Trivial Transciphering With Trivium and TFHE

Thibault Balenbois[1], Jean-Baptiste Orfila[1], and Nigel P. Smart[1,2]

[1] Zama, Paris, France.
[2] COSIC, KU Leuven, Leuven, Belgium.
thibault.balenbois@zama.ai, jb.orfila@zama.ai,
nigel.smart@kuleuven.be/nigel@zama.ai.

Abstract. We examine the use of Trivium and Kreyvium as transciphering mechanisms for use with the TFHE FHE scheme. Trivium was introduced in the eSTREAM project as a general purpose stream cipher, whilst Kreyvium was introduced to strengthen Trivium (in the context of transciphering BGV/BFV ciphertext). Previously both ciphers were investigated for FHE transciphering only in the context of the BGV/BFV FHE schemes; this is despite Trivium and Kreyvium being particularly suited to TFHE. Recent work by Dobraunig et al. gave some initial experimental results using TFHE. We show that these two symmetric ciphers have excellent performance when homomorphically evaluated using TFHE. Indeed we improve upon the results of Dobraunig et al. by at least two orders of magnitude in terms of latency. This shows that, for TFHE at least, one can transcipher using a standardized symmetric cipher (Trivium), without the need for special FHE-friendly ciphers being employed. For applications wanting extra security, but without the benefit of relying on a standardized cipher, our work shows that Kreyvium is a good candidate.

Table of Contents

1 Introduction

A "standard" benchmark for MPC and FHE systems has, since the very early days of implementations of MPC and FHE, been the secure evaluation of symmetric key primitives. For example, the first reported large actively secure MPC computation was an evaluation of the AES function using garbled circuit-based techniques in [PSSW09]. In [PSSW09] an encryption of a single block under AES took around 17 minutes. On the FHE side, the first reported computation of a function under FHE was again that of the AES circuit in [GHS12]. In [GHS12] an encryption of an encryption of a single block under AES took around 18 minutes (with parameters that enable bootstrapping for further computation) or 4 minutes (for parameters which just allow the AES computation). However, due to the packing inherent in the underlying FHE system during this 18 (resp. 4) minutes many such evaluations could be carried out. In particular 180 (resp. 120) blocks can be evaluated at once, resulting in an *amortized* time of six (resp. two) seconds per block. Thus whilst a single block evaluation gives us an 18 (resp 4) minutes *latency* of evaluation, the amortized time of six (resp 2) seconds per block gives a *throughput* of 10 (resp. 30) blocks per second.

Over the intevening years the time it takes, both in MPC and FHE, to evaluate the AES circuit has decreased considerably. For example actively secure MPC evaluation of AES now takes around 7 milliseconds latency on a LAN, with a throughput of 500 blocks per second [GRR$^+$16]. On the FHE side, using the TFHE cipher Stracivskt et al. [SMK22] report a time of four minutes latency to evaluate a single block of AES; where the output can be used in further homomorphic processing (an improvement on the 18 minutes of the prior result in this situation).

In addition, there is now a greater appreciation of why evaluating symmetric ciphers in MPC and FHE is important in applications. The key usage of such operations is as a form of *transciphering*, namely to get data efficiently into an MPC/FHE system[3]. However, for many applications using FHE the latency and throughput from using AES is not good enough.

This has led researchers to develop symmetric ciphers for use specifically in MPC and FHE systems; thus creating so-called MPC- or FHE-friendly symmetric ciphers. Examples of these include LowMC [ARS$^+$15], Elisabeth [CHMS22], FLIP [MJSC16], MiMC [AGR$^+$16], Rubato [HKL$^+$22], FiLIP [MCJS19], Rasta [DEG$^+$18], Dasta [HL20], Fasta [CIR22], Pasta [DGH$^+$21], and Kreyvium [CCF$^+$16] (which we will discuss in more detail later). Some older PRF designs, such as the Naor–Reingold PRF [NR97] and the Legendre PRF [Dam90] have also been analyzed in the context of use as MPC/FHE-friendly ciphers [GRR$^+$16]. There are also MPC/FHE-friendly hash functions based on sponge constructions, which can also be used to create symmetric ciphers; for example Rescue [AAB$^+$20], and Poseidon [GKR$^+$21]. There has also been work on special MPC-friendly modes of operation, e.g. [RSS17]. For such MPC/FHE-friendly ciphers one can obtain (in the actively secure MPC setting) latencies in the order of milli-seconds, and throughputs in the order of thousands of operations per second [GRR$^+$16]. It remains an open problem in obtaining similar timings in the context of FHE transciphering. In this paper we show that with TFHE and Trivium or Kreyvium one is not far off.

Many of these specially designed ciphers have had less analysis than standard ciphers; thus it is unclear whether organizations would be willing to deploy them when compared to a standardized cipher. Indeed the construction of new proposals for MPC/FHE-friendly ciphers seem to come at a rate faster than the communities ability to apply cryptanalytic effort to them. As just one

[3] To transcipher data from an FHE encryption to symmetric encryption, one would perform the symmetric decryption process homomorphically, and then perform some form of distributed decryption to obtain the ciphertext "in the clear".

example, FLIP [MJSC16] was cryptanalyzed in [DLR16]. In addition, the MPC-in-the-Head based signature scheme Picnic [CDG+17] which was submitted to the NIST PQC "non-competition", did not proceed to the final rounds. One of the reasons for this was that Picnic's security was based on the properties of the non-standardized MPC-friendly block cipher LowMC[4]. Thus companies seemingly need to choose between either a slow, but standardized/well scrutinized, traditional cipher, and a fast, but less standardized/less scrutinized, MPC/FHE-friendly cipher.

Much of the development of MPC/FHE-friendly special ciphers has been motivated by the fact that for *most* MPC and FHE systems the underlying plaintext space is a large finite field, i.e. not $\mathbb{F}_2$. Thus much of the prior work has focused on FHE schemes such as BGV [BGV12] and BFV [FV12, Bra12]. However, for FHE systems such as TFHE [CGGI20] the plaintext space is exactly $\mathbb{F}_2$, or $\mathbb{Z}/(2^k)$ for some small value of k. Thus for such an FHE encryption scheme one *might* be able to use a relatively standard cipher, or one closely related to a standardized cipher.

The most promising candidate for such a *standardized* TFHE-friendly cipher is Trivium [De 06]. This was a cipher designed for the eSTREAM project (a competition run via a European project, between 2004 and 2008, in order to identify new stream ciphers). It was designed without any thought of application to MPC or FHE. Indeed, it's main design criteria were to achieve 80-bits of security and to be efficient in hardware, as well as a reasonably efficient software implementation. Trivium ended up in the final eSTREAM portfolio of recommended ciphers, and has been standardized in ISO/IEC 29192-3 [ISO12].

The security of Trivium is well established, with only some attacks on it, or closely related ciphers, having been presented [ADMS09, BKM11, CGB+17, DDGP22, FV14, FWDM18, HJL+20, HLM+20, HL11, KMN12, MB07, WB10, YT18, YT21, ZLL18]. However, the "security margin" for Trivium is now considered to be relatively small.

