D1.2 General MPC Framework

Development of Novel General-Purpose MPC Protocols

Sakina Asadova (PHI), Niek J. Bouman (TUE), Satrajit Ghosh (AU), Paul Koster (PHI), Peter Sebastian Nordholt (ALX), Claudio Orlandi (AU), Berry Schoenmakers (TUE), Peter Scholl (AU), Meilof Veeningen (PHI), Tore Kasper Frederiksen (ALX)

The project SODA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583.

Project Information

Scalable Oblivious Data Analytics

Project number:	731583
Strategic objective:	H2020-ICT-2016-1
Starting date:	2017-01-01
Ending date:	2019-12-31
Website:	https://www.soda-project.eu/

Document Information

Title: D1.2 General MPC Framework

ID:	D1.2	Type:	R	Dissemination level:	PU
		Month:	M21	Release date:	September 30, 2018

Contributors, Editor & Reviewer Information

Contributors (person/partner)	Sakina Asadova (PHI) Niek J. Bouman (TUE) Satrajit Ghosh (AU) Paul Koster (PHI) Peter Sebastian Nordholt (ALX) Claudio Orlandi (AU) Berry Schoenmakers (TUE) Peter Scholl (AU) Meilof Veeningen (PHI) Tore Kasper Frederiksen (ALX)
Editor (person/partner)	Peter Scholl (AU), Claudio Orlandi (AU)
Reviewer (person/partner)	Peter Nordholt (ALX)

Release History

Release	**D**ate issued	**R**elease description / changes made
1.0	September 30, 2018	First release to EU

SODA Consortium

Full Name	Abbreviated Name	Country
Philips Electronics Nederland B.V.	PHI	Netherlands
Alexandra Institute	ALX	Denmark
Aarhus University	AU	Denmark
Göttingen University	GU	Germany
Eindhoven University of Technology	TUE	Netherlands

Table 1: Consortium Members

Executive Summary

This deliverable presents research undertaken by SODA project members on general-purpose protocols for secure computation, which was carried out during the first phase of the project. The main goal of this research is to develop new, optimized protocols and primitives that can be used for applications requiring private data analytics, particularly when the quantity of data becomes large. The scope of the research described in this deliverable covers efficient, general cryptographic protocols for secure two-party and multi-party computation, as well as verifiable computation. More specifically, we present several new multi-party computation (MPC) protocols with improved efficiency characteristics in various measures; these protocols are fully general in the sense that they can be used to privately compute any functionality. We also present several new protocols for more specific, mid-level functionalities. These protocols are useful, general building blocks in many applications of privacy-preserving big data analytics.

We now give a brief summary of the results in this document. See Section 1 for a more detailed exposition.

Our first results focus on the case of secure multi-party computation over rings. Traditional MPC protocols require the function being computed to be expressed as a sequence of arithmetic operations over a finite field, which can lead to great overheads in practice. In contrast, when a ring such as the integers modulo 2^{32} is allowed, computations are much closer aligned to what happens in plaintext computations, and can be much more efficient. We present two new paradigms for efficient secure computation over rings, one in the setting where a majority of the participants are honest, and one where only at least one party is guaranteed to be honest.

Our second area of focus is efficient MPC protocols in the preprocessing model. These protocols are important for complex computations involving large quantities of data, as they allow much of the heavy work to be offloaded into an offline phase that is independent of the parties' inputs. We have several results in this setting, including: (1) More efficient batch verification of multiplication triples in the honest majority setting; (2) A more efficient protocol for oblivious linear function evaluation, and its application to faster 2-party computation; (3) A new approach to MPC with preprocessing, using additively homomorphic commitments instead of information-theoretic MACs to obtain active security at lower cost; and (4) A new approach to constructing *large-scale* multi-party computation protocols in the preprocessing model, when the number of parties grows very large.

In the topic of verifiable computation, we present a construction of *adaptive zero-knowledge SNARKs*, which we show can be applied to obtain a *verifiable* multi-party computation protocol, allowing private computations to be verified, such that even the prover does not know the sensitive data. We do this by building upon the previous Pinocchio and Trinocchio systems for SNARKs and verifiable computation.

We also present several new mid-layer protocols, which are less general than the previous protocols, but still have a wide range of applications. Specifically, we give improved protocols for inexact string matching, data-oblivious array slicing, and privacy-preserving, verifiable fixed-point division. These building blocks can be used in a number of data-mining applications such as private DNA matching and linear programming.

Our final contribution is closer to the implementation layer of the secure computation stack, and is a new approach to implementing finite field arithmetic modulo p in C++. This type of arithmetic is fundamental in all kinds of cryptographic protocols, so is a highly important object of study. We describe a simple and general implementation which obtains comparable performance to specialised, assembly-based code, by exploiting compile-time optimizations with template programming.

About this Document

Role of the Deliverable

This deliverable contains several research results on general-purpose protocols for secure computation, with an overall aim of improving the efficiency of large-scale private computations on big data. It is part of Work Package 1 of the SODA project.

Relationship to Other SODA Deliverables

The basis for this deliverable was the D1.1 deliverable on the state-of-the-art in secure computation, which identified gaps in the existing literature and informed our decisions on which research areas to focus. Deliverable D2.2 also contains secure computation protocols targeted at big data analytics, but with a focus on specific applications such as linear algebra and neural networks, whereas D1.2 is more aimed at general-purpose protocols and primitives. Many of the results from D1.2 are being implemented and will be used to demonstrate practical applications of secure computation in big data, which will be reported on further in deliverables D1.3 and D4.2.

See Section 1.2 for more details on the connections with future SODA work.

Structure of this Document

Section 1 introduces this document and summarises the main contributions and their relationships to the SODA project. Following that, there are 10 technical sections detailing the research carried out. Section 2 describes two new multi-party computation protocols over rings, while Sections 4–7 contain new, efficient multi-party computation protocols in the preprocessing model, and Section 3 shows how to scale up secure computation protocols to handle many parties. Section 8 gives results on verifying private computations using SNARKS, and Sections 9–11 contain new protocols for string matching, array slicing and fixed-point division. Finally, Section 12 describes a novel approach to implementing finite field arithmetic modulo p using templates in C++.

Table of Contents

1 Introduction

1.1 Overview

This report documents the progress made by SODA members in the development of novel, general-purpose protocols for performing big data analytics using secure computation. The research topics covered were chosen based on gaps and limitations in the existing literature that were identified following the previous deliverable D1.1, on the state-of-the-art in secure computation protocols. The work produced in this period has been published at some of the top conferences in cryptography, including the IACR conferences *CRYPTO*, *Asiacrypt* and *PKC*, as well as ACM's flagship *CCS* conference. See Section 1.3 for more details on these publications. Below we summarise the technical contributions of our work.

Secure Computation Over Rings. Following our state-of-the-art study in D1.1, the first major issue we identified was that most practical secure computation protocols require the function being evaluated to be expressed in terms of finite field operations. Since this unnatural requirement can often incur a costly overhead, we chose to focus on constructing *general-purpose, secure computation protocols over rings*, since arithmetic over rings such as the integers modulo 2^k much closer resembles computations carried out in practice. This is because computation modulo 2^k closely matches what happens on standard CPUs, allowing protocol designers to take advantage of the tricks found in this domain. For instance, functions containing comparisons and bitwise operations are typically easier to implement using arithmetic modulo 2^k compared with finite field arithmetic, and also very common in applications of MPC.

In Section 2 we describe two new paradigms for efficient secure computation over rings, one in the setting where a majority of the participants are honest, and one where only at least one party is guaranteed to be honest. Our first protocol is a general compiler from passive to active security, with perfect security and very low overhead. The compiled protocol tolerates a lower corruption threshold than the original protocol, so it is suitable in the honest majority setting. Our second protocol is designed for the dishonest majority setting, where all-but-one of the participants may be corrupted. It is based upon a new type of information-theoretic MAC modulo 2^k, used to extend the SPDZ protocol, which only works over fields, to the ring setting. We believe that this tool will also have applications in other settings.

Efficient Protocols in the Preprocessing Model. The second area we chose to look at is building more efficient, general-purpose protocols in the preprocessing model. The preprocessing model allows much of the heavy computation in an MPC protocol to be offloaded to an ''offline' phase that is independent of the inputs to the computation. This leads to a very efficient online phase, that can even handle large-scale computations such as statistical analyses on big data. However, a serious bottleneck in these scenarios is the preprocessing phase, which is typically around *10 thousand times slower* than the online phase, so still very costly for complex computations. We therefore chose to optimize the preprocessing phase of general, multi-party computation protocols in this model.

In Section 4 we present a technique for efficient batch verification of *multiplication triples*, which are one of the main types of random, correlated data produced in the preprocessing phase. We show how this technique can be applied to greatly improve the preprocessing phase of protocols for secure computation over fields with an honest majority, obtaining a reduction in communication of up to *7 times* compared with the previous fastest protocol by Lindell and Nof [90]. In particular, in the 3-PC setting, each party sends only two field elements per multiplication, while for a general number of

parties each party sends just seven field elements. We also show how to achieve fairness, which Lindell and Nof left as an open problem. A concurrent work by Chida et al. [27] from *CRYPTO 2018* describes a protocol with similar complexity using different techniques; we remark that the performance of their protocol can be further improved by around a factor of 2 using our optimizations.

Next, in Sections 5–6, we study protocols based on *oblivious linear function evaluation* (OLE), which we identified as a key, emerging tool in this area of protocols in the preprocessing model. Our main results are a new protocol for OLE with active security, and a new secure two-party computation protocol with active security based on OLE; both protocols improve significantly on the performance of previous protocols. We believe that the efficiency of our protocols, both in asymptotic and practical terms, establishes OLE and its variants as an important foundation for efficient and scalable secure computation.

Afterwards, in Section 7, we present a new protocol for secure computation which uses additively homomorphic commitments on secret-shared values to prevent active corruptions. This is in contrast to most actively secure multi-party computations over arithmetic circuits against a dishonest majority, which use MACs to prevent active corruptions [21, 43, 55, 78]. Furthermore, our scheme works over arbitrary fields, whereas previous work in the same setting has mostly considered binary, extension or large fields. Finally, our scheme manages to reduce the constants in construction of *multiplication triples* over several previous works.

Large-Scale MPC for Dishonest Majority. In Section 3, we investigate the possibility of extending existing, general-purpose secure computation protocols to a scenario with *many parties*, with the goal of reducing both the amount of data sent, and the number of connections each party has to maintain over the network. Our starting point is the classic GMW and BMR protocols for passively secure multi-party computation; these satisfy the strong security requirement of allowing $n-1$ out of n corrupted parties, however, they scale poorly when n is large — the communication cost is around $n^2\kappa$ bits per AND gate, where κ is a security parameter, e.g. 128. Our main idea is to relax the corruption threshold slightly, so that for large n we can obtain more efficient protocols when, say, 80% of the parties may be corrupt. We do this by modifying the GMW and BMR protocols, secure for $n-1$ corruptions, to use *short cryptographic keys*, with the aim of basing security on the concatenation of all honest parties' keys. This results in a more efficient protocol tolerating fewer corruptions, whilst also introducing an LPN-style syndrome decoding assumption. This is the first time that this corruption setting has been studied in a practical context. Our techniques start to improve upon existing protocols when there are around $n = 20$ parties with $h = 6$ honest parties, and as these increase we obtain up to a 13 times reduction (for $n = 400, h = 120$) in communication complexity for our GMW variant, compared with the best-known GMW-based protocol modified to use the same threshold.

Privacy-Preserving, Verifiable Computations. In Section 8, we show how to practically verify the correctness of the results of computations that have been performed in a privacy-preserving manner. To do this we give a new construction of *adaptive zk-SNARKs* based on Pinocchio, which allows a prover to repeatedly prove statements about some committed input, whilst hiding the input in a zero-knowledge manner. We then show how this can be applied to the Trinocchio system, to obtain a verifiable multi-party computation protocol, where private computations can be repeatedly performed and verified, such that the sensitive data is even hidden from the prover by multi-party computation.

Mid-level Protocols. Our final contributions come under the umbrella of 'mid-level protocols', which are slightly more specialised than general-purpose computations for, say, arithmetic circuits,

but still general enough to be used in a range of applications. Firstly, we present two new tools for oblivious computations on strings; these can be generally applied in a number of areas, but in deliverable D2.2 we have shown that these are particularly useful for privacy-preserving DNA matching. The first tool, given in Section 9, is a technique for oblivious verification of approximate string matching. More concretely, we give a protocol for verifying that a search string occurs at a given position in a reference string, within a bounded edit distance, where the search string, reference string and position can all be kept private during the verification. This procedure can be performed in either a standard two-party scenario with a prover and verifier, or a distributed setting where all values are secret-shared. Secondly, in Section 10 we present a new data-oblivious technique for array slicing, allowing a fixed number of consecutive array elements to be retrieved, where the exact index remains hidden.

Finally, in Section 11 we describe a protocol for proving that a value contained in a commitment (or encryption) is a fixed-point multiplication (or division) of two other committed values. This is an important building block in constructing verifiable computations of more complex functionalities, as needed when using, for example, the Pinocchio system for verifiable computation, or the Trinocchio system for verifiable private computations using MPC.

Implementing Arithmetic Modulo p. We describe a new C++ library for multiprecision arithmetic for numbers in the order of 100–500 bits, i.e., representable with just a few limbs. The library is written in "optimizing-compiler-friendly" C++, with an emphasis on the use of fixed-size arrays and particular function-argument-passing styles (including the avoidance of naked pointers) to allow the limbs to be allocated on the stack or even in registers. Depending on the particular functionality, we get close to, or significantly beat the performance of existing libraries for multiprecision arithmetic that employ hand-optimized assembly code. Most functions in the library are constant-time, which is a necessity for secure implementations of cryptographic protocols. Beyond the favorable runtime performance, our library is, to the best of the author's knowledge, the first library that offers big-integer computations during compile-time. For example, when implementing finite-field arithmetic with a fixed modulus, this feature enables the automatic precomputation (at compile time) of the special modulus-dependent constants required for Barrett and Montgomery reduction. Another application is to parse (at compile-time) a base-10-encoded big-integer literal.

1.2 Impact on Future SODA Deliverables

The theoretical protocols from this deliverable have potential to be used in future proof-of-concept implementations and use-case demonstrators in other parts of the SODA project. In fact, several of the results presented have already had an impact on work carried out by SODA researchers in connection with these tasks. In particular, the SPD$\mathbb{Z}_{2^k}$ protocol from Section 2.5 has being implemented for the latest release (v1.0.0) of the FRESCO framework by SODA researchers at Alexandra Institute. This led to improved performance of the core protocols for secure comparisons and integer multiplications in FRESCO, and will hopefully be used to demonstrate new data-mining applications in future. Furthermore, the string matching protocol from Section 9 has been implemented in MPyC to demonstrate the application of private DNA matching, and Geppetri (Section 8) has been implemented as part of the PySNARK library. We expect that these results, and others, will be reported on in detail as part of the upcoming deliverable from WP4.

1.3 Publications

The research described in this report have been published at the top-tier venues in cryptography, including CRYPTO, Asiacrypt, PKC and ACM CCS. Below we list publications produced by SODA WP1 members as part of this phase. Note that items 3, 5, 6, 12 and 13 have been omitted from this deliverable for space reasons, although they are relevant to the goals of SODA.

