Performance Comparison of Two Generic MPC-frameworks with Symmetric Ciphers

Thomas Lorünser[a] and Florian Wohner[b]
AIT Austrian Institute of Technology, Vienna, Austria

Keywords: Multiparty Computation, Applied Cryptography, Privacy, Benchmarking.

Abstract: Research on multiparty computation (MPC) made substantial progress over recent years. It can be used to protect the privacy of data and users in modern application scenarios like Blockchain and the Internet of Things where different stakeholders want to collaborate. In this work we analyze practical aspects of two generic MPC frameworks, MP-SPDZ and MPyC, to generate new insights into the state-of-the art for generic and platform independent MPC. We implemented various symmetric ciphers and did extensive benchmarking on both frameworks to see how universal and generic they are and if they can be used without special knowledge. We found that the achieved performance cannot be trivially estimated from the algorithms without implementing. The stream cipher Trivium was by far the fastest and most portable in our tests. Contrary to most of existing literature we also addressed non optimal network settings and found surprising results. The asynchronous architecture of MPyC turned out to make more efficient use of the network layer in scenarios with higher network latency and loss and could even compensate for the optimizing compiler used by MP-SPDZ.

1 INTRODUCTION

Multiparty computation (MPC) is a technology for computing on confidential data in a distributed setting, i.e., with multiple nodes holding only fragments of input data. It can be used to decentralize systems where typically a central trusted authority is needed to execute certain functionality. With MPC the functionality is evaluated jointly between multiple parties such that the correctness of the output is guaranteed and the privacy of the inputs of the individual parties is preserved, and only the output of the computation is learned. The concept has been invented more than 30 years ago, but for a long time it was considered only of theoretical interest. However, progress in recent years has led to many interesting applications which can be realized with practical efficiency.

1.1 Motivation

Many of the promising results published in research papers have been obtained under ideal settings typically only considering a very limited scope. Additionally, many of the open source frameworks used to generate the results are not very mature. Experimenting with these frameworks, we found it challenging to correctly and efficiently implement certain algorithms for specific MPC systems and manual optimization was often necessary.

To better understand the potential of MPC in real life application scenarios we analyze the problem of interfacing with MPC by means of reading data from encrypted storage (Happe et al., 2017). To the best of our knowledge, there is so far no empirical study which tries to identify important parameters in the usage of MPC technology in a systematic and platform independent way. We tried to compare similar MPC settings and protocols and therefore selected two of the most actively developed frameworks which run the same protocols under the hood and support the same adversary model. For our study we are working in the honest-majority setting with information theoretical security and only semi-honest adversaries. This is the most basic setting which has all major ingredients to build more complex scenarios. Computationally secure protocols for dishonest majorities are out of scope for this work. It should also be noted that we do not intend to compare ciphers of equal security strength. We are aware that stream ciphers like Trivium do not provide the same security level as AES, however, our goal is to compare how the two

[a] https://orcid.org/0000-0002-1829-4882
[b] https://orcid.org/0000-0002-8641-7522

Lorünser, T. and Wohner, F.
Performance Comparison of Two Generic MPC-frameworks with Symmetric Ciphers.
DOI: 10.5220/0009831705870594
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 587-594
ISBN: 978-989-758-446-6

selected MPC frameworks can deal with the different structures of the various algorithms and what has to be considered in the porting of algorithms to MPC systems.

1.2 Contribution and Structure

We implemented, analyzed, and benchmarked four different ciphers in two MPC frameworks. We found that estimating the performance of algorithms for MPC systems in advance is hard and real performance deviated from the expected in many cases. Our work also shows a performance comparison for networks with higher latency or loss, where we identified some unexpected behavior which suggests that there is room for improvement. In our study Trivium turned out to be most versatile cipher, i.e., it supports the widest range of MPC frameworks, but it has also the lowest security level. We also provide our implementations as open source modules[1].

