Silentium: Implementation of a Pseudorandom Correlation Generator for Beaver Triples

Vincent Rieder[0009−0007−8694−7260]

University of Stuttgart

Abstract. Secure Multi-Party Computation is a privacy-enhancing technology that allows several parties to securely compute on distributed private data. In the line of the well established SPDZ protocol, the by far most expensive task is the generation of Beaver triples in the so called offline phase. Silentium is our implementation of an actively secure offline phase in the form of a Pseudorandom Correlation Generator for Beaver triples (Bt-PCG, Boyle et al. CRYPTO 2020), which, as any PCG, is designed to have low communication. Compared to previous offline phases, their Bt-PCG reduces the communication costs by three orders of magnitude. However, so far efficiency was only estimated. With Silentium, we demonstrate that their Bt-PCG can achieve even better running times than state-of-the-art offline phase implementations in the MP-SPDZ library. To actually achieve such a performance, Silentium comprises a systematic parallelization strategy and implementation-friendly decomposition scenarios of the Bt-PCG into structured modules. Looking forward for large-scale applications on the cloud, Silentium is designed to be versatile to support hardware acceleration in future.

Keywords: Secure Multi-Party Computation · Beaver Triples · Pseudorandom Correlation Generators

Secure multi-party computation (MPC) is a privacy-enhancing technology that allows several parties to evaluate a public function without leaking private inputs and outputs. Silentium is our implementation of a recent protocol for silent MPC [9], i.e., we bring forward practical MPC solutions with low communication. Our motivation is MPC on cloud servers, e.g., with the Carbyne Stack platform[1] based on the SPDZ protocol [14], where typically communication is a bottleneck due to high latency. More generally, the cloud setting is promising to outsource large-scale and industrial applications, e.g., privacy-preserving machine learning. In this sense, the attractiveness of MPC on the cloud is determined not only by communication but also computation. Concretely, Silentium is our implementation of a recent Pseudorandom Correlation Generator (PCG) for the generation of Beaver triples [9], the most expensive, and cryptographically challenging, task in the line of the SPDZ protocol. While this PCG, as any PCG, is designed to have low communication, so far concrete efficiency was

[1] https://carbynestack.io/

only estimated. With Silentium[2] we prove that their PCG is indeed attractive in practice.

1.1 Technical Context

To distinguish our work from other MPC approaches, we mention three main MPC assumptions of Silentium. Firstly, follwoing the SPDZ protocol, Silentium is in the domain of MPC over arithemtic circuits based on additive secret-sharing, in contrast to MPC over binary circuits based on e.g. garbled circuits [33]. Secondly, following the Bt-PCG, Silentium is restricted to the two-party setting, which is already sufficient to outsource a secure computation for any number of parties to (two) cloud servers [3]. Finally, we consider the actively secure Bt-PCG formulation [9], i.e., with strong security guarantees even if one untrusted party might actively deviate from the underlying protocols. This strong scenario gives stronger security guarantees and compliance for industrial applications.

1.2 Silentium as Offline Phase in the Line of SPDZ

The basic MPC framework of our work is the SPDZ protocol [14]. Protocols in the line of SPDZ can be seen as standard approach for MPC over arithmetic circuits (encoding the real-world function to evaluate). The crucial point of the SPDZ protocol is the preprocessing model [3] with two phases. In this model, the actual secure function evaluation (online phase) consumes distributed correlated randomness that is provided by a preceding, input-independent, offline phase. While this separation allows the construction of online phases with little overhead compared to an unprotected (local) function evaluation, the construction of actively secure offline phases is challenging in terms computation and communication. For SPDZ the offline phase mainly refers to the generation of Beaver triples, also known as authenticated multiplication triple, each supporting one multiplication in the online phase. For the cloud context, Silentium targets for large-scale applications, that means for the generation of millions of Beaver triples.

In the last years, the initial offline phase for SPDZ was improved in several directions [3,13,14], with ready-to-use implementations in the MP-SPDZ library [22]. While these typically rely on homomorphic encryption, Silentium can be seen as an alternative offline phase implementation, following the recent design of Pseudorandom Correlation Generators [8]. Concretely we implement a PCG for Beaver triples (Bt-PCG). In general, PCGs come with low-communication by design. [1,6,31]. The basis of Silentium is the Bt-PCG from Boyle et al [9], which is so far the only attractive Bt-PCG (see below).

As a baseline, Silentium realizes the generation of about one million Beaver triples over an 128-bit field (that means 100 MB of correlated randomness) in one batch. For this amount, the Bt-PCG [9] takes a few MB of communication, while

[2] Silentium (latin for silence) denotes periods of silence in monasteries with special rules to bring communication to a minimum, in analogy to the design of PCGs.

the protocols in MP-SPDZ take a few GB of communication. In pure numbers, this reduction promises a game changing advantage in the aforementioned MPC cloud context. With Silentium we demonstrate that on the same time, the Bt-PCG is computationally as good as the protocols in MP-SPDZ.

1.3 Pseudorandom Correlation Generators

The following introduction on PCGs puts light on what actually hides beyond the term Bt-PCG by giving relevant background on PCGs.

In general, PCGs [8] are a recent primitive for MPC offline phases with a focus on low communication. PCGs [5,8] can be seen as a distributed generalization of PRGs to expand small seeds into a large batch of correlated pseud-randomness that is distributed cryptographically close to a given distributed target correlation. A PCG consists of two stages: In a first step, private seeds are generated, which in our use case of an MPC offline phase takes place as actively secure seed generation protocol. In a second step, the seeds can be locally, i.e., without further communication, expanded into a large amount of correlated randomness, e.g., 2^{20} of Beaver triples in case of the Bt-PCG [9]. The point about PCGs is to be compressive in terms of small private seeds, which typically is achieved with an encoding of the target correlation under variations of the learning with error assumption [28]. While the compressive property is challenging to achieve it has two practical advantages, that especially apply to MPC in the cloud context:

- **Low Communication:** When carefully constructed, the small seeds can be generated with low communication, which typically is achieved with function secret sharing [10,11] in the form of distributed point functions (DPF) [17].
- **Reduced Storage:** The storage costs for the seed between the interactive and local phase is cheap, which becomes relevant for large-scale applications with many PCG iterations. Concretely, small seeds enable efficient MPC deployments, where the interactive phase of the PCGs can be scheduled ways before the PCG local phase and the MPC online phase. A dynamic cloud deployment can benefit from that freedom, and for example schedule seed generations at times where high bandwidth is available. Instead, dynamic resource allocation on the cloud allows to schedule local phases just in advance of an MPC online phase activation, which might depend on short-notice user inputs.

PCGs for Beaver triples: It is well known that PCGs in general can achieve very good performances [4,6,7,31,32]. However, the construction of PCGs for Beaver triples turns out to be more challenging compared to other forms of correlated randomness like VOLE. The point is that the non-linear multiplicative property of Beaver triples is in contrast with many linear building blocks in the toolbox of PCGs. Apart from initial, but inefficient, approaches [8], there exists only the Bt-PCG construction [9] for two-party Beaver triples with two follow-up works about optimizations [29] and the multi-party setting [1]. Impressively, for the Bt-PCG the generation of 2^{20} Beaver triples (128 bit field) takes only

0.5 to 30 MB per party, which is three orders of magnitude better than for the protocols in MP-SPDZ. However, there exists no implementation, mainly since the Bt-PCG is based on two special purpose primitives [9]. The first is an actively secure generation protocol for authenticated secret-shared scaled unit vectors (SUVs) [29]. So far, concrete efficiency is only based on related work [7]. The second primitive is the ring-LPN assumption over large fields, generalizing the binary-ring LPN assumption [20]. Hereby the challenge is that the ring-LPN assumptions requires large degree polynomial arithmetic, e.g., the Number Theoretic Transforms with degree 2^{20} over a 128-bit field, which is far beyond sizes for comparable applications like the lattice based protocols standardized by NIST[3].

1.4 Contribution

Silentium is the first implementation of the Bt-PCG [9] (to the best of our knowledge), proving the competitiveness with MP-SPDZ. Effectively, our best setup of the Bt-PCG runs 34% faster than respective benchmarks for LowGear, the fastest offline phase in MP-SPDZ. For fairness, we do not claim to be in fact better: Firstly, we stress that our benchmarks are only preliminary, rather than a comprehensive comparison, e.g., with respect to the selection of parameters. Secondly, while Silentium is standalone offline phase of based on the rather new PCG paradigm, the protocols in MP-SPDZ are much more established and mature in practice.

Silentium takes a two-folded approach to achieve good performance for the Bt-PCG. As a theoretical contribution, Silentium actually comprises a whole framework with systematic strategies for parallelization and decomposition (Section 1.5). On the practical side, Silentium comprises more than 20.000 lines of code, which, for good efficiency on hardware level, are mainly written in C language. The pure size of the code can be justified by the novelty and size of many building blocks, the implementation of the Silentium framework, and a versatile design. The outstanding components are:

- At the core is the first implementation of an actively secure protocol to generate authenticated secret-shared scaled unit vectors(SUVs) [9,29].
- An implementation of the Number Theoretic Transform (NTT) [26] for the extraordinary large degree 2^{20}, based on a from scratch 128-bit field arithmetic implementation. This setting is Bt-PCG specific and much more extensive compared to usual cryptography NTT applications (with degree up to 2^{15} on smaller fields, e.g. [2]).
- Silentium contains two libraries for MPC online phases (as necessary for the Bt-PCG itself): For SPDZ [14], i.e., for arithmetic operations, and for TinyOT [16,27], i.e., for binary operations. We stress that the overhead for their respective offline phases, which are not (yet) covered by Silentium, is expected to be small [9,29].

[3] https://csrc.nist.gov/projects/post-quantum-cryptography

Looking forward to large-scale MPC deployments on the cloud, we design the Silentium framework to be versatile. One goal is to prepare Silentium for special purpose hardware acceleration and thus even better performance for large-scale applications. For example, Silentium allows to easily exchange different components like the field implementation, enabling future use of specialized hardware implementations. A further aspect of versatility is the challenge to find optimal parameters configurations, which we address with our decomposition framework.