This small security margin led Canteaut et al. [CCF+16] to introduce a tiny modification to Trivium, called Kreyvium, in order to boost the security level to 128-bits. In addition, Kreyvium protects against some of the prior attack methodologies on Trivium. The main motivation for introducing Kreyvium was to present an FHE-friendly symmetric primitive with 128-bits of security. Since the introduction of Kreyvium, further cryptanalysis has been performed on Kreyvium [WIM17], and on both Trivum and Kreyvium [HJL+20, HLM+20, YT18]. Theoretical key recovery attacks have been proposed against 839 round Trivium and 891 round Kreyvium [WHT+18], and a distinguisher on 899 round Kreyvium was presented in [WIM17]. A practical key recover attack against 805 round Trivium was presented in [YT21]. This has led both Trivium and Kreyvium to still be considered secure.

1.1 Prior Work on Performance of FHE Transciphering

As discussed above a lot of the prior work has been on special ciphers which work over plaintext spaces of the form $\mathbb{F}_p$, for "large primes" p. The reader is suggested to examine the paper [DGH+21], which not only introduces the cipher Pasta, but also provides extensive implementation experiments on various ciphers, using different FHE libraries.

For the case of $\mathbb{F}_p$, the authors of [DGH+21] show that a block cipher such as Pasta, when used with an FHE-scheme such as BGV or BFV, can transcipher a single block ciphertext, encrypted under Pasta into a ciphertext encrypted under the FHE scheme, in 120 seconds for the case of 17- and 33-bit primes p. They conclude that for such situations Pasta is the preferred cipher.

[4] The NIST report on their choice of SPHINCS+ vs Picnic [NIS22] states "*NIST chose SPHINCS+ largely because it could not confidently quantify the security of LowMC*".

As remarked above Kreyvium was actually introduced in the context of trying to find a cipher which is FHE-friendly. However, the paper [CCF+16] introducing Kreyvium looked at transciphering in the context of FHE schemes such as BGV and BFV, for which it is not ideally suited. The reported performance of Trivium and Krevium in [CCF+16] were of the order of 1000's of seconds for latency, and throughputs of hundreds of bits per minute (when using BGV on a single core machine), with a small improvement in this performance when using BFV. In addition, as the BGV/BFV schemes do not (easily) support bootstrapping the transciphering was done to a levelled FHE scheme, meaning very little output could be obtained before the cipher would need to be re-initialized.

In [DGH+21] the authors report on an implementation of Kreyvium using TFHE, for which Kreyvium is more suited. They present experiments which output 46 bits of output, and which takes 284 seconds to produce this output. To produce 46 bits of output in Kreyvium actually means one has to clock the cipher $1198 = 46 + 1152$ times, since the cipher requires one to discard the first 1152 bits of output. Thus, after this warm-up phase the experiments in [DGH+21] imply one can obtain one bit of output every $284/1198 = 0.237$ seconds. This rate can be continued, since we do no need to reset the cipher, since TFHE supports bootstrapping. In this work we show roughly a 100-fold improvement on this throughput.

Other work, combining TFHE with FHE-friendly ciphers, has concentrated mainly on dedicated (i.e. non-standardized cipher designs). For example [HMR20] gives a time of around 20 seconds per output bit for TFHE, and 1-2 seconds per output bit for TGSW, when evaluating the FiLIP stream cipher [MCJS19]. This was improved to 2.6 ms per bit using the FINAL FHE scheme [BIP+22] (a scheme closely related to TFHE, but based on the NTRU-like as opposed to LWE-like assumptions) in [CDPP22]. However, as we pointed out above ciphers such as FiLIP are not as well cryptanalyzed when compared to standard ciphers such as Trivium.

1.2 Our Contribution

We revisit the ciphers Trivium and Kreyvium in the context of the TFHE homomorphic encryption scheme. We concentrate on obtaining a low latency implementation, which then maximises the throughput. The concentration on latency as opposed to throughput is motivated by application concerns; customers are unlikely to want to wait minutes for an encryption to take place, even if they get 100's of such encryptions per execution.

We show that the *standardized* cipher Trivium is ready for use in FHE applications, and it is already FHE-friendly. Thus there is no need to base application security on one animal in the menagerie of purpose designed, but *non-standardized* MPC/FHE-friendly ciphers. For those users interested in enhanced security, given Trivium's small security margin, we also investigate Kreyvium and show this is also ready for deployment. We feel the potential applicability of Kreyvium in real FHE deployments would warrant standardization of this cipher in the near future.

2 Trivium and Kreyvium

As already remarked in the introduction, Trivium is a well-studied, and standardized stream cipher which aims to provide 80-bits of security. However, cryptanalysis over the last fifteen years has shaved off the security margin that Trivium provides. So whilst it can still be considered secure, it can be said to *only just* provide 80-bits of security. This fact led Canteut et al [CCF+16] to introduce a variant of Trivium, called Kreyvium, which aims to offer 128-bits of security. Interestingly they introduced the cipher exactly in the context of our study, namely homomorphic transciphering. In this section we overview these two stream ciphers and highlight the small differences between them.

2.1 Trivium

The basis of Trivium is a set of three shift registers called **a**, **b** and **c**, of lengths 93, 84 and 111 bits respectively (making 288 bits in total). Once the state has been set up the three shift registers feed into each other via the following equations, over $\mathbb{F}_2$:

$$\begin{aligned} a_i &= c_{i-111} + c_{i-110} \cdot c_{i-109} + c_{i-66} + a_{i-69}, \\ b_i &= a_{i-93} + a_{i-92} \cdot a_{i-91} + a_{i-66} + b_{i-78}, \\ c_i &= b_{i-84} + b_{i-83} \cdot b_{i-82} + b_{i-69} + c_{i-87}. \end{aligned}$$

Notice the regular pattern here: the three top bits of **a**, **b** or **c** are combined with a lower bit (in position 66 or 69) and then with a bit of a second register, to obtain a new bit in the second register.

To initialize the state an 80-bit key $k_0, \ldots, k_{79}$ and an (up to) 80-bit initial value (IV) $v_0, \ldots, v_{79}$ are fed into the lower bits of the **a** and **b** registers, with **a** getting the key, and **b** the IV. The rest of the bits of all registers are set to zero, bar the top three bits of the **c** register, which are set to one. The system is then clocked $4 \cdot 288 = 1152$ times before any keystream is actually used.

The output bit of Trivium is then obtained from the $\mathbb{F}_2$-equation

$$r_i = c_{i-111} + a_{i-93} + b_{i-84} + c_{i-66} + a_{i-66}.$$

The entire algorithm, with some algorithmic optimizations, is given in Figure 1.

Trivium

1. $(s_1, \ldots, s_{93}) \leftarrow (k_0, \ldots, k_{79}, 0, \ldots, 0)$.
2. $(s_{94}, \ldots, s_{177}) \leftarrow (IV_0, \ldots, IV_{79}, 0, \ldots, 0)$.
3. $(s_{178}, \ldots, s_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$.
4. For $i = 1, \ldots, ?$ do
   (a) $t_1 \leftarrow s_{66} + s_{93}$.
   (b) $t_2 \leftarrow s_{162} + s_{177}$.
   (c) $t_3 \leftarrow s_{243} + s_{288}$.
   (d) If $i > 1152$ then output $r_{i-1152} = t_1 + t_2 + t_3$
   (e) $t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$.
   (f) $t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$.
   (g) $t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$.
   (h) $(s_1, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
   (i) $(s_{94}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$.
   (j) $(s_{178}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$.