The remainder of the paper is structured as follows. The related work as well as selected MPC framework are presented in section 1 below. In section 2 the algorithms chosen for implementation for both frameworks are presented, as well as our findings during software implementation and the manual optimizations which were necessary to achieve good performance. The performance results achieved are presented and discussed in section 3. The conclusion of the paper is then presented in section 4.

1.3 Related Work

The most recent and comprehensive work on comparing MPC frameworks was presented in (Hastings et al., 2019) which tried to give an overview on the state of the art from a users point of view and does not focus on a particular technique or scheme. In particular, no performance benchmark was possible because of the very different approaches considered. Additionally, only basic operations in the spirit of "hello world" applications were analyzed, as was shown by the code snippets in the appendix. In our work, we focus on benchmarking more complex algorithms, symmetric ciphers in our case, and MPC with similar protocols and adversary models but different programming paradigms.

Besides comparing MPC frameworks, this work was also motivated by the progress made in developing optimized ciphers for application in multiparty computation (MPC), fully homomorphic encryption (FHE) and zero-knowledge proof systems (SNARKs) (Albrecht et al., 2015), (Albrecht et al., 2016a), (Rechberger et al., 2018), (Grassi et al., 2019) and (Albrecht et al., 2019), which are typically only benchmarked in ideal settings (almost perfect connectivity and fast servers). This somehow contradicts the idea of generic platform-independent secure computation, which should be the goal for widespread use of the technology. In our work we are using these new cipher designs and compare them with AES and well-known stream ciphers also considered interesting candidates for MPC. It is not the goal of our work to absolutely compare the many proposed ciphers, but to see how good their structure is suited in general for framework-independent application and this is only a first step towards broader benchmarking activities.

1.4 MPC Frameworks

Various frameworks have been developed over the last years, specifically in the open source domain. We selected two candidate frameworks, MP-SPDZ and MPyC, which both have the capability to work in the semi-honest setting with honest majority and use secret sharing based protocols. However, because they use a fundamentally different programming model it is interesting to see how they behave for the given task.

1.4.1 MPyC

MPyC (Schoenmakers, 2018) is a fork of the discontinued VIFF framework (Damgård et al., 2009), but only rarely used for MPC benchmarks. In our opinion, it follows an interesting concept which could lead to both easy access for programmers and reasonable performance. It is a Python framework that implicitly represents multi-party computations as graphs of regular Python values, secret-shared values, and operations on them. By overloading Python's regular operators, the whole process is made mostly invisible to the user of the framework. The resulting graph is built at runtime and evaluated asynchronously, so no static analysis, and therefore no optimization, is performed. However, heavily optimized primitives (mostly vector and matrix operations) are available. The framework is passively secure in an honest-majority setting.

Code written in the framework can be hand-optimized with the help of the built-in `gather()` and `_reshare()` methods. Applying the first method on a shared value will run all outstanding asynchronous computations and then unpack the share to return an element of the underlying field. Any further computations on it will then be performed locally only. Applying the second method to this value will return a

[1]Source code on GitHub: https://github.com/Archistar/archistar4mpc-cipher, first published 29.4.2020.

shared value again. Everything that happens in between is therefore part of a single round of communication. Obviously, this can destroy correctness, and it is left to the programmer to ensure that it does not.

Internally, MPyC makes ample use of this facility to improve the performance of its built-in methods and operators, and also to provide efficient vectorized versions of some common operations. We looked at four of those operations and compare them to their unoptimized counterparts in Table 1.

1.4.2 MP-SPDZ

MP-SPDZ (Keller, 2019) is a fork of SPDZ2 which was originally developed at the University of Bristol and is based on the SPDZ type of protocols (Damgård et al., 2012b). Since forking, MP-SPDZ has integrated more and more protocols and is now the most prominent framework used for benchmarking MPC protocols in general. It supports very flexible use of different protocols and also separation of online and off-line phases for performance measurements.

Its approach is to let users write their programs in a Python-like language that is then heavily optimized and compiled to byte-code for a fast virtual machine implemented in C++. The framework implements a wide variety of protocols for several different security models, based on arithmetic as well as Boolean circuits. Both integer and fixed-point numbers are supported and security models of honest majority as well as dishonest majority are supported, even for both semi-honest and malicious adversaries. In this work we were only using the `shamir-party.x` program as this resembles the same protocol used by MPyC.