1.5 Technical Overview and Paper Outline

The preliminary section (Section 2) introduces the basis of SPDZ, that is MPC with additively secret-sharing. Section 3 describes the Bt-PCG [9] and its concrete building blocks. In Section 4, we then perform a cost analysis of the Bt-PCG, identifying relevant cost metrics and suitable actions concerning communication, computation, internal MPC operations, and memory consumption. Based on this analysis, we then derive the Silentium framework. Concretely, we introduce different techniques and scenarios to decompose the Bt-PCG into modules of implementation-friendly size (Section 5). At the core is a parallelization strategy for speed-up, operating on two levels:

- **High-level multi-tasking:** Different modules are supposed to run on different computation units like cores. This allows to balance high memory consumption.
- **Low level multi-threading:** Within each module, we synchronize identical tasks to run arithmetic operations in parallel on hardware level, and to reduce the number of communication rounds.

In Section 6 we make one further step towards an implementation by formalizing the concept of modules as executable programs. Hereby, the modules are configurable in order to make Silentium versatile. Section 7 provides some details of our implementation. The evaluation of Silentium is in Section 8.3, including a comparison to MP-SPDZ benchmarks. With Section 9 we conclude the paper with a list of possible future work.

Acknowledgements: A thanks goes to the Robert Bosch GmbH, Research Campus Renningen, where this project was located.[4] And a thanks goes to Daniele de Bernardini, Giulia Salvatori, and Enrico Sorbera supporting in coding as master students.

[4] This work has been done as part of the CRYPTECS project that received funding from the German Federal Ministry of Education and Research under Grant Agreement No. 16KIS1441 and from the French National Research Agency under Agreement Grant No. ANR-20-CYAL-0006. The author has no competing interest to declare that are relevant to the content of this article.

2 Preliminaries:Additive Secret Sharing

In general, we write $\mathbb{F}$ for a large finite field, where the bit size ν is a security parameter. We write $x \leftarrow S$ to denote that x is sampled uniformly random from a set S. We use upper indices S^c to denote c independent instances of S.

In SPDZ [14] the secure arithmetic circuit evaluation operates on additive secret-sharing with authentication for active security. We write $[x]$ for a secret-shared value $x \in \mathbb{F}$, where both parties P_σ hold a private share x_σ and a private message authentication code (MAC) x'_σ such that $x = x_0 + x_1$ and $\mathrm{m}x = x'_0 + x'_1$, where $\mathrm{m} = \mathrm{m}_0 + \mathrm{m}_1 \in \mathbb{F}$ is an additively shared global MAC key [13]. We write $x' = \mathrm{m} \cdot x$ for short. The MAC is used to ensure active security, i.e., on each secret-shared value $[x]$ that is revealed as output, a MAC check is performed to ensure that the value x was computed as intended without any corruption.

Note that due to linearity of $[\cdot]$, the online phase evaluation of addition gates in an arithmetic circuit can be done locally without communication. In contrast, multiplications are interactive, consuming one Beaver triple $[x], [y], [z]$, where $x, y \leftarrow \mathbb{F}, z = x \cdot y$. Silentium is an offline phase for the generation of Beaver triples[5], making itself use of secret-sharing $[\cdot]$. Hereby, we extend the notation $[\cdot]$ to vectors and polynomials over $\mathbb{F}$, where all coefficients are secret-shared individually. We denote an authenticated secret-shared scaled unit vector, i.e., a secret-shared element in $\mathbb{F}^N$, where exactly one position α has a non-zero payload A, as SUV.

Apart from SPDZ for arithmetic circuits, Silentium uses secret-sharing in the sense of the TinyOT protocol [27] for actively secure evaluation of binary circuits, operating on authenticated secret-shared bits $[\cdot]_2$. We extend this to bit-wise secret-shared integers $[\cdot]$.

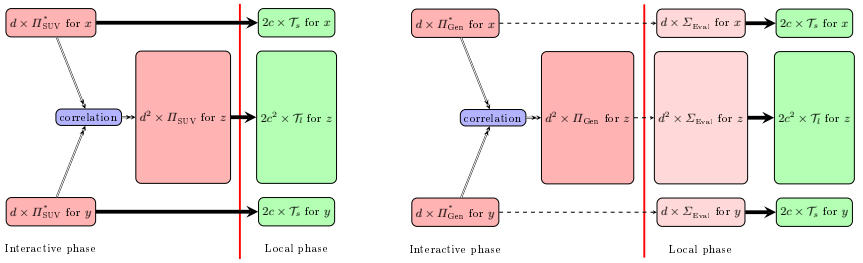
3 Description of the Bt-PCG

The purpose of the Bt-PCG [9] is to generate a large batch of N many uniformly random Beaver triples. As any PCG the Bt-PCG consists of an interactive seed generation phase (Section 3.1) proceeded by a local expansion phase (Section 3.2). In Section 3.3 we draw a whole picture, Fig. 1, distinguishing between a direct and indirect Bt-PCG formulation (Section 3.4). We conclude the description of the Bt-PCG with a discussion on parameters (Section 3.5).

3.1 Interactive Phase and Generation of SUVs

The core of the interactive phase is a novel protocol Π_{SUV} for the generation of SUVs [9]. We always consider Π_{SUV} in optimized form [29]. Given an integer position α, the task of Π_{SUV} is to generate a random SUV with position α and payload A without revealing information about α, A, m. For this, Π_{SUV} processes

[5] We do not explicitly consider the generation of random share $[a]$ (to input private values in the online phase), since this is a sub-task of generating Beaver triples.



(a) Direct PCG (b) Indirect PCG

Fig. 1: Visualization of the Bt-PCG. An execution starts with the Π^*_{SUV} (resp. Π^*_{Gen}) for x, y and ends with the output $[x] \cdot [y] = [z]$ of the LPN transformation $\mathcal{T}$. Double arrows represent the flow of secret-shared positions and payloads, thick arrows represent the flow of large vectors, and dashed arrows the flow of small DPF keys.

the position bit-wise secret shared with $[\cdot]_2$, whereas the payload A and MAC key m, are both secret-shared with $[\cdot]$. The choice of $[\cdot]_2$ and $\cdot[\cdot]$ is due to protocol details [9]. As key feature, the communication of Π_{SUV} only operates on the level of the position and payload, instead of the full SUV of large degree N. This is the core of the global PCG design for low communication.

To describe the indirect Bt-PCG, we also consider a two-step separation of Π_{SUV} over the interactive and local phase. As part of the interactive phase, this approach takes a key generation protocol Π_{Gen} which generates two private keys that are much smaller than the respective SUV. The keys are made such that the parties can expand them into their share of the SUV with a local (non-interactive) algorithm Σ_{Eval}. Hereby, Π_{Gen} is similar to Π_{SUV}, differing only in the output format, and Σ_{Eval} can be seen as a localization of Π_{SUV}, where all information received by interaction is encoded into the private keys. Note that the literature mainly focuses on Π_{SUV}, leaving the description of the two-step approach with $\Pi_{\mathrm{Gen}}, \Sigma_{\mathrm{Eval}}$ as straightforward modification [9,29].

As variations of Π_{SUV} and Π_{Gen} we additionally use protocols $\Pi^*_{\mathrm{SUV}}, \Pi^*_{\mathrm{Gen}}$ that do not expect a secret-shared position and payload as input, but rather include to sample fresh values in an efficient way [29].

3.2 Local Phase and Ring-LPN Transformation

The local phase relies on a special purpose coding-theoretic assumption, which we refer as ring-LPN assumption [9]. The assumption itself is out of the scope of its work, for Silentium the underlying polynomial transformation is sufficient. For this, let $R = \mathbb{F}[X]/F$ be the polynomial ring modulo F, where $F \in \mathbb{F}[X]$ has degree N and is fully reducible into different linear factors, i.e., there exists a ring isomorphism $\psi : R \cong \mathbb{F}^N$, where we identify polynomials in R with their N-dimensional coefficient vectors. For example, if F is the cyclotomic polynomial $F = X^N + 1$, than ψ is efficiently computable with the NTT [26]. Let furthermore $R_q \subset R$ be the set of q-sparse polynomials, i.e., N dimensional vectors with q

arbitrary non-zero positions and respective arbitrary payloads. Let $\langle x, y\rangle_c = \sum_{i=0}^{c-1} x_i y_i$ denote the c-dim scalar product over R.

For the Bt-PCG, consider vectors $u, v \leftarrow \mathcal{H}_q^c$ and respective MACs $u' = \mathrm{m} \cdot u, v' = \mathrm{m} \cdot v$. We write $\otimes$ for tensor products, e.g., $w = u \otimes v \in R^{c \times c}$ is given by $w_{i,j} = u_i \cdot v_j$. Than the ring-LPN assumption implies that the distribution

$$\begin{aligned} \{x, x', y, y', z, z' \mid \rho \leftarrow R^c, u \leftarrow R_q^c, v \leftarrow R_q^c, u' = \mathrm{m} \cdot u, v' = \mathrm{m} \cdot v, \\ w = u \otimes v, w' = \mathrm{m} \cdot w, x = \langle \rho, u\rangle_c, x' = \langle \rho, u'\rangle_c, y = \langle \rho, v\rangle_c, \\ y' = \langle \rho, v'\rangle_c z = \langle \rho \otimes \rho, \rangle_{c\times c}, z' = \langle \rho \otimes \rho, w'\rangle_{c\times c}\} \end{aligned} \tag{1}$$

over R^6 is computationally indistinguishable from sampling $[x], [y], [z]$, where $x, y \xleftarrow{\$} R, z = x \cdot y$. Hence, after applying ψ, Eq. (1) describes nothing else than to sample N independent Beaver triples over $\mathbb{F}$. We denote the respective map

$$\begin{aligned} u \mapsto \psi(\langle u, \rho\rangle), \quad v \mapsto \psi(\langle v, \rho\rangle), \quad w \mapsto \psi(\langle w, \rho \otimes \rho\rangle) \\ u' \mapsto \psi(\langle u', \rho\rangle), \quad v' \mapsto \psi(\langle v', \rho\rangle), \quad w' \mapsto \psi(\langle w', \rho \otimes \rho\rangle) \end{aligned} \tag{2}$$

as ring-LPN transformation $\mathcal{T}$. We write $\mathcal{T}_s, \mathcal{T}_l$ for the individual maps with small c- or large c^2-dimensional input vectors, respectively.