Fig. 1. The Trivium Stream Cipher

2.2 Kreyvium

Kreyvium is very similar to Trivium, except now there is a 128-bit key and a 128-bit IV value, which are held in shift registers **k** and **IV**. The initial state is now defined as follows: The first

93-bits of **k** are placed in the **a** register, the first 84-bits of **IV** are placed in the **b** register, the remaining 44 bits of **IV** are placed in the **c** register, which is then padded with 1 values for all remaining positions, except the final one which is set to zero.

The algorithm proceeds much as before except the registers **k** and **IV** are cyclicly rotated to the right on every clock cycle. The top bit of the **k** register is added into both the output and the update to the **a** register. In addition, the top bit of the **IV** register is added into the update to the **b** register, so we have

$$\begin{aligned} a_i &= c_{i-111} + c_{i-110} \cdot c_{i-109} + c_{i-66} + a_{i-69} + k_{127}, \\ b_i &= a_{i-93} + a_{i-92} \cdot a_{i-91} + a_{i-66} + b_{i-78} + IV_{127}, \\ c_i &= b_{i-84} + b_{i-83} \cdot b_{i-82} + b_{i-69} + c_{i-87}, \\ \mathbf{k} &= \mathbf{k} \ggg 1, \\ \mathbf{IV} &= \mathbf{IV} \ggg 1, \\ r_i &= c_{i-111} + a_{i-93} + b_{i-84} + c_{i-66} + a_{i-66} + k_0. \end{aligned}$$

The entire algorithm, with some algorithmic optimizations, is given in Figure 2, where we mark the changes from Trivium in blue.

Kreyvium

1. $(s_1, \ldots, s_{93}) \leftarrow (k_0, \ldots, k_{92})$.
2. $(s_{94}, \ldots, s_{177}) \leftarrow (IV_0, \ldots, IV_{83})$.
3. $(s_{178}, \ldots, s_{288}) \leftarrow (IV_{84}, \ldots, IV_{127}, \ldots, 1, \ldots, 1, 0)$.
4. For $i = 1, \ldots, ?$ do
   (a) $t_1 \leftarrow s_{66} + s_{93}$.
   (b) $t_2 \leftarrow s_{162} + s_{177}$.
   (c) $t_3 \leftarrow s_{243} + s_{288} + k_{127}$.
   (d) If $i > 1152$ then output $r_{i-1152} = t_1 + t_2 + t_3$
   (e) $t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171} + IV_{127}$.
   (f) $t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$.
   (g) $t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$.
   (h) $(s_1, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
   (i) $(s_{94}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$.
   (j) $(s_{178}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$.
   (k) $(k_0, \ldots, k_{127}) \leftarrow (k_{127}, k_0, \ldots, k_{126})$.
   (l) $(IV_0, \ldots, IV_{127}) \leftarrow (IV_{127}, IV_0, \ldots, IV_{126})$.

Fig. 2. The Kreyvium Stream Cipher

3 Transciphering in TFHE

In this section we outline how transciphering is integrated with the TFHE, and along the way we briefly introduce TFHE for the reader who is new to this FHE scheme.

3.1 Generic Transciphering Protocol

As explained in the introduction, in the context of FHE, transciphering is the method of using an encryption scheme E within the fully homomorphic one FHE. To illustrate the usage, let us assume a classical scenario where a client $\mathcal{C}$ sends their encrypted data to a server $\mathcal{S}$. To simplify, let E be a (standard) symmetric cipher and FHE be a symmetric homomorphic encryption scheme with plaintext space $\mathbb{Z}/p\mathbb{Z}$. Formally these ciphers are given by tuples of algorithms; $\mathsf{E} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ and $\mathsf{FHE} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{EvalCircuit})$. The process is described in Figure 3.

Transciphering

$\mathcal{C}$:

1. $\mathsf{sk_E} \leftarrow \mathsf{E.KeyGen}(1^\lambda)$
2. $(\mathsf{evk_{FHE}}, \mathsf{sk_{FHE}}) \leftarrow \mathsf{FHE.KeyGen}(1^\lambda)$
3. Let $m_1, \ldots, m_k \in \mathbb{Z}/p\mathbb{Z}$ for some $k \in \mathbb{N}$ be the cleartexts.
4. For $i \in [1, \ldots, k], \mathsf{ct}_i \leftarrow \mathsf{E.Encrypt}(\mathsf{sk_E}, m_i)$
5. $\mathsf{ct_{sk_E}} \leftarrow \mathsf{FHE.Encrypt}(\mathsf{sk_{FHE}}, \mathsf{sk_E})$
6. Send to $\mathcal{S}$: $(\mathsf{ct}_0, \cdots, \mathsf{ct}_k, \mathsf{ct_{sk_E}})$

$\mathcal{S}$:

1. For $i \in [1, \ldots, k], \mathsf{ct}'_i \leftarrow \mathsf{FHE.EvalCircuit}(\mathsf{evk_{FHE}}, \mathsf{E.Dec}(\mathsf{ct_{sk_E}}, \mathsf{ct}_i))$

Fig. 3. Generic Transciphering Protocol between a symmetric E and a FHE FHE cryptosystems

In what follows, we instantiate FHE as the TFHE scheme, and E with either Trivium or Kreyvium. The homomorphic evaluation of the decryption circuit starts by generating the output keystream of Trivium or Kreyvium r, in the encrypted domain using the Trivium/Kreyvium instructions. The last step is a homomorphic XOR operation between the input (plaintext) Trivium ciphertext and homomorphically encrypted value of r.

3.2 TFHE Scheme and Large Integer Representation

The key to understanding our optimized implementation of Trivium/Kreyvium one needs to understand how the plaintext space of the TFHE cipher interacts with so-called "programmable bootstrapping". In particular it is not necessary to perform a bootstrapping operation upon every boolean gate operation.

TFHE is a fully homomorphic encryption scheme in which bootstrapping (the algorithm to refresh reduce the noise in a ciphertext after a series of homomorphic operations) has the property that it is programmable. In particular during bootstrapping an arbitrary lookup table can be evaluated homomorphically on the ciphertext. The TFHE scheme relies on the LWE problem (and it's variant RLWE/GLWE). In what follows, we denote an LWE ciphertext of a message $m \in \mathbb{Z}/p\mathbb{Z}$, with the secret key $\mathsf{sk} \xleftarrow{\$} \mathcal{S}^n$ ($\mathcal{S}$ could be a binary, ternary or discrete Gaussian distribution), a mask $\mathbf{a} \xleftarrow{\$} (\mathbb{Z}/q\mathbb{Z})^n$ (with q the ciphertext modulus), a scaling factor $\Delta = \frac{q}{p}$, and some noise $e \xleftarrow{\$} \mathcal{N}_{\sigma^2}$ ($\mathcal{N}_{\sigma^2}$ is a discrete Gaussian of variance σ^2, assumed to be centered in 0), by the equation

$$\mathsf{LWE}^{n,q}_{\mathsf{sk}}(\Delta \cdot m) = (\mathbf{a}, \langle \mathbf{a} \ , \ \mathsf{sk} \rangle + \Delta \cdot m + e \mod q).$$

In practice, for TFHE, one usually selects p to be a small power of two up to 10 bits (say $p = 2, 4$, or 16), and q to be 2^{32} or 2^{64}.