2 ALGORITHMS

In the following section we will briefly review the basic principles and properties of the algorithms we selected for implementation, and discuss our findings. The algorithms have been selected because they are considered to be lightweight or because they were specifically proposed as ciphers optimized for application in MPC settings based on secret-sharing with a low circuit depth for multiplication gates. In general, the algorithms should help to understand how the two approaches of MP-SPDZ and MPyC are able to optimize the processing of the various ciphers and how suitable the structures of the ciphers are for the respective frameworks.

2.1 AES

The advanced encryption standard is the de-facto benchmark when it comes to MPC for evaluation of symmetric ciphers since (Pinkas et al., 2009). It is particularly challenging for systems operating on arithmetic circuits, because AES does not lend itself well to secure computation over prime fields (Damgård et al., 2012a). In spite of this, a lot of progress was made in the evaluation of AES in secret-sharing based systems with (Araki et al., 2016) claiming the best performance. They report on a cluster of three 20-core servers with a 10Gbps connection, which carries out over 1.3 million AES computations per second, processing over 7 billion gates per second. However, these results are achieved with a dedicated protocol for Boolean circuits supporting only three parties, and a lot of parallelization on the block level, which can always be achieved. AES implementations were available in both frameworks and we took them as baselines for our benchmarks.

2.2 ChaCha20

ChaCha20 (Bernstein, 2008) is a stream cipher and the successor to Salsa20, and is one of several novel ciphers recommended for new implementations by the eSTREAM project (De Cannière and Preneel, 2008). It is a typical ARX-cipher, consisting only of unsigned 32-bit integer additions, fixed-width bit rotations, and XORs. In the MPC setting, this mixing of integer and logical operations is a problem and suggests two different implementation strategies: one would be to represent the data as integers and convert to and from a bit-level representation as needed. The other, likely more efficient, strategy would be to use a bit-level representation throughout, and implement the integer addition as a Boolean circuit. In this approach, the multiplicative complexity of the cipher depends on the type of addition circuit used. The possible complexity for adders ranges from linear (ripple-carry) to inverse quadratic (carry-select) to logarithmic (carry-lookahead). For our implementation, we tried only ripple-carry and carry-select adders.

In the first implementation variant, we use the equivalent of unsigned 32-bit integers for the additions, switch to a bit-vector representation for the XOR and bit rotations, then again back to the integer representation for the additions, and so on and so forth. Given the structure of ChaCha20, this necessitates 640 decompositions and 320 recompositions per block of 64 bytes.

In the other implementation variant, we convert the input values only at the beginning and end, and therefore have to perform addition on the bit-level. The simplest way to achieve this is with a ripple-carry adder that takes one vectorized multiplication of all bits of the addends plus 30 multiplications with the carry bit, for a total of 31 communication rounds per addition. At 336 integer additions per block, this is obviously very expensive and can easily be improved. In MP-SPDZ there is a built-in carry-select adder, and in our implementation, compared to the ripple-carry adder, it reduced the rounds of communication necessary for one block from 1731 to 811. As expected, this implementation variant performs better, regardless of which adder is used. Nonetheless, the results confirm that ChaCha20 is not a suitable cipher for MPC.

2.3 Trivium

Trivium, based on a nonlinear feedback shift register (NFSR), is another cipher of the eSTREAM portfolio (De Cannière and Preneel, 2008). It has a simple structure with only bit operations, so that it can be applied to resource-constrained environments such as wireless sensors in IoT. The internal state of Trivium consists of 288 bits. It is initialized by a key and IV of 80 bits each, with all other bits except for the last three set to 0. To complete the initialization, 1152 keystream bits are generated and discarded. The generation of a keystream bit is the same for initialization phase and regular operation: the state is shifted by inserting three new bits, each generated by two XORs and one AND. The XOR of these new bits is the output bit. We chose Trivium because of this very simple construction and the low multiplicative complexity it promised. As detailed, it only takes three multiplications to produce one output bit, but by construction, this operation can be parallelized for at least 64 bits.[2] This means that a fully optimized implementation of Trivium should be able to generate 8 bytes in a single communication round.

