally.

Definition 1. A *homomorphic encryption of* $\mathbb{K}$ is a map $\mathrm{Dec}\colon C \longrightarrow \mathbb{K}$, where C is a set, together with two other maps

$$\langle \mathit{add} \rangle, \langle \mathit{mul} \rangle \colon C \times C \longrightarrow C$$

chosen so that the following diagrams are commutative:

$$\begin{array}{ccc} C \times C & \xrightarrow{\langle \mathit{add} \rangle} & C \\ {\scriptstyle \mathrm{Dec} \times \mathrm{Dec}}\downarrow & & \downarrow{\scriptstyle \mathrm{Dec}} \\ \mathbb{K} \times \mathbb{K} & \xrightarrow{+} & \mathbb{K} \end{array} \qquad \begin{array}{ccc} C \times C & \xrightarrow{\langle \mathit{mul} \rangle} & C \\ {\scriptstyle \mathrm{Dec} \times \mathrm{Dec}}\downarrow & & \downarrow{\scriptstyle \mathrm{Dec}} \\ \mathbb{K} \times \mathbb{K} & \xrightarrow{\cdot} & \mathbb{K} \end{array}$$

In other words, we require $\mathrm{Dec}(\langle \mathit{add} \rangle(x, y)) = \mathrm{Dec}(x) + \mathrm{Dec}(y)$ and $\mathrm{Dec}(\langle \mathit{mul} \rangle(x, y)) = \mathrm{Dec}(x) \cdot \mathrm{Dec}(y)$ for all $x, y \in C$.

The set C (the elements of which are called ciphertexts or ciphers) and the maps $\langle \mathit{add} \rangle$ and $\langle \mathit{mul} \rangle$ are public, whereas the "decryption map" Dec is confidential. Questions of security will be discussed later, but the reader should keep in mind that we assume the presence of an "attacker" having access to many elements $x \in C$ as well as the corresponding value $\mathrm{Dec}(x)$, and that they must not be able to predict the value of $\mathrm{Dec}(y)$ for any new element $y \in C$. In other words, we expect a Chosen Plaintext Attack (CPA), and the goal is to achieve CPA security.

Remark 2. When $\mathbb{K} = \mathbb{F}_2$, we are talking about the encoding of a bit, in such a way that all possible operations on one or two bits can be performed on the ciphers. (For example the operation on one bit which exchanges 0 and 1, or in others words the NOT operation, is $x \mapsto 1 + x$, and can be performed on ciphers as $c \mapsto \langle \mathit{add} \rangle(u, c)$ where u is a cipher of 1.) However, it is well-known that any function $\mathbb{F}_2^n \longrightarrow \mathbb{F}_2$, for any $n \geq 1$, can be realized by composing binary and unary operations: algebraically, this is because any such map can be written as a polynomial, and in computer science this fact is better known as the universality of the NAND gate. Thus when we speak of a homomorphic encryption of $\mathbb{F}_2$, we imply a fully homomorphic encryption scheme. However in practice, if one has an efficient homomorphic encryption of $\mathbb{F}_p$ for some prime p, it may be largely preferable to use it rather than write all numbers in binary and so on.

There are classical examples of encryption protocols which are compatible with a single operation (such as RSA for example). In order to encrypt two operations, in the above sense, the first step is to encode $\mathbb{K}$ in a group, that is to pick an injective map $\mathrm{Enc}\colon \mathbb{K} \longrightarrow E$ where E is a group. There is no loss of generality in assuming that $\mathrm{Enc}(0)$ is the identity element e of E. (In other contexts, the identity element of a group is often denoted by 1, but here there is a risk of confusion with the element $1 \in \mathbb{K}$.)

We need some vocabulary to describe the properties which Enc must satisfy.

Definition 3. Let G be a group and let $f\colon G^n \longrightarrow G$ be a function, where n is an integer. We say that f is *polynomial* when it is constructed entirely from the group law on G; in other words, when it is of the form

$$f(x_1, \ldots, x_n) = a_0 x_{i_0} a_1 x_{i_1} a_2 \cdots a_k x_{i_k} a_{k+1}$$

for a certain integer k, where the $a_0, \ldots, a_{k+1}$ are elements of G (and the indices i_j need not be distinct).

Remark 4. In this definition we do not allow the use of the inverse operation in the group G (so that the definition could be given in a monoid). However, little change would be brought by allowing inverses: in all practical examples, the group G is finite, and for $x \in G$ of finite order s, one has $x^{-1} = x^{s-1}$. Another remark is that constant functions are polynomial (this is the formula above for $k = -1$, or you may take it as a convention if you prefer).

Definition 5. A *homomorphic encoding of* $\mathbb{K}$ in the group E is an (injective) encoding $\mathrm{Enc}\colon \mathbb{K} \longrightarrow E$ together with two polynomial maps $\mathit{add}, \mathit{mul}\colon E^2 \longrightarrow E$ such that the restriction of add (resp. mul) to $\mathrm{Enc}(\mathbb{K})$ corresponds to the sum (resp. product) of $\mathbb{K}$. In other words, we require $\mathit{add}(\mathrm{Enc}(k), \mathrm{Enc}(\ell)) = \mathrm{Enc}(k+\ell)$ and $\mathit{mul}(\mathrm{Enc}(k), \mathrm{Enc}(\ell)) = \mathrm{Enc}(k \cdot \ell)$, for all $k, \ell \in \mathbb{K}$.

We will provide many examples in the next subsection. Let us indicate at once, however, that we are going to show that $E = \mathrm{SL}_3(\mathbb{K})$ can be endowed with a homomorphic encoding of $\mathbb{K}$, for any ring $\mathbb{K}$.

The following construction is central. Assume that E is equipped with a homomorphic encoding of $\mathbb{K}$, and that L is any group on which we have defined a surjective morphism

$$\pi\colon L \longrightarrow E\,.$$

(The letter L is for "lift".) One can then construct a homomorphic encryption of $\mathbb{K}$ from this, by putting $C = \pi^{-1}(\mathrm{Enc}(\mathbb{K}))$ and $\mathrm{Dec} = \mathrm{Enc}^{-1} \circ \pi$, and lifting add and mul to maps $\langle \mathit{add} \rangle, \langle \mathit{mul} \rangle\colon L^2 \longrightarrow L$ in the natural way. In more detail, recall that add is assumed polynomial, so that we may write for $x_1, x_2 \in E$:

$$\mathit{add}(x_1, x_2) = a_0 x_{i_0} a_1 x_{i_1} \cdots a_k x_{i_k} a_{k+1}\,,$$

with $a_i \in E$. Thus it suffices to choose $\hat{a}_i \in L$ such that $\pi(\hat{a}_i) = a_i$ and to put for $x_1, x_2 \in C$:

$$\langle \mathit{add} \rangle(x_1, x_2) = \hat{a}_0 x_{i_0} \hat{a}_1 x_{i_1} \cdots \hat{a}_k x_{i_k} \hat{a}_{k+1}\,.$$

We then have $\pi(\langle \mathit{add} \rangle(x_1, x_2)) = \mathit{add}(\pi(x_1), \pi(x_2))$. Similar considerations apply to mul. It is clear that $\langle \mathit{add} \rangle$ and $\langle \mathit{mul} \rangle$ map C^2 into C, and that we are in the presence of a homomorphic encryption of $\mathbb{K}$.

This construction reduces the problem to the choice of L and π, that is, we are left with the task of encrypting the elements of E in a way which is compatible with the group law.

Remark 6 (vocabulary)**.** It is convenient to slightly abuse the language, and to call E the *group of encodings* or the *group of plaintext messages* (thus giving each element of E the status of "message", even when it is not in $\mathrm{Enc}(\mathbb{K})$). The elements of C are called *ciphers*. In most examples, C is actually a subgroup of L, and we call it accordingly the *group of ciphers* in this case. When the cipher $x \in C$ verifies

$\pi(x) = \mathrm{Enc}(k)$ for some $k \in \mathbb{K}$, or in other words $\mathrm{Dec}(x) = k$, we call x a *cipher of* k.

The ciphers of 0 are precisely the elements of the kernel of π, which will always be denoted by Z.

Remark 7. It is not truly necessary to assume that L is a group, in the considerations above, nor even a monoid: any set equipped with an internal composition law – what is called a *magma* – would suffice. We explore this possibility further when studying rewriting systems that are not confluent, which give rise to a composition law for which associativity is not guaranteed. One could similarly relax the assumptions on E, but no practical example has appeared so far.

2.2. **Homomorphic encodings – hands-on approach.** Here we give a few examples of encodings. We begin by mentioning two results which were already in the literature.

Example 8. In [33], one can find formulae for encoding $\mathbb{F}_2$ into $E = A_5$, along with the corresponding functions add and mul. In [32], the reader will find alternative expressions for the same E, as well as a variant using $\mathbb{F}_3$. □

In both cases, the number of group operations needed for add and mul is rather large. We offer the following family of examples, which works for any commutative ring $\mathbb{K}$ and uses very few operations.

Example 9. We encode the commutative ring $\mathbb{K}$ in $E = \mathrm{SL}_3(\mathbb{K})$ as follows:

$$\mathrm{Enc} : \mathbb{K} \longrightarrow \mathrm{SL}_3(\mathbb{K}), \qquad x \longmapsto \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Let us show how this encoding can be made homomorphic. We see easily that $\mathrm{Enc}(k) \cdot \mathrm{Enc}(\ell) = \mathrm{Enc}(k + \ell)$ for all $k, \ell \in \mathbb{K}$, or in other words, we can define add by $\mathit{add}(x_1, x_2) = x_1 \cdot x_2$.

We describe the multiplication next. We define the matrices:

$$h := \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \qquad g := \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Then we check that for all $k, \ell \in \mathbb{K}$:

$$g \cdot \mathrm{Enc}(k) \cdot g^{-1} = \begin{bmatrix} 1 & -k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$h \cdot \mathrm{Enc}(\ell) \cdot h^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -\ell \\ 0 & 0 & 1 \end{bmatrix}.$$

For any two elements U, V in a group, we write

$$[U, V] = U \cdot V \cdot U^{-1} \cdot V^{-1}$$

for their commutator. One verifies readily that

$$[g \cdot \mathrm{Enc}(k) \cdot g^{-1}, h \cdot \mathrm{Enc}(\ell) \cdot h^{-1}] = \mathrm{Enc}(k \cdot \ell)\,,$$

for $k, \ell \in \mathbb{K}$. In other words, we may put $\mathit{mul}(x_1, x_2) = [gx_1g^{-1}, hx_2h^{-1}]$, and mul is polynomial.

We conclude that the maps Enc, add and mul together form a homomorphic encoding of $\mathbb{K}$. □

This example comes in simple variants. Thus, one may observe that $\mathrm{SL}_3(\mathbb{K})$ can be replaced by $\mathrm{PSL}_3(\mathbb{K})$; more generally, whenever there is a normal subgroup N of E, and the composition

$$\mathbb{K} \xrightarrow{\mathrm{Enc}} E \longrightarrow E/N$$

is injective, then one may choose to work with E/N in place of E. The fact that add and mul can "induce" maps $(E/N)^2 \longrightarrow E/N$ is obvious if one keeps in mind that these functions are polynomial.

Another variant, based on another general fact: if E is a subgroup of E', and if E is endowed with a homomorphic encoding of $\mathbb{K}$, then so is E'. Again, we extend add and mul to E' by exploiting their polynomial nature. For example, as the symmetric group S_7 contains a subgroup isomorphic to $\mathrm{PSL}_3(\mathbb{F}_2)$, we deduce that S_7 can be used for the homomorphic encoding of $\mathbb{F}_2$.

With a little more work, we can even obtain:

Example 10. For practical reasons, it is useful to work with permutation groups of the smallest possible degree. We can modify "by hand" the example of S_7 just given and operate within S_6. The end result is this. Define $\mathrm{Enc}(0) = Id$ (the identity permutation) and $\mathrm{Enc}(1) = (15)(34)$. We put $\mathit{add}(x, y) = xy$ and $\mathit{mul}(x, y) = a_1xa_1a_2ya_2a_1xa_1a_2ya_2$ where $a_1 = (12)(56)$ and $a_2 = (35)$. The reader can check that this is a homomorphic encoding of $\mathbb{F}_2$ in S_6 – there are, after all, very few verifications to make. □

Remark 11. In this last example, assuming a_1a_2 is precomputed, it takes 5 group operations to obtain $\mathit{mul}(x, y) = [(((a_1x)a_1a_2)y)a_2]^2$. One can show that this optimal, at least in the following sense: a homomorphic encoding of $\mathbb{F}_2$ within the group E, with $\mathrm{Enc}(1)$ of order 2 and $\mathit{add}(x, y) = xy$, cannot be realized using 4 or less group operations for mul. This is left as an exercise (one may simply explore the different possibilities).

2.3. **Homomorphic encodings – some theory.** The passage from $\mathrm{SL}_3(\mathbb{K})$ to $\mathrm{PSL}_3(\mathbb{K})$ motivates the search for minimal groups with which we might work. We are going to investigate this in the case of $\mathbb{K} = \mathbb{F}_2$.

Lemma 12. *Assume that the group E possesses a homomorphic encoding of $\mathbb{F}_2$, with $\mathrm{Enc}(0) = e$, the identity of E, and $\mathrm{Enc}(1) = \sigma \in E$. Then σ does not belong to the centre $\mathcal{Z}(E)$ of E. It follows that the group $E/\mathcal{Z}(E)$ possesses a homomorphic encoding of $\mathbb{F}_2$.*

Proof. We proceed by contradiction and suppose that $\sigma \in \mathcal{Z}(E)$. Since $\mathrm{Enc}(\mathbb{K}) = \{e, \sigma\}$ is contained in the centre of E, and since *add* is polynomial, there exists a function $f\colon E^2 \longrightarrow E$ which agrees with *add* on $\mathrm{Enc}(\mathbb{K})$ and is of the form $f(x, y) = ax^n y^m$ for some integers n, m.

We must have $f(e, e) = e$ (since $0 + 0 = 0$ in $\mathbb{F}_2$), hence $a = e$. Then, from $f(\sigma, e) = \sigma$ we get $\sigma^n = \sigma$, so we may continue the proof assuming $n = 1$; similarly, we reduce to $m = 1$, and finally *add* coincides on $\mathrm{Enc}(\mathbb{K})$ with the function f defined by $f(x, y) = xy$. Note also that $f(\sigma, \sigma) = e$ (since $1 + 1 = 0$ in $\mathbb{F}_2$), so $\sigma^2 = 1$.

Let us now turn to the multiplication. Using similar arguments, we see that *mul* coincides on $\mathrm{Enc}(\mathbb{K})$ with a map $g\colon E^2 \longrightarrow E$ of the form $g(x, y) = bx^n y^m$ with $n, m \in \{0, 1\}$ since σ has order 2. From $g(e, e) = e$ we deduce $b = e$; from $g(\sigma, e) = e$ we get $\sigma^n = e$, so $n = 0$, and by symmetry $m = 0$. Finally, g is the constant function equal to e, which is absurd, as the relation $g(\sigma, \sigma) = \sigma$ cannot be satisfied.

The passage from E to $E/\mathcal{Z}(E)$ can then be carried out as described above. □

Corollary 13. *If E is abelian, then it does not possess a homomorphic encoding of $\mathbb{F}_2$.*

This is clear as $\mathcal{Z}(E) = E$ in this case.

Corollary 14. *Let p be a prime number and let E be a p-group. Then E does not possess a homomorphic encoding of $\mathbb{F}_2$.*

Proof. Suppose that E is a group of order p^n possessing a homomorphic encoding of $\mathbb{F}_2$, and assume that it is chosen with n minimal. The group E is nontrivial as it contains $\mathrm{Enc}(\mathbb{K})$, and its centre is then also nontrivial (a basic fact about p-groups). From the lemma we see that $E/\mathcal{Z}(E)$ also has a homomorphic encoding, even though it is a p-group of order less than that of E. This is a contradiction. □

This can be generalized immediately to nilpotent groups, and we conclude that these do not possess homomorphic encodings of $\mathbb{F}_2$.