1. Pinocchio-Based Adaptive zk-SNARKs and Secure/Correct Adaptive Function Evaluation. Meilof Veeningen. *Africacrypt 2017.*
2. TinyOLE: Efficient Actively Secure Two-Party Computation from Oblivious Linear Function Evaluation. Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges and Roberto Trifiletti. *ACM Computer and Communications Security 2017.*
3. Low Cost Constant Round MPC Combining BMR and Oblivious Transfer. Carmit Hazay, Peter Scholl and Eduardo Soria-Vazquez. *Asiacrypt 2017.*
4. Maliciously Secure Oblivious Linear Function Evaluation with Constant Overhead. Satrajit Ghosh, Jesper Buus Nielsen and Tobias Nilges. *Asiacrypt 2017.*
5. Committed MPC — Maliciously Secure Multiparty Computation from Homomorphic Commitments. Tore Frederiksen, Benny Pinkas and Avishay Yanai. *PKC 2018.*
6. Extending Oblivious Transfer with Low Communication via Key-Homomorphic PRFs. Peter Scholl. *PKC 2018.*
7. Committed MPC - Maliciously Secure Multiparty Computation from Homomorphic Commitments. Tore K. Frederiksen, Benny Pinkas, and Avishay Yanai. *PKC 2018.*
8. Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification Peter Sebastian Nordholt and Meilof Veeningen. *ACNS 2018.*
9. Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. Ivan Damgård, Claudio Orlandi and Mark Simkin. *CRYPTO 2018.*
10. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl and Xaoping Xing. *CRYPTO 2018.*
11. TinyKeys: A New Approach to Efficient Multi-Party Computation. Carmit Hazay, Emmanuela Orsini, Peter Scholl and Eduardo Soria-Vazquez. *CRYPTO 2018.*
12. Yes, There is an Oblivious RAM Lower Bound! Kasper Green Larsen and Jesper Buus Nielsen. *CRYPTO 2018*, **Best Paper Award**.
13. Oblivious RAM with Small Storage Overhead Michael Raskin and Mark Simkin. *IACR Cryptology ePrint Archive (in submission.*

2 Efficient Secure Computation in Rings

Secure Multiparty Computaton (MPC) allows a set of participants $P_1, \ldots, P_n$ with private inputs respectively $x_1, \ldots, x_n$ to learn the output of some public function f evaluated on their private inputs i.e., $z = f(x_1, \ldots, x_n)$ without having to reveal *any other information* about their inputs. Seminal MPC results from the 80s [119, 65, 11, 26] have shown that with MPC it is possible to securely evaluate any boolean or arithmetic circuit with information theoretic security, under the assumption that a strict minority of the participants are corrupt, or with computational security when no such honest majority can be assumed.

As is well known, the most efficient MPC protocols are only passively secure. What is perhaps less well known is that by settling for passive security, we also get a wider range of domains over which we can do MPC. In addition to the standard approach of evaluating boolean or arithmetic circuits over fields, we can also efficiently perform computations over other rings. This has been demonstrated by the Sharemind suite of protocols [16], which works over the ring $\mathbb{Z}_{2^\ell}$. Sharemind's success in practice is probably, to a large extent, due to the choice of the underlying ring, which closely matches the kind of ring CPUs naturally use. Closely matching an actual CPU architecture allows easier programming of algorithms for MPC, since programmers can reuse some of the tricks that CPUs use to do their work efficiently.

While passive security is a meaningful security notion that is sometimes sufficient, one would of course like to have security against active attacks. However, the known techniques, such as the GMW compiler, for achieving active security incur a significant overhead, and while more efficient approaches exist, they usually need to assume that the computation is done over a field, and they always have an overhead that depends on the security parameter. Typically, such protocols, like the BeDOZa or SPDZ protocols [14, 40, 35], start with a *preprocessing phase* which generates the necessary correlated randomness [73] in the form of so called *multiplication triples*. This is followed by an information theoretic and therefore very fast *online phase* where the triples are consumed to evaluate the arithmetic circuit. To get active security in the on-line phase, protocols employ information-theoretic MACs that allow to detect whether incorrect information is sent. Using such MACs forces the domain of computation to be a field which excludes, of course, the ring $\mathbb{Z}_{2^\ell}$.

In this section we present two recent methods developed by SODA researchers to address this problem. Both works were presented at CRYPTO 2018.

2.1 A Compiler for Active Security Over Arbitrary Rings

Given the above state of affairs, a very natural question is:

Can we go from passive to active security at a small cost and can we do so in a general way which allows us to do computations over general rings?

In [39] SODA researchers have addressed the above question by making three main contributions:

1. A generic transformation that compiles a protocol with passive security against at least 2 corruptions into one that is actively secure, but against a smaller number of corruptions. This works both for the preprocessing and the standard model. The transformation preserves perfect and statistical security security and its overhead depends only on the number of players, and not on the security parameter. Thus, for a constant number of parties it loses only a constant factor in efficiency.

2. We present a preprocessing protocol for 3 parties. It generates multiplication triples to be used by a particular protocol produced by our compiler. This preprocessing can generate triples over

any ring $\mathbb{Z}_m$ and has constant computational overhead for large enough m; more precisely, if m is exponential in the statistical security parameter. We build this preprocessing from scratch, not by using our compiler. This, together with our compiler, gives a plug-in replacement for the Sharemind protocol as explained below.

3. A generic transformation that works for a large class of protocols including those output by our passive-to-active compiler. It takes as input a protocol that is secure with abort and satisfies certain extra conditions, and produces a new protocol with complete fairness [29]. In security with abort, the adversary gets the output and can then decide if the protocol should abort. In complete fairness the adversary must decide whether to abort without seeing the output. This is relevant in applications where the adversary might "dislike" the result and would prefer that it is not delivered. The transformation has an additive overhead that only depends on the size of the output and not the size of the computation. It works in the honest majority model without broadcast. In this model we cannot guarantee termination in general so security with complete fairness is essentially the best we can hope for.

2.2 Our Contributionss

Our passive-to-active compiler can, for instance, be applied to the straightforward 3-party protocol that represents secret values using additive secret sharing over $\mathbb{Z}_{2^\ell}$ and does secure multiplication using multiplication triples created in a preprocessing phase. This protocol is secure against 2 passive corruptions. Applying our compiler results in a 3-party protocol Π in the preprocessing model that is information theoretically secure against 1 corruption and obtains active security with abort. Π can be used as plug-in replacement for the Sharemind protocol. It has active instead of passive security and is essentially as efficient. This, of course, is only interesting if we can implement the required preprocessing efficiently, which is exactly what we do as our second result, discussed in more detail below.

The compiler is based on the idea of turning each party in the passively secure protocol into a "virtual" party, and then each virtual party is independently emulated by 2 or more of the real parties (i.e., each real party will locally run the code of the virtual party). Intuitively, if the number of virtual parties for which a corrupt party is an emulator is not larger than the privacy threshold of the original protocol, then our transform preserves the *privacy* guarantees of the original protocol. Further, if we can guarantee that each virtual party is emulated by at least one honest party, then this party can detect faulty behaviour by the other emulators and abort if needed, thus guaranteeing *correctness*. Moreover, if we set the parameters in a way that we are guaranteed an honest majority among the emulators, then we can even decide on the correct behaviour by majority vote and get full active security. While this in hindsight might seem like a simple idea, proving that it actually works in general requires us to take care of some technical issues relating, for instance, to handling the randomness and inputs of the virtual parties.

The approach is closely related to replicated secret sharing which has been used for MPC before [95, 57], but to the best of our knowledge, this is the first general construction that transforms an entire passively secure protocol to active security. From this point of view, it can be seen as a construction that unifies and "explains" several earlier constructions.

While our construction works for any number of parties it unfortunately does not scale well, and the resulting protocol will only tolerate corruptions of roughly $\sqrt{n}$ of the n parties and has a multiplicative overhead of order n compared to the passively secure protocol. This is far from the constant fraction of corruptions we know can be tolerated with other techniques. We show two ways to improve

this. First, while our main compiler preserves adaptive security, we also present an alternative construction that only works for static security but tolerates $n/\log n$ active corruptions, and has overhead $\log^2 n$. Second, we show that using results from [28], we get a protocol for any number n of parties tolerating roughly $n/4$ malicious corruptions. We do this by starting from a protocol for 5 parties tolerating 2 passive corruptions, use our result to constructs a 5 party protocol tolerating 1 active corruption, and then use a generic comstruction from [28] based on monotone formulae. Note that a main motivation for the results from [28] was to introduce a new approach to the design of multiparty protocols. Namely, first design a protocol for a constant number of parties tolerating 1 active corruption, and then apply player emulation and monotone formulae to get actively secure multiparty protocols. From this point of view, adding our result extends their idea in an interesting way: using a generic transformation one can now get active and information theoretic security for a constant fraction of corruptions from a seemingly even simpler object: a protocol for a constant number of parties that is *passively* secure against 2 corruptions.

Our second result, the preprocessing protocol, is based on the idea that we can quite easily create multiplication triples involving secret shared values $a,b,c \in \mathbb{Z}_m$ and where $ab = c \bmod m$ if parties behave honestly. The problem now is that the standard efficient approach to checking whether $ab = c \bmod m$ only works if m is prime, or at least has only large prime factors. We solve this by finding a way to embed the problem into a slightly larger field $\mathbb{Z}_p$ for a prime p. We can then check efficiently if $ab = c \bmod p$. In addition we make sure that a,b are small enough so that this implies $ab = c$ over the integers and hence also that $ab = c \bmod m$.

Our final result, the compiler for complete fairness, works for protocols where the output is only revealed in the last round, as is typically the case for protocols based on secret sharing. Roughly speaking, the idea is to execute the protocol up to its last round just before the outputs are delivered. We then compute verifiable secret sharings of the data that parties would send in the last round – as well as one bit that says whether sending these messages would cause an abort in the original protocol. Of course, this extra computation may abort, but if it does not and we are told that the verifiably shared messages are correct, then it is too late for the adversary to abort; as we assume an honest majority the shared messages can always be reconstructed. While this basic idea might seem simple, the proof is trickier than one might expect – as we need to be careful with the assumptions on the original protocol to avoid selective failure attacks.

2.3 Overview of Our Techniques

The goal of our transform is to take a passively secure protocol and convert it into a protocol that is secure against a small number of active corruptions.

For simplicity, let us start with a passively secure n-party protocol ($n \geq 3$) that we will convert into an n-party protocol that is secure against *one* active corruption, in a model where parties can agree on unbiased random bits.

The main challenge in achieving security against an actively corrupted party, is to prevent it from deviating from the protocol description and sending malformed messages. Our protocol transform is based upon the observation that, assuming one active corruption, every pair of parties contains at least one honest party. Now instead of letting the real parties directly run the passively secure protocol, we will let pairs of real parties simulate virtual parties that will compute, using the passively secure protocol, the desired functionality on behalf of the real parties. More precisely, for $1 \leq i \leq n$, the real parties P_i and P_{i+1} will simulate virtual party $\mathbb{P}_i$. In the first phase of our protocol, P_i and P_{i+1} will agree on some common input and randomness that we will specify in a moment. In the second phase, the virtual parties will run a passively secure protocol on the previously agreed inputs and

randomness. Whenever virtual party $\mathbb{P}_i$ sends a message to $\mathbb{P}_j$, we will realize this by letting P_i and P_{i+1} both send the same message to P_j and P_{j+1}. Note that when both P_i and P_{i+1} are honest, these two messages will be identical since they are constructed according to the same (passively secure) protocol, using the same shared randomness and the previously received messages. The "action" of receiving a message at the virtual party $\mathbb{P}_j$ is emulated by having the real parties P_j and P_{j+1} both receive two messages each. Both parties now check locally whether the received messages are identical and, if not, broadcast an "abort" message. Otherwise they continue to execute the passively secure protocol. The high-level idea behind this approach is that the adversary controlling one real party cannot send a malformed message and at the same time be consistent with the other honest real party simulating the same virtual party. Hence, either the adversary behaves honestly or the protocol will be aborted.

Remember that we need all real parties emulating the same virtual party to agree on a random tape and a common input. Agreeing on a random tape is trivial in the $\mathscr{F}_{\mathsf{cflip}}$-hybrid model, we can just invoke $\mathscr{F}_{\mathsf{cflip}}$ for each virtual $\mathbb{P}_i$ and have it send the random string to the corresponding real parties P_i and P_{i+1}. Moreover, in the process of agreeing on inputs for the virtual parties we need to be careful in not leaking any information about the real parties' original inputs. Towards this goal, we will let every real party secret share its input among all virtual parties. Now, instead of letting the underlying passively secure protocol compute $f(x_1,\ldots,x_n)$, where real P_i holds input x_i, we will use it to compute $f'((x_1^1,\ldots,x_n^1),\ldots,(x_1^n,\ldots,x_n^n)) := f(\bigoplus_i x_1^i,\ldots,\bigoplus_i x_n^i)$, where virtual party $\mathbb{P}_i$ has input $\left(x_1^i,\ldots,x_n^i\right)$, i.e. one share of every original input.

As a small example, for the case of three parties, we would get $\mathbb{P}_1 = \{\mathrm{P}_1,\mathrm{P}_2\}$ holding input $\left(x_1^1,x_2^1,x_3^1\right)$, $\mathbb{P}_2 = \{\mathrm{P}_2,\mathrm{P}_3\}$ with input $\left(x_1^2,x_2^2,x_3^2\right)$, and $\mathbb{P}_3 = \{\mathrm{P}_3,\mathrm{P}_1\}$ with $\left(x_1^3,x_2^3,x_3^3\right)$. Since every real party only participates in the simulation of two virtual parties, no real party learns enough shares to reconstruct the other parties' inputs. More precisely, for arbitrary $n \geq 3$ and one corruption, each real party will participate in the simulation of two virtual parties, thus the underlying passively secure protocol needs to be at least passively 2-secure. Actually, each real party will learn not only two full views, but also one of the inputs of each other virtual party, since it knows the shares it distributed itself. As we will see in the security proof this is not a problem and passive 2-security is, for one active corruption, a sufficient condition on the underlying passively secure protocol.

The approach described above can be generalized to a larger number of corrupted parties. The main insight for one active corruption was that each set of two parties contains one honest party. For more than one corruption, we need to ensure that each set of parties of some arbitrary size contains at least one honest party that will send the correct message. Given n parties and t corruptions, each virtual party needs to be simulated by at least $t+1$ real parties. We let real parties $\mathrm{P}_i,\ldots,\mathrm{P}_{i+t}$ simulate virtual party $\mathbb{P}_i$[1]. This means that every real party will participate in the simulation of $t+1$ virtual parties. Since we have t corruptions, the adversary can learn at most $t\,(t+1)$ views of virtual parties, which means that our underlying passively secure protocol needs to have at least passive $\left(t^2+t\right)$-security.

2.4 Related Work

Besides what is already mentioned above, there are several other relevant works. Previous compilers, notably the GMW [65] and the IPS compiler [74, 91], allow to transform passively secure protocols into maliciously secure ones. The GMW compiler uses zero-knowledge proofs and, hence, is not blackbox in the underlying construction. It produces protocols which are far from practically efficient.

[1] Any other distribution of real party among virtual parties that ensures that each real party simulates equally many virtual parties would work as well.

The IPS compiler works, very roughly speaking, by using an inner protocol to simulate the protocol execution of an outer protocol. The outer protocol computes the desired functionality. The inner protocol protocols computes the individual computation steps of the outer protocol. The compiler is blackbox with respect to the inner, but not the outer protocol and it requires the existence of oblivious transfer. It is unclear whether the IPS compiler can be used to produce practically efficient protocols.

In contrast, our compiler does not require any computational assumption and thus preserves any information theoretic guarantees the underlying protocol has. Our transform does not have any large hidden constants and can produce actively secure protocols with efficiency that may be of practical interest.

In a recent work by Furukawa et al. [57], a practically very efficient three-party protocol with one active corruption was proposed. Their protocol uses replicated secret sharing and only works for bits. As the authors state themselves, it is not straightforward to generalize their protocol to more than three parties, while maintaining efficiency. In contrast, our protocol works over any arbitrary ring and can easily be generalized to any number of players. Furthermore our transform produces protocols with constant overhead, whereas their protocol does not have constant overhead.

The idea of using replication to detect active corruptions has been used before. For instance, Mohassel et al. [97] propose a three-party version of Yao's protocol. In a nutshell, their approach is to let two parties garble a circuit separately and to let the third party check that the circuits are the same. Our results in this work are more general in the sense that we propose a general transform to obtain actively secure protocols from passively secure ones. In [47], Desmedt and Kurosawa use replication to design a mix-net with t^2 servers secure against (roughly) t actively corrupted servers. A simple approach to MPC based on replicated secret sharing was proposed by Maurer in [95]. It has been the basis for practical implementations like [16].