3.3 Visualization of the Bt-PCG

Given the previous two sections, we now can draw a complete picture of the Bt-PCG (Fig. 1). First observe that due to linearity, the parties can locally apply $\mathcal{T}_s, \mathcal{T}_l$ to transform a secret-shared version of $w = u \otimes v$ into a batch of N many pseudo-random Beaver triples $[x], [y], [z]$. This already induces the task of the interactive phase, namely to generate a respective triple $[u], [v], [w]$ of secret-shared sparse vectors (SSVs), that are correlated by a polynomial tensor product. These SSV can be generated with Π_{SUV}, since the SSVs can be decomposed into vectors of SUVs $\bar{u}, \bar{v}, \bar{w}$ with coefficients $u_i^k \cdot v_j^l = w_{i,j}^{k,l}, 0 \le i, j < c, 0 \le k, l < q$, where for the w component the degree of the vectors is increased to $2N$. We refer to the SUVs in $\bar{u}, \bar{v}$ as *small* SUVs and to the SUVs in $\bar{w}$ as *large* SUVs (resp. for SSVs).

Altogether, the direct Bt-PCG proceeds as follows (Fig. 1a). In the interactive phase, the computation starts with calls of Π_{SUV}^* to generate the small SSVs $[u], [v]$ with uniformly random payloads $[\alpha]_2, [\beta]_2$ and payloads $[A], [B]$. Afterwards, positions and payloads $[\gamma]_2, [C]$ of the SSVs $[w]$ are computed. This is done with d^2 secret-shared integer additions for $[\gamma]_2$ (with TinyOT AND gates [9,27]) and d^2 secret-shared multiplications for $[C]$ (consuming Beaver triples). Afterwards, Π_{SUV} is used to generate the d^2 many respective SSVs $[w]$. In a separate local phase, the parties can than call $\mathcal{T}_s, \mathcal{T}_l$ on their shares of u, v, w, giving them secret-shares of N Beaver triples $[x], [y], [z]$.

For the indirect approach, Fig. 1b, the difference is that $\Pi_{\mathrm{SUV}}, \Pi_{\mathrm{SUV}}^*$ calls are replaced by $\Pi_{\mathrm{Gen}}, \Pi_{\mathrm{Gen}}^*$ protocol calls. This replaces the output SSVs by a set of respective SUV keys, altogether referred as PCG seed. It is than part of the local phase to recompute the SSVs shares with Σ_{Eval} before running $\mathcal{T}$.

3.4 Indirect and Direct PCG

While the direct PCG is more compact with respect to computation in the local phase, the indirect Bt-PCG is more compact with respect to memory requirements between the interactive and local phase. Whether to use the direct or indirect Bt-PCG is a design choice of the real-world deployment. While the direct Bt-PCG contains less computational steps, the indirect Bt-PCG has the advantage of the seed compression. The latter is not only useful to reduce storage in large-scale applications, but also if the interactive phase is schedule independently of the MPC online phase, e.g. on cloud servers.

The initial presentations of the Bt-PCG [9,29] only consider the direct Bt-PCG, without actually naming it as PCG. This has technical reasons since the initial PCG definition [8] is hard to formalize for active security. More concretely, the mechanism for active security at Π_{SUV} introduces a technical leakage. While this leakage is not critical with respect to security, it makes a formalization in terms of the initial PCG definition very challenging. Still in practice, we do not see a point in not using an indirect, two-step approach, or why not to call the whole construction as PCG. What the indirect PCG actually does compared to the PCG only depends on the different output structure of Π_{SUV} and Π_{Gen}. Since the communicated messages are identical, there is nothing to discuss about security (following the simulation based privacy formalization from the UC framework as used for PCGs [9]). Furthermore, Σ_{Eval} has no effect on the security and privacy, since it is only local.

3.5 Bt-PCG Parameter Choices

To complete the description of the Bt-PCG we comment on the parameter choices for N, c, t. One point is that the Bt-PCG can be slightly improved by replacing the set R_q of sparse vectors, by a set $R_{b,t}$ of regular sparse vectors, where the polynomial is split into b many t sparse blocks. The concrete advantage of this regular variant is that the degree of the SUVs is reduced to $N_b = N/b$ ($2N/b$ respectively), which reduces the costs for the SUV generation in terms of communication, memory consumption, and runtime, while on the other side the security level of the ring-LPN assumption is only slightly decreased [9].

All available parameter recommendations [9] are for fixed $N = 2^{20}, \nu \approx 124, F = X^N + 1$ and different $\lambda = 80, 128, c = 2, 4, 8$ and $d = c \cdot b \cdot t = 32, 40, 64, 96, 156$, where b is a power of 2 in order to divide N_b (see Section 8.3 for concrete combinations). The general relation is that larger values for c allow to use smaller values for $b \cdot t$. Due to out versatile design, Silentium is adaptive to future results on the ring-LPN assumption.

4 Analysis of the Bt-PCG

The Bt-PCG employs a wide variety of tools and cryptographic primitives, which makes it hard to predict bottlenecks and to detect possible improvements in theory only. However, we got more insight during the process of the implementation,

Table 1: Different cost metrics for the direct Bt-PCG. The absolute numbers are per party and with respect to $N = 2^{20}, \nu = 128$, and different parameters (c, b, t).

Nr.	Metric	Asymptotic costs	$(8,1,4)$	$(8,4,1)$	$(2,1,76)$
1a	communication	$\mathcal{O}(d^2(n_b\lambda+\nu))$	0.8 MB	0.8 MB	23.1 MB
1b	exchanged messages	$\mathcal{O}(d^2 n_b)$	$123 \cdot 10^3$	$113 \cdot 10^3$	$2702 \cdot 10^3$
2a	PRG calls (interactive phase)	$\mathcal{O}(d^2 N_b)$	$4.5 \cdot 10^9$	10^9	$87 \cdot 10^9$
2b	$\mathbb{F}$ mults. (interactive phase)	$\mathcal{O}(d^2 N_b)$	$9 \cdot 10^9$	$2 \cdot 10^9$	$195 \cdot 10^9$
3a	opened values	$\mathcal{O}(d^2 n_b)$	$57 \cdot 10^3$	$55 \cdot 10^3$	$1275 \cdot 10^3$
3b	consumed correlated randomness	$\mathcal{O}(d^2(n_b\lambda+\nu))$	1.2 MB	1.2 MB	32.7 MB
4a	size of SSVs $\bar{u}, \bar{v}, \bar{w}$	$\mathcal{O}(d^2 N_b \nu)$	35 GB	9 GB	780 GB
4b	size of SUVs u, v, w	$\mathcal{O}(c^2 N_b \nu)$	1342 MB	1342 MB	134 MB
4c	size of PCG seed	$\mathcal{O}(d^2(n_b\lambda+\nu))$	0.3 MB	0.2 MB	8.7 MB
5	ring mults. ($\mathcal{O}(N \log N)$ $\mathbb{F}$ mults.)	$\mathcal{O}(c^2)$	144	144	12

from which we derive different cost metrics to formalize identified bottlenecks. One step further, for each metric, we derive countermeasures towards an efficient implementation.

Table 1 gives concrete cost values for all metrics. As basis, we consider the reasonable choice $N = 2^{20}$ on a bit field size of 128 (due to security [9]). To provide a general intuition: Each individual SUV has a size of more than 30MB per party, making the Bt-PCG very heavy.

Metric 1. Communication: By design, the Bt-PCG achieves low communication (Table 1, 1.a), which we do not further address since it is already highly optimized [9,29]. We only stress that the practical numbers have a relatively large range, which can be explained with the advantages of the regular variant and the trade-off between c and d (Section 3.5). However, we identify the number of communication rounds, Metric 1b, as a bottleneck. As an upper bound, we count the number of exchanged messages required (per party) when running through all protocols step by step (Table 1, 1.b). Being in the hundred-thousands, containing single bit messages, this number is prohibitively large.

Derived Requirement: Synchronization to pack several messages into larger batches that can be exchanged in one communication round.

Metric 2. Computation (Interactive Phase): The computational costs of the interactive phase are dominated by the SUV generation. Concretely, the costs of Π_{SUV} are dominated by $\mathcal{O}(N)$ calls to a PRG (Metric 2a) and multiplications over $\mathbb{F}$ (Metric 2b), which goes into billions of calls [29]. Hereby, the challenge is, that the operations are separated by interactive steps and distributed among different SUV instances.

Derived Requirements: Use single instruction many data (SIMD) hardware instructions for the PRG and $\mathbb{F}$. Synchronization of different computational steps to increase the parallelization rate.

Metric 3. MPC Operations: Most of the interactive parts are in terms of MPC operations (TinyOT, SPDZ), which can be quantified in the number of opened values, Metric 3a, and amount of consumed correlated randomness, Metric 3b. Both Metrics indicate that the MPC is expensive and requires a careful implementation. Even more, since Π_{SUV} mixes MPC operations with other computational expensive steps from Metric 2, generic MPC solutions can not be applied.

Derived Requirements: From scratch implementation of the MPC operations. The preprocessing stage for TinyOT and SPDZ is beyond our work [9,29].

Metric 4. Memory Consumption: By design, the indirect Bt-PCG compress the target correlation into small seeds, Metric 4c. However, during runtime the Bt-PCG contains a blow-up of pseudo-randomness in terms of the intermediate SUVs and SSVs.