Thanks to the simplicity of Trivium, we were able to realize this parallelisation with a (mostly) straightforward translation of the specification. We implemented the cipher so that it always generates blocks of 64 bits, by simply running the output function 64 times in a loop. MP-SPDZ was able to generate optimal code from this without any additional work on our part, but for the MPyC implementation, we had to do some optimizations by hand finally achieving the same results that the MP-SPDZ had produced automatically.

[2] 72 bits can be computed per round and by turning over the internal state (288 bits) four times, the cipher can be initialized in 16 rounds of communication.

2.4 LowMC

LowMC is a block cipher based on a substitution-permutation network which can be parametrized in a very flexible way. The number of rounds needed to achieve the desired level of security is determined as a function of several parameters. The way this is done is to consider and try to bound all known attacks and choose the number of rounds so that the most effective attack for a particular set of parameters is just not able to violate the security expectation. The first version of this round 'formula' was introduced at Eurocrypt 2015 (Albrecht et al., 2015). Soon after, optimized attacks were demonstrated and as a result, an updated round formula for LowMC was proposed by the designers (Albrecht et al., 2016a).

Implementing LowMC was a surprise in many ways. It has already been observed by (Albrecht et al., 2016a) that the high number of XORs entailed by the matrix operations of LowMC means that they can no longer be considered free. We found this to be true to an even larger extent in our setting. Even though it was clear that the MPyC implementation could only be done on a bit-level representation with all the overhead it entails - using one of the recommended parameter sets (256 bits blocksize, 80 bits keysize, 12 rounds), and keeping in mind that even a small Python integer weighs in at 26 bytes, the multiplication matrices take up almost 50 mega-bytes - we were still negatively surprised by the resulting performance. The MP-SPDZ implementation immediately performed closer to our expectations, achieving 6 KB per second.

2.5 MiMC / HADES

Introduced by (Albrecht et al., 2016b), MiMC is a radically simplified construction based on the idea to explore the typical field size used in MPC. The numbers of rounds is $\lceil \frac{n}{\log_2 3} \rceil$ (with $n \bmod 2 = 1$ as the chosen block size). The round function just adds the key and a round constant, then takes the result to the power of three. To decrypt, the process is performed in reverse, but with the exponent $\frac{2^{n+1}-1}{3}$ instead of 3. As decryption is therefore massively more expensive than encryption, the authors recommend using modes where it is not needed. Encryption, however, should be very efficient. Since the design does not contain any S-boxes and only uses addition and multiplication, the cipher can be evaluated in a binary field without any conversion, and needs only two multiplications per round. Later, (Grassi et al., 2019) developed what they call the HADES design. The core idea of this approach is to apply reduced versions of the nonlinear layer in some rounds. Instantiated for MiMC, this means that the cipher now operates not on a single, but any number of blocks, and in certain rounds, the exponentiation is only applied to some (in this instantiation: the first) of them. Additionally, (Albrecht et al., 2019) presented a generalized MiMC that can cope with prime fields and work on many field elements at once and therefore gives good amortized cost if it can be parallelized. Our work is based on the initial design (Albrecht et al., 2016b) but can be naturally extended to the new versions.

The drawback of all versions of this approach, however, is the large number of rounds required to obtain adequate security. In our tests, we used a block size of 127 bits which works out to 82 rounds. With two multiplications per round, this adds up to 164 communication rounds per block. MiMC is specified for GF(2n), but can also be used in GF(p). We chose the first variant, for which no conversion is necessary, as only XORs and multiplications are performed. Our findings indicate that encryption is indeed fast, as expected. However, the high amount of rounds has a noticeable impact on performance. Only a few measurements were done on decryption because it was immediately obvious that the high cost of reversing the exponentiation made it as slow as predicted.