Programmable bootstrapping (PBS) gives the possibility to homomorphically evaluate almost any univariate functions $f : \mathbb{Z}/p\mathbb{Z} \to \mathbb{Z}/p\mathbb{Z}$. When both q and p are powers of two, the functions f which can be evaluated via the PBS are those which are negacyclic on the input domain to which they are applied. A negacyclic function is one for which we have

$$f(x + p/2) = -f(x) \pmod{p}$$

for $x \in [0, \ldots, p/2)$. Note, we only require negacyclicity on the input values x for which we will apply the function. Often in TFHE this is enabled by adding an extra bit into the plaintext space, which is made to always equal zero. As we shall see one can sometimes uses this extra bit if the function f and the application is designed to cope with it.

In order to extend these ideas to bivariate functions $g : \mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/p\mathbb{Z} \to \mathbb{Z}/p\mathbb{Z}$, the idea is to split p into two parts: the message msg and carry spaces carry. In other words one thinks of the real "message" lying in the range $[0, \ldots, \mathsf{msg} - 1]$, with a carry lying in $[0, \ldots, \mathsf{carry} - 1]$, plus an addition buffer bit to ensure that any bivariate function computing on the message and carry spaces is negacyclic. This leads to setting $p = 2 \cdot \mathsf{carry} \cdot \mathsf{msg}$.

For instance, for a two bits of message space we have $\mathsf{msg} = 4$, and for three bits of carry space we have $\mathsf{carry} = 8$. In most homomorphic operations one wants to pick these values so that $\mathsf{msg} \leq \mathsf{carry}$ in order to help with evaluation of bivariate functions. Then, assuming that the carry space is empty, by concatenating two ciphertexts ct_1 and ct_2 into one (i.e., $\mathsf{ct}_{\text{res}} = \mathsf{msg} \cdot \mathsf{ct}_1 + \mathsf{ct}_2$), we are able to compute a PBS over the two inputs. Note that the carry space is also used as a buffer for levelled operations (i.e., homomorphic operations which do not get immediately followed by a bootstrapping operation). In our most efficient implementation of transciphering we utilized $\mathsf{msg} = 2$ and $\mathsf{carry} = 2$.

In general, an operation called keyswitching preceeds the PBS operation. A keyswitch allows one to transform a ciphertext ct encrypted with a key sk to another ciphertext encrypted under a key sk'. The PBS takes ciphertexts encrypted under sk', and transforms them (during bootstrapping) into ciphertexts encrypted under sk. Thus, in order to be consistent, a keyswitch is computed before the PBS. The first keyswitch changes sk to sk', whereas during the PBS the key is switched from sk' to sk. Thus, the output ciphertext has the same encryption as the input one. Let bsk be a bootstrapping key, ksk a keyswitching key. We use the following signature to design the chaining of such a keyswitch (KS) and PBS as applying a function $f(\cdot)$:

$$\mathsf{ct}_{out}(f(m)) \leftarrow \mathsf{PBS}(\mathsf{bsk}, \mathsf{ksk}, \mathsf{ct}(m), x \mapsto f(x)).$$

with a simple keyswitch denoted by:

$$\mathsf{ct}_{out}(m) \leftarrow \mathsf{Keyswitch}(\mathsf{ksk}, \mathsf{ct}(m))$$

As described in [BBB+23], the original TFHE scheme does not allow working with plaintexts larger than 10 bits. To overcome this constraint, the idea is to apply a radix decomposition on the large plaintext and to encrypt independently each part. More formally, let $\mathsf{pt} \in \mathbb{Z}/P\mathbb{Z}$ be the plaintext, let $\beta \in \mathbb{N}$ be the basis, such that $|\beta| \leq 10$. The β-radix decomposition of pt can be written as: $\mathsf{pt} = \sum_{i=0}^{d-1} \mathsf{pt}_i \cdot \beta^i$, for some $d \in \mathbb{N}$ and $0 \leq \mathsf{pt}_i < \beta$ for $i \in [0, \ldots, d-1]$. Then, an encryption of pt is:

$$\mathsf{ct}(\mathsf{pt}) = \{\mathsf{LWE}^{n,q}_{\mathsf{sk}}(\Delta \cdot \mathsf{pt}_i)\}_{i \in [0,\ldots,d-1]}$$

In what follows, we denote ρ_i the set of parameters associated to a precision p_i. A parameter set contains values ensuring secure LWE instances (i.e., n, q, σ_{LWE},), secure GLWE instances for the bsk (i.e., the GLWE dimension k, the polynomial size N, and the standard deviation σ_{GLWE}) and correctness parameters (i.e., the decomposition bases and levels for the PBS and the KS, β_{PBS}, ℓ_{PBS}, β_{KS}, ℓ_{KS}). A ciphertext associated to a parameter set ρ is written as $ct(\cdot)^{\rho}$.

3.3 Casting between TFHE encryptions

In TFHE, the complexity (and thus the concrete timings) of computing a PBS is linked to the precision of the plaintext. All cryptographic parameters are defined depending on the input precision. We refer to [BBB+23, Fig.8] for more details. This means that choosing the right message precision has a major impact on the performance. In the case of the transciphering, the best precision needed to implement the decryption algorithm of E might not be the same as the one for the following homomorphic operations. The idea is then to be able to cast from one precision to another one.

Here, we focus on an approach allowing casting from a smaller precision p_1 to a larger one p_2 (where $\mathsf{p}_i = \log_2(\mathsf{msg}_i \cdot \mathsf{carry}_i) + 1$). This is because both Trivium and Kreyvium are boolean oriented, whereas the best trade off between precision and computational time, for standard computations on encrypted integer values under TFHE, is around 5 bits of precision. Thus we want to cast from one set of parameters (used for Trivium/Kreyvium evaluation) and another set (used for operations on encrypted integers). The idea of the casting algorithm is first to pack as many ciphertexts as possible into one. This is done by shifting a ciphertext by the size of the message space msg_1. Then, a keyswitch is applied to switch from the first set of parameter to the second. This requires a dedicated keyswitching key, denoted $\mathsf{ksk}_{\rho_1 \to \rho_2}$, going from the parameter set ρ_1 to ρ_2. Finally, a PBS is applied in order to go from the scaling factor Δ_1 to Δ_2. The process is described in Figure 4.

4 Implementation of Trivium in TFHE

There are various design choices in how one could implement transciphering in TFHE. In this section we outline two major ones, the underlying data type used for the homomorphic evaluation of Trivium/Kreyvium and how this data type is casted into the data types used by following homomorphic evaluations.

4.1 Three Potential Underlying Data Types

We examined three underlying methodologies for representing the data within the homomorphic evaluation of Trivium and Krevium.

FheBool: A naïve implementation of the symmetric schemes would use the default API of the library (which we will refer to as the high-level API in what follows). The high-level API provides a `FheBool` type, representing a bit message encrypted using the TFHE scheme. The `FheBool` type internally uses a ciphertext modulus of $q = 2^{32}$, and it computes a bootstrapping operation after each Boolean operations (e.g., AND, OR, XOR, ...) except the NOT one. Since both Trivium and Kreyvium are working with bits, the `FheBool` type seems to be a good fit. It allows the production of a stream of pseudorandom `FheBool`, each being the encrypted version of the actual Trivium and Kreyvium stream. However, this approach does not offer many possibilities to optimize computations.