2.5 SPD$\mathbb{Z}_{2^k}$: MPC mod 2^k for Dishonest Majority

In this section, based on [32], we describe the first practical MPC protocol for computations over the ring of integers modulo 2^k, with active security even when up to $n-1$ out of n participants are corrupted. At the core of our protocol is a scheme for information-theoretic MACs that are homomorphic modulo 2^k; constructing these has been an open problem since the BeDOZa protocol [14] from 2011.

Background on MACs. To obtain active security over fields, the main idea of modern protocols in the dishonest majority setting is to use unconditionally secure message authentication codes (MACs) to prevent players from lying about the data they are given in the preprocessing phase. A typical example is the SPDZ protocol [40, 35], where security reduces to the following game: we have a data value x, a random MAC key α and a MAC $m = \alpha x$, all in some finite field $\mathbb{F}$. The adversary is given x but not α or αx. He may now specify errors to be added to x, α and m, and we let x', α', m' be the resulting values. The adversary wins if $x \neq x'$ and $m' = \alpha' x'$. It is easy to see that the adversary must guess α to win, and so the probability of winning is $1/|\mathbb{F}|$. This authentication scheme is additively homomorphic, which is exploited heavily in the SPDZ protocol and is crucial for its efficiency.

However, the security proof depends on the fact that any non-zero value in $\mathbb{F}$ is invertible, and it is easy to see that if we replace the field by a ring, say $\mathbb{Z}_{2^k}$, then the adversary can cheat with large probability. For instance, in the ring $\mathbb{Z}_{2^k}$ he can choose $x' = x + 2^{k-1}$ and cheat with probability $1/2$. Up to now, it has been an open problem to design a homomorphic authentication scheme that would work over $\mathbb{Z}_{2^k}$ or more generally $\mathbb{Z}_M$ for any M, and is as efficient as the SPDZ scheme.

2.5.1 Our contributions

In this work we solve the above question: we design a new additively homomorphic authentication scheme that works in $\mathbb{Z}_{2^k}$[2], and is as efficient as the standard solution over a field. The main idea is to choose the MAC key α randomly in $\mathbb{Z}_{2^s}$, where s is the security parameter, and compute the MAC αx in $\mathbb{Z}_{2^{k+s}}$. We explain below why this helps. We also design a method for checking large batches of MACs with a communication complexity that does not depend on the size of the batch. We believe that these techniques will be of independent interest.

We then use the MAC scheme to design a SPDZ-style online protocol that securely computes an arithmetic circuit over $\mathbb{Z}_{2^k}$ with statistical security, assuming access to a preprocessing functionality that outputs multiplication triples in a suitable format. The total computational work done is dominated by $O(|C|n)$ elementary operations in the ring $\mathbb{Z}_{2^{k+s}}$, where C is the circuit to be computed. So if $k \geq s$, the work needed per player is equal to the work needed to compute C in the clear, up to a constant factor — as is the case for the SPDZ protocol. As in other protocols from this line of work, the overhead becomes more significant when k is small. Each player stores data from the preprocessing of size $O(|C|(k+s))$ bits. However, the communication complexity is $O(|C|k)$ bits plus an overhead that does not depend on C. This is due to the batch-checking of MACs mentioned above.

Our final result is a protocol for the preprocessing functionality to generate multiplication triples. It has communication complexity $O((k+s)^2)$ bits per multiplication gate, and is roughly as efficient as the MASCOT protocol [79], which is the state of the art for preprocessing over a field using oblivious transfer. Concretely, our triple generation protocol has around twice the communication cost of MASCOT, due to the overhead incurred when we have to work over larger rings in certain scenarios. However, this additional cost seems like a small price to pay for the potential benefits to applications from working modulo 2^k instead of in a field.

2.5.2 Overview of our techniques

For the authentication scheme, as mentioned, we have a data item $x \in \mathbb{Z}_{2^{k+s}}$, a key $\alpha \in \mathbb{Z}_{2^{k+s}}$ and we define the MAC as $m = \alpha x \mod 2^{k+s}$. Note that we want to authenticate k-bit values, so although $x \in \mathbb{Z}_{2^{k+s}}$, only the least significant k bits matter. The adversary is given x, and specifies errors e_x, e_α, e_m, which define modified values $x' = x + e_x, \alpha' = \alpha + e_\alpha, m' = m + e_m$. He wins if $m' = \alpha' x' \mod 2^{k+s}$, but note that since we store data in the least significant k bits only, this is only a forgery if $e_x \mod 2^k \neq 0$. As we show in detail in [32], if the adversary wins, he is able to compute $e_x \alpha \mod 2^{k+s}$. From this, and $e_x \mod 2^k \neq 0$, it follows that the adversary can effectively guess $\alpha \mod 2^s$, which is only possible with probability 2^{-s}.

We also want to batch-check many MACs using only a small amount of communication. The SPDZ protocol [40] uses a method that basically takes a random linear combination of all messages and MACs and checks only the resulting message and MAC. Unfortunately, applying the analysis we just sketched to this scenario does not give a negligible probability of cheating, unless we 'lift' again and compute MACs modulo 2^{k+2s}, but then our storage and preprocessing costs would become significantly bigger. We provide a more complicated but tighter analysis showing that we can still compute MACs mod 2^{k+s} and the batch checking works with $2^{-s+\log s}$ error probability, so we only need increase s by a few bits.

Using these MACs, we can create an information-theoretically secure MPC protocol over $\mathbb{Z}_{2^k}$ in the preprocessing model, similar to the online phase of SPDZ from [35]. To implement the preprocessing phase, we follow the style of MASCOT [79], which uses oblivious transfer to produce shares

[2] We use modulus 2^k throughout, but the scheme easily extends to any modulus.

of authenticated multiplication triples. We first design a protocol for authenticating values using correlated oblivious transfer, which allows creating the secret-shared MACs that will be added to the preprocessing data. This stage is similar to MASCOT, whereby first a passively secure protocol is used to compute shares of the MACs αx_i, for each value x_i that is to be authenticated, and then a random linear combination of these values is opened, and the resulting MAC checked for correctness. The main change we need to make here is that, depending on the size of the x_i's being authenticated, we may need to first compute the MACs over a larger ring in order to apply our analysis of taking random linear combinations.

Once the authentication scheme has been implemented, the main task is to create the multiplication triples needed in the online phase of our protocol. For this we also follow a similar approach to MASCOT, where the overall idea is that each party P_i chooses its shares (a^i, b^i) and then is engaged in an oblivious transfer subprotocol with P_j for each $j \neq i$, where shares of the cross products $a^i b^j$ and $a^j b^i$ are obtained. This yields shares of the product $(\sum_{i=1}^n a^i)(\sum_{j=1}^n b^j) = \sum_{i=1}^n a^i b^i + \sum_{i \neq j}(a^i b^j + a^j b^i)$, as required. Behind this simplification lies the problem that some information about the honest parties' shares can be leaked to a cheating adversary. In MASCOT this potential leakage is mitigated by "spreading out" the randomness by taking random linear combinations on correlated triples (with the same b value). When working over fields, the inner product yields a 2-universal hash function so the new distribution can be argued to be close to uniform using the Leftover Hash Lemma. However, this is not true anymore over rings like $\mathbb{Z}_{2^k}$. We overcome this issue by starting with triples where the shares of a are *bits* instead of ring elements, and then taking linear combinations over the bits. These combinations correspond to a subset sum over $\mathbb{Z}_{2^k}$, which *is* a 2-universal hash function, so allows for removing the leakage.

Additionally, random combinations are used in MASCOT to check the correctness of a triple by "sacrificing" another one. The security argument is that if the adversary manages to authenticate an incorrect triple, then it will have to guess the randomness used in the sacrifice step, which is unlikely. This is argued by deriving an equation from which we can solve for the random value. In order to extend this argument to the ring case, we use the technique sketched at the beginning of this section, working over $\mathbb{Z}_{2^{k+s}}$ to check correctness modulo 2^k.

Related work. There are only a few previous works that study MPC over rings, and none of these offer security against an active adversary who corrupts a dishonest majority of the parties. Cramer et al. showed how to contruct actively secure MPC over black-box rings [33] using secret-sharing techniques for honest majority, but this is only a feasibility result and the concrete efficiency is not clear. As already mentioned, Sharemind [16] allows mixing of secure computation over the integers modulo 2^k with boolean computations, but is restricted to the three-party setting when at most one party is corrupted.

3 Scaling up MPC to Many Parties

This section is based on a work which appeared at *CRYPTO 2018* by Hazay, Orsini, Scholl and Soria-Vazquez [70].

The efficiency of an MPC protocol typically depends highly on how many corrupted parties can be tolerated before security breaks down, a quantity known as the *threshold*, t. With semi-honest security, most protocols either require $t < n/2$, where n is the number of parties, in which case unconditionally secure protocols [12, 26] based on Shamir secret-sharing can be used, or support any choice of t up to $n-1$, as in computationally secure protocols based on oblivious transfer [65, 64]. Interestingly, within these two ranges, the efficiency of most practical semi-honest protocols *does not depend on* t. For instance, the GMW [65] protocol (and its many variants) is *full-threshold*, so supports any $t < n$ corruptions. However, we *do not know* of any practical protocols with threshold, say, $t = \frac{2}{3}n$, or even $t = n/2+1$, that are more efficient than full-threshold GMW-style protocols. One exception to this is when the number of parties becomes very large, in which case protocols based on *committees* can be used. In this approach, due to an idea of Bracha [20], first a random committee of size $n' \ll n$ is chosen. Then every party secret-shares its input to the parties in the committee, who runs a secure computation protocol for $t < n'$ to obtain the result. The committee size n' must be chosen to ensure that not the whole committee is corrupted, so clearly a lower threshold t allows for smaller committees, giving significant efficiency savings. However, this technique is only really useful when n is very large, at least in the hundreds or thousands.

In this work we investigate designing MPC protocols where *an arbitrary threshold for the number of corrupted parties can be chosen*, which are practical even when n is very large. Specifically, we ask the question:

> *Can we design* concretely efficient *MPC protocols where the performance improves gracefully as the number of honest parties increases?*

Note that the performance of an MPC protocol can be measured both in terms of *communication overhead* and *computational overhead*. Using fully homomorphic encryption [61], it is possible to achieve very low communication overhead that is independent of the circuit size [4] even in the malicious setting, but for reasonably complex functions FHE is impractical due to very high computational costs. On the other hand, practical MPC protocols typically communicate for every AND gate in the circuit, and use *oblivious transfer* (OT) to carry out the computation. Fast OT extension techniques allow a large number of secret-shared bit multiplications to be performed using only symmetric primitives and an amortized communication complexity of $O(\kappa)$ [72] or $O(\kappa/\log\kappa)$ [85, 48] bits, where κ is a computational security parameter. This leads to an overall communication complexity which grows with $O(n^2\kappa/\log\kappa)$ bits per AND gate in protocols based on secret-sharing following the [65] style, and $O(n^2\kappa)$ in those based on garbled circuits in the style of [119, 9, 10].

Short keys for secure computation. Our main idea towards achieving the above goal is to build a secure multi-party protocol with h honest parties, by distributing secret key material so that each party only holds a *small part of the key*. Instead of basing security on secret keys held by each party individually, we then base security on the *concatenation of all honest parties' keys*.

As a toy example, consider the following simple distributed encryption of a message m under n keys:

$$\mathsf{E}_k(m) = \bigoplus_{i=1}^{n} \mathsf{H}(i,k_i) \oplus m$$

where H is a suitable hash function and each key $k_i \in \{0,1\}^\ell$ belongs to party P_i. In the full-threshold setting with up to $n-1$ corruptions, to hide the message we need each party's key to be of length $\ell = 128$ to achieve 128-bit computational security. However, if only $t < n-1$ parties are corrupted, it seems that, intuitively, an adversary needs to guess all $h := n-t$ honest parties' keys to recover the message, and potentially each key k_i can be *much less than 128 bits* long when h is large enough. This is because the "obvious" way to try to guess m would be to brute force all h keys until decrypting "successfully".

In fact, recovering m when there are h unknown keys corresponds to solving an instance of the *regular syndrome decoding problem* [6], which is related to the well-known *learning parity with noise* (LPN) problem, and believed to be hard for suitable choices of parameters.

3.1 Our Contribution

In this work we use the above idea of short secret keys to design new MPC protocols in both the constant round and non-constant round settings, which improve in efficiency as the number of honest parties increases. We consider security against a static, honest-but-curious adversary, and leave it for future work to extend our techniques to the malicious case based on, e.g. message authentication codes. Our contribution is captured by the following:

1. We present a GMW-style MPC protocol for binary circuits, where multiplications are done with OT extension using short symmetric keys. This reduces the communication complexity of OT extension-based GMW from $O(n^2\kappa/\log\kappa)$ [85] to $O(nt\ell)$, where the key length ℓ decreases as the number of honest parties, $h = n-t$, increases. When h is large enough, we can even have ℓ *as small as* 1.

   To construct this protocol, we first analyse the security of the IKNP OT extension protocol [72] when using short keys, and formalise the leakage obtained by a corrupt receiver in this case. We then show how to use this version of "leaky OT" to generate multiplication triples using a modified version of the GMW method, where pairs of parties use OT to multiply their shares of random values. We also optimize our protocol by reducing the number of communication channels using two different-sized committees, improving upon the standard approach of choosing one committee to do all the work.

2. Our second contribution is the design of a constant round, BMR-style [9] protocol based on garbled circuits with short keys. Our offline phase uses the multiplication protocol from the previous result in order to generate the garbled circuit, using secret-shared bit and bit/string multiplications as done in previous works [10, 71], with the exception that the keys are shorter. In the online phase, we then use the LPN-style assumption to show that the combination of all honest parties' ℓ-bit keys suffices to obtain a secure garbling protocol. This allows us to save on the key length as a function of the number of honest parties.

   As well as reducing communication with a smaller garbled circuit, we also reduce computation when evaluating the circuit, since each garbled gate can be evaluated with only $O(n^2\ell/\kappa)$ block cipher calls (assuming the ideal cipher model), instead of $O(n^2)$ when using κ-bit keys. For this protocol, ℓ *can be as small as 5* when n is large enough, giving a significant saving over 128-bit keys used previously.

Concrete Efficiency Improvements. The efficiency of our protocols depends on the total number of parties, n, and the number of honest parties, h, so there is a large range of parameters to explore when comparing with other works. We discuss this in more detail in [70]. Our protocols seem most significant in the *dishonest majority* setting, since when there is an honest majority there are unconditionally secure protocols with $O(n \log n)$ communication overhead and reasonable computational complexity e.g. [37], whilst our protocols have $\Omega(nt)$ communication overhead.

Our GMW-style protocol starts to improve upon previous protocols when we reach $n = 20$ parties and $t = 14$ corruptions: here, our triple generation method requires less than *half the communication cost* of the fastest GMW-style protocol based on OT extension [48] tolerating up to $n-1$ corruptions. When the number of honest parties is large enough, we can use *1-bit keys*, giving a *25-fold reduction* in communication over previous protocols when $n = 400$ and $t = 280$. In addition, we describe a simple threshold-t variant of GMW-style protocols, which our protocol still outperforms by 1.1x and 13x, respectively, in these two scenarios.

For our constant round protocol, with $n = 20, t = 10$ we can use 32-bit keys, so the size of each garbled AND gate is $1/4$ the size of [10]. As n increases the improvements become greater, with a *16-fold reduction* in garbled AND gate size for $n = 400$ and $t = 280$. We also reduce the communication cost of *creating* the garbled circuit. Here, the improvement starts at around 50 parties, and goes up to a 7 times reduction in communication when $n = 400$ and $t = 280$. Note that our protocol does incur a slight additional overhead, since we need to use extra "splitter gates", but this cost is relatively small.

To demonstrate the practicality of our approach, in the full paper [70] we also present an implementation of the online evaluation phase of our constant-round protocol for key lengths ranging between $1-4$ bytes, and with an overall number of parties ranging from $15-1000$.

Applications. Our techniques seem most useful for large-scale MPC with around 70% corruptions, where we obtain the greatest concrete efficiency improvements. An important motivation for this setting is privacy-preserving statistical analysis of data collected from a large network with potentially thousands of nodes. In scenarios where the nodes are not always online and connected, our protocols can also be used with the "random committee" approach discussed earlier, so only a small subset of, say, a hundred nodes need to be online and interacting during the protocol.