Derived Requirements: Compact data representation enabling fast memory access. Time-scheduling strategy to balance high memory consumption.

Metric 5. Computation (Local Phase): The transformation $\mathcal{T}$ in the local phase employs ψ and ring multiplications. The challenge is that due to the security the degree is large, e.g, $N = 2^{20}$. Note that comparable applications, e.g., applications in lattices based cryptography go typically up to 2^{15} for even smaller fields, and are hence much cheaper to realize.

Derived Requirements: Careful selection of algorithms and their implementation. Focus on parallelization and fast memory access. GPU or FPGA implementation for better multi-threading (as future work).

4.1 General Challenges of an Implementation

On the global Bt-PCG level, we identify the following challenges with the metrics and their countermeasures:

- There is a trade-off between parallelization and large memory consumption. How can we find a suitable granularity of tasks?
- The Bt-PCG contains several heavy tasks that are on the one hand supposed to run in parallel and on the other hand are time-wise dependent. How can we realize respective time-scheduling of tasks?
- The Bt-PCG employs very different primitives that however rely on common functions, like MPC operations and field arithmetic. How can we connect the different blocks in Fig. 1 with little overhead?
- How do the recommendations for c, b, t, Section 3.5 affect the performance in practice?

Derived Methods: We provide several techniques to decompose the Bt-PCG into handful sizes with the goal to have a suitable granularity for parallelization and time-scheduling of tasks (Section 5). To control the interfaces of the modules,

we equip the modules with additional structure (Section 6). In our implementation modules share common functionalities and data structures with a focus on compact memory representation and synchronization (Section 7). All our constructions are theoretically independent of specific parameter choices.

5 Decomposition of the Bt-PCG

Silentium comprises a framework to decompose the Bt-PCG into implementation-friendly modules. Starting point of all our decomposition scenarios is the visualization in Fig. 1, which already contains a task-oriented decomposition but without taking care about the aspects of our analysis in Section 4. We provide further decomposition techniques in terms of modules as containers for several instances of one task. Hereby we distinguish between different module categories like Cor, SUV, Gen, Eval, LPN, similar as in Fig. 1. For example, a module of the category SUV has the task to generate a given number of SUV instances, while a module of the category LPN has the task to apply the transformation $\mathcal{T}_s, \mathcal{T}_l$ to a specified set of input vectors. In this sense, a decomposition scenario describes how to cluster the relevant instances (e.g. all $2d+d^2$ SUVs to be generated, all d correlation steps, all calls to ψ) into a set of modules. Hereby, setting the number and sizes of modules is a tool for balancing different tasks and costs, e.g., to perform a trade-off between low-level and high-level parallelization. Individual modules are supposed to run on different computation units (high level parallelization), while the low-level parallelization takes place inside the modules.

In Section 5.1, we differentiate between module categories for different tasks. Due to Metric 4, one aspect is, where to perform the aggregation of SUVs to SSVs (Section 5.2). In Section 5.3, we discuss decomposition steps for the block of large SUVs, which can be seen as a blueprint for other decomposition techniques.

5.1 The Module Categories

In the following we formalize the different module categories and discus respective decomposition steps.

The Correlation Cor: The task of Cor is to compute the positions γ and payloads C of the large SUV instances (Section 3.3), which in fact takes secure circuit evaluations with SPDZ (positions) and TinyOT (payloads) [9]. To separate cryptographic techniques we actually use two the module categories, namely Pos, correlating the positions, and Payl, correlating the payloads. For high-level parallelization both module categories can run on parallel cores, using different communication channels. For low-level parallelization, we propose to run all d additions/multiplications in parallel with synchronous communication to address Metric 1b (see Section 7 for details).

In a further step, the tensor sum and product might be distributed to several modules by splitting the d-dimensional tensor operations into k instances of smaller $\tilde{d}$-dimensional tensor operations, where $d = k \cdot \tilde{d}$. However, we stress

that for reasonable Bt-PCG parameters ($d < 100$) a further decomposition tends to be disadvantageous, since, if the actual computational steps become smaller, the time to setup the modules during runtime, e.g. communication channels, becomes more and more dominant.

Modules for the SUV Generation: For the generation of the vectors of SUVs $\bar{u}, \bar{v}, \bar{w}$, we distinguish between the module category SUV for the direct Bt-PCG and the module categories Gen, Eval for the indirect Bt-PCG. We treat SUV* and Gen* as corner cases of SUV, Gen (see Section 6 and [29]). In the following we focus on SUV and treat all categories equal to simplify the description. Still in practice one might distinguish especially between the interactive SUV, Gen and local category Eval. For example, the key generation runs faster since there are no latency issues, hence parallelization is less critical. The versatile module design allows such a separation (Section 6).

One module refers to several SUV instances, or their respective key encoding. In practice, we distribute the SUVs into at least three modules, separating the terms u, v, w as already done in Fig. 1. In Section 5.3, we further decompose the SUV module referring to the w component of size d^2, which is necessary to address the metrics 1 and 4. A similar decomposition scenario can be applied to the smaller components u, v of size d. However, we skip the details since for our current implementation we see no strong need. Depending on parameter choices, such a decomposition might have even disadvantages since the modules become too small, as for the correlation modules.

LPN Modules: For the LPN category, we distinguish between the three components x, y, z of the target Beaver triples, as already done in Fig. 1. Although the costs for the z component are quadratic in c, we do not apply any further decomposition. The point is that the LPN transformation is non-interactive; controlling the Metrics 4 and 5 is less critical than controlling for example latency issues in the interactive phase. Instead, for now, we focus on an efficient implementation of the large degree polynomial operations (Section 7.4).

MAC Checks: The MPC operations in the Pos, Payl, SUV, Gen categories require MAC checks for authentication against malicious parties (Section 2). While one option is to introduce an extra module for MAC checks, we take the opposite approach and include the MAC checks into the individual modules. The advantage is that this isolation avoids the overhead of coordinating the opened values, see Metric 3, during the Bt-PCG execution.

5.2 Aggregation of SUVs to SSVs

While the SUV protocol from Section 3.1 generates individual SUVs, the LPN transformation $\mathcal{T}$ operates on SSVs. At some point this requires to aggregate the vectors $\bar{u}, \bar{v}, \bar{w}$ of SUVs into vectors u, v, w of SSVs, which is a very critical

part of the Bt-PCG. While the memory consumption of the intermediate SUVs is very critical in pure numbers (Metric 4), the aggregation itself is intensive in terms of memory access, i.e., to read in the SUVs and to concatenated them following the polynomial tensor product modulo F.

An open question is where to perform the aggregation: As part of SUV, EVAL or as part of LPN? In case of the direct Bt-PCG: As part of the interactive- or local phase? Since the aggregation is a purely arithmetic tasks, it naturally belongs to the local phase; in the interactive phase, the aggregation might introduce latency. However, for memory efficiency, the aggregation should run as early as possible, i.e., in the interactive phase as part of the SUV generation. In fact, this would reduces the amount of data to be send and stored between different module executions (Metric 4a vs. Metric 4b), respectively the interactive and local phase.

Our design choices is to perform the aggregation in the SUV, EVAL modules, where decomposition of the large SUV instances into several modules respects the the aggregation step (Section 5.3). For each module, all SUV instance are generated at once, which addresses Metric 1b and 2. However, this implies that the memory consumption for Metric 4b reaches a maximum with respect to the number of SUV instances . In particular, large memory consumption is already an issue inside the SUV module and not only between different module categories. Hence, it is only consequent to perform the aggregation as final step of $\Pi_{\mathrm{SUV}}, \Sigma_{\mathrm{Eval}}$ without further delay.

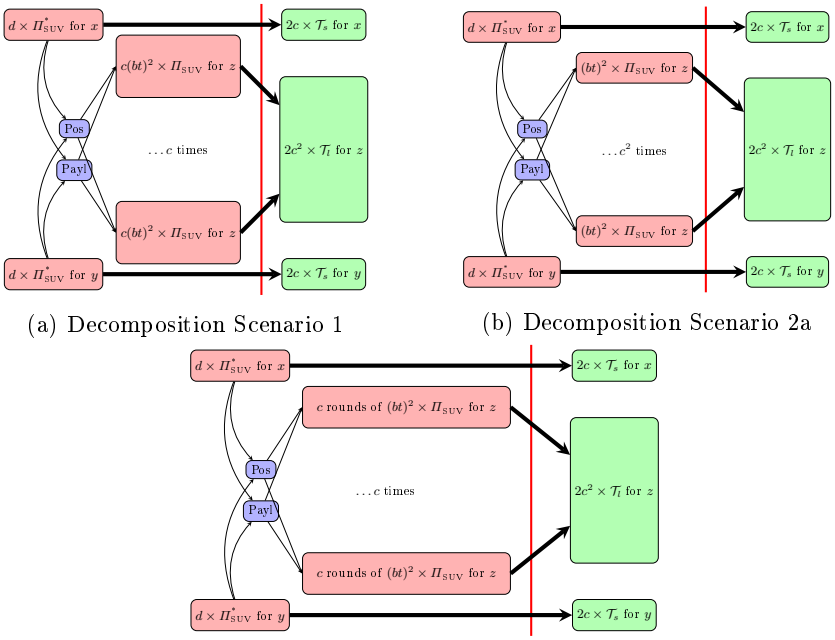
5.3 Decomposition of the Large SUV Component

The generation of the SUV for the z component as visualized in Fig. 1 demands for a further decomposition, e.g., since the Metrics 2 and 4 scale with d^2 and are prohibitively large for low-level parallelization. The opposite to Fig. 1 would be to decompose the Bt-PCG down to the level of its noise entries, i.e., referring to each of the SUVs in $\bar{w}$ as individual modules, which are then processed individually. As discussed for the small SUV instances, POS, PAYL, this goes however too far.