Casting

Conditions:

1. $\mathsf{msg}_1 \cdot \mathsf{carry}_1 \geq \mathsf{msg}_2$;
2. $\mathsf{p}_2 \geq \mathsf{p}_1$

Input:

1. $\mathsf{ksk}_{\rho_1 \to \rho_2}$: keyswitching key from parameter sets ρ_1 to ρ_2
2. $(\mathsf{ksk}_{\rho_2}, \mathsf{bsk}_{\rho_2})$: keyswitching and bootstrapping keys to compute a PBS using the parameter set ρ_2
3. $\mathsf{ct}(m)^{\rho_1} = \{\mathsf{LWE}^{n_1,q_1}_{\mathsf{sk}_1}(\Delta_1 \cdot m_i)\}_{i \in [0,\ldots,\kappa-1]}$: a ciphertext encrypting a message m under parameters ρ_1

Output: A ciphertext $\mathsf{ct}^{\rho_2}(m) = \{\mathsf{LWE}^{n_2,q_2}_{s_2}(\Delta_2 \cdot m'_i)\}$

Algorithm:

1. For $i \in \left[0,\ldots,\left\lfloor \frac{\kappa \cdot \log_2(\mathsf{msg}_1)}{\log_2(\mathsf{msg}_2)} \right\rfloor\right]$:
   (a) // *Packing*
   For $j \in [0,\ldots,\log_2(\frac{\mathsf{msg}_2}{\mathsf{msg}_1})]$:
      i. $\mathsf{ct}^{\rho_1}(m_i) \leftarrow \mathsf{ct}^{\rho_1}_i(m_i) + 2^{j \cdot \log_2(\mathsf{msg}_1)} \cdot \mathsf{ct}^{\rho_1}_j(m_j)$
   (b) // *Switching to the second parameter set*
   $\mathsf{ct}^{\rho_2}(m_i) \leftarrow \mathsf{Keyswitch}(\mathsf{ksk}_{p_1 \to p_2}, \mathsf{ct}(m_i))$
   (c) // *Adjusting to the scaling factor* Δ_2
   $\mathsf{ct}^{\rho_2}(m_i) \leftarrow \mathsf{PBS}(\mathsf{bsk}, \mathsf{ksk}, \mathsf{ct}^{\rho_2}(m_i)', x \mapsto x \ggg \log_2\left(\frac{\Delta_1}{\Delta_2}\right))$
2. Return $\mathsf{ct}^{\rho_2}(m_i)$

Fig. 4. Casting Algorithm between two LWE ciphertexts

FheUint8: A second naïve implementation would use the `FheUint8` type, representing a byte encrypted via the high-level TFHE API. One might be tempted to do this as such a representation could potentially avoid any casting operation after the transciphering algorithm was performed.

In a cleartext implementation, in C say, one might store the Trivium state/key etc in blocks of 8, 32 or 64 bits depending on whether ones native machine type was a byte, long, or longlong. This `FheUint8` representation is the TFHE analogy, where the 8 bits of the data type standin for each byte of the original Trivium or Kreyvium cipher; be it from the key, registers, messages, etc. In practice, all the high-level integer types of the TFHE-rs library are radix represention of the underlying actual integer, using ciphertexts with $\mathsf{msg} = 4$, representing 2-bit input messages. The `FheUint8` is then only a wrapper around four of these ciphertexts ; the equivalent of looking up bits in the registers consists in reconstructiong bytes from two bytes of the registers, a costly operation. However, what this also means is that it is straightforward (i.e., casting is trivial) with this representation to transcipher messages that use the same radix representation, into any other integer type of the TFHE-rs library. The downside of this implementation is its poor performance: using bigger ciphertexts and more complex representations means one needs more costly bitwise operations. By construction, Trivium/Kreyvium does not allow leveraging the potential advantages of this representation. We provide this implementation for comparison as it provides simple casting for further homomorphic operations; however it is very costly in terms of transcipherting. Thus, it probably should never be used in practice because of its poor performance.

Optimized implementation: Our best implementation revolves around a family of types from the TFHE-rs library dubbed `shortints`, where a small integer (of modulus 2, 4, 8, or 16), is encrypted in a single ciphertext, along with a potential carry (empty, or of modulus 2, 4, 8, 16). This carry can hold temporary results during an FHE circuit evaluation, often allowing optimizations. The radix representation of the high-level integers of the TFHE-rs library uses ciphertexts encrypting 2 bits of message and 2 bits of carry.

In this implementation we represented each bit with a different ciphertext, with each of these ciphertexts being the encryption of a 1-bit message and a 1-bit carry, plus a one bit buffer (to enable negacyclic function evaluations via the PBS). Thus we set $p = 8$. This enabled us to take advantage of the fact that this representation does not necessarily need a PBS after each arithmetic operation: for example we can let an addition overflow over the carry bit (a so-called leveled addition in the language of TFHE). Meaning we can perform two (leveled) additions in a row before doing a PBS (or one addition and one bitwise AND for example). This carry bit then needs to be cleaned at the end of each step, which, however, does require a PBS operation. This means we set the PBS to compute the function

$$f : \begin{cases} \mathbb{Z}/8\mathbb{Z} \longrightarrow & \mathbb{Z}/8\mathbb{Z} \\ \quad x \quad \longmapsto x & \pmod 2 \end{cases}$$

Naively, one would assume that this would allow us to process three bit additions, and then apply a PBS to obtain a reduction modulo two. i.e., one could compute

$$\mathsf{PBS}(\mathsf{bsk}, \mathsf{ksk}, x_1 + x_2 + x_3, f) = x_1 \oplus x_2 \oplus x_3,$$

since, if $x_i \in \{0, 1\}$ then (as an integer) the domain of the PBS lies in $\{0, 1, 2, 3\}$, and so the function f is applied to an element of the plaintext space where the buffer bit is equal to zero. Since f it is negacyclic on these inputs when considered as a function.

However, the function f above is also negacyclic on the input value four, as $f(4) = 0 = -0 = -f(0)$. This means we can actually add four values together before needing to perform a PBS operation, i.e., one can compute

$$\mathsf{PBS}(\mathsf{bsk}, \mathsf{ksk}, x_1 + x_2 + x_3 + x_4, f) = x_1 \oplus x_2 \oplus x_3 \oplus x_4,$$

Alas, this idea does not extend as the function f is not negacyclic on the input value five, as $f(5) = 1 \neq 7 = -f(1)$.

Since the PBS operation is the most costly operation (by far), we try to optimize them out of the circuit as much as possible. Every XOR gate is represented by a (leveled) addition in our scheme. Our main steps (executed 64 times in parallel) for Trivium would thus go like this:

- Execute steps 4a, 4b, and 4c as simple (leveled) additions, i.e. with no PBS operation being carried out. [zero PBS]
- Spawn 4 threads:
  - Step 4d:, as two (leveled) additions, along with two 'clean carry' operations [one PBS];
  - Step 4e: as an AND gate [one PBS] followed by two (leveled) additions, then a 'clean carry' operation, [one PBS]; (for Kreyvium an extra 'clean carry' operation is needed, meaning an additional PBS);
  - Step 4f: as an AND gate [one PBS] followed by two (leveled) additions, then a 'clean carry' operation, [one PBS];
  - Step 4g: as an AND gate [one PBS] followed by two leveled additions, then a 'clean carry' operation, [one PBS]; (for Kreyvium an extra 'clean carry' operation is needed, meaning an additional PBS);
- Return: r, t_1, t_2, t_3

Making a total of eight PBS operations spread over the four threads. We can then output the 64 return values, and update each register 64 times. When fully parallelized, this will cost the latency equivalent of two operations PBS per output bit. For Kreyvium one requires ten PBS operations spread over the four threads, with a latency equivalent to three PBS operations per output bit.