An interesting example is safely measuring the Tor network which is among the most popular tools for digital privacy, consisting of more than 6000 relays that can opt-in for providing statistics about the use of the network. Nowadays and due to privacy risks, the statistics collected over Tor are generally poor: There is a reduced list of computed functions and only a minority of the relays provide data, which has to be obfuscated before publishing. Hence, the statistics provide an incomplete picture which is affected by a noise that scales with the number of relays. Running MPC in this setting would enable for more complex, accurate and private data processing, for example through anomaly detection and more sophisticated censorship detection. Moreover, our protocols are particularly well-suited to this setting since all relays in the network must be connected to one another already, by design.

Another possible application is for securely computing the interdomain routing within the Border Gateway Protocol (BGP), which is performed at a large scale of thousands of nodes. A recent solution in the dishonest majority setting [3] centralizes BGP so that two parties run this computation for all Autonomous Systems. Our techniques allow scaling to a large number of systems computing the interdomain routing themselves using MPC, hence further reducing the trust requirements.

Decisional Regular Syndrome Decoding problem. The security of our protocols relies on the *Decisional Regular Syndrome Decoding (DRSD)* problem, which, given a random binary matrix H, is to distinguish between the syndrome obtained by multiplying H with an error vector $e = (e_1, \cdots, e_h)$ where each $e_i \in \{0,1\}^{2^\ell}$ has Hamming weight one, and the uniform distribution. This can equivalently be described as distinguishing $\bigoplus_{i=1}^{h} \mathsf{H}(i, k_i)$ from the uniform distribution, where H is a random function and each k_i is a random ℓ-bit key (as in the toy example described earlier).

We remark that when h is large enough, the problem is *unconditionally hard* even for $\ell = 1$, which means for certain parameter choices in our GMW-based protocol we can use 1-bit keys *without introducing any additional assumptions*. This introduces a significant saving in our triple generation protocol.

Overall, our approach demonstrates a new application of LPN-type assumptions to efficient MPC without introducing asymmetric operations. Our techniques may also be useful in other distributed applications where only a small fraction of nodes are honest.

3.2 Technical Overview

In what follows we explain the technical side of our results in more detail.

Leaky oblivious transfer (OT). We first present a two-party secret-shared bit multiplication protocol, based on a variant of the IKNP OT extension protocol [72] with short keys. Our protocol performs a batch of r multiplications at once. Namely, the parties create r correlated OTs on ℓ-bit strings using the OT extension technique of [72], by transposing a matrix of ℓ OTs on r-bit strings and swapping the roles of sender and receiver. In contrast to the IKNP OT extension and followups, that use κ 'base' OTs for computational security parameter κ, we use $\ell = O(\log \kappa)$ base OTs.

This protocol leaks some information on the global secret $\Delta \leftarrow \{0,1\}^\ell$ picked by the receiver, as well as the inputs of the receiver. Roughly speaking, the leakage is of the form $\mathsf{H}(i, \Delta) + x_i$, where $x_i \in \{0,1\}$ is an input of the receiver and H is a hash function with 1-bit output. Clearly, when ℓ is short this is not secure to use on its own, since all of the receiver's inputs only have ℓ bits of min-entropy (based on the choice of Δ).

MPC from leaky OT. We then show how to apply this leaky two-party protocol to the multi-party setting, whilst preventing any leakage on the parties shares. The main observation is that, when using additive secret-sharing, we only need to ensure that the *sum* of all honest parties' shares is unpredictable; if the adversary learns just a few shares, they can easily be rerandomized by adding pseudorandom shares of zero, which can be done non-interactively using a PRF. However, we still have a problem, which is that in the standard GMW approach, each party P_i uses OT to multiply their share x^i with every other party P_j's share y^j. Now, there is leakage on the *same share* x^i from each of the OT instances between all other parties, which seems much harder to prevent than leakage from just a single OT instance.

To work around this problem, we have the parties add shares of zero to their x^i inputs *before* multiplying them. So, every pair (P_i, P_j) will use leaky OT to multiply $x^i \oplus s^{i,j}$ with y^j, where $s^{i,j}$ is a random share of zero satisfying $\bigoplus_{i=1}^{n} s^{i,j} = 0$. This preserves correctness of the protocol, because the parties end up computing an additive sharing of:

$$\bigoplus_{i=1}^{n}\bigoplus_{j=1}^{n}(x^i \oplus s^{i,j})y^j = \bigoplus_{j=1}^{n} y^j \bigoplus_{i=1}^{n}(x^i \oplus s^{i,j}) = xy.$$

This also effectively removes leakage on the individual shares, so we only need to be concerned with the *sum* of the leakage on all honest parties' shares, and this turns out to be of the form: $\bigoplus_{i=1}^{n}(\mathsf{H}(i,\Delta_i)+x^i)$ which is pseudorandom under the decisional regular syndrome decoding assumption.

We realize our protocol using a hash function with a polynomial-sized domain, so that is can be implemented using a CRS which simply outputs a random lookup-table. This means that, unlike when using the IKNP protocol, we do not need to rely on a random oracle or a correlation robustness assumption.

When the number of parties is large enough, we can improve our triple generation protocol using *random committees*. In this case the amortized communication cost is $\leq n_h n_1(\ell+\ell\kappa/r+1)$ bits per multiplication where we need to choose two committees of sizes n_h and n_1 which have at least h and 1 honest parties, respectively.

Garbled circuits with short keys. We next revisit the multi-party garbled circuits technique by Beaver, Micali and Rogaway, known as BMR, that extends the classic Yao garbling [119] to an arbitrary number of parties, where essentially all the parties jointly garble using one set of keys each. This method was recently improved in a sequence of works [92, 93, 10, 71], where the two latter works further support the Free-XOR property.

Our garbling method uses an expansion function $\mathsf{H}:[n]\times\{0,1\}\times\{0,1\}^{\ell}\rightarrow\{0,1\}^{n\ell+1}$, where ℓ is the length of each parties' keys used as wire labels in the garbled circuit. To garble a gate, the hash values of the input wire keys $k^i_{u,b}$ and $k^i_{v,b}$ are XORed over i and used to mask the output wire keys.

Specifically, for an AND gate g with input wires u,v and output wire w, the 4 garbled rows $\tilde{g}_{a,b}$, for each $(a,b)\in\{0,1\}^2$, are computed as:

$$\tilde{g}_{a,b}=\left(\bigoplus_{i=1}^{n}\mathsf{H}(i,b,k^i_{u,a})\oplus\mathsf{H}(i,a,k^i_{v,b})\right)\oplus(c,k^1_{w,c},\ldots,k^n_{w,c}).$$

Security then relies on the DRSD assumption, which implies that the sum of h hash values on short keys is pseudorandom, which suffices to construct a secure garbling method with h honest parties.

Using this assumption instead of a PRF (as in recent works) comes with difficulties, as we can no longer garble gates with arbitrary fan-out, or use the free-XOR technique, without degrading the DRSD parameters. To allow for arbitrary fan-out circuits with our protocol we use *splitter gates*, which take as input one wire w and provide two outputs wires u,v, representing the same wire value. Splitter gates were previously introduced as a fix for an error in the original BMR paper in [114]. We stress that transforming a general circuit description into a circuit with only fan-out-1 gates requires adding at most a single splitter gate per AND or XOR gate.

The restriction to fan-out-1 gates and the use of splitter gates additionally allows us to garble XOR gates for free in BMR without relying on circular security assumptions or correlation-robust hash functions, based on the FlexOR technique [86] where each XOR gate uses a unique offset. Furthermore, the overhead of splitter gates is very low, since garbling a splitter gate does not use the underlying MPC protocol: shares of the garbled gate can be generated non-interactively. We note that this observation also applies to Yao's garbled circuits, but the overhead of adding splitter gates there is more significant; this is because in most 2-party protocols, the *size* of the garbled circuit is the dominant cost factor, whereas in multi-party protocols the main cost is *creating* the garbled circuit in a distributed manner.

4 Communication-Efficient Honest-Majority MPC from Batch-Wise Multiplication Verification

Recently, there has been a lot of interest in concretely efficient actively secure MPC in the honest-majority setting with abort, in which fewer than $n/2$ out of n parties may be corrupted. In this setting, very efficient solutions are known and it is also possible to achieve *fairness*, i.e., either all parties learn the result or none do, which is not possible without a honest majority.

A number of recent works have achieved particularly striking performance numbers in this setting. Binary circuits can be evaluated at a cost of sending 10 bits per AND gate for three parties due to Furukawa *et. al* [58], and arithmetic circuits can be evaluated at a cost of sending 4 (for $n = 3$), $5(n-1)$, or 42 field elements per multiplication due to Lindell and Nof [90]. However, this still leaves at least a factor four communication increase compared to passive security. Moreover, these best known protocols unfortunately do not satisfy fairness (unlike other honest-majority protocols).

4.1 Contribution

In this work, we improve on the state-of-the-art of concretely efficient honest-majority MPC by further decreasing communication complexity, while also supporting fairness.

First, we improve on all the main three concrete instantiations of Lindell and Nof's protocol framework [90], based on the multiplication protocols of Gennaro *et al.* [60], Damgård and Nielsen [37] and Araki *et. al* [2] (we denote these variants GRR, DN and AFL+ respectively). Concerning communication complexity, we decrease communication in these variants by factors of approximately 2, 5, and 7, respectively. In all cases, the gap between passive and active security becomes only a factor 2. Moreover, in the three-party setting, the best protocol now requires sending just two messages per party per multiplication. In Table 2 we give a more specific overview of our results compared to the three original instantiations of Lindell and Nof. Some of this improvement comes from better utilization of pseudo random secret sharing; but the most significant improvement comes from applying the tool of *batch-wise multiplication verification* introduced by Ben-Sasson *et. al* [13], a technique that allows to check that many multiplications have been performed correctly by essentially checking a single multiplication.

Operation	**GRR**	**GRR-PRNG**	**DN**	**DN-PRNG**	**AFL+**
Random value	0	0	$\lesssim 2$	$\lesssim 1$	0
Opening	$n-1$	$n-1$	$n-1$	$n-1$	1
Passive mult.	$n-1$	$n-t-1$	$\lesssim 6$	$\lesssim 3$	1
LN mul + check	$5(n-1)$	$6(n-t-1)$	$\lesssim 42$	$\lesssim 18$	4
Batch mul + check	$\gtrsim 2(n-1)$	$\gtrsim 2(n-t-1)$	$\gtrsim 12$	$\gtrsim 6$	$\gtrsim 2$

Table 2: Field elements sent per party for the Lindell-Nof protocol instantiated with GRR, DN (both with or without PRNG optimizations) and AFL+ (with PRNG optimization). The number of parties and the threshold is denoted by n and t respectively (generally $n \approx 2t$). Grey areas are our results

We additionally provide a novel three-party protocol, based on the SPDZ protocol Damgård *et. al* [40], that reduces *online* communication from 2 in our protocol described above to $\frac{4}{3}$ messages per party per multiplication. This comes at the expense of requiring a *preprocessing* phase with $\frac{5}{3}$ messages per party per multiplication. Our SPDZ-based protocol also makes heavy use of PRNGs and batch-wise multiplication verification, but additionally incorporates the idea of taking a two-party

protocol in the preprocessing model, and replacing the distributed preprocessing protocol by in-the-plain preprocessing by a third party. This idea was known before but, as far as we know, has never been applied; we extend this idea by allowing the preprocessing to be spread evenly between the three parties. By way of a rough comparison, in the two-party dishonest majority setting, a recent SPDZ variant by Keller *et. al* [80] requires the equivalent of around 130 field elements to be sent per party, highlighting the communication gap between the honest- and dishonest-majority settings.

In both our Lindell-Nof and our SPDZ based protocol, the decrease in communication cost implies an increase in computation cost, but we show that in many practical settings, communication is still the bottleneck.

Finally, we show how to add fairness both of our constructions. We employ general principles to achieve fairness such as using signature-based broadcast for agreement and MACs or signatures to prevent output manipulation. Our solutions are especially crafted to ensure that they add as little practical overhead as possible; in particular, they do not affect the above communication complexity results. This means that communication-efficient, actively secure MPC is possible in practice without having to sacrifice fairness.

4.2 Overview of Techniques

As mentioned above the main tool to improve communication complexity in both our Lindell-Nof and SPDZ based protocols is the use of a sub protocol we denote *batch-wise multiplication verification*. This protocol is used to efficiently check that batches of secret shared multiplication triples were correctly shared.

Batch-wise multiplication verification was introduced by Ben-Sasson *et. al* [13] to improve the asymptotic complexity of verifying preprocessed multiplication triples over small fields. Standard multiplications checks, e.g. based on sacrificing, scale with the security parameter (which is larger than the field size), but using batch-wise multiplication verification, these costs can be spread over a batch.

In particular, given secret-shared values $[a_1]$, ..., $[a_N]$, $[b_1]$, ..., $[b_N]$, $[c_1]$, ..., $[c_N]$, the goal is to verify that $c_i = a_i \cdot b_i$ for all i. This is done by translating these N equalities of field elements into a single equality of polynomials, and verifying this equality based on the Schwartz-Zippel lemma [120, 110]. Fix nonzero $\omega_1, \ldots, \omega_{2N-1}$, and let $A(x), B(x)$ be of degree $\leq N-1$ such that for $i \in [1,N]$, $A(\omega_i) = a_i$ and $B(\omega_i) = b_i$. If we let $C(x) = A(x)B(x)$, then obviously $C(\omega_i) = c_i$ for $i \in [1,N]$, but the converse is also true: if there exists a polynomial $C(x)$ of degree $\leq 2N-1$ such that $C(x) = A(x)B(x)$ and $C(\omega_i) = c_i$ for $i \in [1,N]$, this implies $c_i = a_i \cdot b_i$.

In batch-wise multiplication verification, first, $C(x)$ is constructed by computing $C(\omega_j) = A(\omega_j) \cdot B(\omega_j)$, $j \in [N+1, 2N-1]$ using passively secure MPC and deriving its coefficients by interpolation. Then, A, B, and C are evaluated in a random point $s \notin \{\omega_1, \ldots, \omega_{2N-1}\}$. This can be done with local linear operations given shares of the a_i, b_i, c_i, and $C(\omega_j)$. Finally, a multiplication check protocol is run to check that $A(s) \cdot B(s) = C(s)$. The Schwartz-Zippel lemma, states that for a non-zero degree d polynomial, P, over field $\mathbb{F}$ of and a random $r \in S$ for a finite $S \subseteq \mathbb{F}$ the probability that $P(r) = 0$ is at most $d/|S|$. Thus if $A(s) \cdot B(s) = C(s)$ then with high probability, $A(x) \cdot B(x) = C(x)$ as polynomials and hence $a_i \cdot b_i = c_i$. Note that for each triple, an additional passively secure multiplication is needed, but the multiplication check is performed only once per batch, giving the asymptotic advantage.

Our version of batch-wise multiplication verification protocol follows the basic idea of Ben-Sasson *et. al* [13], but avoids its actively secure $A(s) \cdot B(s) = C(s)$ check. We add a random multiplication triple (a_N, b_N, c_N) to the batch of triples and choose s uniformly at random from $\mathbb{Z}_p$. Then, the values of $A(s), B(s), C(s)$ are uniformly random and can be opened so that the check $A(s) \cdot B(s) = C(s)$

can be performed in the plain. Note that this option was not available to Ben-Sasson *et. al* since they need s from an extension field so $A(s), B(s), C(s)$ are not uniform.

In Lindell and Nof's protocol framework they show that for many secret sharing based MPC protocols that are *passively* secure with an honest majority, we can get *active* security, essentially, by checking the correctness of all multiplications at the end of the protocol. We get our Lindell-Nof based protocol by showing that batch-wise multiplication verification can be used as a replacement for the methods originally proposed by Lindell and Nof.