It is natural at this stage to look at simple groups. In principle, this is an excellent idea, in view of the following theorem:

Theorem 15 (Maurer & Rhodes [29])**.** *Suppose E is a finite, simple, non-abelian group. Then any map $f\colon E^n \longrightarrow E$, for any integer n, is polynomial.*

We deduce, of course, that such an E admits many homomorphic encodings of $\mathbb{K}$ for any $\mathbb{K}$ whose cardinality does not exceed that of E.

In practice, however, even though the proof given in [29] is constructive, and may in principle be used to express any given function f in terms of the multiplication in E, the corresponding expression may involve hundreds or even thousands of multiplications, making it hardly usable in practice.

2.4. **Return to the general discussion.** Having established that examples of homomorphic encodings abound, we return to the general strategy. Assume we have chosen a group E with a homomorphic encoding of the ring $\mathbb{K}$, and we are looking for a group L with a surjective homomorphism $\pi\colon L \longrightarrow E$. The group L is to be made public, while π is the secret key. If we are to achieve CPA security, then the attacker will have access to many elements of $Z = \ker(\pi)$, in fact so many that with overwhelming probability they will have a family of generators for Z; and they must not be able to predict whether a new element $x \in L$ belongs to Z or not. In computational group theory, the *subgroup membership problem* is, precisely, that of deciding whether an element of a group belongs to a given subgroup. Thus we want to set things up so that that the subgroup membership problem is hard.

What sort of group should we employ for L? By this we ask the type of machine representation to use. One naturally turns to permutation groups when working with groups on a computer. However, the available algorithms are so powerful that the subgroup membership problem becomes trivial.

To an extent, the same applies to matrix groups over finite fields. And methods of linear algebra come into play, usually allowing easy solutions for the same problem.

Groups presented by generators and relations are more promising to us, with most tasks being very difficult to perform. However, we still need to be able to store group elements easily, and to multiply them efficiently. For this, we propose the use of *rewriting systems*, to be discussed in the next section.

Before we turn to this, we point out that we will often, though not always, have an "ambient" group G that properly contains L. It sometimes arises naturally, for example G could be a symmetric group or a general linear group. Moreover, it can be advantageous to be in a situation where the attacker, forced to work within G, is unable to identify the elements of L.

That said, in some cases one prefers to work with $G = L$. One reason is that it may be computationally more expensive to derive the "rules" for the rewriting system, as in the next section, for a larger group than L itself.

In terms of notation, in any case, at this point we have the following objects:

$$\begin{array}{ccccccc} Z & \xrightarrow{\subset} & C & \xrightarrow{\subset} & L & \xrightarrow{\subset} & G \\ & & \downarrow{\scriptstyle \pi} & & \downarrow{\scriptstyle \pi} & & \\ & & \mathrm{Enc}(\mathbb{K}) & \xrightarrow{\subset} & E & & \end{array}$$

In the discussion above we have already introduced the groups E and L and the map π; here we have also put $C = \pi^{-1}(\mathrm{Enc}(\mathbb{K}))$ and $Z = \pi^{-1}(\mathrm{Enc}(0)) = \ker \pi$. The deciphering map is $\mathrm{Dec} = \mathrm{Enc}^{-1} \circ \pi$.

The next section, on rewriting systems, is written for a generic group G, and this is consistent with the above, that is, in practice it will the group G appearing in the diagram.

3. Rewriting systems

Rewriting systems are at the heart of our protocol, and it seems useful to briefly present their theory. Chapter 12 of [25] is an excellent reference.

3.1. **First definitions.** When A is a (usually finite) set, we denote by A^* the set of words over A (that is, finite sequences of elements of A, including the so-called *empty word*, denoted by ε). We will say that A is the *alphabet* we are working with. Note that A^* is a monoid under the operation of concatenation; we will write uv for the concatenation of the words u and v.

A *rewriting system* over the alphabet A is a finite set of *rules*, i.e., pairs of words over A, with the suggestive notation $\lambda \longrightarrow \rho$ for the pair (λ, ρ). Such a rule is said to *apply* to a word of the form $u\lambda v$, giving, by definition, the word $u\rho v$.

We write $w_1 \longrightarrow w_2$ to indicate that there exists a rule in the system that applies to w_1 to produce w_2. The notation $w_1 \longrightarrow^* w_2$ indicates that there is a sequence of rule applications leading from the word w_1 to the word w_2. Finally, we write $\longleftrightarrow^*$ for the equivalence relation generated by $\longrightarrow$; in other words, $w_1 \longleftrightarrow^* w_2$ means that one can go from w_1 to w_2 by applying rules *possibly in reverse*.

More pictorially, one can construct a graph whose vertices are the elements of A^*, with a directed edge from w_1 to w_2 whenever $w_1 \longrightarrow w_2$. Then $w_1 \longrightarrow^* w_2$ means that there is a directed path from w_1 to w_2, whereas $w_1 \longleftrightarrow^* w_2$ means that w_1 and w_2 lie in the same connected component (of the underlying undirected graph).

Suppose we have a rewriting system over the alphabet $A = \{x_1, \ldots, x_n\}$, and that we are also given a monoid, which we will denote G since in the overwhelming majority of cases it will in fact be more precisely a group. Let us choose elements $\overline{x}_1, \ldots, \overline{x}_n \in G$. A word $w \in A^*$ gives an element $\overline{w} \in G$ by replacing each occurrence of x_i by $\overline{x}_i$ and multiplying in G, so that the map $A^* \longrightarrow G$ that sends w to $\overline{w}$ is a monoid homomorphism. We will assume that the $\overline{x}_i$ generate G,

so that this homomorphism is surjective. Note that $\overline{w}$ is called the *evaluation* of w (with respect to the choice of generators for G), and we shall also say that w *evaluates* to $\overline{w}$.

We say that the rewriting system is *compatible* with G [with respect to the elements $\overline{x}_i$] if the relation $w_1 \longleftrightarrow^* w_2$ is equivalent to $\overline{w}_1 = \overline{w}_2$. In particular, for every rule $\lambda \longrightarrow \rho$, we must have $\overline{\lambda} = \overline{\rho}$, or more briefly, the "rules hold in G". All the words in the connected component of w, in the sense above, therefore define the same element of G, and compatibility means that this induces a bijection between the connected components and the elements of G.

A word $w \in A^*$ is said to be *reduced* with respect to a rewriting system if no rule applies to w (one also sometimes says that w *reduces to* v to indicate the relation $w \longrightarrow^* v$). A rewriting system is said to be *noetherian* if there is no infinite directed path in the graph defined above, or more informally, if starting from any word and applying rules, one always reaches a reduced word in finite time. Noetherianity is easy to guarantee when one has chosen a total order $<$ on A^*, compatible with concatenation in the obvious sense, such that for each w, there are only finitely many v with $v < w$: it suffices to ensure that for each rule $\lambda \longrightarrow \rho$, we have $\rho < \lambda$.

For example, let us write $<$ for the *shortlex* order on A^*: the relation $v < w$ holds if v is strictly shorter than w, or if v and w have the same length and v comes before w in the lexicographic order. In most of our examples below, each rule $\lambda \longrightarrow \rho$ will satisfy $\rho < \lambda$ in shortlex order. The corresponding systems will thus be automatically noetherian, and usually we will not mention this fact.

Finally, to conclude this long list of definitions, a rewriting system is said to be *confluent* if it is noetherian and if for each word $w \in A^*$, there exists a *unique* reduced word v such that $w \longrightarrow^* v$. It is not hard to show that this additional condition is equivalent to stipulating that each connected component of the graph contains a unique reduced word.

We will also speak, though this is not standard terminology, of a *bounded* rewriting system when there exists an integer N such that every word w reduces to a word of length $\leq N$. Since the alphabet is usually finite, this is equivalent to requiring that there be only finitely many reduced words.

Example 16. On the alphabet $A = \{a, b\}$, define the rewriting system with rules:

(R1) $a^2 \longrightarrow \varepsilon$,
(R2) $b^2 \longrightarrow \varepsilon$,
(R3) $bab \longrightarrow aba$.

(Recall that ε denotes the empty word.) One checks by inspection that the reduced words are ε, a, b, ab, ba, aba.

Let $G = S_3$, the group of permutations of $\{1, 2, 3\}$. Set $\overline{a} = (1, 2)$ and $\overline{b} = (2, 3)$. The relations $\overline{a}^2 = e$, $\overline{b}^2 = e$, and $\overline{b}\overline{a}\overline{b} = \overline{a}\overline{b}\overline{a}$ hold in G (where e is the identity element of G); in the notation above, it follows that $w_1 \longleftrightarrow^* w_2$ implies $\overline{w}_1 = \overline{w}_2$.

Moreover, the 6 reduced words above, when evaluated in G, yield the 6 elements of the group. This shows that these words belong to distinct connected components. In other words, the rewriting system is confluent. At the same time, we have established that the system is compatible with G. □

Example 17. To make a variant, note the relation $(\overline{a}\overline{b})^3 = e$ in G. This suggests replacing (R3) with:

(R3') $(ab)^3 \longrightarrow \varepsilon$.

The resulting rewriting system is still compatible with G – this is rarely the problematic point, and we leave it as an exercise. (The key point is that the three relations $\lambda = \rho$, where $\lambda \longrightarrow \rho$ is one of the three rules, constitute a monoid presentation for G. That this implies compatibility is a relatively straightforward general fact.)

For now, we want to emphasize the differences between the two systems, and in particular the fact that the second example is not confluent. Indeed, the reduced words are now ε, a, b, ab, ba, aba, bab, $abab$, $baba$, $ababa$, $babab$, $bababa$, so there are 12 of them, whereas compatibility with G shows that there are only 6 equivalence classes. On the other hand, the system is still bounded. □

Example 18. Still with the group $G = S_3$, one might be tempted to introduce $\overline{r} = \overline{a}\overline{b}$, which satisfies $\overline{a}\,\overline{r}^2 = \overline{r}\,\overline{a}$, as well as the rewriting system defined over the alphabet $\{a, r\}$ by the rules

(R1) $a^2 \longrightarrow \varepsilon$,
(R2) $r^3 \longrightarrow \varepsilon$,
(R3) $ar^2 \longrightarrow ra$.

This system is still compatible with G. A remarkable feature is that the words $(ar)^n$, for $n \geq 1$, are all reduced: there are infinitely many reduced words, and the system is not even bounded (let alone confluent). □

A major problem is that of finding, given a group or monoid G, a bounded rewriting system which is compatible with G. Before reviewing the possible approaches, let us cite a result from [35] (see Proposition 2.7 in particular):

Theorem 19. *Let G be a monoid, let $\overline{x}_1, \ldots, \overline{x}_n \in G$ be generators, and let us use the alphabet $A = \{x_1, \ldots, x_n\}$. Fix an appropriate order $<$ on A^*. Then there exists a unique reduced, confluent rewriting system on this alphabet which is compatible with G, and such that each rule $\lambda \longrightarrow \rho$ satisfies $\rho < \lambda$.*

Here a rewriting system is called "reduced" when for each rule $\lambda \longrightarrow \rho$, neither λ nor ρ contains the left hand side of *another* rule. Also note that this theorem allows rewriting systems with an infinite number of rules, but the system mentioned is finite if G is finitely presented (in particular, if G is finite).

This result is not constructive and does not provide a way of writing down the rules (for this, read the next subsections). However, the uniqueness statement is very convenient, allowing us to speak of "the" rewriting system defined by the choice of generators of G (for an order which is understood, usually shortlex).

3.2. **The Knuth-Bendix algorithm.** This celebrated result must be mentioned in any survey of the theory of rewriting systems, even though in practice it cannot be applied successfully in the examples of interest to us. Thus we remain brief. In fact, we only state a particular case.

Theorem 20. *Resume the setting of Theorem 19, with G finite. Suppose we have a presentation for G as a monoid, given by a finite list of relations $\lambda_i = \rho_i$, $1 \leq i \leq r$, where λ_i and ρ_i are words on the alphabet A. Assume $\rho_i < \lambda_i$ for each index i, and consider the initial rewriting system given by the rules $\lambda_i \longrightarrow \rho_i$.*

Then there exists an algorithm which terminates in finite time, taking the rules $\lambda_i \longrightarrow \rho_i$ as input, and returning the reduced, confluent rewriting system which is compatible with G.

Example 21. Let us return to Example 17, with its three rules (R1), (R2) and (R3'). Certainly the relations $a^2 = \varepsilon$, $b^2 = \varepsilon$ and $(ab)^3 = \varepsilon$ form a presentation for S_3 (as a monoid or as a group). If we feed these three rules to the Knuth-Bendix algorithm (using the shortlex order), it returns the three rules of Example 16. To give an idea of what the algorithm does, it will examine the left hand sides *ababab* and *bb* and make them overlap, forming the word *abababb*; the latter can be reduced to *ababa* using (R2), and to *b* using (R3'), so the algorithm "discovers" the relation $ababa = b$ and adds the rule $ababa \longrightarrow b$. Reducing *ababaa* in two different ways gives the rule $abab \longrightarrow ba$. Finally, reducing *aabab* in two different ways, we find $bab \longrightarrow aba$ as requested.

Example 22. Consider $G = S_n$, the symmetric group of degree n. We choose the generators $\sigma_i = (i, i+1)$ for $1 \leq i < n$. We use the very well-known presentation given by the relations $\sigma_i^2 = \varepsilon$ for all i, $\sigma_i\sigma_j = \sigma_j\sigma_i$ when $|j - i| \geq 2$, and $\sigma_{i+1}\sigma_i\sigma_{i+1} = \sigma_i\sigma_{i+1}\sigma_i$ for $1 \leq i < n - 1$.

Working with the shortlex order (and of course $\sigma_i < \sigma_{i+1}$), we can use the Knuth-Bendix algorithm on the rules $\sigma_i^2 \longrightarrow \varepsilon$, $\sigma_j\sigma_i \longrightarrow \sigma_i\sigma_j$ when $j > i + 1$, and $\sigma_{i+1}\sigma_i\sigma_{i+1} \longrightarrow \sigma_i\sigma_{i+1}\sigma_i$.

When $n = 3$, the algorithm returns the set of rules as such, and we recover the previous example. For $n > 3$ however, there are $n^2 - 2n + 1$

rules: the unmodified $\sigma_i^2 \longrightarrow \varepsilon$, $\sigma_j\sigma_i \longrightarrow \sigma_i\sigma_j$ when $j > i+1$, and

$$\sigma_i\sigma_{i-1}\cdots\sigma_j\sigma_i \longrightarrow \sigma_{i-1}\sigma_i\sigma_{i-1}\cdots\sigma_j$$

whenever $j < i$. The first mention of this in print seems to be Le Chenadec's PhD thesis [27].

This example is exceptionally simple. For $n = 10$, there are less than 100 rules, while the rewriting system corresponding to a random choice of generators for S_{10} has typically several millions of rules.

3.3. **The Froidure-Pin algorithm.** This is an alternative algorithm for computing the reduced, confluent rewriting system corresponding to a choice of generators in a finite monoid G (for shortlex, as usual), first introduced in [19].

It is very easy to summarize the algorithm. Suppose we have computed the list of all reduced words of length $\leq n$, as well as the set of all rules of the form $\lambda \longrightarrow \rho$ with the length of λ also $\leq n$ (for $n = 0$ we have the empty word and no rules). We examine each reduced word w of length n, and form wa where a is a single letter; we arrange for the words wa to come out in lexicographic order. If some rule applies to wa, we discard it. Otherwise, we evaluate $\overline{wa} \in G$ and check if it is of the form $\overline{u}$ for some reduced word u already computed; if so, we have discovered the rule $wa \longrightarrow u$, if not, we have a new reduced word wa. The algorithm terminates if we have not found a single new reduced word of length $n+1$. This will eventually happen, since we assume that G is finite and there are, at any time, fewer reduced words than elements of G.

Example 23. Let us return to Example 18 and run the Froidure-Pin algorithm with $\overline{a} = (12)$ and $\overline{r} = (1, 3, 2)$ (since we have an initial set of rules, we could equally well have run the Knuth-Bendix algorithm on them, of course, but note that for Froidure-Pin the input is really just the generators). The output is:

- $a^2 \longrightarrow \varepsilon$
- $r^3 \longrightarrow \varepsilon$
- $ara \longrightarrow r^2$
- $ar^2 \longrightarrow ra$
- $rar \longrightarrow a$
- $r^2a \longrightarrow ar$.

