Trivial Transciphering With Trivium and TFHE

Thibault Balenbois\textsuperscript{1}, Jean-Baptiste Orfila\textsuperscript{1} and Nigel P. Smart\textsuperscript{1,2}

\textsuperscript{1} Zama Inc., Paris, France.
\textsuperscript{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. Originally these two 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 transpose 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** ................................................. 3
   1.1 Prior Work on Performance of FHE Transciphering .......... 4
   1.2 Our Contribution ........................................... 5
2. **Trivium and Kreyvium** ..................................... 6
   2.1 Trivium .................................................... 6
   2.2 Kreyvium .................................................. 6
3. **Transciphering in TFHE** .................................. 8
   3.1 Generic Transciphering Protocol .......................... 8
   3.2 TFHE Scheme and Large Integer Representation ............ 8
   3.3 Casting between TFHE encryptions ........................ 9
4. **Implementation of Trivium in TFHE** ....................... 10
   4.1 Multithreading Strategy .................................. 10
   4.2 Three Potential Underlying Data Types .................... 11
   4.3 Transciphering ............................................ 12
5. **Experimental Evaluation** ................................ 13

References .................................................................. 14
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 intervening 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, and out of, an MPC/FHE system. 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 (so should it be Fasta?) than the communities ability to apply cryptanalytic effort to them. As just one 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. 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 and BFV. However, for FHE systems such as TFHE 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. 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.

The security of Trivium is well established, with only some attacks on it, or closely related ciphers, having been presented. However, the “security margin” for Trivium is now considered to be relatively small.

This small security margin led Canteaut et al. 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, and on both Trivium and Kreyvium. Theoretical key recovery attacks have been proposed against 839 round Trivium and 891 round Kreyvium, and a distinguisher on 899 round Kreyvium was presented. A practical key recover attack against 805 round Trivium was presented. 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 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 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.

As remarked above Kreyvium was actually introduced in the context of trying to find a cipher which is FHE-friendly. However, the paper introducing Kreyvium looked at transciphering in the context of FHE schemes such as BGV and BFV, for which it is not ideally suited. The

---

3 The NIST report on their choice of SPHINCS+ vs Picnic states “NIST chose SPHINCS+ largely because it could not confidently quantify the security of LowMC.”
reported performance of Trivium and Kreyvium 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 \( \mathbb{F}_2 \):

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

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.

**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 \( \text{IV} \). 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 \( \text{IV} \) are placed in the \( b \) register, the
remaining 44 bits of \( \mathbf{IV} \) 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 \( \mathbf{IV} \) 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 \( \mathbf{IV} \) register is added into the update to the \( b \) register, so we have

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

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. \((s_{178}, \ldots, s_{288}) \leftarrow (IV_{84}, \ldots, IV_{127}, \ldots, 1, \ldots, 1)\).
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 \( \mathbf{FHE} \). To illustrate the usage, let us
assume a classical scenario where a client C sends their encrypted data to a server 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; $E = (\text{KeyGen}, \text{Encrypt}, \text{Decrypt})$ and $FHE = (\text{KeyGen}, \text{Encrypt}, \text{Decrypt}, \text{EvalCircuit})$. The process is described in Figure 3.

**Transciphering**

C:
1. $sk_E \leftarrow E.\text{KeyGen}(1^λ)$
2. $(evk_{FHE}, sk_{FHE}) \leftarrow FHE.\text{KeyGen}(1^λ)$
3. Let $m_1, \ldots, m_k \in \mathbb{Z}/p\mathbb{Z}$ for some $k \in \mathbb{N}$ be the cleartexts.
4. For $i \in [1..k]$, $ct_i \leftarrow E.\text{Encrypt}(sk_E, m_i)$
5. $ct_{sk_E} \leftarrow FHE.\text{Encrypt}(sk_{FHE}, sk_E)$
6. Send to S: $(ct_0, \ldots, ct_k, ct_{sk_E})$