In our SPDZ based protocol we start from the idea that, in two-party SPDZ, we can replace the expensive, actively secure preprocessing phase used to generate multiplication triples, by a trusted third party dealer. This trusted dealer would simply generate the triples locally and then secret share them between the two parties. Using the batch-wise multiplication verification protocol to check that the dealer does this correctly, the need for trust in the dealer is then removed and we get a actively secure three-party protocol in the honest majority setting.

4.3 Related Work

Several recent works are closely related to this work. Concerning efficient honest-majority MPC, the most relevant work is the framework for communication-efficient MPC of Lindell and Nof [90] that forms the basis of our first protocol. It is also the closest competitor in terms of overall communication complexity that we are aware of. Another recent honest-majority MPC framework is our other work described in Section 2.1. Although that construction is quite a bit less communication-efficient than the one presented in this section, it does work for arbitrary rings as opposed to just fields.

Concerning the technique of batch-wise multiplication verification, the groundwork was laid out in several earlier works. Ben-Sasson *et al.* [13] first proposed batch-wise multiplication verification. As discussed below, there it was used to get an asymptotic result; we are not aware of works using it to improve practical performance. Works such as the Pinocchio verifiable computation system by Parno *et al.* [102] and the Trinocchio protocol, of Schoenmakers *et. al* [109], that combines it with MPC were a main inspiration to start seeing batch-wise multiplication verification also as a tool that may deliver practical efficiency. Corrigan-Gibbs and Boneh [30] first proposed to use batch-wise multiplication verification where one party provides data and a number of other parties verify it, as in our SPDZ-based protocol; but there it is not for performing the MPC but for checking its inputs.

5 Maliciously Secure Oblivious Linear Function Evaluation with Constant Overhead

This section is based on work published at Asiacrypt 2017 by Ghosh, Nielsen and Nilges [62].

Oblivious evaluation of functions is an essential building block in cryptographic protocols. The first result in the area is a protocol for oblivious transfer (OT), which was introduced in the seminal work of Rabin [104]. Here, a sender can specify two bits s_0, s_1, and a receiver can learn one of the bits s_b depending on his choice bit b. It is guaranteed that the sender does not learn b, while the receiver learns nothing about s_{1-b}. Kilian [82] subsequently showed that OT is in fact already complete, which means that it allows the (oblivious) evaluation of *any* function. The work on OT spawned the field of multiparty computation (MPC), which considers the generalized case of several parties obliviously evaluating generic circuits.

While there has been tremendous progress in the area of generic MPC over the last three decades, there are certain classes of functions that can be evaluated more efficiently by direct constructions instead of taking the detour via MPC. In this context, Naor and Pinkas [98] introduced oblivious polynomial evaluation (OPE) as an useful primitive. OPE deals with the problem of evaluating a polynomial P on an input α obliviously, i.e., the sender specifies the polynomial P but does not learn α, while the receiver learns $P(\alpha)$ but nothing else about P.

A special case of OPE, called oblivious linear function evaluation (OLE, sometimes also referred to as OLFE, or OAFE for affine functions) has been considered, in particular due to potential applications in MPC protocols for arithmetic circuits. Instead of evaluating an arbitrary polynomial P, the receiver wants to evaluate a linear or affine function $f(x) = ax + b$. Ishai et al. [75] propose a passively secure protocol for oblivious multiplication which uses a similar approach as [98], and can be easily modified to give a passively secure OLE. Based on stateful tamper-proof hardware [77] as a setup assumption, [50] build an unconditionally UC-secure protocol for OAFE.

Currently, all of the above mentioned actively secure realizations of OPE or OLE require rather expensive computations or strong setup assumptions. In contrast, the most efficient passively secure constructions built from noisy encodings and OT require only simple field operations. However, to date a direct construction of a maliciously secure protocol in this setting has been elusive. One approach to achieve this would be to apply a compiler like [74] to the passively secure protocols, which can result in an actively secure protocol with a constant overhead compared to the passively secure protocol. But such a transformation typically incurs a large constant, resulting in efficiency only in an asymptotic sense. Thus, the most efficient realizations possibly follow from applying the techniques used for the precomputation of multiplied values in arithmetic MPC protocols such as SPDZ [40] or MASCOT [79].

5.1 Our Contribution

In [62] we present a UC-secure protocol for oblivious linear function evaluation in the OT-hybrid model based on noisy encodings. The protocol is based on the semi-honest secure implementation of OLE by Ishai et al. [75], which is the most efficient protocol for passively secure OLE that we are aware of. Our actively secure protocol only has a constant overhead of 2 compared to the passively secure original construction. In numbers, this means:

- We need 16 OTs per OLE, compared to 8 for the semi-honest protocol [75].
- We communicate $16 \cdot (2 + c_{\mathsf{OT}}) \cdot n + 8$ field elements for n multiplications, compared to $8 \cdot (2 + c_{\mathsf{OT}}) \cdot n$ in [75], where c_{OT} is the cost for one OT.

	Assumption	OTs	Expon.	Security
[25]	OT	$O(d\kappa)$	0	passive
[99]	OT & Noisy Encodings	$O(d\kappa \log \kappa)$	0	passive
[75]	OT & Noisy Encodings	$O(d)$	0	passive
[69]	CRS & DCRP	0	$O(ds)$	UC
[68]	DDH	0	$O(d)^*$	active
This work	**OT & Noisy Encodings**	$O(d)$	0	**UC**

Table 3: Overview of OPE realizations, where d is the degree of the polynomial, κ is a computational security parameter and s a statistial security parameter ([69] propose $s \approx 160$). We compare the number of OTs and exponentiations in the respective protocols.

- The computational overhead is twice that of the semi-honest case.

One nice property of [75] and the main reason for its efficiency is that it directly allows to multiply a batch of values. This property is preserved in our construction, i.e., we can simultaneously evaluate several linear functions.

In order to achieve our result we solve the long standing open problem of finding an actively secure OLE/OPE protocol which can directly be reduced to the security of noisy encodings (and OT). This problem was not solved in [98] and has only been touched upon in follow-up work [75, 81]. The key technical contribution of the paper is a reduction which shows that noisy encodings are robust against leakage in a strong sense, which allows their application in a malicious setting. As a matter of fact, our robustness results are more general and extend to *all* noisy encodings.

An immediate application of our UC-secure batch-OLE construction is a UC-secure OPE construction. The construction is very simple and has basically no overhead over the OLE construction. We follow the approach taken in [99], i.e., we use the fact that a polynomial of degree d can be decomposed into d linear functions. Such a decomposed polynomial is evaluated with the batch-OLE and then reconstructed. UC-security against the sender directly follows from the UC-security of the batch-OLE. In order to make the protocol secure against a cheating receiver, we only have to add one additional check that ensures that the receiver chooses the same input for each linear function. Table 3 compares the efficiency of our result with existing solutions in the literature.

We point out that [68] only realizes OPE in the exponent, which is still sufficient for many applications, but requires additional work to yield a full fledged OPE. In particular, this might entail additional expensive operations. Another important factor regarding the efficiency of OT-based protocols is the cheap extendability of OT [72, 78] which shows that the asymptotic price of OT is only small constant number of applications of a symmetric primitive like for instance AES. Therefore, the concrete cost of the OTs is much less than the price of exponentiations if d is sufficiently large, or if several OPEs have to be carried out. In such a scenario, we get significant improvements over previous (actively secure) solutions, which always require expensive operations in the degree of the polynomial.

5.2 Technical Overview

At the heart of our constructions are noisy encodings. These were introduced by Naor and Pinkas [98] in their paper on OPE and provide a very efficient means to obliviously compute multiplications. A noisy encoding is basically an encoding of a message via a linear code that is mixed with random values in such a way that the resulting vector hides which elements belong to the codeword and which

elements are random, thereby hiding the initial message. In a little more detail, the input $\mathbf{x} \in \mathbf{F}^t$, for a field $\mathbf{F}$, is used as t sampling points on locations α_i of an otherwise random polynomial P of some degree $d > t$. Then the polynomial is evaluated at e.g., $4d$ positions β_i, and half of these positions are replaced by uniformly random values, resulting in the encoding $\mathbf{v}$. It is assumed that this encoding is indistinguishable from a uniformly random vector.[3]

Robustness of noisy encodings The main problem of using noisy encodings in maliciously secure protocols is that the encoding is typically used in a non-black-box way. On one hand this allows for very efficient protocols, but on the other hand a malicious party obtains knowledge that renders the assumption that is made on the indistinguishability of noisy encodings useless. In a little more detail, consider a situation where the adversary obtains the encoding and manipulates it in a way that is not specified by the protocol. The honest party only obtains part of the encoding (this is usually necessary even in the passively secure case). In order to achieve active security, a check is performed which is supposed to catch a deviating adversary. But since the check is dependent on which part of the encoding the honest party learned, this check actually leaks some non-trivial information to the adversary, typically noisy positions of the codeword.

We show that noisy encodings as defined by [99, 75] are very robust with respect to leakage. In particular, we show the following theorem that is basically a stronger version of a result previously obtained by Kiayias and Yung [81].

Theorem 5.1 *(informal) For appropriate choices of parameters, noisy encodings are resilient against non-adaptive leakage of $O(\log \kappa)$ noisy positions.*

In a little more detail, we show that a noisy encoding generated as described above remains indistinguishable from a random vector of field elements, even for an adversary that is allowed to *fix* the position of f noisy positions. Fixing f positions is of course stronger than being able to leak f positions. The security loss incurred by the fixing of f positions is 3^f.

We then show that an adversary which is given a noisy encoding cannot identify a super-logarithmic sized set consisting of only noisy positions.

Theorem 5.2 *(informal) For appropriate choices of parameters, an adversary cannot identify more than $O(\log \kappa)$ noisy positions in a noisy encoding.*

These theorems together show that we can tolerate the leakage of any number of noisy positions that might be guessed. This is the basis for the security of our protocol. Note that tolerance to leakage of a set of noisy positions that might be guessed is not trivial, as we are working with an indistinguishability notion. Hence leakage of a single bit might *a priori* break the assumption. However in [62] we show that noisy encodings are robust against these kind of leakage.

Efficient OLE from noisy encodings. We build a UC-secure OLE protocol inspired by the passively secure multiplication protocol of Ishai et al. [75]. Let us briefly recall their construction on an intuitive level. One party, let us call it the sender, has as input t values $a_1, \ldots, a_t \in \mathbf{F}$, while the receiver has an input $b_1, \ldots, b_t \in \mathbf{F}$. A set of distinct points $\alpha_1, \ldots, \alpha_{n/4}$ is fixed. The high-level idea is as follows: both sender and receiver interpolate a degree $n/4 - 1$ polynomial through the points (α_i, a_i) and (α_i, b_i) (picking a_i, b_i for $i > t$ randomly), to obtain $A(x)$ and $B(x)$, respectively. They also

[3] The problem is related to efficient polynomial reconstruction, i.e., decoding Reed-Solomon codes, and as such well researched. The parameters have to be chosen in such a way that all known decoding algorithms fail.

agree on n points $\beta_1, \ldots, \beta_n$. Now the receiver replaces half of the points $B(\beta_i)$ with uniformly random values (he creates a noisy encoding) and sends these n values $\bar{B}(\beta_i)$ to the sender. He keeps track of the noiseless positions using an index set L. The sender draws an additional random polynomial R of degree $2(n/4-1)$ to mask the output. He then computes $Y(\beta_i) = A(\beta_i) \cdot \bar{B}(\beta_i) + R(\beta_i)$ and uses these points as input into a $n/2-1$-out-of-n OT, from which the receiver chooses the $n/2-1$ values in L. He can then interpolate the obtained points of $Y(\beta_i)$ to reconstruct Y and learn $a_i \cdot b_i + r_i$ in the positions α_i. This also directly yields an OLE: the polynomial R is simply used as another input of the sender, since it can be generated identically to A.

The passive security for the sender follows from the fact that the receiver obtains only $n/2-1$ values and thus R completely masks the inputs $a_1, \ldots, a_t$. Passive security of the receiver follows from the noisy encoding, i.e., the sender cannot learn B from the noisy encoding.

In order to achieve actively secure OLE from the above protocol, we have to ensure several things: first of all, we need to use an actively secure k-out-of-n OT. But instead of using a black-box realization, which incurs an overhead of $n \log n$ on the number of OTs, we use n 1-out-of-2 OTs and ensure that the right number of messages was picked via a secret sharing, which the receiver has to reconstruct. This protocol first appeared in [111]. It does not have active security against the sender, who can guess some choice bits. A less efficient but active secure version was later given in [44], using verifiable secret sharing. We can, however, use the more efficient but less secure original variant as we can tolerate leakage of a few choice bits in the overall protocol.

Secondly, we also need to make sure that the parties used the right inputs in the computation, i.e., valid polynomials A, B and R. In order to catch deviations, we add two checks—one in each direction. The check is fairly simple: one party selects a random point z and the other party sends a pair $A(z), R(z)$, or $B(z), Y(z)$ respectively. Each party can now locally verify that the values satisfy the equation $A(z) \cdot B(z) + R(z) = Y(z)$.

As it turns out, both of these additions to the protocol, while ensuring protocol compliance w.r.t. the inputs, are dependent on the encoding. But this also means that a malicious sender can do selective failure attacks, e.g., it inputs incorrect shares for the secret sharing, and gets some leakage on the "secret key" of the encoding. This problem does not occur when considering semi-honest security.

5.3 Noisy Encodings

The security of our protocols is based on a noisy encoding assumption. Very briefly, a noisy encoding is an encoding of a message, e.g., via a linear code, that is mixed with random field elements. It is assumed that such a codeword, and in particular the encoded message, cannot be recovered. This assumption seems reasonable due to its close relationship to decoding random linear codes or the efficient decoding of Reed-Solomon codes with a large fraction of random noise.

Noisy encodings were first introduced by Naor and Pinkas [98], specifically for the purpose of realizing OPE. Their encoding algorithm basically generates a random polynomial P of degree $k-1$ with $P(0) = x$. The polynomial is evaluated at $n > k$ locations, and then $n-k$ positions are randomized. Generalizing the approach of [99], Ishai et al. [75] proposed a more efficient encoding procedure that allows to encode several field elements at once instead of a single element, using techniques of [53]. Basically, they use Reed-Solomon codes and then artificially blind the codeword with random errors in order to mask the location of the codeword elements in the resulting string.

The encoding procedure depicted in Figure 1 is nearly identical to the procedure given in [75], apart from the fact that we do not fix the signal-to-noise ratio (because this will be dependent on the protocol). We also allow to pass a set of points $\mathscr{P}$ as an argument to Encode to simplify the description of our protocol later on. This change has no impact on the assumption, since these points

are made public anyway via G.

$\mathsf{Encode}_{n,\rho}(\mathbf{x}, \mathscr{P})$

Generator $\mathscr{G}(n,\rho)$ for Reed-Solomon code: Let the output be (G,H,L).

- Let $\ell = n - \rho$ and $k = \frac{\ell-1}{2} + 1$.
- Let $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_n$ be the points in $\mathscr{P}$. If $\mathscr{P} = \emptyset$, pick these points randomly from $\mathbf{F}$.
- Define the $n \times k$ matrix G such that for any $\mathbf{u} \in \mathbf{F}^k$, $(G\mathbf{u})_i = P(\beta_i)$ for $i = 1, \ldots, n$, where P is the unique degree $k-1$ polynomial such that $P(\alpha_i) = u_i$ for $i = 1, \ldots, k$.
- Pick $L \subset [n]$ with $|L| = \ell$ uniformly at random.
- Let H be the $k \times 2k-1$ matrix such that $(H\mathbf{v}_{|L})_i = Q(\alpha_i)$, where Q is the unique degree $2(k-1)$ polynomial such that $Q(\beta_j) = \mathbf{v}_j$ for all $j \in L$.

Encoding: Let the private output be $(G,H,L,\mathbf{v})$ and the public output be $(G,\mathbf{v})$.

- Let $(G,H,L) \leftarrow \mathscr{G}(n,\rho)$.
- Pick a random $\mathbf{u} \in \mathbf{F}^k$ conditioned on $u_i = x_i$ for $i = 1, \ldots, t$. Compute $G\mathbf{u} \in \mathbf{F}^n$.
- Pick a random vector $v \in \mathbf{F}^n$, conditioned on $v_i = (G\mathbf{u})_i$ for $i \in L$. Output $\mathbf{v}$.