□

For our purposes, it is of the utmost importance to be able to work with *random* generators, and compute the corresponding rules. In this case, the Froidure-Pin algorithm tends to give better results than the Knuth-Bendix algorithm.

Moreover, we can decide to stop the Froidure-Pin algorithm at any point, in particular when the rewriting system appears to be bounded – perhaps in some weak, probabilistic sense, which is checked empirically.

Example 24. We pick the following generators for S_9 randomly, using GAP: $\overline{a} = (1,5)(2,4,8,7,9,3,6)$ and $\overline{b} = (1,7,9,3)(2,5,6)$. The Froidure-Pin algorithm computes very rapidly the $104,110$ rules which constitute the complete rewriting system. The longest left hand side of a rule is of length 22.

However, with only $6,325$ rules, of length no more than 17, we obtain a rewriting system which appears bounded for all practical purposes (we comment on this phrase below).

Finally, it is possible to mix the two approaches, letting Froidure-Pin run for a while, and then applying Knuth-Bendix to get the remaining rules.

3.4. **The use of rewriting systems for homomorphic encryption; pseudo-boundedness.** We want to bridge explicitly the material from the previous section with the present considerations.

Suppose we have picked a group G, with a subgroup L as in the previous section, itself equipped with a homomorphism $\pi\colon L \longrightarrow E$. The idea is to pick random generators for G, and to compute a corresponding rewriting system. The latter is made public. If we let $\mathscr{G}$ denote the set of reduced words, with composition law "concatenate-then-reduce", then we obtain a magma with a homomorphism $\varphi\colon \mathscr{G} \longrightarrow G$.

When the rewriting system is confluent, the magma $\mathscr{G}$ is a group and φ is an isomorphism. However, if we stop the Froidure-Pin algorithm before it is completed, then $\mathscr{G}$ is certainly not a group (the composition law which we have defined is not associative).

We will not insist on mentioning $\mathscr{G}$. It seems much simpler, in practice, to deal with words and keep in mind that they represent elements of G. At any point, we may decide to reduce a word using the rules, typically because we want to shorten it, and this does not change the underlying element of G being represented. (Every now and then we may abuse the language and speak, say, of an element of L to mean a word which represents an element of L; we do this below.)

The attacker sees many words which represent elements of Z, or of $C \smallsetminus Z$, in the notation of §2.4. Their task is to predict, given a new word which is known to represent an element of C, whether it represents an element of Z (breaking the scheme is obviously equivalent to being able to identify the ciphers of 0).

We can now expand on what we mean by a rewriting system which "appears bounded for all practical purposes". The reason we need the rules at all is that we want to store the elements of G with a finite, indeed bounded, amount of memory, so each element of G should be

representable by a word whose length is less than some absolute constant. However, we can relax this a little bit. In practice, we are going to see elements of L, and only their images under π are of importance, and we may multiply them by elements of $Z = \ker \pi$ without changing their decryption. Very often, if we do this repeatedly and reduce the result with the rules every time, we can find very short words for the ciphers *even if the rewriting system is not strictly speaking bounded*. In practice, we check that the reduced words appear to be of bounded length by performing some reductions and concatenations at random, and this works well (see Appendix A for implementation details). We say that a rewriting system is *pseudo-bounded* when it passes our test. The process of finding a shorter word using multiplications by elements of Z we call *randomized reduction*.

4. The protocol

We present our entire protocol, for the homomorphic encryption of the elements of a commutative ring $\mathbb{K}$, in the standard fashion. Note that, as presented, the scheme allows many variants; to clarify things, we present a possible choice at every step, with some parameters kept generic. This is then illustrated in two ways. First, we provide a toy example, with concrete values and all computations done. The latter is of course too small to be secure, but we think its simplicity will help the reader understand the general case. Second, we have exposed many implementation details in Appendix A.

4.1. **The complete protocol (private key version).**

4.1.1. *Encoding.* We pick a group E and a homomorphic encoding of $\mathbb{K}$ within E.

> We proceed as in Example 10, which we reproduce here for convenience. We have $\mathbb{K} = \mathbb{F}_2$, $E = S_6$, we define $\mathrm{Enc}(0) = Id$ and $\mathrm{Enc}(1) = (15)(34)$. We put $\mathit{add}(x, y) = xy$ and $\mathit{mul}(x, y) = a_1xa_1a_2ya_2a_1xa_1a_2ya_2$ where $a_1 = (12)(56)$ and $a_2 = (35)$.

4.1.2. *Key generation.* We choose a group G containing a subgroup L which is endowed with a group homomorphism $\pi\colon L \longrightarrow E$. We pick an integer d and a set of d generators $\overline{x}_1, \ldots, \overline{x}_d$ for G; these generators constitute the private key. Next, we compute the rules of the corresponding rewriting system, for example using the Froidure-Pin algorithm.

> Resuming the example, we pick $G = S_n$ for some $n \geq 6$, and $L = S_6 \times S_{n-6}$ (here S_{n-6} is seen as acting on $\{7, 8, \ldots, n\}$). The map π projects onto the first factor.

4.1.3. *Encryption.* To encode $k \in \mathbb{K}$, pick an element $x \in L$ such that $\pi(x) = \mathrm{Enc}(k)$. Then solve the word problem in G is order to write x as a product $x = \overline{x}_{i_1} \cdots \overline{x}_{i_k}$. Construct the word $w = x_{i_1} \cdots x_{i_k}$.

Optionally, reduce w using the rewriting rules. The word w is the encoding of x.

> In the current example, put $x_1 = (15)(34) \in G$. To obtain x as above, we pick any $z \in Z = \{1\} \times S_{n-6}$ uniformly at random and put $x = zx_1$. Note that the word problem is easily solved in a permutation group, using software such as GAP. Please also see Appendix A for improvements.

We note that the functions add and mul, which are part of the encoding, involve some constants taken from G – in the discussion after Definition 5, these are called a_i. It is necessary for the homomorphic operations (see below) to compute, exactly as above, an element $\hat{a}_i \in L$ such that $\pi(\hat{a}_i) = a_i$, and to write it as a word w_i in the generators. This word may optionally be reduced, and is made public.

> In our example we have just $a_1 = (12)(56)$ and $a_2 = (35)$, we can take $\hat{a}_i = a_i$ since E is identified with a subgroup of L here, and two words w_1 and w_2 are computed and made public.

4.1.4. *Decryption.* To decrypt the word $x_{i_1} \cdots x_{i_k}$, compute the product $g = \overline{x}_{i_1} \cdots \overline{x}_{i_k}$ in G. Things have been arranged so that this is an element of L, to which π may be applied. Then $\pi(g) = \mathrm{Enc}(k)$ for some $k \in \mathbb{K}$, and the original word decrypts to k. We write $\mathrm{Dec}(w)$ for the decryption of the word w.

> Back to our example again, the element g belongs to $S_6 \times S_{n-6}$, and its decryption is 0 if it fixes the points $1, 2, 3, 4, 5, 6$, while it is 1 if its restriction to this set of six points gives the permutation $(15)(34)$. In the end we see that it is enough to compute $g(1)$ which is 1 or 5 according as the decryption of the message is 0 or 1.

4.1.5. *Homomorphic operations.* The functions add and mul are polynomial, so are given by formulae which can be lifted to formulae on words, with w_i replacing a_i, and the basic operation being concatenation. This defines two functions $\langle \mathit{add} \rangle$ and $\langle \mathit{mul} \rangle$. When c_1 and c_2 are two ciphers, with $\mathrm{Dec}(c_1) = k$ and $\mathrm{Dec}(c_2) = \ell$, then $c_3 = \langle \mathit{add} \rangle(c_1, c_2)$ is another cipher with $\mathrm{Dec}(c_3) = k + \ell$. Likewise $\mathrm{Dec}(c_4) = k \cdot \ell$ when $c_4 = \langle \mathit{mul} \rangle(c_1, c_2)$. Optionally, we may reduce c_3 and c_4 using the rules.

> In the running example the operations on words are $\langle \mathit{add} \rangle(x, y) = xy$ and $\langle \mathit{mul} \rangle(x, y) = w_1 x w_1 w_2 y w_2 w_1 x w_1 w_2 y w_2$.

4.2. **A toy example.** We describe here a toy example, which is not secure but easily discussed. We take $n = 9$ and $d = 8$. We pick the following 8 elements of $G = S_9$ at random: $\overline{a} = (1, 7, 4, 2, 6)(3, 5, 9, 8)$, $\overline{b} = (1, 2, 3)(5, 6, 7, 9)$, $\overline{c} = (1, 3)(2, 8)(4, 9, 6, 7, 5)$, $\overline{d} = (1, 8, 3, 5, 7, 6, 2, 9, 4)$, $\overline{e} = (1, 3, 6, 2, 8, 4, 5, 7)$, $\overline{f} = (1, 2, 6, 4, 9, 8, 5, 7)$, $\overline{g} = (1, 5)(2, 9, 4, 7)(3, 6, 8)$, $\overline{h} = (1, 3, 7, 4, 8, 6, 9)$.

We check that they do generate G. Then we compute the rules. The *complete* set of rules, which we can obtain on this rather small example, has 976,242 rules such as

- $bcbdefa \longrightarrow dbgbb$,

- $gchfhcd \longrightarrow adbagcg$,
- $habhba \longrightarrow eddcdb$,
- $haghbfe \longrightarrow bdbghgg$,
- $aacecch \longrightarrow gghhhb$,
- $gahbggh \longrightarrow ddebbf$,
- $debghcf \longrightarrow fgea$,

and so on.

As will be clarified below (see §6.1), there are just $(n-6)! = 3! = 6$ ciphers for each message. In this case they are

$$\{\varepsilon,\ eeffhaf,\ ddgdfa,\ afedg,\ afcfgbf,\ bafdaf\}$$

for the message 0 and

$$\{aehbfcf,\ dhcfed,\ adhcbc,\ cachbf,\ fhabhe,\ dfbbc\}$$

for the message 1.

We can alternatively decide to stop computing the rules when the system appears pseudo-bounded (as in §3.4). This happens already with 118,451 rules. In this case we can produce hundreds of ciphers of 0, for example $baabebcfaafhc$, $aaahhfdbhf$, $afbfhcagbhhbebfbgaaecff$, $bhbfbebggehe$, $cedhadfbhcfcdbcfca$, $bagfhabdeae$, $afbefcahhbb$, $ddadbddbcfg$, $gefcefbcbcd$, $afcadecbdbebd$, $bagfhabdeae$, $ddbaghdaddgbbag$, $bbbchbcbe$, $aadeggfaadceb$, $edcafdcdhhggfhfha$, $cbecahcfdc$, $bcagcgahhefb$, $eeefbbebghc$, $fffffffffdcbahadeg$, $bbacdbbdgecdebcbefdbbcfdbgce$, $fabdgefagdhcd$, $bgccaaeebcdd$, $afdcfdahaha$, $cgdadhbc$, $cfabcfbfdbbbfbedgfb$, $bffbacdhgdgchec$, $bdfbgfafbbg$, $bddcfaheedgec$, $bbbcbbcbbcbbchbcbe$, $aebefhbdeghe$, $bfaaegahcc$, $bbbdhbbhhahfe$, $bbagfbhaecedaeg$, $bcbabaefegchc$, $dbcgbgecechc$, $cbcfbdedega$, $cadebbebbca$, $agcgbhgedbdbdbd$, ...

One way to produce such extra ciphers is to start with an initial batch, obtained as described above, and to multiply them together, optionally reducing them. This is similar to the "public-key variant" discussed below.

A word about the size of messages: a random word of length 10,000 can be reduced to one of length 12, on average.

4.3. **Public-key variant.** As with any homomorphic encryption scheme, it is easy to turn a private-key protocol into a public-key protocol. One solution (but the reader may envisage many variants) is as follows. Say $\mathbb{K} = \mathbb{F}_p$, the field with p elements, where p is a prime. First, we construct a large database D of ciphers of 0 and make it public. To create a new cipher of 0, pick a collection of elements of D at random and apply to them the operation $\langle \mathit{add} \rangle$ repeatedly; then, reduce the resulting word.

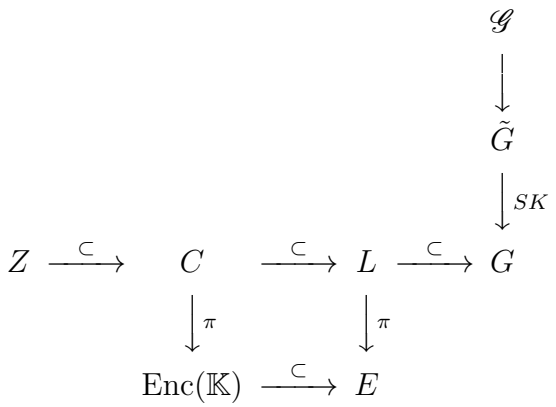
We also compute a single cipher of 1, say w, and make it public. To create a cipher of k mod p, apply $\langle \mathit{add} \rangle(w, -)$ repeatedly $(k-1)$ times, and call the result w_k. Then compute a cipher of 0, says w_0, as above.

Then the word $w_k w_0$ is a random cipher of k, and again we reduce it using the rules. We point out that the reduction process efficiently obfuscates the way a word was constructed.

Remark 25. Here the reductions mentioned are mandatory, for security reasons. Indeed, a word which is a concatenation of public ciphers can be decrypted immediately, while the reduction process will obfuscate the origin of the ciphers thus obtained.

5. Interlude: summary of notation

The main objects of interest in the paper appear in the next diagram:



Here is a short description:

- E is a group with a homomorphic encoding of the commutative ring $\mathbb{K}$. Typically $\mathbb{K} = \mathbb{F}_2$ and $E = S_6$.
- G is a group, and L is a subgroup which maps onto E. So far we have suggested $G = S_n$ and $L = S_6 \times S_{n-6}$, while later sections of the paper suggest $G = S_n \rtimes S_n$.
- $C = \pi^{-1}(\mathrm{Enc}(\mathbb{K}))$ and $Z = \pi^{-1}(\mathrm{Enc}(0)) = \ker \pi$. The deciphering map is $\mathrm{Dec} = \mathrm{Enc}^{-1} \circ \pi$.
- G has a rewriting system, with alphabet $A = \{x_1, \ldots, x_d\}$ and set of rules R, corresponding to generators $\overline{x}_1, \ldots, \overline{x}_d$ for G. We put $\tilde{G} = \langle A | R \rangle$, the group generated by generators and relations by A and R (this is the first place where $\tilde{G}$ is introduced in the paper).
- $\mathscr{G}$ is the set of reduced words with the concatenate-then-reduce law. As we will see, some of the attacks will be based on viewing $\mathscr{G}$ as an object existing independently. It is not a group in general, but merely a magma. There is a homomorphism $\mathscr{G} \longrightarrow \tilde{G}$ taking x_i to the element with the same name in $\tilde{G}$.
- The map from $\tilde{G}$ to G, taking x_i to $\overline{x}_i$, is called SK where the letters are suggestive of "Secret Key". Formally, we have defined the tuple $(\overline{x}_1, \ldots, \overline{x}_d)$ as the secret key; hopefully it is clear how the secret key and the homomorphism SK determine

one another uniquely, and are essentially two facets of the same thing.

6. Security evidence: analysis

In this section and the next, we provide strong evidence for the security of GRAFHEN. We start by reviewing as many attacks as possible in §6, having consulted with experts in computational group theory, notably Florent Hivert and James Mitchell. We can fend off many of these by simply choosing our parameters appropriately. To establish immunity against the other attacks, some enhancements must be brought to our basic procedure, and we turn to this in §7.

Although all the necessary notation was given in compact form in the previous section, which the reader can use as a reference, here is a more leisurely recap with extra comments. Of course we have our group G; we also have an alphabet A, and a rewriting system, that is a set of rules R for the words on A, which is compatible with G. When useful, we may write $\mathscr{G}$ for the set of reduced words, with the concatenate-then-reduce law.