4.2 Transciphering

By the definition of transciphering, we are using a different integer representation in the cipher than the one used in the high level integer types for the data. Thus, we need to switch between the keys used in the FHE evaluation of Trivium and Kreyvium, to the keys corresponding to the integers that we actually want to transcipher in the higher level application. In other words we need to cast the underlying data type of the homomorphic encryption from one which is preferred for Trivium/Kreyvium evaluation, into one which is preferred for further (application specific) homomorphic evaluation.

Following Figure 5, the client is using Trivium/Kreyvium to encrypt its messages whereas the secret key is encrypted using TFHE. On the server side, Trivium/Kreyvium is ran in the encrypted domain. As previously described, the best precision (i.e., cryptographic parameter set) to homomorphically compute the symmetric encryption scheme differs from the one used to compute over homomorphic integers. For example one may be operating on homomorphically encrypted integers of 16, 32 or 64 bits in length (i.e., `FheUint16`, `FheUint32`, `FheUint64`). Since Trivium has a natural parallel execution of 64-bits in parallel, this is relatively easy to translate into the domain over which one is computing if the integer length is less than 64. In contrast, the input ciphertexts are encrypted under Trivium using r, leading to $\mathsf{ct}(r) = \mathsf{LWE}^{n_1,q_1}_{\mathsf{sk}_1}(\Delta_1 \cdot r)$ in the FHE-encrypted domain. So we need to transform these ciphertexts into ciphertexts which encrypted the same message under TFHE.

This is done quite easily by seeing the Trivium ciphertexts (denoted $\mathsf{ct}^{\mathsf{Trivium}}(\cdot)$) as trivial TFHE ciphertexts. The idea is first to split $\mathsf{ct}^{\mathsf{Trivium}}(\cdot) = b_{63}\|b_{62}\|\ldots\|b_0$ (with $b_i \in \mathbb{F}_2$) into blocks of two bits. Each chunk is now seen as a trivial LWE ciphertext: $\mathsf{ct}(b_{2i}\|b_{2i+1}) = \mathsf{LWE}^{n_2,q_2}_{\vec{0}}(\Delta_2 \cdot (b_{2i}\|b_{2i+1}) = (\vec{0},\ \Delta_2 \cdot (b_{2i}\|b_{2i+1}))$, so that the ciphertext $\mathsf{ct}^{\rho_2}(m)$ encrypting the 64-bit m is equal to $\{\mathsf{ct}^{\rho_2}(b_{2i}\|b_{2i+1})\}_{i\in[0,\ldots,31]}$. This step is obviously adaptable to any message space msg, and is generally denoted by:

$$\mathsf{ct}^{\rho_2}[\mathsf{ct}^{\mathsf{Trivium}}(m)] \leftarrow \mathsf{TrivialSplitting}(\log_2(\mathsf{msg}_2), \mathsf{ct}^{\mathsf{Trivium}}(m))$$

Now, the encrypted randomness $\mathsf{ct}(r)$ needs to be cast from the precision $\mathsf{p}_1 = 2$ (with $\mathsf{msg}_1 = \mathsf{carry}_2 = 2$) to $\mathsf{p}_2 = 16$. This is achieved by the process described in Figure 4. This can also be parallelized, with one thread per pair of bits, so 32 threads per step. Each of these threads will perform a leveled addition, an LWE keyswitch, and a bitshift (this last one will also perform a PBS). After all this is done, we have produced a stream of ciphertexts, interoperable via FHE with the radix representation of any high-level integer of TFHE-rs. Finally, for transciphering, we then XOR each of the resulting ciphertext with an element of the radix representation of a `FheUint64`, again done 32 times in parallel, and each of these XOR operations also requiring a PBS. All in all, this transciphering step costs a latency of two PBS operations when fully parallelized.

Transciphering between Trivium and TFHE

Notations:

- $\mathsf{sk}_{\mathsf{Trivium}}$: Trivium secret key
- **IV**: Trivium input vector
- $\mathsf{Encrypt}^*$: random generation of 64 bits using Trivium (i.e., the XOR step is ignored)
- sk_{ρ_i}: TFHE secret key associated to the parameter set ρ_i
- $\mathsf{ksk}_{\rho_i}, \mathsf{bsk}_{\rho_i}$: Evaluation keys (i.e., keyswitching and bootstrapping)

$\mathcal{C}(\mathsf{sk}_\mathsf{T}, \mathsf{sk}_{\rho_1}, m)$:

1. $\mathbf{IV} \overset{\$}{\leftarrow} \mathbb{F}_2^{80}$
2. $r \leftarrow \mathsf{Trivium.Encrypt}^*(\mathsf{sk}_T, \mathrm{IV})$
3. $\mathsf{ct}^{\mathsf{Trivium}}(m) \leftarrow m \text{ XOR } r$
4. Send to $\mathcal{S}$: $(\mathsf{ct}^{\mathsf{Trivium}}(m), \mathbf{IV}, \mathsf{ct}^{\rho_1}(\mathsf{sk}_T))$

$\mathcal{S}(\mathsf{ksk}_{\rho_1\to\rho_2}, \{\mathsf{ksk}_{\rho_i}, \mathsf{bsk}_{\rho_i}\}_{i\in[1,2]})$:

1. $\mathsf{ct}^{\rho_1}(r) \leftarrow \mathsf{TFHE.EvalCircuit}((\mathsf{ksk}_{\rho_1}, \mathsf{bsk}_{\rho_1}), \mathsf{Trivium.Encrypt}^*(\mathsf{ct}^{\rho_1}(\mathsf{sk}_T), \mathbf{IV}))$
2. $\mathsf{ct}^{\rho_2}(r) \leftarrow \mathsf{TFHE.Casting}(\mathsf{ksk}_{\rho_1\to\rho_2}, \mathsf{bsk}_{\rho_2}, \mathsf{ksk}_{\rho_2}, \mathsf{ct}^{\rho_1}(r))$
3. $\mathsf{ct}^{\rho_2}[\mathsf{ct}^{\mathsf{Trivium}}(m)] \leftarrow \mathsf{TrivialSplitting}(\log_2(\mathsf{msg}_2), \mathsf{ct}^{\mathsf{Trivium}}(m))$
4. $\mathsf{ct}^{\rho_2}(m) \leftarrow \mathsf{TFHE.EvalCircuit}((\mathsf{ksk}_{\rho_2}, \mathsf{bsk}_{\rho_2}), \mathsf{ct}^{\rho_2}[\mathsf{ct}^{\mathsf{Trivium}}(m)] \text{ XOR } \mathsf{ct}^{\rho_2}(r))$

Fig. 5. Transciphering Algorithm using TFHE and Trivium.

5 Experimental Evaluation

In the last section we detailed how we implemented Trivium and Kreyvium using TFHE. In this section we explain how we implemented the transciphering operations ontop of the **TFHE-rs** library.