Figure 1: Encoding procedure for noisy encodings.

[98] propose two different encodings and related assumptions, tailored to their protocols. One of these assumptions was later broken by [15] and [17], and a fixed version was presented in [99]. We are only interested in the unbroken assumption. The same assumption was used by [75] and we will adopt their notation in the following.

Assumption 1 *Let κ be a security parameter and $n, \rho \in \mathsf{poly}(\kappa)$. Further let $x, y \in \mathbf{F}^{t(\kappa)}$. Then the ensembles $\{\mathsf{Encode}_{n,\rho}(x)\}_\kappa$ and $\{\mathsf{Encode}_{n,\rho}(y)\}_\kappa$ are computationally indistinguishable, for $t \leq \frac{\ell}{4}$.*

As it is, our security reductions do not hold with respect to Assumption 1, but rather a variant of Assumption 1 which was already discussed in [98]. Instead of requiring indistinguishability of two encodings, we require that an encoding is pseudorandom. In order for these assumption to hold, [75] propose $n = 4\kappa, \rho = 2\kappa + 1$ as parameters, or $n = 8\kappa, \rho = 6\kappa + 1$ on the conservative side.

Assumption 2 *Let κ be a security parameter and $n, \rho \in \mathsf{poly}(\kappa)$. Further let $x \in \mathbf{F}^{t(\kappa)}$. Then the ensembles $\{\mathsf{Encode}_{n,\rho}(x)\}_\kappa$ and $\{G \leftarrow \mathscr{G}(n,\rho), v \leftarrow \mathbf{F}^n\}_\kappa$ are computationally indistinguishable, for $t \leq \frac{\ell}{4}$.*

Clearly, Assumption 2 implies Assumption 1, while the other direction is unclear. Apart from being a very natural assumption, Kiayias and Yung [81] provide additional evidence that Assumption 2 is valid. They show that if an adversary cannot decide for a random position i of the encoding whether it is part of the codeword or not, then the noisy codeword is indeed pseudorandom.

5.4 Constant Overhead Oblivious Linear Function Evaluation

Oblivious linear function evaluation (OLE) is the task of computing a linear function $f(x) = ax + b$ in the following setting. One party, lets call it the sender S, provides the function, namely the values a and b. The other party, the receiver R, wants to evaluate this function on his input x. This task becomes non-trivial if the parties want to evaluate the function in such a way that the sender learns

nothing about x, while the receiver learns only $f(x)$, but not a and b. OLE can be seen as a special case of oblivious polynomial evaluation (OPE) as proposed by Naor and Pinkas [98], where instead of a linear function f, the sender provides a polynomial p.

Ideal Functionality: The efficiency of our protocol follows in part from the fact that we can directly perform a batch of multiplications. This is reflected in the ideal UC-functionality for $\mathscr{F}_{\text{OLE}}^{\text{t}}$ (cf. Figure 2), which allows both sender and receiver to input vectors of size t.

Functionality $\mathscr{F}_{\text{OLE}}^{\text{t}}$

1. Upon receiving a message $(\texttt{inputS}, \mathbf{a}, \mathbf{b})$ from S with $\mathbf{a}, \mathbf{b} \in \mathbf{F}^t$, verify that there is no stored tuple, else ignore that message. Store $\mathbf{a}$ and $\mathbf{b}$ and send a message $(\texttt{input})$ to $\mathscr{A}$.
2. Upon receiving a message $(\texttt{inputR}, \mathbf{x})$ from R with $\mathbf{x} \in \mathbf{F}^t$, verify that there is no stored tuple, else ignore that message. Store $\mathbf{x}$ and send a message $(\texttt{input})$ to $\mathscr{A}$.
3. Upon receiving a message $(\texttt{deliver}, \mathsf{S})$ from $\mathscr{A}$, check if both $\mathbf{a}, \mathbf{b}$ and $\mathbf{x}$ are stored, else ignore that message. Send $(\texttt{delivered})$ to S.
4. Upon receiving a message $(\texttt{deliver}, \mathsf{R})$ from $\mathscr{A}$, check if both $\mathbf{a}, \mathbf{b}$ and $\mathbf{x}$ are stored, else ignore that message. Set $y_i = a_i \cdot x_i + b_i$ for $i \in [t]$ and send $(\texttt{output}, \mathbf{y})$ to R.

Figure 2: Ideal functionality for an oblivious linear function evaluation.

Our Protocol: Our starting point is the protocol of Ishai et al. [75] for *passively* secure batch multiplication. Their protocol is based on noisy encodings, similar to our construction. We will now briefly sketch their construction (with minor modifications) and then present the high-level ideas that are necessary to make the construction actively secure.

In their protocol, the receiver first creates a noisy encoding $(G, H, L, \mathbf{v}) \leftarrow \mathsf{Encode}(\mathbf{x})$ (as described in Section 5.3, Figure 1) and sends $(G, \mathbf{v})$ to the sender. At this point, the locations $i \in L$ of $\mathbf{v}$ hide a degree $\frac{\ell-1}{2}$ polynomial over the points $\beta_1, \ldots, \beta_n$ which evaluates to the input $\mathbf{x} = x_1, \ldots, x_t$ in the positions $\alpha_1, \ldots, \alpha_t$. The sender picks two random polynomials A and B with the restriction that $A(\alpha_i) = a_i$ and $B(\alpha_i) = b_i$ for $i \in [t]$. The degree of A is $\frac{\ell-1}{2}$, and the degree of B is $\ell - 1$.[4] This means that B completely hides A and therefore the inputs of the sender. Now the sender simply computes $w_i = A(\beta_i) \cdot v_i + B(\beta_i)$. Sender and receiver engage in an ℓ-out-of-n OTs, and the receiver picks the ℓ positions in L. He applies H to the obtained values and interpolates a polynomial Y which evaluates in position α_i to $a_i \cdot x_i + b_i$.

We keep the generic structure of the protocol of [75] in our protocol. In order to ensure correct and consistent inputs, we have to add additional checks. The complete description is given in Figure 3, and we give a high-level description of the ideas in the following paragraph.

First, we need to ensure that the receiver can only learn ℓ values, otherwise he could potentially reconstruct part of the input. Instead of using an expensive ℓ-out-of-n OT, we let the sender create a (ρ, n)-secret sharing (remember that $\rho + \ell = n$) of a random value e and the share s_i in the i'th OT, letting tge other message offered be a random value t_i. Depending on his set L, the receiver chooses t_i or the share s_i. Then he uses the shares to reconstruct e and sends it to the sender. This in turn might leak some information on L to the sender, if he can provide an inconsistent secret sharing. We thus force the sender to commit to e and later provide an unveil. Here the sender can learn some information on L, if he cheats but is not caught, but we can use our results from the previous section

[4] The value ℓ is fixed by the encoding, but we require that ℓ is uneven due to the fact that we have to reconstruct a polynomial of even degree $\frac{\ell-1}{2} + \frac{\ell-1}{2} = \ell - 1$, which requires ℓ values.

to show that this leakage is tolerable. The receiver can proceed and provide the encoding $\mathbf{v}$, which allows the sender to compute $\mathbf{w}$.

Second, we have to make sure that the sender computes the correct output. In order to catch a cheating sender, we add a check to the end of the protocol. Recall that the receiver knows the output Y. He can compute another polynomial X of his input and then pick a uniformly random challenge z_{R}. He sends it to the sender, who has to answer with $A(z_{\mathsf{R}}), B(z_{\mathsf{R}})$. Now the receiver can verify that $Y(z_{\mathsf{R}}) = A(z_{\mathsf{R}})X(z_{\mathsf{R}}) + B(z_{\mathsf{R}})$, i.e., the sender did not cheat in the noiseless positions. Again this leaks some information to the sender, but with the correct choice of parameters this leakage is inconsequential.

Security against a malicious receiver basically follows from the passively secure protocol. We only have to make sure that the extraction of his input is correct and that no information about the sender's inputs is leaked if e is incorrect. We thus mask the w_i by a one-time-pad and add the following check. This time the sender chooses z_{S} and the receiver has to answer with $X(z_{\mathsf{S}}), Y(z_{\mathsf{S}})$, which enforces correct extraction.

For the full proof and the OPE protocol we refer the reader to [62].

Protocol Π_{OLE}

Let $\mathscr{P} = \{\alpha_1, \ldots, \alpha_{\frac{\ell+1}{2}}, \beta_1, \ldots, \beta_n\}$ be a set of publicly known distinct points in $\mathbf{F}$. Further, let SS be a (ρ, n) secret sharing and COM be an OT-based commitment scheme. Set $\rho = \frac{3}{4}n$, $n = 8k$ and $\ell = n - \rho$.

1. Sender (Input $\mathbf{a}, \mathbf{b} \in \mathbf{F}^t$):
   - Draw a random polynomial A of degree $\frac{\ell-1}{2}$ with $A(\alpha_i) = a_i$ and a random polynomial B of degree $\ell - 1$ with $B(\alpha_i) = b_i \; \forall i \in \{1, \ldots, t\}$.
   - Draw a uniformly random vector $\mathbf{t} \in \mathbf{F}^n$.
   - Draw a random value $e \in \mathbf{F}$ and compute $\mathbf{s} \leftarrow \mathsf{SS.Share}(e)$. Further compute $(\mathsf{com}, \mathsf{unv}) \leftarrow \mathsf{COM.Commit}(e)$.
   - Send com to the receiver and engage in n OT instances with input (t_i, s_i) for instance i.
2. Receiver (Input $\mathbf{x} \in \mathbf{F}^t$):
   - Start the encode procedure $\mathsf{Encode}_{n,\rho}(\mathbf{x}, \mathscr{P})$ and obtain $(G, H, L, \mathbf{v})$. Interpolate a polynomial X through the points (β_i, v_i) for $i \in L$.
   - For each OT instance i, if $i \in L$, set $\texttt{choice}_i = 0$, otherwise set $\texttt{choice}_i = 1$.
   - Obtain ℓ values t_i and ρ values $\tilde{s}_i$. Compute $\tilde{e} = \mathsf{SS.Reconstruct}(\tilde{\mathbf{s}}_{|\neg L})$.
   - Send $\tilde{e}$ to the sender.
3. Sender: Check if $\tilde{e} = e$, if not abort. Send unv to the receiver.
4. Receiver: Check if $\mathsf{COM.Open}(\mathsf{com}, \mathsf{unv}, \tilde{e}) = 1$, abort if not. Send $(G, \mathbf{v})$ to the sender.
5. Sender: Compute $\tilde{w}_i = A(\beta_i) \cdot v_i + B(\beta_i) + t_i$ for $i \in \{1, \ldots, n\}$. Send $\tilde{\mathbf{w}} = (\tilde{w}_1, \ldots, \tilde{w}_n)$ to the receiver.
6. Receiver: Set $w_i = \tilde{w}_i - t_i$ for $i \in L$ and interpolate the degree $\ell - 1$ polynomial Y through the points (β_i, w_i) for $i \in L$. Draw $z_{\mathsf{R}} \in \mathbf{F} \setminus \mathscr{P}$ uniformly at random and send z_{R} to the sender.
7. Sender: Draw $z_{\mathsf{S}} \in \mathbf{F} \setminus \mathscr{P}$ uniformly at random and send $\big(A(z_{\mathsf{R}}), B(z_{\mathsf{R}}), z_{\mathsf{S}}\big)$ to the receiver.
8. Receiver:
   - Check if $A(z_{\mathsf{R}}) \cdot X(z_{\mathsf{R}}) + B(z_{\mathsf{R}}) = Y(z_{\mathsf{R}})$ and abort if not.
   - Send $(X(z_{\mathsf{S}}), Y(z_{\mathsf{S}}))$ to the sender and output $\mathbf{y} = H\mathbf{w}_{|L}$.
9. Sender: Check if $A(z_{\mathsf{S}}) \cdot X(z_{\mathsf{S}}) + B(z_{\mathsf{S}}) = Y(z_{\mathsf{S}})$ and abort if not.

Figure 3: Actively secure realization of $\mathscr{F}_{\text{OLE}}$ in the OT-hybrid model.

6 TinyOLE: Efficient Actively Secure Two-Party Computation from Oblivious Linear Function Evaluation

This section builds upon the previous section by showing how to construct an efficient secure two-party computation protocol based on the new OLE protocol. It is based on work published at CCS 2017 by Döttling et al. [49].

In [49], we introduce a new approach to actively secure 2PC which can securely compute an arithmetic circuit C over any field while using black-box access to only a very small constant number of OLEs over $\mathbf{F}$ per arithmetic gate in C. Specifically we use 22 OLE per multiplication gate in C, while at least 8 OLE are necessary to compute an authenticated multiplication even with semi-honest security. Additionally, the parties only have to perform a constant number of additions and multiplications. Since one OLE over many fields can in turn be implemented with communication $O(\kappa)$, where κ is the security parameter, under standard assumptions this gives a protocol with communication $O(|C|\kappa)$. This means the protocol is asymptotically as good as [74] when the field size is large, but the concrete complexity is much better. At the same time, the number of OLEs used by our protocol per *arithmetic* gate is significantly less than the number of OTs used by [100] per *Boolean* gate.

The only previous protocol we are aware of which achieves the same communication complexity as our protocol using black-box access to a general assumption is [75]. In [75] they show how to evaluate an arithmetic circuit C while using a constant number of black-box accesses to homomorphic encryption per gate. This gives a complexity of $O(|C|\kappa)$, where κ is the size of a ciphertext. The protocol, however, goes via the general framework in [74] and as such the big-O notation hides some fairly large constants. Our approach can therefore be seen as combining the good asymptotic complexity of [75] with the same good practical efficiency of [100].

Another potential advantage of our protocol over [75] and the other protocols for actively secure arithmetic computation mentioned above is that they use black box access to homomorphic encryption, whereas we use OLE. It seems to be a plausible conjecture that we might at some point be able to construct a protocol for actively secure OLE extension *a la* actively secure OT extension from [100], i.e. a protocol which given a few seed OLEs can generate any polynomial number of OLEs while using only a few applications of an efficient symmetric primitive per generated OLE. This hope is based on the similarity between OT and OLE. On the other hand there does not seem to be similar support for the hope that we might construct a homomorphic encryption extension, i.e.,, to implement any polynomial number of accesses to homomorphic encryption given only a few seed applications of homomorphic encryption plus access to for instance a hash function. If one could construct a protocol for efficient actively secure OLE extension, our protocol would immediately yield a very practical protocol, which would outcompete [100] even if using both protocols to compute a Boolean circuit.

6.1 Our Techniques

Our approach is somewhat reminiscent of the approach in [100]. We start with an arithmetic version of [65]. A value $a \in \mathbf{F}$ is represented by $a = a_1 + a_2$ where a_2 is uniformly random and a_i is known only by P_i. Secure addition is straight forward: $c_i = a_i + b_i$. To securely compute a representation of $c = ab$ one again uses that $ab = a_1b_1 + a_1b_2 + a_2b_1 + a_2b_2$. To compute a secure representation of a troublesome term like $c = a_1b_2$ use one OLE. Let the sender P_1 of the OLE pick uniformly random c_1 and input $(a,b) = (a_1, -c_1)$ and let the receiver input $x = b_2$. Then they invoke the OLE to compute $c_2 = ax + b = a_1b_2 - c_1$. Clearly $c_2 + c_1 = a_1b_2$. We then construct in an offline phase a large number of authenticated triples on random values and then we consume a small constant number of these in the online phase when evaluating an arithmetic gate. This forces the parties to input the

right values to all gates and hence gives active security. We essentially generate one authenticated multiplication by running one OLE on random values and then letting the parties commit to their values. We use that OLEs themselves can be seen as commitments to the value x to efficiently compute these commitments. Then similar to [100] we compute a test on each intermediate value to check that the commitments were computed correctly. We call this technique fingerprinting. These fingerprints are significantly different from the test in [100] which used a hash function. We only use black-box access to OLE. Furthermore, we manage to work around the selective error problem encountered in [100]. Our test in principle has a similar problem with a selective attack, but we can carefully arrange the test such that the sender of a and b needs to guess the random field element x to not get detected. This immediate gives security $|\mathbf{F}|^{-1}$, which is negligible for large enough fields. Developing the new test required developing significantly new techniques which exploit essentially that we are in the arithmetic setting where both parties have a large random input. As a result of our improved test for correct commitment, we do not need to apply the bucketing technique of [100] which immediately saves us an asymptotic factor of $O(s/\log(s))$. At the same time our test is simple enough that we get a very practical protocol.