The attacker sees words and rules. Thus they have access, in principle, to the group $\tilde{G}$ defined by generators and relations as $\tilde{G} = \langle A|R\rangle$. The secret key, consisting of the generators of G which the alphabet names, can be interpreted as a homomorphism $SK\colon \tilde{G} \longrightarrow G$. The latter is very often an isomorphism (for example, when the rules are computed with the Froidure-Pin algorithm which we keep running for long enough), but not necessarily, as we suggest below (for clarity, we point out that it is an easy exercise to show that SK is an isomorphism just when the rewriting system is "compatible with G" as defined in §3.1, which we often assume, but in the next section we open the door to systems with fewer rules).

On the other hand there is a surjective map $\mathscr{G} \longrightarrow \tilde{G}$; this may also be an isomorphism when the system is confluent, but this is not typical, and in general $\mathscr{G}$ is not a group or even a monoid (its law may not be associative).

As pointed out in §2.4, the security of our scheme is based on the hardness of the subgroup membership problem in a group presented by generators and relations, namely $\tilde{G}$ (the attacker must decide whether a given word belongs to the subgroup mapping to Z under SK, of which they have seen many members). This is undecidable in general: if the subgroup membership problem could be decided, the particular case of the trivial subgroup would show that the word problem is decidable in such groups, which it is not, see [31]. There are even groups with decidable word problem and undecidable subgroup membership problem, see [30]. Of course, the attacker is not required to solve the problem in general but in the particular case of $\tilde{G}$.

We distinguish two types of attacks. By an attack *on the representation* we mean, broadly, any attack which aims at mapping $\tilde{G}$ into a permutation group P (or some other type of group with solvable subgroup membership problem) in such a way that the ciphers can be told apart in P. Thus this includes any attack which leads to the discovery of the secret key itself, namely the generators of G giving access to SK. For this reason, we may speak informally of an attack *on the key* instead of an attack "on the representation".

The other category is that of attacks *on the ciphers*, by which we mean any attack that will lead to deciphering some messages, but without providing a way of trivially deciphering all of them.

We start with attacks on the representation.

6.1. **Brute-force attack.** We want to estimate the effort required for a brute-force attack on the key itself. Recall that we have an alphabet, say $A = \{x_1, x_2, \dots, x_d\}$, and the key consists of corresponding elements $\overline{x}_1, \overline{x}_2, \dots, \overline{x}_d$ of a (known) group G, which are generators. We wish to count the number of keys, and first we must agree on what we consider "truly different" keys.

Suppose $K = (\overline{x}_1, \dots \overline{x}_d)$ is a key, and that R_K is the corresponding full set of rules (as in Theorem 19, and we reserve the letter R to mean the published rules); suppose $K' = (\overline{x}'_1, \dots, \overline{x}'_d)$ is another key, with set of rules $R_{K'}$. We call the two keys *equivalent* when $R_K = R_{K'}$. Note that the attacker only has access to the set of rules (or rather, in practice, to a subset of it), and has no way of telling two equivalent keys apart. On the other hand, observe:

Lemma 26. *Assume that the subgroup membership problem has an easy solution in the finite group G (which is the case when G is a permutation group for example). Also assume that the homomorphism SK above realizes an isomorphism $\tilde{G} \cong G$. If we use a certain key for encryption, and the attacker has access to an equivalent key, then they can decrypt all messages.*

Proof. The key obtained by the attacker defines, using evaluation of words, a homomorphism $\varphi\colon A^* \longrightarrow G$. The latter induces an homomorphism $\tilde{G} \longrightarrow G$ also written φ; indeed $\tilde{G}$ is built using the public rules from the set R, and these are valid rules for the attacker's key. Then φ is onto and so must be an isomorphism because $\tilde{G}$ and G have the same order by assumption.

A word w is a cipher of 0 just when $SK(w) \in Z$, that is $w \in SK^{-1}(Z)$. This happens precisely when $\varphi(w) \in Z' := \varphi \circ SK^{-1}(Z)$.

The attacker has access to many ciphers of 0, providing generators for Z' upon applying φ. Thus the condition $\varphi(w) \in Z'$ may be checked by solving the subgroup membership problem in G. □

Naturally, if the attacker should use a key which is not equivalent to the one employed for encryption, then the rules of the rewriting system could not be applied, lest they should give absurd results. All in all, it seems reasonable to consider two keys as "truly different" just when they are not equivalent.

Then we note the following.

Lemma 27. *Assume SK is an isomorphism. The number of equivalence classes of keys is approximately*

$$\frac{|G|^d}{|Aut(G)|}\,.$$

See the proof for the sense in which this is an approximation.

Proof. The group $Aut(G)$ acts on the set of keys, that is, the set of d-tuples of elements of G generating G; the automorphism $\alpha \in Aut(G)$ takes $(\overline{x}_1, \ldots, \overline{x}_d)$ to $(\alpha(\overline{x}_1), \ldots, \alpha(\overline{x}_d))$.

It is clear that two keys in the same orbit are equivalent, for each relation (or rule) satisfied by one tuple will be satisfied by the other. Conversely, when two keys are equivalent, we may resume the notation from the previous proof to see that they are related by the action of $\alpha = \varphi \circ SK^{-1}$.

Thus the number of equivalence classes is precisely the number of orbits of $Aut(G)$ on the set of keys. We note that the action is free, for the condition $\alpha(\overline{x}_i) = \overline{x}_i$ for all indices i implies that α is the identity (as the $\overline{x}_i$'s generate G). Thus each orbit is of size $|Aut(G)|$.

As a result, the number of orbits is $r/|Aut(G)$ where r is the number of keys. Now, when d is large enough, a random d-tuple of elements of G will generate G with overwhelming probability, so that the number of keys is roughly the size of G^d. □

Remark 28. Here are some additional comments on the probabilistic argument at the end of this proof. If M is a maximal (proper) subgroup of G, then the probability that a random d-tuple of elements of G belongs to M^d is

$$\frac{|M^d|}{|G^d|} = \left(\frac{|M|}{|G|}\right)^d \leq \frac{1}{2^d}\,.$$

Thus if G possesses m distinct maximal subgroups, the probability that a random d-tuple of elements of G does not generate G (in other words, is contained in at least one maximal subgroup) is certainly less than $m/2^d$.

This is a very crude estimate, and the probability is often much lower. For example, the probability that two random elements of S_n generate S_n or A_n tends to 1 rapidly as n increases (cf. [14], [17]).

Remark 29. We have relied a couple of times on the assumption that SK be an isomorphism. This is extremely likely to be the case in

practice, as the definition of $\tilde{G}$ exposes millions of rules satisfied by the generators of G. In the unlikely event that SK should fail to be an isomorphism, however, the security would be increased. Indeed, Lemma 26 could potentially fail, so that an attacker in the possession of a key which is equivalent to the secret key might still be unable to decipher the messages.

Example 30. In the example suggested in the previous section, with $G = S_n$ with $n > 6$, we have $Aut(S_n) \cong S_n$, so that the number of truly different keys is $n!^{d-1}$.

Let us describe a little more concretely what the brute force attack might look like. The attacker is looking for generators of G, and these must satisfy the rules; that is, for each rule $\lambda \longrightarrow \rho$, we must have $\overline{\lambda} = \overline{\rho}$, where the evaluation of words is with respect to putative elements $\overline{x}_1, \ldots, \overline{x}_d$. This is really the only condition, for suppose conversely that generators of G have been found, satisfying the rules. Even if we assume that only a subset of the confluent rewriting system was made public, the remaining rules could be in theory recovered (assuming SK is an isomorphism) using the Knuth-Bendix algorithm, and from the way this algorithm works, we see that the attacker's generators would also satisfy the "missing" rules. From the uniqueness statement of Theorem 19, we see that the rules defined by the proposed generators are just the rules defined by the key, or in other words, the attacker has found an equivalent key.

Let us see how the attacker can take more concretely advantage of the fact that the original key itself is not needed. We may imagine a situation as follows: the group is $G = S_{13}$, and the attacker sees that the order of $\overline{x}_1$ must be 7 (for example they might see the rule $x_1^7 \longrightarrow \varepsilon$). We see that $\overline{x}_1$ must be a conjugate of the cycle $(1, 2, 3, 4, 5, 6, 7)$. Thus we may as well assume that $\overline{x}_1 = (1, 2, 3, 4, 5, 6, 7)$, since we know we can conjugate our generators without changing the equivalence class of the key. That is one less generator to look for, and we see how there remains $|G|^{d-1}$ choices to explore.

We also point out that the exploration is not completely of a "brute force" type, as the attacker might wish (this is only an example) to consider the order of the generators first, then they can consider the orders of the products of two generators, and so on.

Here we must point out what a drastic difference it makes when one chooses a scheme with $L = G$. In this situation, as π is now defined on all of G, the attacker will reason that only the images $\pi(\overline{x}_i)$ matter, and they will look for elements in E satisfying the rules, rather than in G. Of course E is typically much smaller than G. Also, although the details may depend on the chosen encoding, it is likely that two such d-tuples of generators of E which are related by an element of $Aut(E)$ will be equally valuable to the attacker. Thus the size of search space is

closer to $|E|^d/|Aut(E)|$. In order to secure the system, we recommend higher values of d.

6.2. **Todd-Coxeter.** Suppose Γ is a finitely presented group, and Λ is a subgroup given by a list of generators. Assume that Λ has finite index in Γ. Then the Todd-Coxeter algorithm computes in finite time the action of Γ on Γ/Λ (a permutation is given for each generator of Γ). See Chapter 5 in [25].

The attacker may wish to apply this with $\Gamma = \tilde{G} = \langle A|R\rangle$, the group generated by the alphabet A subject to the relations given by the set of rules R. The attacker can pick a subgroup Λ at random, and if Todd-Coxeter completes in a reasonable time, they get a homomorphism from Γ into a symmetric group. With any luck, this homomorphism is injective (this is very likely to happen with a group such as $G = S_n$ which possesses so very few normal subgroups, given that one often has $\tilde{G} \cong G$). In this case, the attacker has a representation of $\tilde{G}$ as a permutation group, in which the subgroup membership problem is easy to solve, and the scheme is broken.

It is notoriously difficult to estimate the complexity of the Todd-Coxeter algorithm. At the very least, we can say that the bottleneck is usually the amount of necessary memory, which can become enormous. Also, the greater the order of Γ, and the greater the index of Λ in Γ, the more difficult it will be to see Todd-Coxeter run to the end (in principle it can run with Γ infinite, as long as Λ does have finite index in Γ). Thus we protect our scheme against such attacks by choosing G large enough; we may additionally arrange the choice of generators so that it is more difficult to exhibit a subgroup of small index. We recommend using stress tests.

The Todd-Coxeter can be used in a different way, and this is our first exploration of an attack on the ciphers (a more technical explanation will be given in Appendix B). Modern implementations of the algorithm can stop before the calculation is complete, and produce useful intermediate data about the action. In particular, a sort of reduction process becomes possible: given an element of Γ written as a word w, a shorter word w' can be produced, such that w and w' act in the same way on Γ/Λ, or at least such that $w\Lambda = w'\Lambda$. If w' is the empty word, we deduce that $w \in \Lambda$. Thus if the attacker tries this with $\Lambda = SK^{-1}(Z)$, the group generated by the ciphers of 0, then membership in Z can be proved in some cases. This attack works well with small groups, and again, we protect the scheme against it by choosing G large enough. Again, see Appendix B for more.

We consider a couple of extra attacks on the representation.

6.3. **Reducing the number of generators, first method.** We describe two attacks which rely on an application of the Froidure-Pin algorithm to $\mathscr{G}$.

In principle, this algorithm was designed to be applied to a monoid, and the magma $\mathscr{G}$ might not be a monoid, having a composition law which might not be associative. However, Froidure-Pin will run anyway, and it will produce rules which are true for *some* order of evaluation. This turns out to be very useful.

For the first variant of this, pick two letters from the alphabet, say x_1 and x_2, and view them as elements of $\mathscr{G}$. Then run the Froidure-Pin algorithm on x_1 and x_2, and hope to find a good number of relations between them. Any rule $\lambda \longrightarrow \rho$ thus found is certainly a true rule in G when the words (in $\mathscr{G}$) are evaluated, and we have found relations between $\overline{x}_1$ and $\overline{x}_2$.

The attacker may then use brute force to find which pairs $(\overline{x}_1, \overline{x}_2)$ of elements of G satisfy the relations. They are, in some sense, back to the attack of §6.1, but crucially d was lowered to 2.

Having found candidates for $\overline{x}_1$ and $\overline{x}_2$, one may return to Froidure-Pin with x_1, x_2 and x_3, where x_3 is another letter taken from our alphabet. One obtains relations, looks for $\overline{x}_3$ by brute force, then moves on to x_1, x_2, x_3, x_4, and so on.

6.4. **Reducing the number of generators, second method.** Here is a variant of the previous attack which can work even if the Froidure-Pin algorithm finds *no rules at all* between x_1 and x_2. Recall that, given a monoid M and generators $x_1, x_2 \in M$, the Froidure-Pin algorithm stores many words on the alphabet $\{x_1, x_2\}$ together with their evaluations, which are elements of M.

With $M = \mathscr{G}$ just as above, we end up with a collection of words using just the two letters x_1, x_2, and their evaluations in $\mathscr{G}$ which are words on the whole alphabet $A = \{x_1, \ldots, x_d\}$. With any luck, we can find a word in x_1, x_2 which evaluates to a word in x_1, x_2, x_i for some other index i, and the letter x_i is made redundant (as $\overline{x}_i$ can be expressed in terms of $\overline{x}_1$ and $\overline{x}_2$). If we have enough relations, we may find that all the generators are redundant except for the first two, and we have again reduced to $d = 2$.

Here we may point out that a more naive approach is also possible: one can inspect the *given* rules (rather than trying to produce new ones), and look for a rule of the form $\lambda \longrightarrow \rho$ where λ and ρ are not written with the same letters (perhaps with ρ using one more letter than λ for example).

All these attacks encourage us to produce rules with some care, as does the next attack (on the ciphers). The "admissible rules" that we will restrict to are described in the next section.

6.5. **On the number of ciphers; attack by random reduction.** Keeping our notation as above, the number of different "lifts" for a single message in E is the order of $Z = \ker \pi$, and the number of different ciphers for the same message is the number of reduced words

w whose evaluation is in Z. We may safely assume that the attacker will reduce any word for us, so we should not count the non-reduced words here (although they are technically valid ciphers, of course). If the rewriting system is confluent, there is just one reduced word for each element of G (and $\mathscr{G}$, $\tilde{G}$ and G are all isomorphic). In practice however, we work with non-confluent systems, and as the reader has seen in §4.2, it becomes possible to create more ciphers.

In the example suggested in §4, for which $L = S_6 \times S_{n-6}$, we have $Z = \{1\} \times S_{n-6}$ so that there are at least $(n-6)!$ ciphers for each message, but possibly many more.

It is generally not hard, in concrete examples, to show that the number of ciphers of 0 is in fact infinite: it suffices to find (by random search) a word w which is a cipher of 0, is reduced, has length greater than the left hand side of any rule, and is such that w^2 is also reduced. In this case all the words w^n are ciphers of 0, and none can be reduced.

Does this improve security in a serious way? The answer is "no" without extra effort of our part. The attacker in this situation will use *randomized reduction*: as explained in §3.4, the idea is to combine any cipher c with a random cipher of 0, say z, thus forming $c' = \langle \mathit{add} \rangle(c, z)$ which is another cipher with the same decryption as c; then c' is reduced using the rules, and kept as a replacement if it is shorter that c. The process is then repeated.

Randomized reduction reduces very efficiently the length of ciphers. In the case where we have artificially a large numbers of ciphers for each message because the rewriting system is not confluent, the "extra" ciphers are long words and will be filtered out. To give an idea, in the example of §4.2, in which Z has order 6 (which is ridiculously small), the randomized reduction of a cipher of 0 produces the empty word after just a few tries (and of course the attacker will recognize the empty word as a cipher of 0).

We conclude that the order of Z must be large enough to guarantee the existence of sufficiently many ciphers for each message, each of roughly the same length.

However, this is not the end of the story. In the next section, we explain how one can choose the rules in a certain way, thus departing from the usual Froidure-Pin algorithm, to prevent random reduction from breaking the cipher (it may still be used conveniently to reduce exceedingly long ciphers, as one occasionally encounters, but will not produce very short ones).