S:
1. For $i \in [1..k]$, $ct_i' \leftarrow FHE.\text{EvalCircuit}(evk_{FHE}, E.\text{Dec}(ct_{sk_E}, ct_i))$

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

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 $sk \overset{S}{\leftarrow} \mathcal{S}$ (S could be a binary, ternary or discrete Gaussian distribution), a mask $a \overset{S}{\leftarrow} \mathbb{Z}/q\mathbb{Z}$ (with $q$ the ciphertext modulus), a scaling factor $\Delta = \frac{q}{2p}$, and some noise $e \overset{S}{\leftarrow} \mathcal{N}_{\sigma^2}$ ($\mathcal{N}_{\sigma^2}$ is a discrete Gaussian of variance $\sigma^2$, assumed to be centered in 0), by the equation

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

Programmable bootstrapping (PBS) gives the possibility to homomorphically evaluate any univariate functions $f(\cdot) : \mathbb{Z}/p\mathbb{Z} \rightarrow \mathbb{Z}/p\mathbb{Z}$. In order to extend this to bivariate functions $g(\cdot, \cdot) : \mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/p\mathbb{Z} \rightarrow \mathbb{Z}/p\mathbb{Z}$, the idea is to split $p$ into two parts: the message msg and carry spaces carry. For instance, for a two bits of message space we have $msg = 4$, and for three bits of carry space we have $carry = 8$. We pick these values so that $msg \leq carry$ 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., $ct_{\text{res}} = msg \cdot ct_1 + ct_2$), 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 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 is going 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 $f(\cdot)$:

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

with a simple keyswitch denoted by:

$$ct_{out}(m_i) \leftarrow Keyswitch(ksk, ct(m))$$

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 $pt \in \mathbb{Z}/p\mathbb{Z}$ be the plaintext, let $\beta \in \mathbb{N}$ be the basis, such that $|\beta| \leq 10$. The $\beta$-radix decomposition of $pt$ can be written as:

$$pt = \sum_{i=0}^{d-1} pt_i \cdot \beta^i,$$

for some $d \in \mathbb{N}$ and $0 \leq pt_i < \beta$ for $i \in [0, d-1]$. Then, an encryption of $pt$ is:

$$ct(pt) = \{LWE_{sk}^{n,q}(pt_i)\}_{i \in [0,d-1]}$$

In what follows, we denote $\rho_i$ the set of parameters associated to a precision $p_i$. A parameter set contains values ensuring secure LWE instances (i.e., $n$, $q$, $\sigma_{LWE}$), secure GLWE instances for the $bsk$ (i.e., the GLWE dimension $k$, the polynomial size $N$, and the standard deviation $\sigma_{GLWE}$) and correctness parameters (i.e., the decomposition bases and levels for the PBS and the KS, $\beta_{PBS}$, $\ell_{PBS}$, $\beta_{KS}$, $\ell_{KS}$). A ciphertext associated to a parameter set $\rho$ 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 a approach allowing casting from a smaller precision $p_1$ to a larger one $p_2$ (where $p_i = \log_2(\text{msg}_i \cdot \text{carry}_i)$). 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 $\text{msg}_i$. Then, a keyswitch is applied to switch from the first set of parameter to the second. This requires a dedicated keyswitching key, denoted $ksk_{\rho_1 \rightarrow \rho_2}$, going from the parameter set $\rho_1$ to $\rho_2$. Finally, a PBS is applied in order to go from the scaling factor $\Delta_1$ to $\Delta_2$. The process is described in Figure[4].
### Casting

**Conditions:**
1. \( \text{msg}_1 \cdot \text{carry}_1 \geq \text{msg}_2 \);
2. \( p_2 \geq p_1 \)

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

**Output:** A ciphertext \( \text{ct}^{\rho_2}(m) = \{ \text{LWE}_{\rho_2}^{n_2}, q_2, s_2^{\rho_2}(\Delta_2 \cdot m'_i) \}_{i \in [0, \kappa - 1]} \)