The optimal solution is somewhere in between, depending on the parameters c, b, t, N. In the following we describe such decompositions for SUV, the other categories GEN, EVAL are similar. Note that SUV includes the aggregation step, where the factor c, of the parameter product $d = c \cdot b \cdot t$, has a special role since it separates individual SSVs. As a natural consequence, we map all SUVs referring to one SSV into the same module.

As a first step, we propose a decomposition with c many modules of size $s = c(bt)^2$, see Fig. 2a. For the Bt-PCG parameter recommendations, it holds $128 \leq s \leq 11552$. Since each of the $2N$ dimensional SUVs takes 67 MB (for $b = 1$), even for $s = 128$ the memory consumption is still prohibitively large for low-level parallelization. Hence we propose to continue the decomposition and to use c^2 many modules of size $(bt)^2$ (Fig. 2b).

Separating the large SUV instances as in Figs. 2a and 2b results in reasonable modules sizes with respect to memory. However, the number of SUV modules is given by c in the first scenario Fig. 2a and by c^2 in the second scenario Fig. 2b,



(a) Decomposition Scenario 1

(b) Decomposition Scenario 2a

(c) Decomposition Scenario 2b

Fig. 2: Decomposition scenarios of the Bt-PCG. An execution starts with the Π^*_{SUV}executions and ends with the output the LPN transformation $\mathcal{T}$. Thin arrows represent the flow of positions or payloads, thick arrows represent vectors of SSVs.

where the recommendations are $c \in \{2, 4, 8\}$. While the first scenario with $c = 2$ employs two parallel computation units for the z component (in the sense of our high-level parallelization), the scenario with $c = 8$ would employ 64 parallel computation units. Even if these are available for the Bt-PCG deployment, setting up 64 modules and coordinating the parallel units, adds some overhead. To address this, we propose to take only c modules of size $s = (bt)^2$, which all run on c rounds (Fig. 2c). The concrete advantage is that compared to a separation to independent modules, the iterative calls share the same infrastructure. like communication channels and memory, which reduces the setup costs.

6 Structured Modules and Bt-PCG Programs

The decomposition scenarios, as visualized in Fig. 2 are static in the sense that they do not specify how to schedule the high-level parallelization or how to link the modules at runtime. In preparation of our implementation, the purpose of this section is to formulate a Bt-PCG execution in terms of our module concept. Our implementation is then a straightforward translation into a Bt-PCG script, calling small C programs, one for each module (Section 7). Hereby, the decom-

position scenario is determined by the script, while the Bt-PCG parameters are passed in at run time.

6.1 Bt-PCG Programs

We now sketch one exemplary program for the decomposition scenario Fig. 2c, formalizing the description from Section 3.3 in terms of our module concept. To keep the interactive phase as compact as possible we assume to have a strict time-wise separation of the interactive and local phase. In our example, the parties agreed on Fig. 2b, whereas the parameters N, c, b, t are generic.

- Before running the Bt-PCG, make sure to provide ρ and to have sufficiently many correlated randomness available for the internal SPDZ and TinyOT operations. For now, we only provide an insecure fake generation, where one party samples and distributes all correlated randomness. While this is sufficient for testing, a real offline phase for the Bt-PCG is future work.
- **Small SUV instances:** Run the two SUV modules for the x, y component. Since they are independent of each other, they are scheduled parallel, using two different communication channels. According to Section 5.3, we model the optimization by SUV* as a specification of SUV modules, which needs to be set by the Bt-PCG program. Each SUV* protocol gives to kinds of outputs. A set of c SSVs, which are large in size for a later use in the local phase; and d secret-shares of positions and payloads, which are subsequently processed in the correlation step.
- **Correlation Step:** Execute the Pos and Payl modules, in parallel. The output are blocks of secret-shared positions and payloads, referring to the blocks of the SUV modules for z. Note that hereby, we implicitly assume that the Pos and Payl modules take care about splitting the output data into respective blocks, which allows a better memory management by implementation specific implementations at an early stage. As a consequence, the Bt-PCG program needs to further specify the two correlation modules, i.e., they require all three parameters c, b, t (not only d) and the decomposition scenario of the z component.
- **Large SUV instances:** Run all large SUV modules, in parallel. Ideally, all modules are scheduled at once, such that the Bt-PCG requires no further coordination and such that there is less latency to wait for all modules to finish in the end. At this point, the transition form Fig. 2b to Fig. 2c becomes relevant: If the number of modules in Fig. 2b is too large for a given number of computation units, one can delegate the coordination of sequential SUV instance calls to the execution inside modules.
- **Local Phase:** Locally run the LPN for the x, y component in parallel, followed by one call for the z component. The point of separating the module for z is that our implementation of LPN internally distributes operations between different computations units (for each of the c^2 SSV), which would be in concurrence to run all three LPN modules at once.

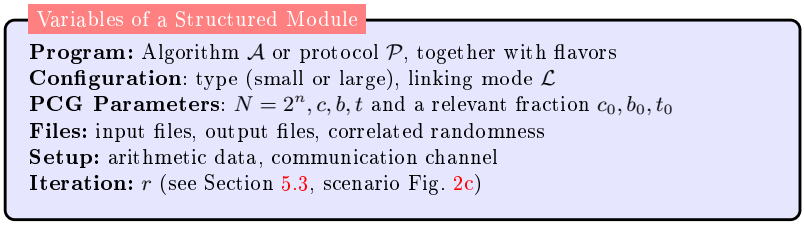


Fig. 3: List of module variables

We stress that it is part of the Bt-PCG program to split the correlated randomness into respective blocks following the decomposition. Hereby a security restriction is that each entry of the correlated randomness can be only used once. Furthermore, for efficiency reasons one might assign exactly as much correlated randomness as needed by a module execution, even if this depends on the PCG parameter choices.

6.2 Structured Modules

For an implementation, the module categories SUV, Gen, Eval, Pos, Payl, LPN are often not precise enough. For example SUV does not distinguish between Π_{SUV} and Π^*_{SUV} or other algorithmic branches, that might be selected depending on available resources. We now describe a respective configuration, either at compile time, i.e., one can select between different module programs for the Bt-PCG program, or at runtime, which can be controlled by program variables. For the latter we extend the modules from Section 5 to structured modules. The general template of a structured module is depicted in Fig. 3, where for practical reasons we distinguish between the following variable types:

Program Variables: The program variable specifies the algorithm (LPN, Eval), or protocol (SUV, Gen, Pos, Payl) that runs inside each module. For generality, one program can have different flavors. Static global flavors, e.g. the choice of the finite field or F, are better addressed at compile-time, while smaller adaptive flavors, e.g., to turn off/on a specific resource or further specify arithmetic choices, might be set at run-time.

Configuration Variables: The main tool to differentiate between different modules are the configuration variables. The type variable differentiates between small and large instances, e.g., between the components x, y and z, which for example comes with different degrees, N or $2N$, and either refers to d or d^2. Furthermore, the type specifies whether to run SUV or SUV* (Gen, Gen*), in fact only changes the input/output format. In general, the configuration variables simplify the deployment of the Bt-PCG, since there is no need to provide different programs that are almost identical up to a few sub-routines.

PCG Parameters: To set the context, the modules get the global PCG parameters $N = 2^n, c, b, t$ (Section 3.5). For example, the value n and b determine the SUV degree N_b to run the SUV generation. However, each module only process specific instances of a task (e.g. a set of SUVs or positions), and these need to be specified as well. For example in Fig. 2c, the large SUVs run on $c(bt)^2$. This can be specified by additionally passing in $b_0 = b, t_0 = t$, and, to indicate that the module execution refers to only one SSV, $c_0 = 1$. Although we do not cover such decompositions in Section 5, this can be technically be similarly used to split along the variables b, t. Note that the fractional parameters c_0, b_0, t_0 only describe the the scope of the internal instances. We model the proper linking to specific items implicitly through the input variables.

File Variables The file variable describe the input and output of each module, covering secret-shared positions and payloads, SUVs, DPF keys, the MAC, as well as files with correlated randomness or the LPN parameter ρ. Note that the PCG parameters and configuration variables only specify the task, but that they do not map to specific instances of SUVs, SSVs etc. Instead, the idea is that the file variable include this information, i.e. they only contain data for the relevant instances. It is hence the task of the Bt-PCG program to assign the file variables accordingly, using a proper indexing. Instead, the technical splitting (and merging) of the data into (from) blocks is part of the module executions, e.g., the correlation steps returns individual files referring to each SUV module execution. Still during the correlation, all d^2 positions/payloads are computed in parallel.

Setup Variables: With setup variables we refer to information that is relevant from the perspective of an implementation, e.g., setting up communication channels or providing arithmetic data. Apart from small values like an identifier of the parties and their communication ports, the prime or polynomial F, this might even refer to larger data like lookup tables for the NTT.

Iteration r: In Section 5.3 we propose to run the SUV modules on several iteration c in order to reduce setup costs and the number of communication channels. The iteration variable r sets the respective number of iterations. For simplicity we let all iterations refer to the same variables, with the exception that the file variables need to be extended to cover r rounds.

7 Implementation

In this section, we describe how we actually implement the Bt-PCG. For each module category we implement a C program; the Bt-PCG itself is a script that runs through a given decomposition. Follwing the recommendations in Section 4, we implement the arithmetic (Section 7.1) and the internal MPC operations (Section 7.2) from scratch. Furthermore, we synchronize the SUV generation,

Section 7.3. Finally in Section 7.4, we show how we optimize the transformation $\mathcal{T}$ with respect to the number of calls to ψ, which we instantiate as the NTT.

7.1 Arithmetic Library

To structure our code, we put the field and ring arithmetic into a separate library, providing different implementations that can be selected for the Bt-PCG at compile time. Concretely, we support a 128-bit prime field $\mathbb{F}$, as standard for SPDZ, and $N = 2^{20}$, as recommended for the Bt-PCG. For the choice of R see Section 7.4. Currently, we provide two implementations for the field arithmetic. Our initial approach is build on the GNU Multiple Precision Arithmetic Library (denoted as GMP version)[6]. Since this turns out to be inefficient (Section 8.1), the second implementation is from scratch, based on an 16-bit Montgomery representation for multiplication [21] and inversion [30]. Hereby, we run the whole Bt-PCG on the Montgomery representation, i.e., we avoid costly transformation between the Montgomery and standard field representation.