6.6. **Using relations between ciphers.** The next attack is a very generic one: almost any scheme based on groups à la Nuida may be attacked in this way. Thus it must be taken very seriously. In fact the only assumption is that the map Dec should be a homomorphism, with values in the additive abelian group $\mathbb{K}$ – more concretely, this means

that when the homomorphic encoding was chosen, the homomorphic addition $\langle \mathit{add} \rangle$ was taken to be $\langle \mathit{add} \rangle(x, y) = xy$, and this is extremely typical.

Pick a cipher y with $\mathrm{Dec}(y)$ known, and let x be another cipher. Now run the Froidure-Pin algorithm on x and y (this is in the group or monoid at hand, so for us it is in $\mathscr{G}$). Any relation obtained between x and y gives a relation in the ring $\mathbb{K}$, and it may allow us to express $\mathrm{Dec}(x)$ in terms of $\mathrm{Dec}(y)$.

For example, suppose $\mathbb{K} = \mathbb{F}_2$ and y is a cipher of 1. Assume we have discovered the relation $yxy^2 = x^2y$. Then $\mathrm{Dec}(y) + \mathrm{Dec}(x) + 2\,\mathrm{Dec}(y) = 2\,\mathrm{Dec}(x) + \mathrm{Dec}(y)$, which simplifies to $\mathrm{Dec}(x) = 0$, revealing the deciphered value of x.

One protection against this attack is to use for L a group which possesses many different homomorphisms to $\mathbb{K}$: if we stick to $K = \mathbb{F}_2$, this would mean picking L of the form $L = \mathbb{F}_2^m \times H$ for some group H, and making sure that the first m "coordinates" of any cipher (of either 0 or 1) are random. Thus if there exists, for each pair (x, y) of ciphers, a homomorphism from L to $\mathbb{F}_2$ mapping the pair to any element of $\mathbb{F}_2^2$, it will be impossible to tell what Dec does by the above method. In other words: the attack described is based on the hope that any homomorphism $L \longrightarrow \mathbb{F}_2$ at all is strongly constrained, and we work against that.

Another way to protect the scheme is to choose the rules in such a way that it is very hard to find any relation between x and y. We turn to this now.

7. Security evidence: enhancements

In the previous section, we have explained how to protect our scheme against all attacks envisaged, with the exception of those presented in §6.3 and §6.4. In §7.1 below, we explain how to select the rewriting rules in such a way that immunity is guaranteed. Moreover, these "new" rules also offer alternative (and better) protection against the attack of §6.6, and also against the random-reduction of §6.5; this protection is so effective that the requirement that Z be large can be largely ignored.

Moreover, in §7.2, we describe a simple method to create rewriting systems for larger groups. This is important for security, as we have seen that several attacks encourage us to pick G of large order and Z of large index. This can now be achieved easily.

7.1. **Admissible rules.** The long list of attacks just given provides motivation for the following definition. A rule $\lambda \longrightarrow \rho$ will be called *admissible*, with respect to the choice of an integer k, when:

(1) both λ and ρ use the entire alphabet,
(2) both λ and ρ are of length no less than k,
(3) λ and ρ have no prefix and no suffix in common.

Remark 31. Of course the third condition is merely here because, as the rules correspond to identities in a group (for us), one could simplify any common prefix or suffix, and produce a shorter, valid rule, which in turn may not satisfy the first two conditions.

Having chosen generators $\overline{x}_1, \ldots, \overline{x}_d$ for a group G, and an integer k, we can produce a set of corresponding admissible rules (over the alphabet $\{x_1, \ldots, x_d\}$ as usual), so that the resulting rewriting system is pseudo-bounded. For this, we merely modify the Froidure-Pin algorithm so that it rejects any rule $\lambda \longrightarrow \rho$ which is not admissible (so λ is considered as a reduced word in the rest of the procedure). We need to produce more rules to reach pseudo-boundedness, but otherwise this works well in practice.

Such a rewriting system exhibits much less of the group structure of G. We can illustrate this with $\mathscr{G}$ – recall that this is our notation for the set of reduced words with the concatenate-then-reduce law. With admissible rules, there are no relations at all between the elements $x_1, \ldots, x_{d-1}$ of $\mathscr{G}$ (with the last generator x_d left out), and these generate a monoid isomorphic to that of all words on the alphabet $\{x_1, \ldots, x_{d-1}\}$. This is in sharp contrast, of course, with the structure of the finite group G.

In terms of security, restricting to admissible rules solves many problems at once. It is obvious that the attacks of §6.3 and §6.4 will not work anymore. Relations between ciphers become extremely hard to discover, and the attack of §6.6 cannot be carried out in practice. Last but not least, it turns out that the random reduction attack of §6.5 is also rendered inefficient. Perhaps surprisingly, we end up with a very large number of ciphers of 0, even with the order of Z fairly small.

7.2. **Semidirect products.** Next we present a simple trick allowing us to reach larger groups. Obviously we need our group G to be as large as possible, to fend off the brute-force attack and the Todd-Coxeter attack, but large groups require large numbers of rules.

First we present the construction in a naive way. Given a rewriting system for G on the alphabet A, with rules R_1, and another one also for G on the alphabet B with rules R_2, we try to combine them and get a larger rewriting system. We note that the rules in R_i can be interpreted as rules for words on the alphabet $A \cup B$, for $i = 1, 2$. If we want to get a bounded rewriting system with minimal effort from this, one possibility is as follows: for $a \in A$ and $b \in B$, find a word w over the alphabet A such that $\overline{w} = \overline{b}\overline{a}\overline{b}^{-1} \in G$; then consider the rule $ba \longrightarrow wb$, which involves both alphabets (and which is motivated by the fact that $\overline{b}\overline{a} = \overline{w}\overline{b}$). Write $R_1 \rtimes R_2$ for the rewriting system on the alphabet $A \cup B$ consisting of $R_1 \cup R_2$ together with all such "commutation rules". It is clear that $R_1 \rtimes R_2$ is bounded if R_1 and R_2

both are; in fact any word w on $A \cup B$ can be reduced to a concatenation uv with u resp. v a reduced word on A resp. B.

The form of the reduced words just given makes it plausible that $R_1 \rtimes R_2$ should not be "compatible" with G but with a much larger group; although we see from the outset that the larger group should have a homomorphism onto G since words on $A \cup B$ can be evaluated to elements of G, and all the rules in $R_1 \rtimes R_2$ hold true in G.

The reader who is familiar with semidirect products will guess what the "larger group" is immediately. Recall that, whenever the group H acts (on the left) on the group N by automorphisms, we can form the semidirect product $N \rtimes H$, which consists of all pairs $(n, h) \in N \times H$, with the following group law:

$$(n_1, h_1)(n_2, h_2) = (n_1(h_1 \cdot n_2), h_1 h_2) \,.$$

The following is then a simple exercise in decoding the definitions:

Lemma 32. *The rewriting system $R_1 \rtimes R_2$ is compatible with $G \rtimes G$, where G acts on itself by conjugation.*

Proof. In general there is an embedding of N into $N \rtimes H$ given by $n \mapsto (n, 1)$, whose image is a normal subgroup; there is also an embedding of H into $N \rtimes H$ given by $h \mapsto (1, h)$, whose image is not normal in general.

With this in mind, for $a \in A$ consider $\tilde{a} = (a, 1) \in G \rtimes G$, and for $b \in B$ consider $\tilde{b} = (1, b)$. This allows us to evaluate any word w over $A \cup B$ to give an element $\tilde{w} \in G \rtimes G$.

We check readily that for each rule $\lambda \longrightarrow \rho$ of $R_1 \rtimes R_2$, we have $\tilde{\lambda} = \tilde{\rho}$ in $G \rtimes G$. As a result, when two words w_1 and w_2 satisfy $w_1 \longleftrightarrow^* w_2$ for the rewriting system $R_1 \rtimes R_2$, we have $\tilde{w}_1 = \tilde{w}_2$.

Now suppose conversely that $\tilde{w}_1 = \tilde{w}_2$. We may as well assume that $w_1 = u_1 v_1$ with u_1 a word over A and v_1 a word over B; likewise assume $w_2 = u_2 v_2$. Thus $\tilde{w}_1 = (\overline{u}_1, \overline{v}_1) = \tilde{w}_2 = (\overline{u}_2, \overline{v}_2)$. This identity in $G \rtimes G$ implies $\overline{u}_1 = \overline{u}_2$ and $\overline{v}_1 = \overline{v}_2$. As we assume that R_1 is a rewriting system which is compatible with G, we have $u_1 \longleftrightarrow^* u_2$, and similarly $v_1 \longleftrightarrow^* v_2$. It follows that $w_1 \longleftrightarrow^* w_2$. □

So we obtain a rewriting system for the group $G \rtimes G$, which is much larger than G, essentially for free. To see what we can do with it, we must notice the following at once.

Lemma 33. *Let G act on itself by conjugation (on the left), and form the semidirect product $G \rtimes G$. Then $G \rtimes G \cong G \times G$. In particular, there are two distinguished homomorphisms p and f from $G \rtimes G$ onto G, defined by $p(x, y) = y$ and $f(x, y) = xy$.*

Proof. An easy proof of the first statement is obtained by considering the normal subgroup $N = G \times \{1\}$ in $G \times G$, and the "diagonal" group $\Delta = \{(g, g) \mid g \in G\}$. It is clear that $N \cap \Delta = \{1\}$ and that

$N\Delta = G \times G$, so by a basic criterion for semidirect products, we have $G \times G \cong N \rtimes \Delta$. Moreover N and Δ are both isomorphic to G and the result follows.

For a more explicit proof, consider $G \rtimes G$ and its projection $p\colon G \rtimes G \longrightarrow G$ defined by $p(x, y) = y$. Crucially, the map $f\colon G \rtimes G \longrightarrow G$ defined by $f(x, y) = xy$ is also a homomorphism. Combining p and f, we obtain a map $\varphi\colon G \rtimes G \longrightarrow G \times G$ given by

$$(x, y) \mapsto (f(x, y), p(x, y)) = (xy, y)\,.$$

It is clear that φ is an isomorphism, with inverse $(X, Y) \mapsto (XY^{-1}, Y)$. □

The map f is crucial to us. Let us describe how it works, in terms of words. In the proof of Lemma 32, we have written $\tilde{w}$ for the evaluation in $G \rtimes G$ of a word w on the alphabet $A \cup B$. Now for $a \in A$, the map f takes $\tilde{a} = (\overline{a}, 1)$ to $\overline{a}$, and likewise for $b \in B$ we see that f takes $\tilde{b} = (1, \overline{b})$ to $\overline{b}$. Let us add, much more informally, that when one works with rewriting systems one is rapidly tempted to forget the "bars" and to rely on context to distinguish a from $\overline{a}$, resulting in the observation that f maps a to a and b to b which, as it turns out, does not create confusion. When seeing a word on the alphabet A, we may think of the corresponding element of G, which can be identified with a subgroup of $G \rtimes G$ if we prefer, and likewise for words on B; when a word mixes both alphabets, the element can only be interpreted in $G \rtimes G$.

Here is how we use all this in practice. We usually start with a choice of group G for our protocol, with initial choices of L_0 and $\pi_0\colon L_0 \longrightarrow E$, in obvious notation. Then we consider $G \rtimes G$ and, for example, put $L = f^{-1}(L_0)$ and $\pi = \pi_0 \circ f\colon L \longrightarrow E$. Thus the group $Z = \ker(\pi)$ is much larger than $Z_0 = \ker(\pi_0)$ (in fact its size has been multiplied by $|G|$).

Alternatively, as we may wish to increase the *index* of Z_0 as well as its size, we may pick a smaller group for L (see the example that follows).

Then we compute two sets of rules R_1 and R_2 for G and combine them, as above, into a rewriting system $R_1 \rtimes R_2$ for $G \rtimes G$.

Example 34. Say we start with $G = S_{11}$ and $L_0 = S_6 \times S_5$, with projection map $\pi_0\colon L_0 \longrightarrow S_6$. Thus $Z_0 = \ker(\pi_0)$ has order $5! = 120$ and index $11!/5! = 332,640$. We turn to $G \rtimes G$ and consider

$$L = \{((z, x), (1, x^{-1})) \mid (z, x) \in S_6 \times S_5\}\,.$$

This is a subgroup of $G \rtimes G$ and it is endowed with the homomorphism $\pi\colon L \longrightarrow E = S_6$ given by $\pi((z, x), (1, x^{-1})) = z$. The kernel Z of π is again isomorphic to S_5 so its order is again $5! = 120$, but its index in $G \rtimes G$ is now $(11!)^2/5! = 13,277,924,352,000$ which is more than enough to protect us against an attack using Todd-Coxeter as in §6.2.

For yet another way of choosing L, see §8 below. Many variants can be envisaged.

Remark 35. Whatever the variant, the decryption process relies on the map f, and to compute its values one needs the full key, consisting of all the elements $\overline{a} \in G$ for $a \in A$ and also all $\overline{b} \in G$ for $b \in B$. The reader is invited to contemplate how the "obvious" way of creating a rewriting system for $G \times G$ from two such systems for G (by requiring letters from different alphabets to simply commute) would give rise to a protocol for which half the alphabet can in fact be ignored. The construction above truly mixes the two keys.

Remark 36. While the semidirect product construction is a very efficient protection against attacks on the ciphers, it does not increase the resistance of the scheme against attacks on the key. Indeed, if the attacker succeeds in finding the secret key corresponding to the "left" copy of G (on the alphabet A), or if they merely can represent efficiency this copy of G, then one can compute the action of the second copy of G on the first by conjugation. This is usually (depending on G) enough to break the second key.

7.3. **Automorphisms.** The last "enhancement" which we present is rather a drastic variant on our protocol. It remains a engineering challenge to work efficiently with the automorphisms to be introduced next, but on the other hand the level of security provided is very appealing.

The idea is to start with a group, which we are going to call G_0 this time, which is equipped with a rewriting system as above, and to work with $G = \mathrm{Aut}(G_0)$, the group of automorphisms of G_0. Concretely, if we have chosen the generators $\overline{x}_1, \dots, \overline{x}_d$ for G_0 (so that the corresponding alphabet is $\{x_1, \dots, x_d\}$) then an element $\alpha \in G$ can be reconstructed from the list $\alpha(\overline{x}_1), \dots, \alpha(\overline{x}_d)$. In turn, if we choose a word w_i with $\overline{w}_i = \alpha(\overline{x}_i)$ for each index i, then we see that α can be stored on a computer as the finite list of words $w_1, \dots, w_d$.

In this representation, if we want to compute $\alpha \circ \beta$ where $\alpha, \beta \in G$, we start with the words $w_1, \dots, w_d$ resp. $w'_1, \dots, w'_d$ representing α resp. β; we consider each w'_j and replace in it each occurrence of x_i by w_i, that is we form $w''_j := w'_j(w_1, \dots, w_d)$, in standard notation. Then the words $w''_1, \dots, w''_d$ represent $\alpha \circ \beta$, since

$$\overline{w}''_j = w'_j(\overline{w}_1, \dots, \overline{w}_d) = w'_j(\alpha(\overline{x}_1), \dots, \alpha(\overline{x}_d)) = \alpha(w'_j) = \alpha(\beta(\overline{x}_j))\,.$$

Briefly, composition is substitution.

For example, if we start with $G_0 = S_n$ with $n > 6$, then as is classical one has $\mathrm{Aut}(S_n) \cong S_n$, so what we have is another representation of the symmetric group on a computer. The rest of the protocol (with the choice of E and L and so on) can be carried out without changes.

The great advantage of this variant is that the subgroup membership problem seems exceedingly difficult to deal with, in a group presented

as $\mathrm{Aut}(G_0)$, with G_0 as above – in the sense that there is essentially nothing in the literature to even get us started. We seem to benefit from an excellent protection against all ciphers attacks. (Attacks on the representation of G_0 could still work, of course, and the same precautions must be taken.)

We shall not try to substantiate any security claim. For now, as this protocol uses rather large ciphers and leads to potentially heavy computations for the homomorphic operations, we are bound by practical considerations to postpone a fuller investigation.