Unfortunately our new test does not translate back to the OT based protocol from [100], as there the value corresponding to our x is the choice bit of the receiver which can of course be guessed with good probability. This shows that an essential prerequisite for our efficiency improvement is moving to the arithmetic setting with a large field, and we indeed exploit the arithmetic structure of the OLE primitive in many places throughout our protocols.

We believe that the efficiency of our protocols, both in asymptotic and practical terms, establishes OLE as an important foundation for efficient actively secure arithmetic 2PC.

6.2 The Dealer Protocol

In this section we briefly discuss the protocol to generate shared and authenticated multiplication triples, which uses an ideal OLE functionality $\mathbf{F}_q$-OLE. We call this protocol the dealer protocol. The detailed description of the protocol can be found in [49]. This is the off-line phase of the 2PC protocol. The online phase of the protocol is same as the online phase of SPDZ [40] or MASCOT [79].

We will start with a high level overview of our dealer protocol. In the semi-honest case, the purpose of the dealer-protocol is to provide random triples $(a_\mathsf{A}, b_\mathsf{A}, c_\mathsf{A})$ to A and $(a_\mathsf{B}, b_\mathsf{B}, c_\mathsf{B})$ to B such that the relation

$$(a_\mathsf{A} + a_\mathsf{B}) \cdot (b_\mathsf{A} + b_\mathsf{B}) = c_\mathsf{A} + c_\mathsf{B} \tag{1}$$

holds. We can expand this to

$$a_\mathsf{A} b_\mathsf{A} + a_\mathsf{A} b_\mathsf{B} + a_\mathsf{B} b_\mathsf{B} + a_\mathsf{B} b_\mathsf{A} = c_\mathsf{A} + c_\mathsf{B} \ . \tag{2}$$

In the simple semi-honest protocol A chooses the values a_A and b_A at random and B the values a_B and b_B. Notice that the terms $c^l_\mathsf{A} = a_\mathsf{A} b_\mathsf{A}$ and $c^l_\mathsf{B} = a_\mathsf{B} b_\mathsf{B}$ can be locally computed by A and B respectively. Thus, we need to securely compute the mixed terms $a_\mathsf{B} b_\mathsf{A}$ and $a_\mathsf{A} b_\mathsf{B}$. To do so, we will introduce random offsets r_A and r_B (chosen at random by A and B respectively). We can rewrite Equation (2) as

$$(a_\mathsf{A} b_\mathsf{A} - r_\mathsf{A} + a_\mathsf{A} b_\mathsf{B} + r_\mathsf{B}) + (a_\mathsf{B} b_\mathsf{B} - r_\mathsf{B} + a_\mathsf{B} b_\mathsf{A} + r_\mathsf{A}) = c_\mathsf{A} + c_\mathsf{B} \ . \tag{3}$$

This suggests the following protocol to compute the triples $(a_\mathsf{A}, b_\mathsf{A}, c_\mathsf{A})$ and $(a_\mathsf{B}, b_\mathsf{B}, c_\mathsf{B})$:

1. A and B choose $(a_\mathsf{A}, b_\mathsf{A}, r_\mathsf{A})$ and $(a_\mathsf{B}, b_\mathsf{B}, r_\mathsf{B})$ locally at random.

2. A and B compute $c^i_\mathsf{A} \leftarrow \mathbf{F}_q\text{-OLE}(a_\mathsf{B}, r_\mathsf{B}; b_\mathsf{A})$ and $c^i_\mathsf{B} \leftarrow \mathbf{F}_q\text{-OLE}(a_\mathsf{A}, r_\mathsf{A}; b_\mathsf{B})$.

3. A sets $c_\mathsf{A} = c^l_\mathsf{A} - r_\mathsf{A} + c^i_\mathsf{A}$ and B sets $c_\mathsf{B} = c^l_\mathsf{B} - r_\mathsf{B} + c^i_\mathsf{B}$.

Correctness of the protocol follows immediately from Equation (3). Semi-honest privacy of the protocol follows from the fact that c^i_A and c^i_B are independent of b_B and b_A and thus reveal nothing about these values (the remaining computations are local).

We will now augment this protocol into a (still semi-honestly secure) protocol such that A and B receive MACs on the other parties' triples. The party A chooses a global MAC key Δ_B and specific keys $K_{a_\mathsf{B}}, K_{b_\mathsf{B}}$ and K_{r_B}. Likewise, B chooses a global MAC key Δ_A and specific keys $K_{a_\mathsf{A}}, K_{b_\mathsf{A}}$ and K_{r_A}. Now, A commits to a_A by running $M_{a_\mathsf{A}} \leftarrow \mathbf{F}_q\text{-OLE}(\Delta_\mathsf{A}, K_{a_\mathsf{A}}; a_\mathsf{A})$. This MAC can be opened by A by sending a_A and M_{a_A} to B. The remaining MACs $(M_{b_\mathsf{A}}, M_{r_\mathsf{A}})$ and $(M_{a_\mathsf{B}}, M_{b_\mathsf{B}}, M_{r_\mathsf{B}})$ are computed likewise.

After the above protocol is finished, A obtains MACs on the values c^l_A and c^i_A by $M_{c^l_\mathsf{A}} \leftarrow \mathbf{F}_q\text{-OLE}(\Delta_\mathsf{A}, K_{c^l_\mathsf{A}}; c^l_\mathsf{A})$ and $M_{c^i_\mathsf{A}} \leftarrow \mathbf{F}_q\text{-OLE}(\Delta_\mathsf{A}, K_{c^i_\mathsf{A}}; c^i_\mathsf{A})$, where $K_{c^l_\mathsf{A}}$ and $K_{c^i_\mathsf{A}}$ are sampled on the fly by B. Likewise, B commits to c^l_B and c^i_B by $M_{c^l_\mathsf{B}} \leftarrow \mathbf{F}_q\text{-OLE}(\Delta_\mathsf{B}, K_{c^l_\mathsf{B}}; c^l_\mathsf{B})$ and $M_{c^i_\mathsf{B}} \leftarrow \mathbf{F}_q\text{-OLE}(\Delta_\mathsf{B}, K_{c^i_\mathsf{B}}; c^i_\mathsf{B})$.

Now, A and B can locally compute MACs on c_A and c_B by using the additively homomorphic properties of the MAC scheme. Specifically, A computes $M_{c_\mathsf{A}} = M_{c^l_\mathsf{A}} - M_{r_\mathsf{A}} + M_{c^i_\mathsf{A}}$ and $K_{c_\mathsf{B}} = K_{c^l_\mathsf{B}} - K_{r_\mathsf{B}} + K_{c^i_\mathsf{B}}$, whereas B computes $M_{c_\mathsf{B}} = M_{c^l_\mathsf{B}} - M_{r_\mathsf{B}} + M_{c^i_\mathsf{B}}$ and $K_{c_\mathsf{A}} = K_{c^l_\mathsf{A}} - K_{r_\mathsf{A}} + K_{c^i_\mathsf{A}}$.

This concludes the description of the semi-honestly secure scheme. We now provide a mechanism to enforce semi-honest behaviour by both parties. The main ideas of this technique can be sketched as follows. To enforce that the parties input the right values in different $\mathbf{F}_q$-OLE-instances, we compute several values twice, in two different ways. We call this technique *fingerprinting*.

Since the protocol is entirely symmetric, we will describe the fingerprinting sub-protocols only from the view of A in this outline.

- First, we want to make sure that A inputs the correct value c^l_A to obtain $M_{c^l_\mathsf{A}}$. At the same time, B must not use a different Δ_A than for the other MACs. We compute a check value γ^1 as follows. A and B use another $\mathbf{F}_q$-OLE to compute $\sigma^1_\mathsf{A} \leftarrow \mathbf{F}_q\text{-OLE}(-K_{a_\mathsf{A}}, K_{c^l_\mathsf{A}} + \gamma^1; b_\mathsf{A})$, where B chooses $\gamma^1 \leftarrow_\$ \mathbf{F}_q$. Now A locally computes $b_\mathsf{A} M_{a_\mathsf{A}} + \sigma^1 - M_{c^l_\mathsf{A}}$, which evaluates to γ^1 if $M_{c^l_\mathsf{A}}$ was computed with the correct c^l_A and Δ_A.

- Secondly, we have to ensure that c^i_A is correctly computed. Another OLE is used to compute $\sigma^2 \leftarrow \mathbf{F}_q\text{-OLE}(M_{a_\mathsf{B}}, M_{r_\mathsf{B}}; b_\mathsf{A})$. A locally computes $\sigma^{2\prime} = \Delta_\mathsf{B} c^i_\mathsf{A} + K_{a_\mathsf{B}} b_\mathsf{A} + K_{r_\mathsf{B}}$, and if B used the correct inputs, it holds that $\sigma^{2\prime} = \sigma^2$.

- Now that we know that c^i_A is correct, we have to verify that the MAC $M_{c^i_\mathsf{A}}$ was generated correctly. Towards this, we first observe that we can create $M_{c^i_\mathsf{A}}$ in a different way than described above: $M_{c^i_\mathsf{A}} \leftarrow \mathbf{F}_q\text{-OLE}(\Delta_\mathsf{A} a_\mathsf{B}, \Delta_\mathsf{A} r_\mathsf{B} + K_{c^i_\mathsf{A}}; b_\mathsf{A})$ yields $M_{c^i_\mathsf{A}} = \Delta_\mathsf{A}(a_\mathsf{B} b_\mathsf{A} + r_\mathsf{B}) + K_{c^i_\mathsf{A}}$. Now we create another check value γ^2 by using an additional authenticated value s_A. A locally computes $d = c^i_\mathsf{A} - s_\mathsf{A}$ and $\gamma^2 = M_{c^i_\mathsf{A}} - M_{s_\mathsf{A}}$, and sends d to B. B locally computes $\Delta_\mathsf{A} d + K_{c^i_\mathsf{A}} - K_{s_\mathsf{A}}$. Again, if A used the correct input for the MAC generation, and B as well, both parties will obtain the same γ^2.

- It remains to bind the input of A to the before authenticated value b_A. We therefore add a final check γ^3: A obtains $\sigma^3_\mathsf{A} \leftarrow \mathbf{F}_q\text{-OLE}(\Delta_\mathsf{A}, \sigma^3_\mathsf{B}; b_\mathsf{A})$. Now A locally computes $\gamma^3 = M_{b_\mathsf{A}} - \sigma^3_\mathsf{A}$, while B locally computes $K_{b_\mathsf{A}} - \sigma^3_\mathsf{B}$.

Protocol	Security	Field	Comm./Triple
SPDZ [40]	active	$\mathbf{F}_p$, 128-bit	430kBit
	covert	$\mathbf{F}_p$, 128-bit	132kBit
MASCOT [79]	active	$\mathbf{F}_q$, 128-bit	360kBit
	active	$\mathbf{F}_q$, 64-bit	106kBit
This work	active	$\mathbf{F}_q$, 128-bit	180kBit
	active	$\mathbf{F}_q$, 64-bit	90kBit

Table 4: Comparison of communication overhead per authenticated triple with existing solutions (taken from [79]).

In a final step we add all γ^i to a global check value γ^{glo}. Then A commits to its version of γ^{glo}, B sends its version to A, and A has to unveil. If the values do not match, the protocol is aborted.

6.3 Efficiency of our Approach

We give a short comparison with recent 2PC protocols that follow the same paradigm of implementing an arithmetic black box.

In order to evaluate the efficiency of our approach, we use the OLE protocol by SODA researchers described in Section 5 based on noisy encodings, which seems to have the highest practical efficiency both communication-wise and computation-wise. From that work, we can deduce the communication cost in field elements by

$$c_{\mathsf{OLE}} = \alpha(2 + c_{\mathsf{OT}})$$

for a single OLE. Here α is depending on the security parameters (basically the noise rate of the encoding) of the underlying assumption. According to [62] $\alpha = 8$ is a more optimistic choice, and $\alpha = 16$ a more conservative one. The cost of OT c_{OT} can be fixed to 2 field elements (when using OT extension). Assuming $\alpha = 16$, this means we need 64 field elements per OLE. Combining this with the overhead of our protocol we get that an authenticated triple costs

$$c_{\mathsf{triple}} = 2(c_{\mathsf{OLE}} \cdot 11 + 4) = 1416$$

field elements. It might be convenient to implement the OLE and then derandomize some of values for our protocol, because there are some dependencies between the inputs of the OLEs. But this will only induce an small additive overhead to c_{triple} (there are at most 22 values that have to be derandomized per triple), which is insignificant in comparison to the cost of the OLEs. All in all, we can bound the number of field elements by 1440.

Based on the above analysis we estimate the communication overhead of our protocol per authenticated triple and compare this in Table 4 with existing solutions. Looking at those numbers, it is interesting to see that our approach beats previous solutions even for smaller fields or with weaker security guarantees.

We currently have no implementation of the OLE protocol of [62], so we cannot give a fair comparison of the computational overhead with respect to other approaches. But we want to highlight that the OLE protocol only requires basic field operations and polynomial interpolation, which has a computational overhead of $n \log n$ for n OLEs. Our protocol itself only needs a constant number of basic field operations. It therefore seems a reasonable assumption that network bandwidth is the limiting factor for the triple generation, and not the computational overhead. This favors our approach over previous solutions.

7 Committed MPC

In this section we present a new maliciously secure MPC protocol based on secret-sharing, which is secure against a malicious and dishonest majority. Specifically our protocol allows up to $n-1$ out of n participants to be maliciously corrupted and unlike most previous work in this field, our protocol works over any field, even if it is not large or an extension field. One specific approach to efficient MPC, which has gained a lot of traction is based on secret sharing [65, 11, 8]: Each party secretly shares his or her input with the other parties. The parties then parse f as an arithmetic circuit, consisting of multiplication and addition gates. In a collaborative manner, based on the shares, they then compute the circuit, to achieve shares of the output which they can then open. Using the secret-sharing approach opens up the possibility of malicious parties using "inconsistent" shares in the collaborative computation. To combat this, most protocols add a MAC on the true value shared between the parties. If someone cheats it is then possible to detect this when verifying the MAC [40, 43, 100].

7.1 Our contributions

In our protocol we take a different approach to ensure correctness: We have each party commit to its shares towards the other parties using an additively homomorphic commitment. We then have the collaborative computation take place on the commitments instead of the pure shares. Thus, if some party tries to change its share during the protocol then the other parties will notice when the commitments are opened in the end (as the opening will be invalid).

By taking this path, we can present the following contributions:

1. An efficient and black-box reduction from random multi-party homomorphic commitments, to two-party additively homomorphic commitments. We describe this in detail in Section 7.4.

2. Using these multi-party commitments we present a new secret-sharing based MPC protocol with security against a majority of malicious adversaries. Leveraging the commitments, our approach does not use any MAC scheme and does not rely on a random oracle or any specific number theoretic assumptions. We describe this in detail in Section 7.5.

3. The new protocol has several advantages over previous protocols in the same model. In particular our protocol works over fields of arbitrary characteristic, independent of the security parameter. Furthermore, since our protocol computes over committed values it can easily be composed inside larger protocols. For example, it can be used for computing committed OT in a very natural and efficient way.

4. We suggest an efficient realization of our protocol, which only relies on a PRG, coin-tossing and OT[5]. We give a detailed comparison of our scheme with other dishonest majority, secret-sharing based MPC schemes, showing that the efficiency of our scheme is comparable, and in several cases preferable, over state-of-the-art.