Advantages of our $\mathbb{F}$ implementation: The main reasons why we took the effort for an implementation of $\mathbb{F}$ is to better control and optimize the NTT and the transition form SUVs to SSVs. Compared to the GMP version, we can have a specific optimizations for a 128-bit field. Especially since the GMP library comes with a significant overhead with respect to memory allocation, which however is a bottleneck according to Metric 4, Table 1.

Data Structures: We use C data structures to implement the field arithmetic. This standardizes the template inside the module implementations, while the implementation of the field arithmetic can be easily exchanged during compile time. Concretely, we employ the following data structures:

- DATA to store relevant arithmetic parameters, e.g, the prime, and additionally data, e.g., the roots of F, to speed up certain operations.
- NUM for individual elements in $\mathbb{F}$, supporting standard operations like addition, multiplication and inversion.
- SUV to store SUVs, i.e. N_b dimensional vectors of field elements. The idea is to have a data structure for vectors with a focus on entry-wise field arithmetic as used inside Π_{SUV}, Π_{Gen}, and Σ_{Eval}. Putting the individual field elements into one data structure, reduces the overhead from wrapping around NUM data types and gives direct access to all coefficients at once. The latter introduces better options for parallelization of entry-wise operations.
- POLY for elements in R, i.e., N-dimensional polynomials, supporting an implementation of the ring-LPN transformation $\mathcal{T}$, e.g., ψ and scalar products, and supporting the aggregation of SUVs (SUV data type) to SSV (POLY data type), following the parameter indexing and output formating described in Section 6.2.

[6] https://gmplib.org/

Data Representation and Parallelization: All operations on NUM, SUV, POLY can be activated on arrays, typically with the length equal to the module size. By this design feature, the low-level parallelization is assigned into our arithmetic libary, that can be exchanged at compile time depending on the actual needs. For now, we use the generic OpenMP programming interface for shared-memory multi-processing[7]. Additionally to keep the memory overhead minimal, which is required by Metric 4, we represent each data structure, e.g. one SUV, POLY or one array of NUM, as unstructured arrays of C data types, e.g., uint16_t.

7.2 Special Purpose Implementation of MPC Operations

As a requirement for Metric 3 on MPC operations, we implement small MPC libraries for TinyOT and SPDZ. Similar to the arithmetic implementation, the MPC implementation can be exchanged at compile time, e.g., to test different approaches or for hardware specific implementations. Currently, we only provide one implementation for SPDZ and TinyOT respectively, with the following two features:

Synchronization with Arithmetic Implementation: All our MPC functions operate on arrays, of NUM for SPDZ, and of C char values for TinyOT, where typically the length is given by the module size s. In this sense, our SPDZ implementation inherits the low-level parallelization from our implementation of NUM. For TinyOT we apply similar techniques for better efficiency. In fact, this vectorization is a countermeasure against Metric 1b: Since the interactive parts are synchronized, the number of communication rounds becomes constant, independently of the modulus size s.

Special Purpose MPC Functions: In general, implementing the MPC operations from scratch allows us to customize MPC operations to the needs of the Bt-PCG that are beyond standard additions and multiplications. Our MPC library includes functions for

- the tensor product $[C] = [A] \otimes [B]$ inside PAYL, taking care about the indexing of c, b, t and separation of the output into blocks (Section 6.1)
- the tensor sum $\gamma = \alpha \boxplus \beta$ inside POS, taking care about the indexing of c, b, t and separation of the output into blocks (Section 6.1). Furthermore, the addition of bit-wise secret shared values is done with a minimal number of interactive AND operations [9,25].
- a mixed multiplication of authenticated bits (TinyOT) with private bit-strings, as required by $\Pi_{\mathrm{SUV}}, \Pi_{\mathrm{Gen}}$ [9].
- a relaxed version of secure multiplications over $[\cdot]$ to specifically optimize $\Pi_{\mathrm{SUV}}, \Pi_{\mathrm{Gen}}$ in terms of communication [29].

[7] https://www.openmp.org/

We stress that Silentium does not include an offline phase implementation for MPC operations, especially not for correlated OT and AND triples [9,19]. For our benchmarks, the correlated randomness is sampled locally by one of the parties and than distributed in advance.

7.3 Synchronized Implementation of the SUV Generation

To increase the low-level parallelization rates and to reduce the communication rounds (Metric 1b), we implement Π_{SUV}, and similarly $\Pi_{\mathrm{Gen}}, \Sigma_{\mathrm{Eval}}$ in such a way that one invocation actually generates several SUV instances in a synchronous way. Shortly speaking, we achieve this by vectorizing the arithmetic and MPC operations according to the previous sections. This address the field operations of Metric 2b. Even more effective for efficiency is that the synchronization enables better SIMD rates for the PRGs invocations (Metric 2a), which we describe in the following.

Intuition of the PRG in Π_{SUV}: To actually formulate our implementation techniques, we provide the following intuition of Π_{SUV}, which consists of two parts: A tree phase processing the secret-shared position and a field phase processing the secret-shared payload. The tree phase additionally contains a verification mechanism to detect actively corrupted parties. Synchronization of the field phase is fully covered by our synchronization of SPDZ and the field arithmetic. We now describe the role of the PRG calls in the tree phase. This phase runs through a GGM binary tree [18] where each party P_σ generates node labels $s_\sigma^{i,j} \in \{0,1\}^\lambda$, $0 \le i \le n, 0 \le j < 2^i$ that for each level i fulfill a tree invariant, depending on the position of the target SUV. Concretely, starting with private nodes $s_\sigma^{0,0}$, the parties run trough the tree levels by applying a length-doubling PRG : $s_\sigma^{i,j} \to (\bar{s}_\sigma^{i+1,2j}, \bar{s}_\sigma^{i+1,2j+1})$. However, the new values $\bar{s}_\sigma^{i+1,\cdot}$ do not fulfill the tree invariant. They actually need to be corrected to $s_\sigma^{i+1,\cdot}$, which the parties either achieve with interactive TinyOT operations ($\Pi_{\mathrm{SUV}}, \Pi_{\mathrm{Gen}}$) or locally from the DPF key (Σ_{Eval}).

Realization of the PRG: For the concrete implementation of the length-doubling PRG, we follow recommendations of previous work [11,15] and combine two independent calls of the AES block cipher, which for the tree phase is required in the ECB mode. As AES implementation we use the OpenSSL library, accessing Intels AES-NI SIMD hardware module[8]. We point out that due to the interactive correction steps, this parallelization is restricted to fixed tree levels, which means that the achieved parallelization rate for one SUV vary from 1 to $N = 2^n$, e.g. $n = 20$. However, when using SIMD operations, the time difference between calling one or approximately one million PRG almost vanishes.

[8] https://openssl-library.org/

Synchronization of the PRG: Since we already synchronize the interactive steps among s SUV instances, it is straightforward to combine the PRG calls of the s SUV instances. That means, for each tree level, we can increase the parallelization rate by a factor s, keeping the runtime time for the SIMD operation almost constant. On a technical site, we avoid unnecessary data shifting and formatting, and treat several SUV protocols as one and operate on large arrays. For example for the strings $\{0,1\}^\lambda$ we use one C char array of dimension $sN\lambda$, that can be directly passed into one SIMD PRG operation.

Parameter Restrictions: Finally, we stress that for the AES-NI module, the maximal throughput size is restricted to 2^{31} bytes, which implies a restriction of $n_b + 1 + \log(s) < 27$ within the modules for large SUVs. This restriction has practical consequences for our deployment of the Bt-PCG (Section 8), since it restricts the Bt-PCG parameter recommendations for deployments of Silentium.

7.4 Optimized Ring-LPN Transformation

The costs of the local phase, i.e., for the ring-LPN transformation $\mathcal{T}$ in Eq. (2), are dominated by multiplications in R and the isomorphism ψ (Metric 5). We actually avoid explicit polynomial multiplications in R and operate on ψ. Inspired by the NTT-based polynomial multiplication, we use the homomorphic property $f \cdot g = \psi^{-1}(\psi(f) * \psi(g))$, where $*$ refers to coefficient wise multiplication, to multiply in $f, g \in R$. Then, the observation for $\mathcal{T}$ is that the call of ψ^{-1} is redundant. Concretely, writing out the scalar product in Eq. (2) allows to shift in ψ, such that ψ^{-1} cancels out:

$$u \mapsto x = \sum_{i=1}^{c} \psi(u_i) * \psi(r_i) \quad w \mapsto z = \sum_{i,j=1}^{c} \psi(w_{i,j}) * \psi(r_i) * \psi(r_j), \tag{3}$$

and similar for $v \mapsto y$. Now, observer that on the one hand ψ is an isomorphism, i.e., it is bijective, and that the only assumption on $r \in R^c$ is to be uniformly random. Hence, we can avoid all respective calls of ψ by computing

$$u \mapsto x = \sum_{i=1}^{c} \psi(u_i) * \tilde{r}_i, \quad w \mapsto z = \sum_{i,j=1}^{c} \psi(w_{i,j}) * \tilde{r}_i * \tilde{r}_j, \tag{4}$$

and similar for $v \mapsto y$, where $\tilde{r} \leftarrow R^c$ is uniformly random. Taking into account that the parties operate on secret shared values that consist as well of the linear MAC component, this shows how to realize the local phase with only $4c + 2c^2$ calls to ψ (with each $\mathcal{O}(N \log N)$ field multiplications), as well as $4Nc + 3Nc^2$ field multiplications for $*$, where the product $\tilde{r}_i * \tilde{r}_j$ needs to be computed only once.