8. Recommended parameters & a benchmark

We end the paper with a realistic example.

First, let us recall the original recommended parameters which appeared in an earlier version of this paper (these are used in some of the challenges we have made public). We shall work with $E = S_{11}$, and leave the choice of encoding open. As $S_6 \subset S_{11}$, one may choose the encoding proposed in Example 10. It is possible to do much better with S_{11}, but the exploration of efficient encodings will be the subject of a subsequent publication.

- Let $G = S_{11}$. We pick 5 generators $\overline{x}_1, \ldots, \overline{x}_5$ at random. Usually, we ensure that the subgroup generated by $\overline{x}_i$ and $\overline{x}_j$, for any pair (i, j), is all of G (otherwise the attacker has access to "easy" subgroups of G which might be fed to the Todd-Coxeter algorithm as in §6.2).
- Use our modified Froidure-Pin algorithm as in §7.1 to obtain a pseudo-bounded rewriting system on the alphabet $\{x_1, \ldots, x_5\}$ whose rules are admissible. We obtain about 20 million rules.
- Repeat the first two steps, and obtain a second rewriting system for G on the alphabet $\{y_1, \ldots, y_5\}$.
- Combine the two rewriting systems into one for $G \rtimes G \cong G \times G$, as described in §7.2.
- Put $E = S_{11}$ and $L = E \rtimes S_8$, a subgroup of $G \rtimes G$. The homomorphism $f : G \rtimes G \longrightarrow G$ described in §7.2 (which maps x_i and y_i to "the element with the same name in G") takes L into E, and this defines our map π.
- In practice, to encrypt $e \in E$, we pick $x \in S_8$ at random and form $(ex^{-1}, x) \in L$.

However, following the beautiful attack by Jules Dumezy in [16], it has become evident that we should not use for G a group of low degree (that is, acting on a finite set of small cardinality) such as S_{11}. In fact, if there are d generators and G has degree n, the proposed attack requires $O(2^{dn})$ steps in the worst-case, so that $d = 5$ and $n = 11$ is obviously not enough.

We recommend using $G = \mathrm{PSL}(2, 343)$ in lieu of S_{11}. This group G is a subgroup of S_{344}, and more importantly, it connot be embedded in S_n if $n < 344$. Also, it contains $E = \mathrm{PSL}(2, 7)$ as a subgroup (note that $343 = 7^3$), and since $\mathrm{PSL}(2, 7) \cong \mathrm{PSL}(3, 2)$, this group E is equipped with a homomorphic encoding of $\mathbb{F}_2$.

As a replacement for the subgroup S_8 appearing above, we can use the stabilizer of a point in S_{344}, that is $G \cap S_{343}$, which has index 344 in G. Everything is otherwise conducted as just described.

It seems that the attack now requires $O(2^{1,720})$ steps, although this is worst-case estimate, and further investigation is required.

We can now describe a benchmark we have run, in order to get a first idea of the performance we can expect. It is notoriously difficult to design a benchmark which is truly informative, so we will do our best to explain what is being measured.

On the one hand, we have considered OpenFHE, a popular, high-quality library which is a current standard. From the webside at https://openfhe.org/ we quote : "OpenFHE is an open-source project that provides efficient extensible implementations of the leading post-quantum Fully Homomorphic Encryption (FHE) schemes". It is a library written in C++, supporting all major FHE schemes, including the BGV, BFV, CKKS, DM (FHEW), and CGGI (TFHE) schemes.

We have used the default bootstrapping method, which is GINX (aka TFHE), with a security of 100 bits. We have homomorphically computed, for various encrypted bits (elements of $\mathbb{F}_2$) the AND operation (that is, the product).

On the other hand, we have measured the performance of our implementation of GRAFHEN, written in Rust. The parameters are chosen exactly as suggested above. We add that the messages are elements of $S_6 \subset E = S_{11}$, and in the expression (ex^{-1}, x) just given for the ciphers, we have in fact $e = my$ with $m \in S_6$ (the message) and $y \in S_5$ taken randomly (viewing $S_6 \times S_5$ as a subgroup of S_{11} as usual). The encoding is that of Example 10, so we are encrypting elements of $\mathbb{F}_2$.

We also mention that we have decided to rely only on rules $\lambda \longrightarrow \rho$ with the length of ρ strictly less than that of λ. We need to compute more rules in order to achieve pseudo-boundedness with this constraint, but this speeds up the reduction process. All in all, the particular key we have constructed involves just under 40 million rules.

To give an idea of the security afforded, we rely on Lemma 27 (and Remark 36): the number of ("truly different") keys is $11!^4$, which is about 2^{101}.

We have measured the (average) time needed to concatenate and reduce two words, which were themselves reduced. This is how one performs homomorphically the group operation of E on encrypted data.

As the encoding employed uses 5 such operations for a homomorphic multiplication (see Remark 11), we have multiplied the measured time by 5 to obtain the time needed to perform the AND operation on two bits with our implementation.

The computations were performed on a MacBook with an Apple M4 Pro CPU, with 14 cores and 48 GB of RAM. Here are the results:

- OpenFHE : $26.4ms$
- GRAFHEN : $7.56\mu s$

In this proof-of-concept implementation, GRAFHEN is thus already 3500 times faster than OpenFHE. Moreover, much more is being done to optimize the core implementation, which is built on top of an entirely novel scheme, and will benefit greatly from dedicated engineering.

Appendix A. A complete example

We will now provide a complete description of the practical steps necessary to produce a key and work with our protocol. We will include some code, and this will use Sagemath: recall that Sagemath is based on the familiar Python language, and includes many specialized packages, most notably for us the GAP system. (For readers familiar with GAP but not Sagemath, we point out that the `libgap` object in Sagemath allows access to GAP functions.)

Please keep in mind that our goal here is pedagogy, and all code is written so as to be easy to understand, and not, as will be amply evident, to be efficient.

A.1. **Initial choices.** One must start by picking a group E with an encoding. In this appendix we shall stick to Example 10, with $E = S_6$, and we are thus encoding a single bit.

Next we must choose G. Among other choices, we recommend using the symmetric group $G = S_n$, and this is what we will focus on in this appendix, so one must choose an integer n. We recommend $n \geq 11$ for security (large values of n will make it difficult to compute the rules, see below). One also needs to pick d generators for G, where the integer d has been chosen, and we recommend $d \geq 4$, again for security (large values of d will produce too many rules).

In order to pick random generators in practice, one can proceed as follows. Suppose we have:

```
n= 11
d= 5
G= libgap.SymmetricGroup(n)
```

Then the following picks d random generators, which together comprise our secret key:

```
while True:
  gens= [ G.Random() for _ in range(d) ]
```

```
  if G.Subgroup(gens) == G:
    break
```

(Keep in mind that GAP's algorithm for producing random elements of a group will always produce the same sequence of elements, so you must take care to properly initialize GAP's pseudorandom number generator with a crypto-secure seed.)

We will need names for the chosen elements. Assuming $d \leq 26$ we may as well use:

```
alphabet= "abcdefghijklmnopqrstuvwxyz"[:d]
```

We will frequently need to evaluate words in this alphabet, to recover the corresponding permutations. This can be achieved thus:

```
evaldict= dict( (letter,g) for letter,g in
      zip(alphabet,gens))
def wordev(w):
  return prod(evaldict[letter] for letter in w)
```

Thus `wordev("ab")` will return the product of the first two elements in the list `gens`.

We must also choose L and a homomorphism $\pi\colon L \longrightarrow E$. Here we shall use $L = S_6 \times S_{n-6}$, where the homomorphism π is the projection onto the first factor, while its kernel Z is $\{1\} \times S_{n-6}$.

This translates to:

```
E= SymmetricGroup(6)
L= E.DirectProduct(libgap.SymmetricGroup(n-6))
Z= L.Embedding(2).Image()
pi= L.Projection(1)
```

A.2. **Producing elements.** We need to be able to perform the following task: given $x \in E$, find $y \in L$ such that $\pi(y) = x$. Moreover, we should find a word in our alphabet which evaluates to y.

There are two ways to go about this, and it is best to use both. The first is GAP's ability to solve the word problem in a permutation group (this boils down to the Schreier-Sims algorithm). For this, we start with:

```
hom= G.EpimorphismFromFreeGroup()
```

Then `hom.PreImagesRepresentative(g)` returns, for any $g \in G$, a word which evaluates to the permutation g. Unfortunately, this word is in another alphabet, and more importantly, it involves the inverses of the generators, while we generally prefer to stick to powers of the original generators. Thus we need the somewhat awkward:

```
orders= [ int(g.Order()) for g in gens ]
def gap_word_to_string(word):
```

```
  L = [word.Subword(i,i) for i in
      range(1, word.Length()+1)]
  s = "".join(str(letter) for letter in L)
  for i in range(len(gens)-1, -1, -1):
    name = "x"+str(i+1)
    newname = alphabet[i]
    order = orders[i]
    s = s.replace(name+"^-1", newname*(order -1))
    s = s.replace(name, newname)
  return s
```

Armed with this we can define:

```
def solve_word_problem(g):
  gapword= hom.PreImagesRepresentative(g)
  return gap_word_to_string(gapword)
```

For example

```
solve_word_problem(wordev("a"))
```

should return `"a"`.

Perhaps a better example is the construction of the words needed, as described in §4.1.3, in order to perform the homomorphic operations. As already noted in that paragraph, the elements $a_1 = (12)(56)$ and $a_2 = (35)$ are trivially lifted to L since we may take $\hat{a}_i = a_i$ for $i = 1, 2$ (with the notation above, a_i plays the role of x and $\hat{a}_i$ plays the role of y). What requires work is to write these as words in the alphabet, and we now have the code for this:

```
a1= solve_word_problem(libgap.eval("(1,2)(5,6)"))
a2= solve_word_problem(libgap.eval("(3,5)"))
```

We can also use the above to produce ciphers. A cipher of 0 can be created with:

```
c0= solve_word_problem(Z.Random())
```

A cipher of 1 can be created with:

```
c1= solve_word_problem(
    libgap.eval("(1,5)(3,4)")*Z.Random() )
```

To see why the permutation $(1, 5)(3, 4)$ comes up, go back to Example 10.

We can improve on this, however. The above method for producing ciphers of 0 will always produce the same word for a given element of Z, even though we have argued in the paper that many more ciphers are available. Moreover, when translating the GAP words into usual strings, the inverses are replaced by powers of the generators, making

it easy for an attacker to guess the orders of the generators, and it is best to avoid this.

The idea is simple: as E is small enough, if we pick elements of L at random sufficiently many times, we will end up with an element mapping to any $x \in E$ given in advance; moreover, we can draw the *words* at random rather than the actual elements.

Let us start with generators for L. We have plenty of choices; for example we may rely on the fact that S_m is always generated by the "Coxeter" generators $(i, i+1)$ with $1 \leq i < m$.

```
gensL= [ libgap.eval("("+str(i)+","+str(i+1)+")")
    for i in [1..5] + [6..(n-1)] ]
```

Next we write these as words in our alphabet:

```
wordsL= [ solve_word_problem(x) for x in gensL ]
```

Here is how we can find a random word evaluating to an element $y \in E$, which in turn maps to the given $x \in E$:

```
def find_word_for(x):
  while True:
    word= ""
    for _ in range(100):
      word= word+random.choice(wordsL)
    y= wordev(word)
    if x == pi.Image(y):
      return word
```

With this, one can create a cipher of 0 with:

```
c0= find_word_for(E.One())
```

and a cipher of 1 with:

```
c1= find_word_for(libgap.eval("(1,5)(3,4)"))
```

This will produce a plethora of different ciphers.

A.3. **Performing the homomorphic operations.** Again we follow the recipe given in Example 10. We see that homomorphic addition is just concatenation:

```
def hom_add(x, y):
  return x+y
```

Homomorphic multiplication is given by:

```
a1a2= a1 + a2
def hom_mult(x, y):
  z= a1 + x + a1a2 + y + a2
  return z + z
```

A.4. **Decryption.** This is trivially achieved as follows:

```
def decrypt(c):
  g= wordev(c)
  if g == E.One():
    return 0
  else:
    return 1
```

Please note that the function `wordev` used here relies on the list `gens`, which is the secret key.

A.5. **Computing the rules.** We have indicated all the steps to produce ciphers, operate on them, and decipher them. However, it is of the utmost importance in practice to be able to reduce the size of words. So we need to compute the rules of the rewriting system.

This is achieved by the Froidure-Pin algorithm, see [19]. There is an implementation available as part of the `libsemigroups` library, written in C++, which can be interfaced with Python.

The algorithm takes as input a family of generators for a finite monoid, and also an alphabet for naming these. Its output is a set of rules, in fact the rules described by Theorem 19 for the shortlex ordering, which can be converted to a Python dictionary. For example, if called with the generators $(1, 2)$ and $(2, 3)$ for the group S_3 (seen as a monoid), with the alphabet `"ab"`, the output would be

```
{ "aa" : "", "bb" : "", "bab" : "aba"}
```

exactly as in Exemple 16.

Let us now describe how to reduce a word with respect to a set of rules. To do so effectively, one should use automata, as described in [25] (see §13.1.3 there). However, here is a toy implementation that one can use:

```
def naive_reduce(word, rules) :
  w= word
  while True:
    lhs= possible_reduction(w, rules)
    if lhs is None:
      return w
    else:
      w= w.replace(lhs, rules[lhs])
```

In turn this uses:

```
def possible_reduction(word, rules):
  for lhs in rules:
    if word.find(lhs) > -1:
      return lhs
```

```
return None
```

We have preferred to use a custom implementation of the Froidure-Pin algorithm. One reason is the ability to filter the rules, for example to exclude the rules which are too short (see §7.1 for more about this). Another reason is that we want to be able to stop the algorithm when just "enough" rules have been computed, and not wait for the complete set of rules (which might not fit into the computer's memory anyway).

By "enough" rules, we mean for the rewriting system to be pseudo-bounded, as defined (in loose terms) in §3.4. Very concretely, given a collection of rules, we perform the following test:

- pick 10 random words in the given alphabet, each of length 10,000 ;
- reduce these words using the rules ;
- compute the average length ℓ after this reduction has been performed ;
- concatenate the 10 reduced words to form w, and reduce again w using the rules, calling the result w'.

When this is done, if the length of w' is less than 3ℓ, we consider the rewriting system to be pseudo-bounded. Obviously one can try different constants here (the number 10, the length 10,000, the factor 3).

Appendix B. Computational attacks, by James Mitchell (University of St-Andrews)

In this appendix we discuss possible attacks on the ciphers related to the finitely presented groups and monoids generated according to the scheme in the paper. We only consider those attacks that permit the decryption of messages encoded using the scheme, not attacks on the key itself, which appears to be a significantly harder problem.

The input used in the attacks described in this section are:

(1) a finite monoid presentation $\mathcal{P} = \langle A|R\rangle$ (possibly defining a group) defining a monoid M;
(2) a set of words Z that encode 0;
(3) a tuple of challenge words $S = (S(0), S(1), \ldots, S(n))$ encoding 0 and 1;
(4) a tuple of solutions $T \in \{0,1\}^{|S|}$ such that the ith term $T(i) = 0$ if and only if $S(i)$ belongs to Z.

The aim of the attacks is to determine which of the words in S correspond to 0 (those belonging to the subgroup $\langle Z\rangle$ generated by Z) and which correspond to 1 (those not belonging to $\langle Z\rangle$).

The main tool used for the attacks is the Todd-Coxeter algorithm for semigroups and monoids [13] (numbers in brackets now refer to the additional bibliography which follows). This algorithm is implemented

in the C++ library LIBSEMIGROUPS [11] and was used through the Python bindings LIBSEMIGROUPSPYBIND11 [12].

In what follows we will use the language of monoids, it might be worth noting that the notion of 1-sided congruences for monoids generalises the notation of subgroups for groups. The Todd-Coxeter algorithm attempts to determine the action of the monoid M on the least 1-sided congruence ρ on M containing $Z \times 1$. There are other algorithms for computing finitely presented monoids (namely the Knuth-Bendix algorithm [26], the low index congruence algorithm [1], and several specialised algorithms such as those for the so-called *small overlap monoids*). The Knuth-Bendix and the low index congruence algorithms can, in theory at least, be applied to the presentations produced as part of the scheme in the paper, but it is widely accepted within the computational group theory community that Todd-Coxeter is the best choice for computing finite groups and monoids. Small overlap monoids are infinite, and so techniques for small overlap monoids cannot be applied here. A later version of this appendix will contain details of the other approaches also.