3 PERFORMANCE EVALUATION

In this section we report our performance comparison results. We started from an ideal setting with three nodes and an almost ideal network setting, i.e., all nodes running on the same physical host. All tests were done on rather standard hardware, namely a Dell Latitude E7440 notebook with a Core i7-4600U CPU running on 2.1 Ghz with 4 cores. In all our tests we did not try to optimize the overall throughput by exhausting all hardware resources through massive block level parallelism, we only intended to measure the time for one block (or a fixed number of blocks) in sequential mode. Compared to other works which optimize overall throughput by parallelization, we think it is essential to understand the behavior of a single block. In particular, we wanted to also understand how the frameworks behave in practical network settings from LAN configurations to worldwide deployments as well as how many nodes can reasonably be supported for the given task.

3.1 Performance of Basic Operations

As a basis for understanding of the expected performance, and as a reference for the actual platform we use, we provide the actual measured latency for basic operations in Table 1.

Table 1: Time (ms) for 100 basic operations over vectors *a*, *b* of length *n*.

n	MP-SPDZ	MPyC	MPyC Vec
	map(operator.add, a, b)		vector_add(a, b)
1	<1	7	14
10	<1	46	19
100	<1	353	71
	map(operator.mul, a, b)		schur_prod(a, b)
1	12	59	60
10	13	258	82
100	17	2309	336
	reduce(operator.add, a)		mpc.sum(a)
1	<1	<1	9
10	<1	33	12
100	<1	507	18
	reduce(operator.mul, a)		mpc.prod(a)
1	<1	1	12
10	97	381	163
100	1017	3834	416

In MP-SPDZ, latency for addition is vanishingly small; for multiplication, it grows slightly (but clearly sublinearly) with increasing vector size, except for the multiplicative reduction, which exhibits an almost exactly linear slowdown with increasing input size. This may seem surprising at first, but is actually to be expected. Contrary to the other three operations under test, the multiplicative reduction cannot be performed in one communication round. As it is written, the multiplication operations form a linear list, so one should expect it to scale linearly with input size. The only optimization possible would be to organize the multiplications as a tree, so that multiplications on the same level could be performed in parallel, and the whole operation would scale logarithmically. Such an optimization is, however, not trivial to find and prove correct. It is therefore not surprising that the MP-SPDZ compiler did not perform it.

In MPyC, the unvectorized operations scale about linearly with vector size. The vectorized operations are faster, but only the sum is clearly sublinear in its behavior, scaling somewhat similar to MP-SPDZ's vector multiplication; the other operations exhibit a noticeable slowdown as their input vectors grow in size.

3.2 Performance of Cipher Implementations

The benchmarking results of cipher implementations with an almost ideal network (localhost) is shown in

Table 2. In MP-SPDZ AES latency is about 20 times lower than in MPyC with optimized code. Chacha20, the lightweight stream cipher, did not perform better. In MP-SPDZ it took almost the same time and in MPyC it took even twice the time of AES. Most surprisingly LowMC performed worse than expected. Leaving aside the initialization time it took twice the time of AES on MP-SPDZ and was 6 times slower on MPyC. MiMC, the second cipher designed specifically for MPC, performed very well. It could not achieve a speed-up on MP-SPDZ but performed 10 times faster on MPyC for encryption. Because of its design, decryption takes substantially more time (between a factor of 30 and 50), which makes it less attractive for applications. Trivium, another stream cipher under test, performed exceptionally well on both frameworks. On MP-SPDZ it was more than 5 times faster and in MPyC with certain manual optimization we achieved roughly a 20 time speed-up compared to AES. However, it has to be noted that Trivium in the standard configuration has only a security level of 80 bits compared to the other ciphers which provide at least 128 bits. Nevertheless, the structure of Trivium seems well suited for MPC implementations and should be used as a basis for future designs with stronger security levels.

Table 2: Ms / Byte for encryption in 3-party MPC, no latency.