**Algorithm:**
1. For \( i \in \left[ 0; \left\lfloor \frac{\log_2(\text{msg}_1)}{\log_2(\text{msg}_2)} \right\rfloor \right] \):
   (a) // **Packing**
      For \( j \in [0; \log_2(\text{msg}_1)] \):
      i. \( \text{ct}^{\rho_1}(m_i) \leftarrow \text{ct}^{\rho_1}(m_i) + 2^j \cdot \log_2(\text{msg}_1) \cdot \text{ct}^{\rho_1}(m_j) \)
   (b) // **Switching to the second parameter set**
      \( \text{ct}^{\rho_2}(m_i) \leftarrow \text{Keyswitch}(\text{ksk}_{\rho_1 \rightarrow \rho_2}, \text{ct}(m_i)) \)
   (c) // **Adjusting to the scaling factor** \( \Delta_2 \)
      \( \text{ct}^{\rho_2}(m_i) \leftarrow \text{PBS}(\text{bsk}, \text{ksk}, \text{ct}^{\rho_2}(m_i), x \mapsto x \gg \log_2 \left( \frac{\Delta_1}{\Delta_2} \right) \)
2. Return \( \text{ct}^{\rho_2}(m_i) \)

---

**Fig. 4.** Casting Algorithm between two LWE ciphertexts

### 4 Implementation of Trivium in TFHE

There are various design choices in how one could implement transciphering in TFHE. In this section we outline the ones we investigated.

#### 4.1 Multithreading Strategy

We chose to implement the Trivium and Kreyvium encryption schemes using the [TFHE-rs library](https://github.com/zama-ai/tfhe-rs).

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 is composed by 3 or 4 independent computation blocks.

In the case of Trivium, 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, 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.

---

[Available from](https://github.com/zama-ai/tfhe-rs)
4.2 Three Potential Underlying Data Types

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

**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.

**FheUint8:** A second naïve implementation would use the *FheUint8* type, representing a byte encrypted via the high-level TFHE API. Each byte standing in for 8 bits of the original Trivium or Kreyvium cipher; be it from the key, registers, messages, etc. Using encrypted bytes is mirroring a cleartext implementation on a modern machine, where bytes are the logical data unit. In practice, all the high-level integer types of the TFHE-rs library are radix representation of the underlying actual integer, using ciphertexts with $msg = 4$, 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 reconstructing bytes from two bytes of the registers, a costly operation. However, what this also means is that it is straightforward 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 completeness, but 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 use 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. 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 levelled addition in the language of TFHE). Meaning we can perform two (levelled) 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.

Since the PBS operation is the most costly operation (by far), we tried to optimize them out of the circuit as much as possible. Every XOR gate was be represented by a (levelled) 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, then a ”clean carry” operation [one PBS];
  • Step 4e: as an AND gate [one PBS] followed by two (leveled) additions, then a ”clean carry” operation, [one PBS] (for Kreyvium the clean carry is replaced with a proper addition and a 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];
– Return: r, t1, t2, t3

Making a total of seven 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.

4.3 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 corresponding 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.

Following Figure 3, 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 (e.g., FheUint64). Then, the encrypted randomness is: 
\[ ct(r) = LWE_{sk, q_1}(\Delta_1 \cdot r) \]

In contrast, the input ciphertexts are encrypted under Trivium, 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 \( ct^{Trivium}(\cdot) \)) as trivial TFHE ciphertexts. The idea is first to split \( ct^{Trivium}(\cdot) = b_{63} || b_{62} || \ldots || b_0 \) (with \( b_i \in \mathbb{F}_2 \)) into blocks of two bits. Each chunk now seen as a trivial LWE ciphertext: 
\[ ct(b_{2i} || b_{2i+1}) = LWE_{sk, q_2}^{\Delta_2}(b_{2i} || b_{2i+1}) \]
so that the ciphertext \( ct^{p_2}(m) \) encrypting the 64-bit \( m \) is equal to \( \{ct(b_{2i} || b_{2i+1})\} \). This step is obviously adaptable to any message space \( msg \), and is generally denoted by:

\[ ct^{p_2} \leftarrow \text{TrivialSplitting}(\log_2(msg_2), ct^{Trivium}(m)) \]