The action of M on ρ is constructed as a specific type of digraph, which is essentially a deterministic finite state automata without accept states. We refer to such digraphs as *word graphs*. Suppose that Γ is the word graph of the action of M on ρ. Then, roughly speaking, the Todd-Coxeter algorithm produces a sequence of word graphs $\Gamma_0, \Gamma_1, \ldots$ that approximate the word graph Γ. The initial word graph Γ_0 has a single node ε corresponding to the empty word, and no edges. For every $i \geq 0$, there is a homomorphism from the graph Γ_i to Γ_{i+1}. The sequence $\Gamma_0, \Gamma_1, \ldots$ also converges to Γ in a certain precise categorical sense, even if Γ is infinite. As such if a word $w \in A^*$ labels a path in Γ_i, then w labels a path in every Γ_j where $j \geq i$. It follows that if $w \in S$ labels a path starting and ending at node ε in any Γ_i, then w also labels a path starting and ending at node ε in Γ_j for every $j \geq i$. As such to determine whether $w \in S$ encodes 0, it suffices to find any approximation Γ_i of Γ where w labels a path starting and ending at ε.

We have considered first a series of challenges based on presentations of the symmetric group S_n for various values of n. These do not involve a semidirect product construction, and are thus much less secure than the examples obtained using the recommended parameters in §8 of the paper. In Figure 1, we have plotted the time taken for the approach above to correctly decode at least 90% of values in the challenge words S by comparison with the provided solutions T. The x-axis corresponds to n.

All of the benchmarks were run on a server with a 2.20GHz Intel Xeon CPU with 16 cores, and 128GB of memory. Figure 1 appears to show that both the amount of time and memory required to successfully break these challenges increase exponentially. These figure represent

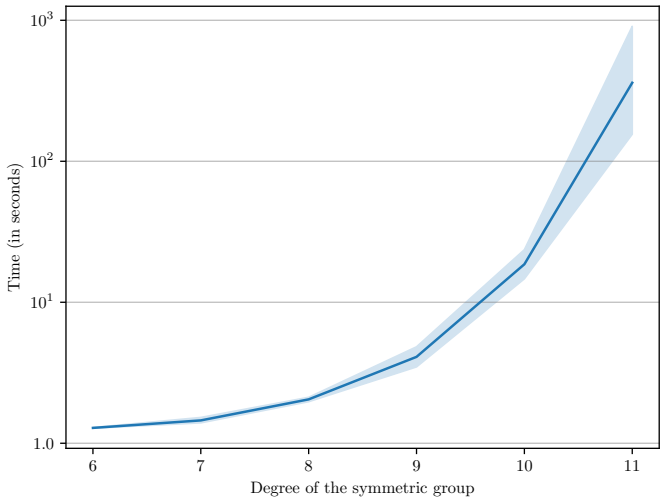


FIGURE 1. Time versus symmetric group degree for the non-semidirect product schemes

an upper bound on the run time for these computations. It is possible that by tweaking the settings for the Todd-Coxeter algorithm in LIBSEMIGROUPS [11] that these times could be decreased to some extent, but it seems unlikely that this would change the overall exponential behaviour. Also we are solving a strictly easier problem than an actual attacker would have to because we are provided with the solution T. This allows us to accurately measure the progress of the computation, and to terminate it early. Without the solution T, it is significantly harder to measure how close to completion the algorithm is.

There were several challenges that we could not break, using the technique outlined above or in any other way, starting with the challenges created as above for $n = 12$. Extrapolating the above curve could lead one to predict that the scheme could still be broken in a matter of hours, but in practice things appear much more complicated.

Besides, we have attempted to break further examples involving a semidirect product $S_n \rtimes S_n$, as recommended in the paper. We have not yet been able to break any of these as soon as $n > 7$, and certainly not for $n = 11$.

To the best of our knowledge at the time of writing, LIBSEMIGROUPS [11] is the only implementation of Todd-Coxeter that provides access to the approximation word graphs Γ_i in the sequence converging to Γ. It might still be possible to use other implementations of Todd-Coxeter (such as those in GAP [20] or ACE [24]) to break some of the challenges. However, there are also several disadvantages to these tools: they were written in the 90s and 00s with the computational

resources of the time in mind; they were not designed to cope with presentations as large as those produced by the scheme in the paper.

Appendix C. A failed attack on GRAFHEN

Discussions about GRAFHEN's security are more than welcome, and every researcher is warmly encouraged to scrutinize GRAFHEN's internals and get in touch with us. We will also setup a bounty challenge in the near future, as a friendly way to invite all sorts of attacks.

In [23], Remi Geraud-Stewart from Amazon claims to develop such an attack. So far, this attempt has failed. We shall presently explain that:

(1) the attack is not relevant to GRAFHEN with its recommended parameters, as it starts from a wrong assumption;
(2) there are severe mathematical mistakes in the argument;
(3) various modifications made in an updated version of [23] do not improve the situation;
(4) our experimental results show that the attack does not provide the claimed advantage; also, the distinguisher, when trying to guess the value of a uniformly chosen encrypted bit, guesses "zero" about 97% of the time;
(5) the code proposed by the author of the attack, when tried on a realistic example, does not provide a positive advantage.
(6) The latest update of [23] (as of April, 29th, 2026) sheds considerable light on the fundamental misconception which underlies the attack, and we have perhaps reached the end of the affair.

C.1. **GRAFHEN does not use confluent systems.** (The first sections are a reaction to the original version of [23]; below we comment on the later additions.)

A fatal flaw appears in the second paragraph of Geraud-Stewart's note, in which it is claimed that the rewriting systems used by GRAFHEN are confluent, which they most certainly are not (according to the recommended parameters).

This point is made repeatedly in our paper, see for example:

- §3.4 where we explain our preference for rewriting systems which are pseudo-bounded instead – confluent systems are too difficult to produce, with a number of rules which is not manageable;
- §6.2 where we explain that using non confluent systems allows a greater number of ciphers for each message, and in fact, an infinite number – indeed if our rewriting systems were confluent, the question of the number of different ciphers would be a security issue much more pressing than Geraud-Stewart's attempt;
- §7.1 which strengthens the previous point, by restricting attention to rules of a certain form, thus preventing the rewriting system from being confluent.

This undue assumption is used in Geraud-Stewart's note in crucial ways. For example, Lemma 3 (which has several issues, see below) applies the pigeon-hole principle to a function ℓ which is ill-defined (for there is no single reduced word representing a given element) and takes its values in an infinite set (for arbitrary long reduced words exist). The proof of Lemma 2 assumes that a certain function φ is a random bijection, when in fact it is (ill-defined and) not a bijection at all. And so on.

It is unclear whether one could repair the broken arguments in this note, and on the other hand, it is also doubtful whether the statements themselves, if they could be proved true, would have any value. Indeed, the purported attack gives an advantage which is bounded below by $\delta/(1+\Delta)$ where δ is some positive constant and Δ is the maximal length of a reduced word in the system. However, with our non confluent rewriting systems, Δ is not bounded and the provided lower bound appears to be 0.

At this point the reader may wonder if an attacker could not simply turn the rewriting system into a confluent one: after all, the Knuth-Bendix algorithm achieves precisely this. It is an assumption of the GRAFHEN scheme that this cannot be done in practice. We are guilty of having kept this assumption silent. Let us now state:

Conjecture. *The rewriting systems used by the GRAFHEN protocol, with the recommended parameters, cannot be made confluent, with realistic ressources, using the Knuth-Bendix algorithm or otherwise.*

C.2. **Mathematical mistakes.** One could wonder whether the rest of the attack could apply, not to GRAFHEN but to some hypothetical, closely related cryptographic scheme which would rely on confluent rewriting systems. However, we can at least identify the following mathematical mistakes, which would have to be adressed.

C.2.1. *Group theory.* Geraud-Stewart insists several times that the conjugation action of the symmetric group on itself is trivial, whereas this is not the case: the conjugation action of a group on itself is trivial if and only if the group is abelian.

Likewise, it is wrongly claimed that the rules $b_i a_j \to a_j W(i,j)$ (to keep the notation as in the note) involve elements $W(i,j)$ which depend only on i; this cannot possibly hold, for (using bars for evaluations) we have $\overline{W(i,j)} = \overline{a}_j^{-1}\overline{b}_i\overline{a}_j$, and if this were independent of j, then $\overline{a}_j$ would be independent of j as well (an element in a group with trivial centre such as the symmetric group is fully determined by its action on the group by conjugation, and in turn this is determined by the action on a set of generators, such as the b_i's). The $\overline{a}_j$'s being chosen randomly, this cannot be assumed.

The confusion seems to arise from our Lemma 33, which proves that $G \rtimes G \cong G \times G$ whenever the semidirect product is made using the conjugation action. It must be borne in mind that $G \times G$ does contain two copies of G which do not commute with one another, as is made explicit in the proof of our Lemma 33: consider $G \times \{1\}$ and the diagonal copy $\{(g, g) \mid g \in G\}$.

These erroneous observations are presented as central to the philosophy of the attack. As a result, it seems natural to conclude that the intuition behind Geraud-Stewart's approach is fundamentally flawed.

C.2.2. *Probability.* The proof of Lemma 3 appears wrong. The context is this: on a finite set G, we have two probabity distributions D_0 and D_1 which satisfy

$$\sum_{g \in G} |D_0(g) - D_1(g)| = 2\delta > 0\,.$$

We have also a function $\ell\colon G \longrightarrow \{0, 1, ..., \Delta\}$. It is claimed that "the pigeon-hole principle" implies that for some r with $0 \leq r \leq \Delta$ we have

$$(*) \qquad |Prob_{D_0}[\ell = r] - Prob_{D_1}[\ell = r]| \geq \frac{2\delta}{\Delta + 1}\,.$$

On the one hand, it is a simple exercise to prove an inequality in the other direction, that is, there must exist an r such that

$$|Prob_{D_0}[\ell = r] - Prob_{D_1}[\ell = r]| \leq \frac{2\delta}{\Delta + 1}\,.$$

On the other hand, the claimed inequality cannot hold in general for an arbitrary function ℓ: for example if ℓ is constant, the left hand side of (*) is always 0, while the right hand side is positive.

If the author relies on special properties of the particular function ℓ under scrutiny, this should be explained.

C.3. **Comments on an updated version.** Shortly after we made the present explanations publicly available, an updated version of [23] was produced. Geraud-Stewart in the new note proposes the exact same attack. Various statements which were problematic, as pointed out above, were deleted, and a lot of interesting extra material was added. This leaves much to be desired, however, as the deleted statements are not really replaced by anything satisfactory, and the extra material is not related to the issues we raise. (Below we shall comment on the third version of [23], which appeared a little later.)

A particularly pregnant problem is the persistent fact that our rewriting systems are not confluent. Geraud-Stewart has removed the adjective "confluent" everywhere, but the mathematical consequences do not vanish. The author defines Δ to be the maximum length of a reduced word, and we have explained that this is infinite. He wrongly

claims, in the proof of Theorem 1, that Δ is the diameter of the corresponding Cayley graph.

Since this erroneous statement comes back, it is worth stating the following: if a rewriting system is confluent, and if it uses the shortlex order, then the maximum length of a reduced word is the diameter of the corresponding Cayley graph. This does not hold for non confluent rewriting systems, and thus does not apply to GRAFHEN (the statement may even fail for some confluent rewriting systems using for example a wreath product ordering, but that is another story). The bottom line is that *the language of Cayley graphs is not appropriate for non confluent rewriting systems*. All the statements in Geraud-Stewart's note which involve Cayley graphs should be disregarded, including Theorem 1.

The advantage offered by the proposed distinguisher is, the author claims,

$$(*) \qquad \frac{\delta_\ell}{1+\Delta}\,.$$

We have explained that the denominator of (*) is not bounded. There are issues with the numerator, too. We have explained above that *some* property of the function ℓ should be used. Geraud-Stewart has replaced a quantity δ by δ_ℓ which appropriately depends on ℓ, and hastens to add that he cannot estimate it, see first sentence after Proposition 4 (it is not even proved that $\delta_\ell > 0$, even though this is a required hypothesis in Lemma 3). Instead he offers a heuristic which is based on Cayley graphs, and explains that a full proof would require an investigation of the Cayley graph. We have made the point that this is ill-informed.

In §4.1 Geraud-Stewart attempts to get rid of the denominator, with an argument which incorrectly assumes that Δ is finite. The variant of the attack thus requires to check membership in a potentially infinite set S^*. Note that S^* is also potentially empty (if δ_ℓ should be 0). The fact is that the probabilities involved in the definition of S^* seem very difficult to compute, and there are infinitely many of them. At the very least this attack is not an explicit one.

Various issues of well-definedness are identified. Let us mention:

- Lemma 3 still tries to use the pigeon-hole principle on an infinite set.
- After Definition 1, the author acknowledges that φ is not a well-defined map, but goes on to treat it as such anyway, defining a "function" ψ in terms of φ, using φ in Lemma 1 and its proof.
- As a result, the distributions D_0 and D_1 which are defined in Lemma 1 in terms of φ, are not well-defined either.
- Etc.

These could probably be repaired by a careful rewording. In fact in our experiment below, we propose some tentative interpretation, hoping to help. We note that we are led to understand the probabilities on sets of words rather than group elements, and it is likely that a formalized definition of D_0 and D_1 would introduce them as distributions on (infinite) sets of words; there remains quite a lot of work to make sense of this.

At this stage it is not obvious at all that, assuming D_0 and D_1 can be defined without ambiguity, we should have $D_0 \neq D_1$, and thus $\delta_\ell > 0$.

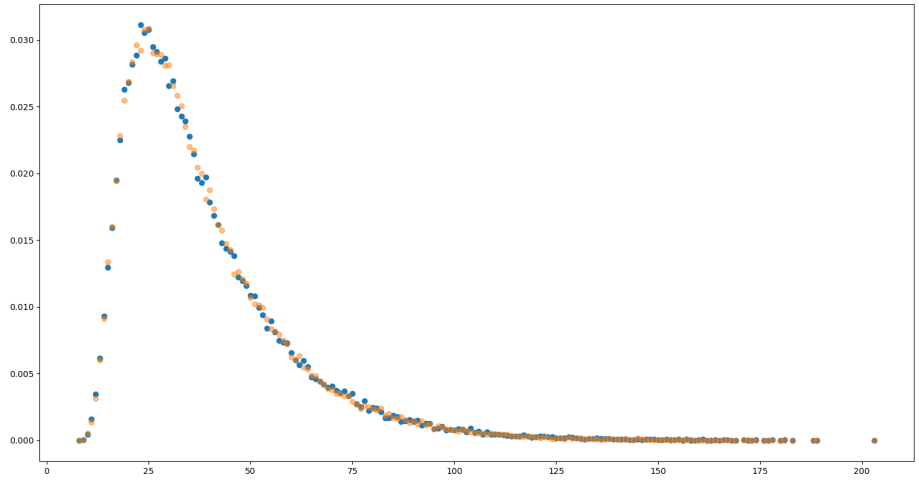
C.4. **Experimentation.** We have implemented the attack of the article [23] on a concrete example, to see whether an advantage was provided.

We start with a training set of 100,000 ciphers of 0 and 100,000 ciphers of 1, with their known cleartext value. We estimate the probability $\Pr_{D_0}[\ell = r]$ and $\Pr_{D_1}[\ell = r]$, for all integers r observed during the experience. Or rather, as ℓ is really not a well-defined function on G (on which the distributions D_0 and D_1 live, at least officially), what we estimate is this: for a cipher w, let w^* be the result of the operation proposed in the attack (replacing letters in the second alphabet by letters in the first, and then reducing), and let us write W_b for the set of ciphers of b, for $b = 0, 1$; then $\Pr_{D_b}[\ell = r]$ should be understood as $\Pr[\ell(w^*) = r | w \in W_b]$.