Cipher	MPyC unopt		MPyC opt		MP-SPDZ	
	init	enc	init	enc	init	enc
AES-128	-	-	-	110	-	5
ChaCha20	-	-	-	216	-	6
LowMC	764	679	772	637	28	9
MiMC	-	-	-	11	-	5
Trivium	3300	23	780	6	18	<1

3.3 Network Latency and Loss

Besides the basic performance in ideal settings it is also important to investigate other aspects that are relevant for the real-world performance of MPC but sometimes get sidelined: network latency and network loss. One stated reason for not caring (too much) about it is that it is assumed that performance degrades linearly with increasing latency. This is indeed what we found in our experiments. What was surprising though was the size of the constant factor, and that it was significantly different between the frameworks.

We measured the behaviour of the implementations under increasing network delay and loss. The measurements were done on the very same hardware with all processes running on the same host and by using kernel level `netem` features to simulate network delay and loss. To make the results easier to compare, we performed computations that used about 1000 rounds of communication, then normalized the obtained time to one communication round.

The results of our experiments with variable latency are shown in Figure 1 and Figure 2 for a one-way delay from 0 to 50 ms. In the ideal zero-latency setting, a multiplication round takes less than a millisecond in both MPyC and MP-SPDZ; every 5 ms of one-way delay (which can be estimated to equal about 10 ms of round-trip delay) adds about 5 ms to MPyC, but almost 10 ms to MP-SPDZ. This means that the seemingly superior performance of MP-SPDZ versus MPyC disappears rapidly with increasing latency. The situation is slightly, but not fundamentally, different when we look at batched multiplications. Repeating the same test with vectors of length 100, with zero latency we get about 3 ms and again less than one ms for MPyC and MP-SPDZ respectively, and once again an increase of about 5 ms vs. almost 10 ms for 5 ms of one-way delay. This means that multiplications (looked at in isolation) are faster in MPyC than MP-SPDZ even in low-latency contexts. The break-even point for Trivium, on the other hand, is around 35 ms in our measurements, which is in a realistic range for reasonable distributed deployments which go beyond a single data center implementation.

All this indicates that although the Python based implementation of MPyC may be slower and seemingly less capable than MP-SPDZ with its impressive optimizing compiler, MPC operations are ultimately bound by the network, and MPyC profits from Python's highly-optimized asynchronous network stack.

In a second series of experiments we tried to simulate packet loss. The results are shown in Figure Figure 3 and the slowdown is comparable to the one we observed with one-way delay.

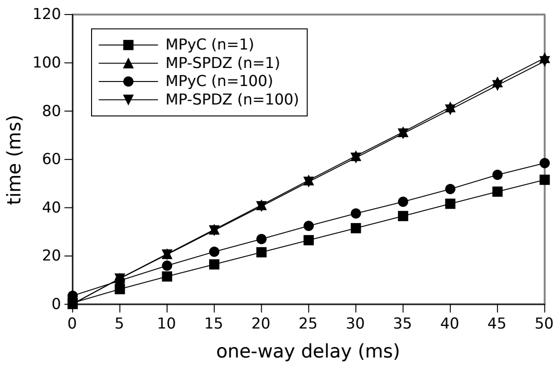

Figure 1: Time for one multiplication with increasing network delay.

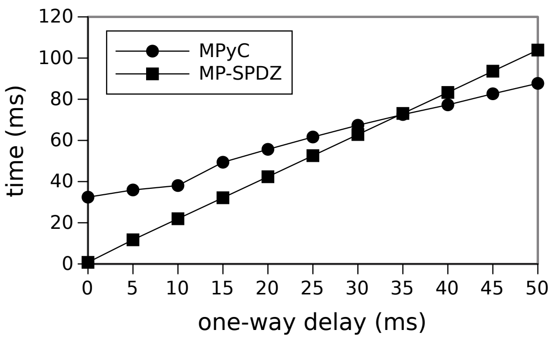

Figure 2: Time per communication round of Trivium with increasing network delay.