5.1 Multithreading Strategy

We chose to implement the Trivium and Kreyvium encryption schemes using the **TFHE-rs** library[5]. In all of the following cases we used multithreading to process 64 bits in parallel (or 8 bytes, when applicable). Additionally, in each of the 64 (or 8) threads, we further subdivide the workload as much as possible since the algorithms are composed of 3 or 4 independent computation blocks.

We explain how the Trivium and Kreyvium design allows us to clock 64-bits of output in one execution; with maximum thread utilization. Recall, for Trivium and Kreyvium, that processing 64 bits of state in parallel is enabled by the ciphers design.

In addition in the case of Trivium (see Figure 1), the steps 4a, 4b, and 4c can be done in parallel, and after that the steps 4d, 4e, 4f, and 4g can be done in parallel. In the case of Kreyvium (see Figure 2), the same parallelization scheme would work: first steps 4a, 4b, and 4c, and then the sets 4d, 4e, 4f, and 4g. The total maximum number of threads that can be used at one time is then $64 \times 4 = 256$. This can potentially be achieved with an actual machine. However to simplify the implementation and handle a possibly low CPU count, we use `Rayon` (a Rust crate for multithreading). The advantage is that it does not instantiate more threads than the CPU count, but rather launches 256 jobs that are to be consumed by the actual launched threads.

[5] Available from `https://github.com/zama-ai/tfhe-rs`.

We now turn to our experimental evaluation of the transciphering operation. Recall we maintain two parameter sets, given in Table 1, one to compute homomorphically the ciphers Trivium and Kreyvium, and one to compute generic TFHE computations, we also maintain keyswitching keys to go between the two representation. Each parameter set is defined to offer 128-bit security, and to guarantee an error probability bound on computation of 2^{-40}.

Parameter		Trivium/Kreyvium Evaluation Parameters (ρ_1)	TFHE Integer Evaluation Parameters (ρ_2)	Key Switching $\mathsf{ksk}_{\rho_1 \to \rho_2}$ Parameters
LWE dimension	n	684	742	/
GLWE dimension	k	3	1	/
Polynomial size	N	512	2048	/
LWE standard deviation	σ_{LWE}	2.04378×10^{-5}	7.06984×10^{-6}	/
GLWE standard deviation	σ_{GLWE}	3.45253×10^{-12}	2.94036×10^{-16}	/
PBS base log	$\log_2(\beta_{\mathsf{PBS}})$	18	23	/
PBS level	ℓ_{PBS}	1	1	/
KeySwitch base log	$\log_2(\beta_{\mathsf{KS}})$	4	3	1
KeySwitch level	ℓ_{KS}	3	5	15
Message Space	msg	2	4	/
Carry Space	carry	2	4	/

Table 1. Cryptographic parameters

We can now outline our experimental results. All execution times were obtained on an AWS m6i.metal machine, with 128 virtual CPUs, 512 GB of RAM, and a clock speed of 3.5 GHz. Our implementation takes some advantage of native CPU instructions, such as SIMD and AVX instructions. We timed the four values;

- The *warm-up time*. This is the average time to execute $1152/64 = 18$ 64-bit cycles of the main loop. This is the delay one needs to pay when initializing the symmetric ciphers with a new homomorphically encrypted key.
- The *latency*. This is the average time difference between the 30'th and the 31'st round of producing 64-bit outputs. This measures the time a user needs to wait, having processed one block of 64-bits, before the next block is ready.
- The *throughput*. This is the average number of bits per second produced by the cipher, after the warmup phase, when run for a minute on the above processor with no other operations being carried out.
- The *transciphering*. This is time needed to fully transcipher a `FheUint64` ciphertext (the most expensive one), including the generation of the 64 bits (this was not done on the implementations that used the `FheBool` type, as key switching in this context was not directly available).

Our results, averaged over 100 executions, are given in Table 2. Thus after the warmup phase, we are able to obtain a sustained throughput of over 500 bits per second (resp. over 400 bits per second) for Trivium (resp. Kreyvium). This equates to a transciphering speed of under 300 ms per 64-bit plaintext block.

Encryption Scheme	FHE Type	Warm-Up (ms)	Latency (ms)	Throughput (bit/s)	Transciphering (ms)
Trivium	`FheBool`	2676	161	398	n/a
Trivium	`FheUint8`	12483	714	90	980
Trivium	Optimized version	2259	121	529	259
Kreyvium	`FheBool`	2828	168	381	n/a
Kreyvium	`FheUint8`	12932	768	83	1043
Kreyvium	Optimized version	2883	150	427	291

Table 2. Run time results

Acknowledgements

The authors would like to thank Christian Rechberger and Samuel Tap for helpful conversations during the work on this paper. The work of the third author was supported by CyberSecurity Research Flanders with reference number VR20192203, by the FWO under an Odysseus project GOH9718N.

References

AAB+20. Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Transactions on Symmetric Cryptology*, 2020(3):1–45, 2020.

ADMS09. Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In Orr Dunkelman, editor, *Fast Software Encryption – FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 1–22, Leuven, Belgium, February 22–25, 2009. Springer, Heidelberg, Germany.

AGR+16. Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 191–219, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.

ARS+15. Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

BBB+23. Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (t) fhe. *Journal of Cryptology*, 36(3):28, 2023.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

BIP+22. Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 188–215, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany.

BKM11. Julia Borghoff, Lars R. Knudsen, and Krystian Matusiewicz. Hill climbing algorithms and Trivium. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *SAC 2010: 17th Annual International Workshop on Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 57–73, Waterloo, Ontario, Canada, August 12–13, 2011. Springer, Heidelberg, Germany.

Bra12. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

CCF+16. Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In Thomas Peyrin, editor, *Fast Software Encryption – FSE 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 313–333, Bochum, Germany, March 20–23, 2016. Springer, Heidelberg, Germany.

CDG+17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1825–1842, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

CDPP22. Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. SortingHat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 563–577, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

CGB+17. Marco Cianfriglia, Stefano Guarino, Massimo Bernaschi, Flavio Lombardi, and Marco Pedicini. A novel GPU-based implementation of the cube attack - preliminary results against Trivium. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*, volume 10355 of *Lecture Notes in Computer Science*, pages 184–207, Kanazawa, Japan, July 10–12, 2017. Springer, Heidelberg, Germany.

CGGI20. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

CHMS22. Orel Cosseron, Clément Hoffmann, Pierrick Méaux, and François-Xavier Standaert. Towards case-optimized hybrid homomorphic encryption - featuring the elisabeth stream cipher. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part III*, volume 13793 of *Lecture Notes in Computer Science*, pages 32–67, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany.

CIR22. Carlos Cid, John Petter Indrøy, and Håvard Raddum. FASTA - A stream cipher for fast FHE evaluation. In Steven D. Galbraith, editor, *Topics in Cryptology – CT-RSA 2022*, volume 13161 of *Lecture Notes in Computer Science*, pages 451–483, Virtual Event, March 1–2, 2022. Springer, Heidelberg, Germany.

Dam90. Ivan Damgård. On the randomness of Legendre and Jacobi sequences. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO'88*, volume 403 of *Lecture Notes in Computer Science*, pages 163–172, Santa Barbara, CA, USA, August 21–25, 1990. Springer, Heidelberg, Germany.