NTT Implementation: The implementation of $\mathcal{T}$ is covered by our library for POLY, where we run the operations $\psi(u_i) \star \tilde{r}_i$, respectively for the other

components, in parallel, using OpenMP. On top of that, the implementation of POLY internally makes use of multi-threading. In fact, due to the large degree, e.g., $N = 2^{20}$, an efficient implementation of ψ turns out the expensive and challenging.

Right now, we fix $N = 2^{20}$ and support the most straightforward choice $R = \mathbb{F}[X]/(X^{2^{20}} - 1)$ (where p divides $2N$). With other words, we implement ψ as the well established NTT. For the NTT implementation we implement the Gentleman-Sandé and Cooley-Tukey approach. Both are implemented in an inline-version with reduced multiplications [26]. For a better efficiency, we use a look-up table for roots of unity, concretely this is part of the DATA structure.

We stress that our NTT implementation is only a baseline for further work. A small step is to replace $F = X^{2^{20}} - 1$ by $X^{2^{20}} + 1$, as initially proposed [9]. Technically, this implies a transition from the NTT to the nega-cyclic NTT, taking three more entry-wise vector multiplications. Long term steps are algorithmic optimizations and hardware acceleration for the NTT, e.g., running the whole transformation $\mathcal{T}$ on a GPU for a better multi-threading of the large degree NTT butterfly multiplications (Section 9).

8 Evaluation

The goal of Silentium is to be competitive with previous offline phase implementations in MP-SPDZ. The following provides benchmarks for a comparision. For a comprehensive evaluation of our decomposition methods and the individual modules, we refer to Section 8. For all our benchmarks we use two servers in a LAN setting, with 24 cores, 2.80GHz, and 256GB RAM.

Furthermore, we provide an Silentium internal evaluation of our SUV,GEN and EVAL modules (Section 8.1) to demonstrate the effect of our theoretical framework. Since we are the first who implement Π_{SUV} and the Bt-PCG, we focus on running time numbers, rather than other metrics like communication which are already discussed in previous work [9,29].

8.1 Evaluation of the SUV Generation

We focus our evaluation of the SUV generation modules on four different aspects. The first is the difference with respect to the module categories, the second is the synchronization, Section 7.3, the third are different parameter choices, and the last is about different configurations, e.g. to compare our field arithmetic implementation (as selected throughout our benchmarks) with the GMP version (Section 7.1).

Differences between SUV, GEN, EVAL: The relative values in Table 2 relate the running time between the different module categories. For the generation of one SUV, the modules SUV, GEN only differ in the output structure (SUV vs. DPF key). Since this implies that GEN skips aggregation into SSV, the GEN module runs much faster than the SUV module, especially for large

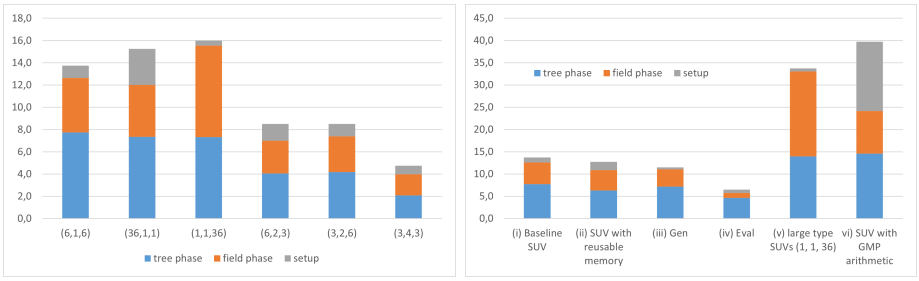
Table 2: Running times of SUV, Gen, Eval modules in seconds for different sizes $d = 2^k - 1$, where $c = 1, t = d$ to include the SSV aggregation. For practical reasons we set the size $s = 2^k - 1$ since according to Section 7.3 the maximum we can run is $s = 127 = 2^7 - 1$. The factor values give the ratio of running times between s many sequential calls (with size 1) and one synchronized module call. The category relation (cat.rel.) values are the ration between the times for Gen (resp. Eval) to the times of SUV. To set the context for the indirect Bt-PCG, Gen+Eval sums up the running times of Gen and Eval.

	SUV		Gen			Eval			Gen+Eval	
size s	time	factor	time	factor	cat. rel.	time	factor	cat. rel.	time	cat. rel.
1	1.3	1	1.3	1	98 %	0.3	1	25%	1.6	123%
3	3.4	1.2	2.8	1.4	84%	0.8	1.2	24%	3.7	108%
7	6.0	1.5	4.9	1.9	81%	1.9	1.2	31%	6.7	111%
15	9.5	3.1	8.1	2.4	86%	3.7	1.3	39%	11.8	124%
31	13.5	2.1	11.1	3.7	82%	6.8	1.5	50%	17.9	132%
63	21.5	3.9	16.5	5.0	77%	13.4	1.6	62%	29.9	139%
127	43.1	3.9	28.4	5.9	66%	27.0	1.5	63%	55.4	129%

s (up to 66%). The Eval module runs much faster than SUV, Gen, mainly since it is non-interactive. However, with growing size s, the running times of Eval gets closer to Gen, which indicates that other factors like memory and computation (PRG, SSV aggregation in Eval) become more dominant. Finally, Table 2 shows that the indirect SUV generation with Gen and Eval takes 10% to 40% longer compared to running the SUV module directly.

Synchronization: Table 2 shows that synchronization is effective, where the effect refers to the synchronization of the PRG, arithmetic operations and MPC in total (Section 7). For the SUV module, synchronization with $s = 3$ reduces the running time about a factor 1.2, while for larger s the reduction grows to a factor up to 3.9, worth of two minutes for $s = 127$. The reason why the synchronization is more effective for larger module sizes is most likely due to increased (PRG) parallelization rates (Section 7.3) and decreasing latency due to less communication rounds, (Metric 1b). The latter explains why the advantage of the synchronization is less significant for the non-interactive Eval category. Furthermore, the synchronization for Gen is even more effective, which indicates that the synchronization of the PRG is more effective than the synchronization of the SSV aggregation (skipped by Gen).

Effect of Bt-PCG Parameters: In Fig. 4a we compare the running times of the SUV module for different combinations of (c, b, t). Hereby, we differentiate between its tree- and field phase (Section 7.3), and the time to setup the module. The latter includes to setup the communication channel, memory allocation, and MAC checks. Fig. 4a shows that the costs are more or less equally distributed between the tree and field phase. While the tree phase is independent of the parameter choices (for fixed depth n_b) the field phase depends on



(a) SUV module (small SUV) for different (c, b, t).

(b) Different modules and configurations, with $(c, b, t) = (6, 1, 6)$, resp. $(1, 1, 36)$.

Fig. 4: Running time of SUV generation for different parameter choices, module categories, and configurations. The values $N = 2^{20}$ and $s = 36$ are fixed.

the costs for the aggregation, which takes more field operations for larger t, b. Furthermore, the setup step contributes to the runtime significantly, driven by the large memory consumption, Metric 4. For example, for larger c the setup time increases, mainly due to the increased memory allocation for the POLY output objects. Note, that for choices of smaller c, this allocation is hidden inside the field phase, where respective SUV are aggregated before the aggregation to POLY objects, representing SSVs. Finally Fig. 4a confirms a positive effect of the regular variant with $b = 2, 4$ (up to ten seconds): The running time of the tree phase roughly halves when the SUV degree N_b halves as expected due to the linear number of PRG calls. Furthermore the smaller SUV degree effectively reduces the running-time of the field phase by a even larger factor, depending on parameter choices, mainly due to the reduction of memory consumption.

Module Configurations: In Fig. 4b we compare the runtime for selected variations of the SUV generation, covering:

(i) As baseline we take SUV as in Fig. 4a with $(c, b, t) = (6, 1, 6)$.
(ii) A version of SUV, selected at compile time, where most of the memory allocation inside Π_{SUV}, e.g. for AES ciphertexts and SUV data types, is offloaded to the setup stage. This version increases the setup costs but reduces the general module costs since it is more latency-friendly. Note that we introduced this version to support module iterations ($r > 1$, Section 6.2), since it allows to reuse allocated memory over several iterations.
(iii) The GEN, EVAL modules to visualize how their running times splits between the tree and field phase. Hereby, it is remarkable that the tree phase of EVAL is quite fast, although it includes the SSVs aggregation step. This indicates that for the interactive modules SUV, GEN, the interactive parts (verification) of the tree phase are relevant as well, compared to the non-interactive SSV aggregation.
(v) A call of the SUV referring to large SUV instances (specified as confuguration variable). To have a comparable size to the other scenarios, we use

$(c^2, b^2, t^2) = (1, 1, 36)$. The cost are increased (more than a factor two, mainly due to the larger degree of $2N$ (which is more memory intensive), and the more complex aggregation (the setup is negligible since we set $c^2 = 1$).

(vi) A version of SUV with the GMP library for the arithmetic, Section 7.1. The costs are much higher compared to the baseline (i) using our own field arithmetic. The reason is the dynamic memory management of the GMP library, which not only implies a larger setup stage (memory allocation), but also slows down the tree- and field phase by seconds.

8.2 Evaluation of the Bt-PCG Decompositions

In Table 4 we provide running times for the direct and indirect Bt-PCG, with explicit numbers for the categories SUV, Gen, Eval, and LPN. Hereby we give the total times for each block of modules that are executed in parallel according to Section 6.1. Note that we do not explicitly list the modules Pos and Payl, since their running time (about one second) is not significant due to our synchronization (Section 7.2).