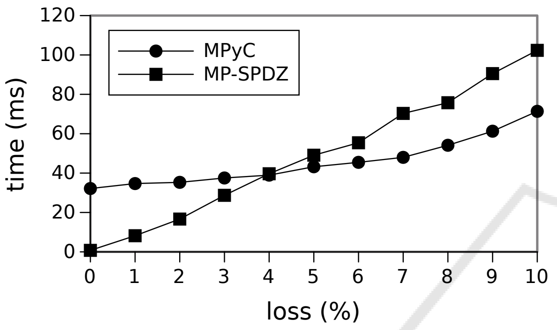

Figure 3: Time per communication round of Trivium with increasing network loss.

3.4 Scalability

Although MPC is very appealing, in reality most systems only support two or three nodes. From an application point of view this is often disappointing, as in many use cases more parties than that want to collaborate. For example, in a secure auction many more users are submitting bids and if they are not MPC nodes in their own right they have to trust the MPC nodes not to collude, and as shown in (Framner et al., 2019) non-collusion assumptions are not well accepted. From the protocol it is clear that the communication overhead limits the number of nodes for particular computations. In Figure 4 we show the results for more parties. Here, MP-SPDZ performs better with increasing number of parties than MPyC, which experiences a significant slowdown.

4 CONCLUSIONS

In this work we compared two generic MPC frameworks based on the same linear secret-sharing protocols, but a fundamentally different software approach. MP-SPDZ is the most used software framework to benchmark algorithms and is supposed to perform

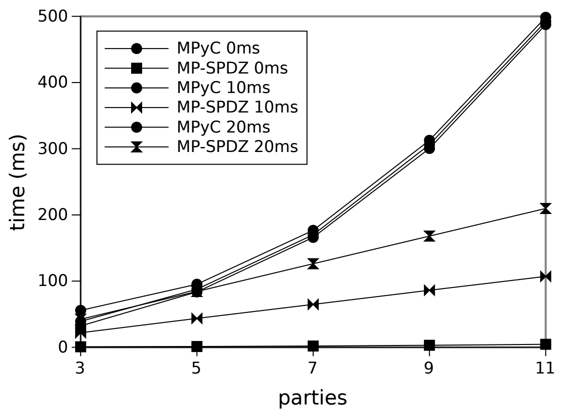

Figure 4: Time per communication round of Trivium with increasing number of parties.

best. MPyC on the other hand is very flexible and easy to use. For our tests we implemented various symmetric ciphers from the literature, some of them optimized for MPC, and did extensive testing and benchmarking on both frameworks. The goal was to understand how universal and generic the available MPC software is and to which extent they can be used without special knowledge about core protocols.

From our tests we learned that even for the most versatile software frameworks available and the basic MPC protocols it is hard to get things right, i.e. MPC is still far from being usable by software developers not familiar with the field. We showed that the practical performance cannot be trivially estimated from the algorithms to be implemented by estimating the multiplicative depth in advance, often additional work (e.g. bit decomposition) significantly penalizes the performance of the algorithms and manual optimization is needed. We also found that some of the algorithms that were specifically developed to have lower numbers of multiplications did not perform as expected, and that some stream ciphers are not well suited to MPC.

Trivium was found to be the best cipher for platform-independent application in MPC and we recommend it for cases where the lower security parameter is not an issue or it can be combined with other mechanisms (Sell et al., 2018). Recently, (Canteaut et al., 2018) proposed Kreyvium to improve on this, but an interesting question remains open: can the Trivium approach be scaled up to generate more than 64 (and maybe even arbitrarily many) keystream bits per round of communication without weakening security or exponentially increasing the size of the internal state?

Contrary to most of existing literature we also addressed non-optimal network settings. In our tests we found that although MP-SPDZ performs by far better in high throughput low latency settings, it gets surprisingly outperformed by MPyC in scenarios with higher network latency. The asynchronous architecture of MPyC guarantees more efficient use of the network layer in this scenarios which could even compensate for the optimizing compiler used by MP-SPDZ. However, for the scalability in the number of parties we found the opposite; here MP-SPDZ behaved as expected and MPyC seems to experience significant slowdowns.