Of course the sets W_0 and W_1 are infinite, and one should perhaps be careful with the choice of probability measure on W_b. Let us agree, however, that a good estimate is obtained by a frequency count, running our encryption function repeatedly. This is surely what is meant in [23].

It seems important to bridge our notation with that in [23], and so below we shall stick to Geraud-Stewart's conventions, which should be understood as shorthands for probabilities computed as indicated here.

Remark 37. The picture displayed below gives an idea of the two distributions; an informal comment is that they appear similar. A much more decisive comment, even at this early stage, is that we have observed reduced words of length above 200. This conflicts with Geraud-Stewart's claim that reduced words with GRAFHEN should be "trivially" (see Remark 2 p.8) of length no more than 55. It is unclear where the bound $n(n-1)/2$ comes from (it is the diameter of the Cayley graph taken with respect to the Coxeter generators $(i, i+1)$, but we use randomly chosen generators, and beside, Cayley graphs are not relevant here).



We choose r_0 such that $|\Pr_{D_0}[\ell = r_0] - \Pr_{D_1}[\ell = r_0]|$ is maximal; it turns out that $r_0 = 23$. We choose this value of r_0 to define the distinguisher described in the article.

We then use the distinguisher on a test set of 10,000 ciphers of 0 and 1 (independent from the training set, with an equal proportion of each). We estimate $|\Pr_{D_0}[\ell = r_0] - \Pr_{D_1}[\ell = r_0]|$, with the above convention (this is a frequency count on the test set). This value is the supposed advantage of the distinguisher. Our experimental result is

$$\left|\frac{151}{5002} - \frac{149}{4998}\right| \approx 0.000376.$$

This should be compared with the value $2^{-7.3} \approx 0.0063$ predicted in [23].

While the discussion in [23] is around the "advantage", it seems also telling to observe that, while facing a test set which contained 50% of ciphers of 0, and 50% of ciphers of 1, the distinguisher has "guessed" that the decrypted value was 0 with frequency 9700/10000, that is, on 97% of the samples.

Remark 38. One may wonder whether we have used sufficiently many samples, of course. Our first answer is that working with 100,000 ciphers was sufficiently time-consuming that we did not wish to wait, say, 1000 times longer and thus delay exceedingly the publication of this note. Moreover, one should keep in mind that the process described above is made of two steps: a "training step" which allows to guess a value of r_0, and then a "testing step" with this value of r_0. By necessity, what we are probing is the efficiency of the combined two steps. The situation would be different if the "right" value of r_0 was provided by an oracle, and we only had to estimate the advantage, for we could then argue quantitatively. As things stand, it seems difficult to choose a number of samples which would be to everyone's

satisfaction, and our experiment is to be understood as a best-effort approximation.

C.5. **Additional experimentation.** The latest version of [23] proposes a variant, which is accompanied by Python code. Let us comment on both.

While the theory developed in the first part of the paper claims to establish that the length of the modified word w^* leaks information about the cipher, this idea is now abandoned altogether. Instead, the new attack is this:

- Given a cipher $c = (w_A, w_B)$, one construct a vector $v(c)$ by counting the "Letter-frequency feature" of $\mathrm{NF}_A(w_A \cdot \mathrm{Lower}_\sigma(w_B))$ for each bijection σ from the alphabet B to A. This gives a vector $v(c)$ of integers of length $(d! \times d) = 5! \times 5 = 600$.
- Then author trains a model to predict the clear text from $v(c)$.
- Then the model is used to predict the clear text of any cipher text.

The code provided by the author is a toy example, which is confluent, with $n = 7$. There are only 1728 distinct cipher texts in total, and the model is trained using the totality of all these ciphers.

The author then tests the model on a test set of 168 ciphers, and shows that the model has 100 per cent accuracy.

Our concerns about this methodology is that the model has seen every single cipher during the training step, or in other words every cipher in the test step has been seen during the training step. Hence we suspect that the model is overfitted.

The function v described above is defined on a set of cardinality 1728, and its range is that of 600-tuples of integers; even if one assumes a bound on the said integers, the range is clearly much larger than the domain, and there is nothing to prevent v from being injective. On the other hand, there is no special property of v (at least none that is put forward by the author) that would cause it to have collisions. It seems that the attack consists in labelling all the ciphers injectively, and then looking up the label. This of course would not scale up to realistic parameters.

To prove our suspicions, we have tried with $n = 11$ (and a rewriting system which is not confluent, of course). We have used the provided Python code as-is, and here we must pause to mention a possible bias in favour of the attack: the code does not perform the reduction of words using the rules; instead the words are evaluated using the private key, and then one looks up in a table of precomputed short words. While for confluent rewriting systems this provides the same answer as reduction with the rules, this is obviously not the case with non confluent systems. Thus the code (which cannot be made to run without the private key)

has a strong advantage, in that it reduces the words much more than the rules do.

Still, the attack fails to exploit this advantage, as we proceed to showcase.

With $n = 11$ there are now around 78×10^9 group elements to choose from when building a cipher. Normally each such group element can be written in infinitely many different ways as a reduced word, but this decisive advantage of GRAFHEN is short-circuited by Geraud-Stewart's code which uses the private key.

However, the greater number of ciphers is enough to prevent the attack from scaling up. We have trained the model by using a random set of 100 000 ciphers of 0 and 100 000 ciphers of 1. While the model has 100 percent accuracy on the training set, it fails have a positive advantage on an independent test set. This confirms that the model can predict correctly only ciphers seen during the training step.

C.6. **Summary.** Geraud-Stewart's note presents an attack on the GRAFHEN scheme, introducing a distinguisher and a couple of variants. The distinguisher is claimed to provide an advantage of the form

$$\frac{\delta_\ell}{1+\Delta}\,.$$

Our response can be summarized thus:

- A careful analysis of the argument has revealed several gaps in the theoretical explanations. We remain unconvinced that the advantage should be given by the above ratio.
- In any case, the author is not able to estimate the numerator δ_ℓ, and we have explained that the denominator $1 + \Delta$ is not bounded.
- Our numerical experiences show that, on the one hand, the advantage seems much lower than the predicted value, and on the other hand, that the distinguisher is almost "constant", guessing the value 0 on 97% of the (uniformly chosen) encrypted bits.
- The experiences presented by Geraud-Stewart appear to consist merely of an overfitted model, which does not scale up. We have tried the proposed Python code with realistic parameters, and this proved entirely unsuccessful.

C.7. **Epilogue.** A very last update of [23] was made (around April 29th, 2026), and perhaps it explains just where the fundamental misconception is. Consider the following erroneous statements, all found on page 22:

- "the randomness of encryption (ν uniform in L, s uniform in S_{n-k})"
- "the encryption procedure produces a specific ciphertext"
- "the set of possible cross-reduced words w^* is finite"

This is then repeated in various synonymous ways. To be clear: *the encryption function for GRAFHEN does not produce a specific word for a given group element, but introduces randomness.*

This is quite plain already in §A.2, where we present a toy version in Python for the encryption function. In fact, in a more professional implementation of GRAFHEN, in which randomness is taken very seriously, we proceed as follows, in order to find a word representing a given $g \in G$, where a rewriting system has been computed:

- we pick a word w at random (it could be of any length, but for concreteness say we have taken w of length $1,000$);
- we compute $\overline{w} \in G$ using the secret key;
- we find a word v such that $\overline{v} = \overline{w}^{-1}g$; there are many algorithms for this, and we usually use Meet-In-The-Middle, which produces a v which is typically of length between 10 and 20;
- we take the word $u = wv$ and reduce it using the rules, obtaining u' (of length anywhere between 50 and 300 in the example) which is the required word that evaluates to g.

Thus randomness goes far beyond the mere choice of g itself.

This fundamental issue seems to be at the root of all the misunderstandings in [23]. It explains why the author insists that the confluence of the rewriting system is not important – while in fact, the confluence of the system would collapse all the randomness just described, and always produce the same word for a given g. (As pointed out above, in this case the number of different ciphers would be way too low and we would have a serious security concern.)

Having elucidated what is perhaps the "original sin" in [23], we certainly stand by all the objections made above (Cayley graphs are irrelevant, φ is ill-defined, as are D_0 and D_1, etc).

Should the author decide to reemploy some of the ideas involved here in a brand new paper, we would be happy to have a look at it.

As a final word, we do want to respond to Geraud-Stewart's claim that our experiment in §C.5 relied on statistics about the word length rather than the frequency of letters. Never mind the fact that statistics on the word length are the main method suggested in [23], especially in the original version, and that it is confusing to see them dismissed entirely now; the fact is that we looked at letter frequencies. *We have used the provided Python code, on a realistic example, and we have not met any success.*

References

[1] Marina Anagnostopoulou-Merkouri, Reinis Cirpons, James D. Mitchell, and Maria Tsalakou, *Computing finite index congruences of finitely presented semigroups and monoids*, 2023.

[2] Dan Boneh and Alice Silverberg, *Applications of multilinear forms to cryptography*, Topics in algebraic and noncommutative geometry (Luminy/Annapolis, MD, 2001), Contemp. Math., vol. 324, Amer. Math. Soc., Providence, RI, 2003, pp. 71–90. MR 1986114

[3] Zvika Brakerski, *Fully homomorphic encryption without modulus switching from classical gapsvp*, Advances in Cryptology – CRYPTO 2012, Part I (Berlin, Heidelberg), Lecture Notes in Computer Science, vol. 7417, Springer Berlin Heidelberg, 2012, pp. 868–886.

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan, *(Leveled) fully homomorphic encryption without bootstrapping*, ACM Trans. Comput. Theory **6** (2014), no. 3, Art. 13, 36. MR 3255281

[5] Philippe Chartier, Michel Koskas, Mohammed Lemou, and Florian Méhats, *Homomorphic sign evaluation with a RNS representation of integers*, Advances in cryptology—ASIACRYPT 2024. Part I, Lecture Notes in Comput. Sci., vol. 15484, Springer, Singapore, [2025] ©2025, pp. 271–296. MR 4872750

[6] Philippe Chartier, Michel Koskas, Mohammed Lemou, and Florian Méhats, *Fully homomorphic encryption on large integers*, Cryptology ePrint Archive, 2024, Preprint.

[7] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song, *Bootstrapping for approximate homomorphic encryption*, Advances in cryptology—EUROCRYPT 2018. Part I, Lecture Notes in Comput. Sci., vol. 10820, Springer, Cham, 2018, pp. 360–384. MR 3794791

[8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song, *Homomorphic encryption for arithmetic of approximate numbers*, Advances in cryptology—ASIACRYPT 2017. Part I, Lecture Notes in Comput. Sci., vol. 10624, Springer, Cham, 2017, pp. 409–437. MR 3747705

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène, *Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds*, Advances in cryptology—ASIACRYPT 2016. Part I, Lecture Notes in Comput. Sci., vol. 10031, Springer, Berlin, 2016, pp. 3–33. MR 3598072

[10] ———, *Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE*, Advances in cryptology—ASIACRYPT 2017. Part I, Lecture Notes in Comput. Sci., vol. 10624, Springer, Cham, 2017, pp. 377–408. MR 3747704

[11] Reinis Cirpons, Joseph Edwards, and James Mitchell, *libsemigroups: C++ library for semigroups and monoids, version 3.1.3*, 2025.

[12] Reinis Cirpons, Joseph Edwards, James Mitchell, Maria Tsalakou, and Murray Whyte, *libsemigroups_pybind11: Python package that wraps the functionality of the c++ library libsemigroups. version 1.0.1*, 2025, Python binding for libsemigroups.

[13] T. D. H. Coleman, J. D. Mitchell, F. L. Smith, and M. Tsalakou, *The todd–coxeter algorithm for semigroups and monoids*, Semigroup Forum **108** (2024), no. 3, 536–593.

[14] John D. Dixon, *The probability of generating the symmetric group*, Math. Z. **110** (1969), 199–205. MR 251758

[15] Léo Ducas and Daniele Micciancio, *FHEW: bootstrapping homomorphic encryption in less than a second*, Advances in cryptology—EUROCRYPT 2015. Part I, Lecture Notes in Comput. Sci., vol. 9056, Springer, Heidelberg, 2015, pp. 617–640. MR 3344940

[16] Jules Dumezy, *A practical key-recovery attack on GRAFHEN*, Cryptology ePrint Archive, 2026, Preprint.

[17] Sean Eberhard and Stefan-Christoph Virchow, *The probability of generating the symmetric group*, Combinatorica **39** (2019), no. 2, 273–288. MR 3962902
[18] Junfeng Fan and Frederik Vercauteren, *Somewhat practical fully homomorphic encryption*, Cryptology ePrint Archive, 2012, Preprint.
[19] Véronique Froidure and Jean-Eric Pin, *Algorithms for computing finite semigroups*, Foundations of computational mathematics (Rio de Janeiro, 1997), Springer, Berlin, 1997, pp. 112–126. MR 1661975
[20] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.14.0*, 2024.
[21] Craig Gentry, *Fully homomorphic encryption using ideal lattices*, STOC'09—Proceedings of the 2009 ACM International Symposium on Theory of Computing, ACM, New York, 2009, pp. 169–178. MR 2780062
[22] ———, *Homomorphic encryption: a mathematical survey*, ICM—International Congress of Mathematicians. Vol. 2. Plenary lectures, EMS Press, Berlin, [2023] ©2023, pp. 956–1006. MR 4680274
[23] Remi Geraud-Steward, *Grafhen is not ind-cpa secure*, Cryptology ePrint Archive, 2026, Preprint, https://eprint.iacr.org/2026/700.pdf.
[24] George Havas and Colin Ramsay, *Advanced coset enumerator (ace)*, Available from http://staff.itee.uq.edu.au/havas, 1995, C program for Todd-Coxeter coset enumeration; version 5.7.0 released April 2025.
[25] Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien, *Handbook of computational group theory*, Discrete Mathematics and its Applications (Boca Raton), Chapman & Hall/CRC, Boca Raton, FL, 2005. MR 2129747
[26] Donald E. Knuth and Peter B. Bendix, *Simple word problems in universal algebras*, Word Problems (John Leech, ed.), Pergamon Press, 1970, Available as PDF in some online archives, pp. 263–297.
[27] Philippe Le Chenadec, *Canonical forms in finitely presented algebras*, 7th international conference on automated deduction (Napa, Calif., 1984), Lecture Notes in Comput. Sci., vol. 170, Springer, Berlin, 1984, pp. 142–165. MR 778045
[28] Vadim Lyubashevsky, Chris Peikert, and Oded Regev, *On ideal lattices and learning with errors over rings*, Advances in cryptology—EUROCRYPT 2010, Lecture Notes in Comput. Sci., vol. 6110, Springer, Berlin, 2010, pp. 1–23. MR 2660480
[29] W. D. Maurer and John L. Rhodes, *A property of finite simple non-abelian groups*, Proc. Amer. Math. Soc. **16** (1965), 552–554. MR 175971
[30] K. A. Miha˘ ilova, *The occurrence problem for direct products of groups*, Mat. Sb. (N.S.) **70(112)** (1966), 241–251. MR 194497
[31] P. S. Novikov, *Ob algoritmičeskoĭ nerazrešimosti problemy toždestva slov v teorii grupp*, Izdat. Akad. Nauk SSSR, Moscow, 1955, Trudy Mat. Inst. Steklov. no. 44. MR 75197
[32] Koji Nuida, *Towards constructing fully homomorphic encryption without ciphertext noise from group theory*, International Symposium on Mathematics, Quantum Theory, and Cryptography (MQC 2019), 2020, Preprint, https://eprint.iacr.org/2014/097.pdf.
[33] Rafail Ostrovsky and William E. Skeith III, *Algebraic lower bounds for computing on encrypted data*, Cryptology ePrint Archive, 2007, Preprint, https://eprint.iacr.org/2007/064.pdf.
[34] Oded Regev, *On lattices, learning with errors, random linear codes, and cryptography*, J. ACM **56** (2009), no. 6, Art. 34, 40. MR 2572935

[35] Charles C. Sims, *Computation with finitely presented groups*, Encyclopedia of Mathematics and its Applications, vol. 48, Cambridge University Press, Cambridge, 1994. MR 1267733