DDGP22. Stéphanie Delaune, Patrick Derbez, Arthur Gontier, and Charles Prud'homme. A simpler model for recovering superpoly on trivium. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021: 28th Annual International Workshop on Selected Areas in Cryptography*, volume 13203 of *Lecture Notes in Computer Science*, pages 266–285, Virtual Event, September 29 – October 1, 2022. Springer, Heidelberg, Germany.

De 06. Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC 2006: 9th International Conference on Information Security*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186, Samos Island, Greece, August 30 – September 2, 2006. Springer, Heidelberg, Germany.

DEG+18. Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: A cipher with low ANDdepth and few ANDs per bit. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 662–692, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

DGH+21. Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Pasta: A case for hybrid homomorphic encryption. Cryptology ePrint Archive, Report 2021/731, 2021. `https://eprint.iacr.org/2021/731`.

DLR16. Sébastien Duval, Virginie Lallemand, and Yann Rotella. Cryptanalysis of the FLIP family of stream ciphers. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 457–475, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`.

FV14. Pierre-Alain Fouque and Thomas Vannet. Improving key recovery to 784 and 799 rounds of Trivium using optimized cube attacks. In Shiho Moriai, editor, *Fast Software Encryption – FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 502–517, Singapore, March 11–13, 2014. Springer, Heidelberg, Germany.

FWDM18. Ximing Fu, Xiaoyun Wang, Xiaoyang Dong, and Willi Meier. A key-recovery attack on 855-round Trivium. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 160–184, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

GHS12. Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

GKR$^+$21. Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 519–535. USENIX Association, August 11–13, 2021.

GRR$^+$16. Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. MPC-friendly symmetric key primitives. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 430–443, Vienna, Austria, October 24–28, 2016. ACM Press.

HJL$^+$20. Yonglin Hao, Lin Jiao, Chaoyun Li, Willi Meier, Yosuke Todo, and Qingju Wang. Links between division property and other cube attack variants. *IACR Transactions on Symmetric Cryptology*, 2020(1):363–395, 2020.

HKL$^+$22. Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Jooyoung Lee, and Mincheol Son. Rubato: Noisy ciphers for approximate homomorphic encryption. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 581–610, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

HL11. ZhenYu Huang and Dongdai Lin. Attacking Bivium and Trivium with the characteristic set method. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT 11: 4th International Conference on Cryptology in Africa*, volume 6737 of *Lecture Notes in Computer Science*, pages 77–91, Dakar, Senegal, July 5–7, 2011. Springer, Heidelberg, Germany.

HL20. Phil Hebborn and Gregor Leander. Dasta – alternative linear layer for Rasta. *IACR Transactions on Symmetric Cryptology*, 2020(3):46–86, 2020.

HLM$^+$20. Yonglin Hao, Gregor Leander, Willi Meier, Yosuke Todo, and Qingju Wang. Modeling for three-subset division property without unknown subset - improved cube attacks against Trivium and Grain-128AEAD. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 466–495, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.

HMR20. Clément Hoffmann, Pierrick Méaux, and Thomas Ricosset. Transciphering, using FiLIP and TFHE for an efficient delegation of computation. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *Progress in Cryptology - INDOCRYPT 2020: 21st International Conference in Cryptology in India*, volume 12578 of *Lecture Notes in Computer Science*, pages 39–61, Bangalore, India, December 13–16, 2020. Springer, Heidelberg, Germany.

ISO12. ISO. ISO/IEC 29192-3:2012: Information technology – Security techniques – Lightweight cryptography – Part 3: Stream ciphers, 2012.

KMN12. Simon Knellwolf, Willi Meier, and María Naya-Plasencia. Conditional differential cryptanalysis of Trivium and KATAN. In Ali Miri and Serge Vaudenay, editors, *SAC 2011: 18th Annual International Workshop on Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 200–212, Toronto, Ontario, Canada, August 11–12, 2012. Springer, Heidelberg, Germany.

MB07. Alexander Maximov and Alex Biryukov. Two trivial attacks on Trivium. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007: 14th Annual International Workshop on Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 36–55, Ottawa, Canada, August 16–17, 2007. Springer, Heidelberg, Germany.

MCJS19. Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. Improved filter permutators for efficient FHE: Better instances and implementations. In Feng Hao, Sushmita Ruj, and

Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India*, volume 11898 of *Lecture Notes in Computer Science*, pages 68–91, Hyderabad, India, December 15–18, 2019. Springer, Heidelberg, Germany.

MJSC16. Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards stream ciphers for efficient FHE with low-noise ciphertexts. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 311–343, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

NIS22. NIST. Status report on the third round of the NIST Post-Quantum Cryptography standardization process. `https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf`, 2022.

NR97. Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th Annual Symposium on Foundations of Computer Science*, pages 458–467, Miami Beach, Florida, October 19–22, 1997. IEEE Computer Society Press.

PSSW09. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

RSS17. Dragos Rotaru, Nigel P. Smart, and Martijn Stam. Modes of operation suitable for computing on encrypted data. *IACR Transactions on Symmetric Cryptology*, 2017(3):294–324, 2017.

SMK22. Roy Stracovsky, Rasoul Akhavan Mahdavi, and Florian Kerschbaum. Faster evaluation of AES using TFHE. Poster Session, FHE.Org - 2022, 2022.

WB10. Kenneth Koon-Ho Wong and Gregory V. Bard. Improved algebraic cryptanalysis of QUAD, Bivium and Trivium via graph partitioning on equation systems. In Ron Steinfeld and Philip Hawkes, editors, *ACISP 10: 15th Australasian Conference on Information Security and Privacy*, volume 6168 of *Lecture Notes in Computer Science*, pages 19–36, Sydney, NSW, Australia, July 5–7, 2010. Springer, Heidelberg, Germany.

WHT+18. Qingju Wang, Yonglin Hao, Yosuke Todo, Chaoyun Li, Takanori Isobe, and Willi Meier. Improved division property based cube attacks exploiting algebraic properties of superpoly. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 275–305, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

WIM17. Yuhei Watanabe, Takanori Isobe, and Masakatu Morii. Conditional differential cryptanalysis for kreyvium. In Josef Pieprzyk and Suriadi Suriadi, editors, *ACISP 17: 22nd Australasian Conference on Information Security and Privacy, Part I*, volume 10342 of *Lecture Notes in Computer Science*, pages 421–434, Auckland, New Zealand, July 3–5, 2017. Springer, Heidelberg, Germany.

YT18. Chen-Dong Ye and Tian Tian. A new framework for finding nonlinear superpolies in cube attacks against Trivium-like ciphers. In Willy Susilo and Guomin Yang, editors, *ACISP 18: 23rd Australasian Conference on Information Security and Privacy*, volume 10946 of *Lecture Notes in Computer Science*, pages 172–187, Wollongong, NSW, Australia, July 11–13, 2018. Springer, Heidelberg, Germany.

YT21. Chen-Dong Ye and Tian Tian. A practical key-recovery attack on 805-round trivium. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part I*, volume 13090 of *Lecture Notes in Computer Science*, pages 187–213, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany.

ZLL18. Xiaojuan Zhang, Meicheng Liu, and Dongdai Lin. Conditional cube searching and applications on Trivium-variant ciphers. In Liqun Chen, Mark Manulis, and Steve Schneider, editors, *ISC 2018: 21st International Conference on Information Security*, volume 11060 of *Lecture Notes in Computer Science*, pages 151–168, Guildford, UK, September 9–12, 2018. Springer, Heidelberg, Germany.