ACKNOWLEDGEMENTS

This work has received funding from the Austrian Research Promotion Agency (FFG) through project FlexProd (GA No. 871395) and from the European Union's Horizon 2020 research and innovation programme under GA No. 830929 (CyberSec4EU).

REFERENCES

Albrecht, M., Rechberger, C., Schneider, T., Tiessen, T., and Zohner, M. (2016a). Ciphers for MPC and FHE. Cryptology ePrint Archive, Report 2016/687.

Albrecht, M. R., Grassi, L., Perrin, L., Ramacher, S., Rechberger, C., Rotaru, D., Roy, A., and Schofnegger, M. (2019). Feistel Structures for MPC, and More. In *Computer Security – ESORICS 2019*, Lecture Notes in Computer Science, pages 151–171. Springer.

Albrecht, M. R., Grassi, L., Rechberger, C., Roy, A., and Tiessen, T. (2016b). MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Advances in Cryptology - ASIACRYPT 2016, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, pages 191–219.

Albrecht, M. R., Rechberger, C., Schneider, T., Tiessen, T., and Zohner, M. (2015). Ciphers for MPC and FHE. In *Advances in Cryptology - EUROCRYPT 2015, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 430–454.

Araki, T., Furukawa, J., Lindell, Y., Nof, A., and Ohara, K. (2016). High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 805–817, New York, New York, USA. ACM Press.

Bernstein, D. J. (2008). ChaCha, a variant of Salsa20.

Canteaut, A., Carpov, S., Fontaine, C., Lepoint, T., Naya-Plasencia, M., Paillier, P., and Sirdey, R. (2018). Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. *Journal of Cryptology*, 31(3):885–916.

Damgård, I., Geisler, M., Krøigaard, M., and Nielsen, J. B. (2009). Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography - PKC 2009, Irvine, CA, USA, March 18-20, 2009. Proceedings*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer.

Damgård, I., Keller, M., Larraia, E., Miles, C., and Smart, N. P. (2012a). Implementing AES via an Actively/Covertly Secure Dishonest-Majority MPC Protocol. In *Security and Cryptography for Networks - 8th International Conference, {SCN} 2012, Amalfi, Italy, September 5-7, 2012. Proceedings*, pages 241–263.

Damgård, I., Pastro, V., Smart, N., and Zakarias, S. (2012b). Multiparty computation from somewhat homomorphic encryption. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

De Cannière, C. and Preneel, B. (2008). *Trivium*, pages 244–266. Springer Berlin Heidelberg, Berlin, Heidelberg.

Framner, E., Fischer-Hübner, S., Lorünser, T., Alaqra, A. S., and Pettersson, J. S. (2019). Making secret sharing based cloud storage usable. *Information and Computer Security*.

Grassi, L., Lüftenegger, R., Rechberger, C., Rotaru, D., and Schofnegger, M. (2019). On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy. Cryptology ePrint Archive, Report 2019/1107.

Happe, A., Wohner, F., and Lorünser, T. (2017). The Archistar Secret-Sharing Backup Proxy. ARES '17, pages 88:1–88:8, New York, NY, USA. ACM.

Hastings, M., Hemenway, B., Noble, D., and Zdancewic, S. (2019). SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, pages 479–496, Los Alamitos, CA, USA. IEEE Computer Society.

Keller, M. (2019). Multi-Protocol SPDZ. https://github.com/data61/MP-SPDZ, accessed 2020-04-28.

Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. (2009). Secure Two-Party Computation Is Practical. In *Advances in Cryptology - ASIACRYPT 2009, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer.

Rechberger, C., Soleimany, H., and Tiessen, T. (2018). Cryptanalysis of Low-Data Instances of Full LowMCv2. *{IACR} Trans. Symmetric Cryptol.*, 2018(3):163–181.

Schoenmakers, B. (2018). MPyC–Python Package for Secure Multiparty Computation. In *Theory and Practice of Multi-Party Computation 2018 - TPMPC 2018*, Aarhus.

Sell, L., Pohls, H. C., and Lorunser, T. (2018). C3S: Cryptographically combine cloud storage for cost-efficient availability and confidentiality. In *CloudCom 2018*.