The numbers in Table 2 are with respect to all parameter recommendations (Section 3.5) that satisfy the condition on the maximum module size from Section 7.3. We checked the condition for all decomposition scenarios depicted (Figs. 1 and 2). It turns out that all possible choices, except one are with respect to Scenario 2, an none of the parameter works without decomposition (Fig. 1). This confirms the need to decompose the Bt-PCG. Concretely, we choose Scenario 2b, instead of Scenario 2a, clustering several iterative calls within our module programs. Table 4 allows the following interpretations:

- **Costs of SUV generation:** The costs of the Bt-PCG are dominated by the generation of the d^2 many large SUVs. Compared to that, the small SUV instances are cheap, so we list them only for the SUV category.
- **Scenario 1** turns out to be the best with respect to the SUV generation. This can be explained with a better high-level parallelization, instead of iterative calls inside the modules. Unfortunately, Scenario 1 is limited to one parameter set.
- **The regular variant**, i.e. $b > 1$ gives a clear advantage for the SUV generation, see for example the last two rows in Table 4. A real evaluation of the regular variant would require more secure parameter sets.
- **The LPN module** executions have costs of $\mathcal{O}(c^2 N \log N)$ field multiplications (Section 7.4). Indeed, running time for LPN only depend on c, with a clear advantage for $c = 4$ over $c = 8$. Note that the parameter recommendations [9] even cover $c = 2$. However this requires larger values of d, which is out of the scope with respect to our restriction on the module sizes.
- **Module Categories:** The time-wise relation between the modules SUV, Gen, Eval reflects the results from Section 8.1, however in detail different effects are hard to quantify. Altogether, the indirect approach is up to half a minute slower compared to the direct approach.

– **The local phase** takes 17% to 76% of the complete running time. This range can be explained by different choices on c and the advantage of Scenario 1.

Throughput of Triples and Communication: PCGs are designed to have low communication, thus a natural question is the relation between computation, in terms of running time, and communication. In theory, the ring-LPN security implies that the price for speeding up the local phase with a smaller c, is to increase d and hence the communication [9]. Table 4 confirms that relation. In general, the relation between communication and computation is hard to quantify with the restricted amount of our benchmarks. Relevant factors are the security parameter λ, the regular variant and the decomposition scenario.

8.3 Comparison to MP-SPDZ

As already known from theory, the Bt-PCG reduces the communication costs by three orders of magnitude, which for 2^{20} triples means a reduction from a few GB to a few MB, see Tables 3 and 4. To compare the running times, we run several Beaver triple generation protocols in MP-SPDZ [13,22] on the same local servers as our Bt-PCG benchmarks (Table 3). It turns out that our work outperforms the protocols in MP-SPDZ. While the benchmarks for MP-SPDZ range between 400 and 7400 triples per second, our Bt-PCG implementation achieves a throughput between 2900 and 11200 triples per second (Table 4), for the same security level $\lambda = 128$. LowGear is the only protocol that runs faster than some of our Bt-PCG programs. For the limitations of our comparison, we refer to Section 1.4.

Table 4 provides further data for different combination of parameters and decomposition, which we derive from the parameters of the theoretical Bt-PCG evaluation[9] and concrete memory limitations of our implementation (Section 7). This might give an intuition how performances is determined by the small and large SUV instances, or how performance differs between the module categories SUV, Gen, Eval. An important takeaway from Table 4 is that the time-relation between the interactive phase and local phase highly depends on the parameters and the direct or indirect Bt-PCG approach. All these freedoms need to be considered when it comes to an deployment.

9 Conclusion and Open Topics

Our work confirms that the Bt-PCG [9] is competitive with previous offline phases in MP-SPDZ, which is especially attractive for real-world applications in the cloud context. Towards such applications, we conclude with a list of further optimizations and issues that Silentium leaves open:

– Our benchmarks are on two local servers. How is the performance on two distant (cloud) severs with higher latency? While the Bt-PCG benefits from low communication, delays induced by high memory consumption might become more significant.

Table 3: Throughput in triples per second, and communication in GB per triple 2^{20} triples for different protocols in MP-SPDZ [22] (named as in [24]). All protocols are with respect to a 128 bit field and 128 bit computational security. (S)HE refers to (somewhat) homomorphic encryption, and OT to oblivious transfer extension.

Protocol	MP-SPDZ Program	Primitives	tr/s	GB
SPDZ 1 [14]	simple-offline.x	SHE	2800	4
High Gear [24]	simple-offline.x -g	semi-HE	2700	4
Low Gear [24]	pairwise-offline.x	semi-HE	7400	2
SPDZ 2 [13]	cnc-offline.x	SHE	300	59
MASCOT [23]	ot-offline.x	OT	1900	47

Table 4: Running times for the generation of $N = 2^{20}$ Beaver triples for different parameters with the direct and indirect Bt-PCG, including explicit numbers for the parallel execution of the SUV, Gen, Eval modules with respect to small and large SUV instances. The local percentage is the percentage the local phase takes of the full running time. The last section gives the performance in terms of triples per second (tr/s,rounded) and communication per 2^{20} triples in MB for the direct Bt-PCG. We do not explicitly list the time to run the Cor modules since it takes only about 1 second for all scenarios.

						SUV		Gen	Eval	LPN		direct		indirect		direct	
λ	d	c	b	t	Fig. 2	small	large	large	large	small	large	total	local	total	local	tr/s	MB
80	32	8	4	1	2b	15	190	159	69	11	78	293	30%	333	49%	3600	0.8
128	40	8	1	5	2b	17	257	188	105	11	78	362	24%	402	50%	2900	1.2
80	40	4	2	5	2b	10	180	111	106	6	20	217	12%	255	53%	4800	1.2
80	32	8	2	2	2b	8	134	115	36	11	78	230	38%	250	51%	4600	0.7
80	32	8	4	1	1	7	32	26	13	11	78	127	69%	135	76%	8300	0.7
80	32	8	4	1	2b	5	102	96	19	11	78	194	45%	208	52%	5400	0.7
128	64	4	8	2	2b	6	112	65	69	6	20	144	17%	165	58%	7300	2.8
128	64	4	16	1	2b	4	63	39	35	6	20	94	28%	106	59%	11200	2.7

- In Section 5, we focus on those decomposition scenarios that we actually implement, leaving details about further decomposition techniques as future work. Is it possible to differentiate the decomposition scenarios for different hardware environments and network settings? How can one find the best decomposition scenario in a given context?
- Future work might continue the initial security analysis of the new ring-LPN assumption [9], especially with the goal to provide further parameter sets for the Bt-PCG.
- Silentium does not cover the preprocessing for the internal MPC (TinyOT, SPDZ), which needs to be addressed for any real-world deployment.
- We see high potential in hardware acceleration for arithmetic operations ($\mathcal{T}$, the NTT, PRG) on GPUs or FPGAs for better multi-threading. Hereby, a challenge might be the high data transfer between different devices, especially with respect to latency in the interactive phase.

– Can the techniques of Silentium be applied to other PCGs? Can the modular approach of Silentium be used to compose other types of correlated randomness efficiently (e.g. circuit-depending tuples or multi-party triples).

References

1. Damiano Abram and Peter Scholl. Low-communication multiparty triple generation for spdz from ring-lpn. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *Public-Key Cryptography – PKC 2022*, Cham, 2022. Springer International Publishing.
2. Nabil Alkeilani Alkadri, Johannes Buchmann, Rachid El Bansarkhani, and Juliane Krämer. A framework to select parameters for lattice-based cryptography. Cryptology ePrint Archive, Paper 2017/615, 2017.
3. Rikke Bendlin, Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*. Springer Berlin Heidelberg, 2011.
4. Maxime Bombar, Geoffroy Couteau, Alain Couvreur, and Clément Ducros. Correlated pseudorandomness from the hardness of quasi-abelian decoding. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, Cham, 2023. Springer Nature Switzerland.
5. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector ole. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2018. Association for Computing Machinery.
6. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, Cham, 2022. Springer Nature Switzerland.
7. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2019. Association for Computing Machinery.
8. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, Cham, 2019. Springer International Publishing.
9. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *Advances in Cryptology – CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II*, Berlin, Heidelberg, 2020. Springer-Verlag.
10. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*. Springer Berlin Heidelberg, 2015.
11. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2016. Association for Computing Machinery.

12. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd$\mathbb{Z}_{2^k}$: Efficient mpc mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, Cham, 2018. Springer International Publishing.
13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS*, volume 8134 of *LNCS*. Springer, 2013.
14. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*. Springer Berlin Heidelberg, 2012.
15. Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2017. Association for Computing Machinery.
16. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to mpc with preprocessing using ot. In *Proceedings, Part I, of the 21st International Conference on Advances in Cryptology – ASIACRYPT 2015 - Volume 9452*, Berlin, Heidelberg, 2015. Springer-Verlag.
17. Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*. Springer Berlin Heidelberg, 2014.
18. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 2019.
19. Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round mpc combining bmr and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, Cham, 2017. Springer International Publishing.
20. Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar, and Krzysztof Pietrzak. Lapin: An efficient authentication protocol based on ring-lpn. In Anne Canteaut, editor, *Fast Software Encryption*. Springer Berlin Heidelberg, 2012.
21. Dan Page Joppe W. Bos, Thorsten Kleinjung. *Efficient Modular Arithmetic*, page 223–250. London Mathematical Society Lecture Note Series. Cambridge University Press, 2021.
22. Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2020. Association for Computing Machinery.
23. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 830–842, New York, NY, USA, 2016. Association for Computing Machinery.
24. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making spdz great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, Cham, 2018. Springer International Publishing.
25. Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *Cryptology and Network Security*, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
26. Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security*, Cham, 2016. Springer International Publishing.

27. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*. Springer Berlin Heidelberg, 2012.
28. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 2009.
29. Vincent Rieder. Generation of authenticated secret-shared scaled unit vectors for beaver triples. In Maria Eichlseder and Sébastien Gambs, editors, *Selected Areas in Cryptography – SAC 2024*, pages 54–83, Cham, 2025. Springer Nature Switzerland.
30. Erkay Savas and Cetin Koc. Montgomery inversion. *Journal of Cryptographic Engineering*, 8, 09 2018.
31. Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-ole: Improved constructions and implementation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1055–1072, New York, NY, USA, 2019. Association for Computing Machinery.
32. Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated ot with small communication. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2020. Association for Computing Machinery.
33. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986.