Now, the key needs to be cast from the precision \( p_1 = 2 \) (with \( msg_1 = carry_2 = 2 \)) to \( p_2 = 16 \). This is achieved by the process described in Figure 4. 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. 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). 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.
Notations:
- \(sk_T\): Trivium secret key
- \(IV\): Trivium input vector
- \(\text{Encrypt}^*\): random generation of 64 bits using Trivium (i.e., the XOR step is ignored)
- \(sk_{\rho_i}\): TFHE secret key associated to the parameter set \(\rho_i\)
- \(ksk_{\rho_i}, bsk_{\rho_i}\): Evaluation keys (i.e., keyswitching and bootstrapping)

\[C(sk_T, sk_{\rho_{1}}, m):\]
1. \(IV \leftarrow \mathbb{F}_2^{80}\)
2. \(r \leftarrow \text{Trivium.Encrypt}^*(sk_T, IV)\)
3. \(ct^{\text{Trivium}}(m) \leftarrow m \text{ XOR } r\)
4. Send to \(S\): \((ct^{\text{Trivium}}(m), IV, ct^{\rho_1}(sk_T))\)

\[S(ksk_{\rho_{1} \rightarrow \rho_{2}}, \{ksk_{\rho_{1}}, bsk_{\rho_{1}}\}_{i \in [1,2]}):\]
1. \(ct^{\rho_1}(r) \leftarrow \text{TFHE.EvalCircuit}((ksk_{\rho_2}, bsk_{\rho_2}, \text{Trivium.Encrypt}^*(ct(sk_T), IV))\)
2. \(ct^{\rho_2}(r) \leftarrow \text{TFHE.Casting}(ksk_{\rho_1 \rightarrow \rho_2}, bsk_{\rho_2}, ksk_{\rho_2}, ct(r))\)
3. \(ct^{\rho_2}(ct^{\rho_1}(m)) \leftarrow \text{TriviumSplitting}(\log_2(msg_2), ct^{\text{Trivium}}(m))\)
4. \(ct^{\rho_2}(m) \leftarrow \text{TFHE.EvalCircuit}(ksk_{\rho_2}, bsk_{\rho_2}, ct^{\rho_2}(ct^{\rho_1}(m) \text{ XOR } r))\)

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; basing our implementation on top of the TFHE.rs library. We explain how the Trivium and Kreyvium design allows us to clock 64-bits of output in one execution; with maximum thread utilization.

Recall we maintain two parameter sets, give 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 \(2^{-40}\).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Trivium/Kreyvium Evaluation Parameters ((\rho_1))</th>
<th>TFHE Integer Evaluation Parameters ((\rho_2))</th>
<th>Key Switching Parameters ((ksk_{\rho_1 \rightarrow \rho_2}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWE dimension</td>
<td>(n)</td>
<td>684</td>
<td>742</td>
</tr>
<tr>
<td>GLWE dimension</td>
<td>(k)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Polynomial size</td>
<td>(N)</td>
<td>512</td>
<td>2048</td>
</tr>
<tr>
<td>LWE standard deviation</td>
<td>(\sigma_{\text{LWE}})</td>
<td>(2.04378 \times 10^{-5})</td>
<td>(7.06984 \times 10^{-6})</td>
</tr>
<tr>
<td>GLWE standard deviation</td>
<td>(\sigma_{\text{GLWE}})</td>
<td>(3.45253 \times 10^{-12})</td>
<td>(2.94036 \times 10^{-16})</td>
</tr>
<tr>
<td>PBS base log</td>
<td>(\log_2(\beta_{\text{PBS}}))</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td>PBS level</td>
<td>(\ell_{\text{PBS}})</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Keyswitch base log</td>
<td>(\log_2(\beta_{\text{KS}}))</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Keyswitch level</td>
<td>(\ell_{\text{KS}})</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Message Space</td>
<td>(msg)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Carry Space</td>
<td>(carry)</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

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, 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.

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

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**


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...