7.2 Overview of our techniques

We depart from any (possibly interactive) two-party additively homomorphic commitment scheme. To achieve the most efficient result, without relying on a random oracle or specific number theoretic

[5]OT can be efficiently realized using an OT extension, without the usage of a random oracle, but rather a type of correlation robustness, as described in [5].

assumptions, we consider the scheme of [54], since it has been shown to be highly efficient in practice [101, 105]. This scheme, along with others [34, 23, 22] works on commitments to vectors of m elements over some field $\mathbb{F}$. For this reason we also present our results in this setting. Thus any of these schemes could be used.

The first part of our protocol constructs a large batch of commitments to random values. The actual value in such a commitment is unknown to any party, instead, each party holds an additive share of it. This is done by having each party pick a random message and commit to it towards every other party, using the two-party additively homomorphic commitment scheme. The resulted multi-party commitment is the sum of all the messages the parties committed to, which is uniformly random if there is at least one honest party. We must ensure that a party commits to the same message towards all other parties, to this end the parties agree on a few (random) linear combinations over the commitments, which are then opened and being checked.

Based on these random additively shared commitments, the parties execute a preprocessing stage to construct random multiplication triples. This is done in a manner similar to MASCOT [79], yet a bit different, since our scheme supports computation over arbitrary small fields and MASCOT requires a field of size exponential in the security parameter. More specifically the Gilboa protocol [63] for multiplication of additively shared values is used to compute the product of two shares of the commitments between each pair of parties. However, this is not maliciously secure as the result might be incorrect and a few bits of information on the honest parties' shares might be leaked. To ensure correctness cut-and-choose and sacrificing steps are executed. First, a few triples are opened and checked for correctness. This ensures that not all triples are incorrectly constructed. Next, the remaining triples are mapped into buckets, where some triples are sacrificed to check correctness of another triple. At this point all the triples are correct except with negligible probability. Finally, since the above process grants the adversary the ability to leak some bits from the honest parties shares, the parties engage in a combining step, where triples are randomly "added" together to ensure that the result will contain at least one fully random triple.

As the underlying two-party commitments are for *vectors* of messages, our protocol immediately features single-instruction multiple-data (SIMD), which allows multiple simultaneously executions of the same computation (over different inputs). However, when performing only a single execution we would like to use only one element out of the vector and save the rest of the elements for a later use. We do so by preprocessing reorganization pairs, following the same approach presented in MiniMAC [43, 36, 42], which allows us to perform a linear transformation over a committed vector.

With the preprocessing done, the online phase of our protocol proceeds like previous secret-sharing based MPC schemes such as [40, 79, 43]. That is, the parties use their share of the random commitments to give input to the protocol. Addition is then carried out locally and the random multiplication triples are used to interactively realize multiplication gates. We outline the complexity of our scheme in Table 5.

7.3 Related Work

There is a plethora of literature on the secret-sharing based MPC in the setting of a dishonest and malicious majority. One of the most families of protocols in this setting might be SPDZ [40, 79]. This scheme is very similar to ours, but uses information theoretic MACs instead of commitments and unfortunately only works over fields of size $\Omega(2^s)$. However, another family of protocols called TinyOT [100, 87, 21] allows computation over the binary field instead. Like SPDZ, TinyOT also requires a MAC on the secret-shared valued, this MAC is of size s per secret-shared bit. Our scheme also facilitates computation over the binary field, but in the amortized sense, using suitable commitments

	Rand, Input	Add	Mult
OTs	0	0	$27m\log(\lvert\mathbb{F}\rvert)n(n-1)$
Two-party Commitments	0	0	$81n(n-1)$
Random coins	$\log(\lvert\mathbb{F}\rvert)$	0	$108\log(\lvert\mathbb{F}\rvert)$

Table 5: Amortized complexity of each instruction of our protocol (Rand,Input,Add and Mult), when constructing a batch of 2^{20} multiplication triples, *each* with m independent components among n parties. The quadratic complexity of the number of two-party commitments reflects the fact that our protocol is constructed from any two-party commitment scheme in a black-box manner, and so each party independently commits to all other party for every share it posses.

only give constant overhead per secret-sharing bit rather than s bits from the MACs for the TinyOT family. Another family of protocols, called MiniMAC [43, 36, 42] manages to overcome the $O(s)$ overhead per bit when working in binary fields. In fact this family of protocols allow constant overhead MACs, in the amortized sense, per bit for secure computation. Unfortunately the authors of [43] did not describe how to realize the preprocessing needed. Neither did the follow up works [36, 42]. The only efficient[6] preprocessing protocol for MiniMAC that we know of is the one presented in [55] based on OT extension. However this protocol has it quirks in that it only works over fields of characteristic 2 and that the ideal functionality it implements is different from the one in [43, 36, 42] in that is gives the adversary slightly more power.

There is one other previous work, by Damgård and Orlandi [38], which also considers a maliciously secure secret-sharing based MPC in the dishonest majority setting using additively homomorphic commitments. However, unlike ours, their protocol only works over large arithmetic fields and uses a very different approach. Specifically they use the cut-and-choose paradigm along with packed secret-sharing in order to construct multiplication triples. Furthermore, to get random commitments in the multi-party setting, they require usage of public-key encryption for each commitment. Thus, the amount of public-key operations they require is linear in the amount of multiplication gates in the circuit to compute. In our protocol it is possible to limit the amount of public-key operations to be linear in the security parameter, as we only require public-key primitives to bootstrap the OT extension. Finally, they rely on commitments being non-interactive. In particular, that a commitment is a static value which can be broadcast. Our scheme does not require non-interactive commitments and can thus use recent, highly efficient commitment extension schemes such as [54].

7.4 Multi-party Commitments

In this section we show how to convert any homomorphic two-party commitment scheme into the multi-party kind we need. To clarify, by multi-party commitment we mean that we consider a set of parties where a party can commit to a specific value towards all other parties. The value can then be opened to one specific party or all parties at a later point in time. Furthermore in our functionality we also allow the sampling of a commitment to a random value unknown to all parties.

Preprocessing. The parties produce a batch of multi-party commitments rather than one at a time. Assume the parties wish to produce γ commitments, each party picks $\gamma+s$ uniformly random messages from $\mathbb{F}^m$. Each party commit to each of these $\gamma+s$ messages towards each other party using

[6] I.e. one that does not use a generic MPC protocol to do the preprocessing.

different instances of the two-party commitment functionality, and thus different internal randomness.

Note that a malicious party might use the two-party commitment scheme to commit to different messages toward different parties, which leads to an incorrect multi-party commitment. To thwart this, we have the parties execute random linear combination checks as done for batch-opening of commitments in [54]: The parties invoke the coin-tossing protocol to agree on a $s \times \gamma$ matrix, $\mathbf{R}$ with elements in $\mathbb{F}$. In the following we denote the element in the qth row of the kth column of $\mathbf{R}$ by $\mathbf{R}_{q,k}$. Every party computes s random linear combinations of the opening information that it holds toward every other party. Similarly, every party computes s combinations of the commitments that it obtained from every other party. The coefficients of the qth combination are determined by the q'th row $\mathbf{R}$ and the qth vector from the s "extra" committed messages added to the combination. That is, let the $\gamma+s$ messages committed by party P_i toward P_j be $\mathbf{x}_1^{i,j},\ldots,\mathbf{x}_{\gamma+s}^{i,j}$ and see that the qth combination from P_i to P_j is $\left(\sum_{k\in\gamma}\mathbf{R}_{q,k}\cdot\mathbf{x}_k^{i,j}\right)+\mathbf{x}_{\gamma+q}^{i,j}$. Then P_i open the result to P_j, who checks that it is correct. If P_i was honest it committed to the same values towards all parties and so $\mathbf{x}_k^i=\mathbf{x}_k^{i,j}=\mathbf{x}_k^{i,j'}$ for all $k\in[\gamma+s]$ and $j\neq j'\in[p]\setminus\{i\}$. Likewise for the other parties, so if everyone is honest they all obtain the same result from the opening of the combination. Thus, a secure equality check would be correct in this case. However, if P_i cheated, and committed to different values toward different parties than this is detected with overwhelming probability, since the parties perform s such combinations. This means that the parties now have γ random commitments. That is, to the values $\mathbf{x}_k=\sum_{i\in[n]}\mathbf{x}_k^i$ for $k\in[\gamma]$.

Online. Each party does a partial opening (see below) of a random, unused commitment towards the party that is supposed to give input. Based on the opened message the inputting party computes a *correction value*. That is, if the random commitment, before issuing the input command, is a shared commitment to the value $\mathbf{x}$ and the inputting party wish to input $\mathbf{y}$, then it computes the value $\varepsilon=\mathbf{y}-\mathbf{x}$ and sends this value to all parties. Since the party giving input is the only one who knows the value $\mathbf{x}$, and it is random, this does not leak.

To open a commitment to $\mathbf{x}$, for every pair $i\neq j$ for $i,j\in[n]$ party P_i opens $\mathbf{x}^i$ towards party P_j using the two-party commitment functionality. Finally every party P_j computes $\mathbf{x}=\sum_{i\in[n]}\mathbf{x}^i$. It is easy to see that this also yields additive homomorphic openings following directly from the homomorphism of the underlying two-party commitment scheme and the distributive property.

7.5 Committed Multi-party computation

In the malicious, dishonest majority setting, our protocol, as many other protocols, works on preprocessed random Beaver triples. These are constructed based on the random multi-party commitments described above. Specifically we construct triples of values $(\mathbf{x},\mathbf{y},\mathbf{z})$ such that $\mathbf{x}*\mathbf{y}=\mathbf{z}$. Note that a single multiplication triple is actually three multi-party commitments to values from $\mathbb{F}^m$ such that $\mathbf{z}$ is the result of a componentwise multiplication of $\mathbf{x}$ and $\mathbf{y}$. The construction of triples is done in a batch similarly to recent preprocessing protocols [79, 90]. Specifically this consists of four parts briefly described below:

1. **Construction.** Using OT for multiplication [63], the parties first *construct* multiplication triples that may be "malformed" and "leaky" in case of a malicious adversary. Here malformed means that they are incorrect, i.e. $\mathbf{x}*\mathbf{y}\neq\mathbf{z}$ and "leaky" means that the adversary has tried to guess the value of the share of an honest party.

2. **Cut-and-Choose.** The parties select τ triples at random which they check for correctness, where τ is the bucketing parameter. If any of these triples are malformed then they abort.

Otherwise, when mapping the remaining triples into buckets, with overwhelming probability all buckets will contain at least one correct triple.

3. **Sacrificing.** The remaining triples (from the cut-and-choose) are mapped to buckets, τ triples in each bucket such that at least one of the triples is correct. Each bucket is then tested to check its correctness where by only a single multiplication is being output while the other $\tau - 1$ are being discarded. This step guarantees that either the output triple is correct or a malformed triple is detected, in which case the protocol aborts.

4. **Combining.** As some of the triples may be "leaky" this allows the adversary to carry a selective attack, that is, to probe whether its guess was correct or not. If the guess is affected by the input of an honest party then it means that the adversary learns that input. Thus, as the name suggests, the goal of this step is to produce a non-leaky triple by combining τ triples, which are the result of the sacrificing step (and thus are guaranteed to be correct), where at least one of the τ is non-leaky.

With multiplication triples in place it is easy to realize a full MPC protocol. Basically it will work almost the same as for MASCOT [79], with the exception that no MAC checks are done as part of opening values as authenticity is handled through the commitments.

7.5.1 Optimizing for large fields.

If the field we compute in contains at least 2^s elements, then the construction of multiplication triples becomes much lighter. First see that in this case it is sufficient to only have two triples per bucket for sacrificing. This is because the probability of passing a sacrificing test on two triples, where one is incorrect, is $|\mathbb{F}|^{-1} \leq 2^{-s}$. Next we see that it is possible to eliminate the combining step on the $\mathbf{y}$ components of the triples. This follows since a selective failure attack can now only be done by the sender in the OT protocol used to compute the product. Before, this could also be done by a malicious receiver since it could guess the sender's input with probability $|\mathbb{F}|^{-1}$, which is not negligible when $|\mathbb{F}|$ is small. Finally, a larger field also allows a smaller bucketing parameter for combining. To conclude, even when using a very conservative bound on bucket size, we get that it now takes only $6m\log(|\mathbb{F}|)$ OTs, amortized, when constructing 2^{21} triples, instead of $27m\log(|\mathbb{F}|)$ when $s = 40$.

8 Pinocchio-Based Adaptive zk-SNARKs and Secure / Correct Adaptive Function Evaluation

Recent advances in SNARKs (Succinct Non-interactive ARguments of Knowledge) are making it more and more feasible to outsource computations to the cloud while obtaining cryptographic guarantees about the correctness of their outputs. In particular, the Pinocchio system [59, 102] achieved for the first time for a practical computation a verification time of a computation proof that was actually faster than performing the computation itself.

In Pinocchio, proofs are verified with respect to plaintext inputs and outputs of the verifier; but in many cases, it is useful to have computation proofs that also refer to committed data, e.g., provided by a third party. Ideally, such proofs should be *adaptive*, i.e., multiple different computations can be performed on the same commitment, that are chosen after the data has been committed to; and *zero-knowledge*, i.e., the commitments and proofs should reveal no information about the committed data. This latter property allows proofs on sensitive data, and it allows extensions like Trinocchio [109] that additionally hide this sensitive data from provers by multi-party computation.

Although several approaches are known from the literature, no really satisfactory practical adaptive zk-SNARK exists. The recent "hash first" proposal [52] shows how to make Pinocchio adaptive at low overhead, but is unfortunately not zero-knowledge. On the other hand, Pinocchio's successor Geppetto [31] is zero-knowledge but not adaptive: multiple computations can be performed on the same data but they need to be known before committing. The asymptotically best known SNARKS combining the two properties have a large practical overhead: [94] because it relies on the impractical subset-sum language; other constructions (e.g., [31, 52]) because they rely on including hash evaluation in the computation. Finally, [7] enables Pinocchio proofs on authenticated data with prover complexity equal to the best known, but verification time is linear in the number of committed inputs.

In this work, we give a new Pinocchio-based adaptive zk-SNARK that solves the above problems. We further apply this to make Trinocchio work on computation-independent commitments, present tooling to easily program flexible verifiable computations (with or without MPC), and use it to build a prototype in a medical research case study.

8.1 Adaptive zk-SNARKs based on Pinocchio

The central contribution here is an adaptive zk-SNARK based on Pinocchio. We obtain our Pinocchio-based adaptive zk-SNARK by generalising the role of the $\langle Z\rangle_1$ element of the Pinocchio proof. In Pinocchio, proof elements $\langle V\rangle_1$, $\langle W\rangle_1$, and $\langle Y\rangle_1$ are essentially weighted sums $\sum_j \mathbf{x}_j\langle v_j\rangle_1$, $\sum_j \mathbf{x}_j\langle w_j\rangle_2$, $\sum_j \mathbf{x}_j\langle y_j\rangle_1$ over elements $\langle v_j\rangle_1$, $\langle w_j\rangle_2$, $\langle y_j\rangle_1$ from the CRS, with the weights given by the witness part of the QAP's solution vector $\mathbf{x}$. The $\langle Z\rangle_1$ element ensures that these weighted sums consistently use the same witness. This is done by forcing the prover to come up essentially with $\beta\cdot(\langle V\rangle_1 + \langle W\rangle_2 + \langle Y\rangle_1)$ given only elements $\langle\beta\cdot(v_j+w_j+y_j)\rangle_1$ in which v_j, w_j, and y_j occur together. The essential idea is of our construction is to use the $\langle Z\rangle_1$ element also to ensure consistency to external commitments.

In more detail, in earlier works [31, 109], it was noted that the Pinocchio $\langle V\rangle_1, \langle W\rangle_2, \langle Y\rangle_1$ elements can be divided into multiple "blocks" $(\langle V_i\rangle_1, \langle W_i\rangle_2, \langle Y_i\rangle_1, \langle Z_i\rangle_1)$. Each block contains the values of a number of variables of the QAP solution, which is enforced by providing $\langle z_j\rangle_1 = \langle\beta_i\cdot(v_j+w_j+y_j)\rangle_1$ elements only for the indices j of those variables. Our core idea is use external commitments of the form $\sum_k \mathbf{v}_k\cdot\langle x^k\rangle_1$ (that can be re-used across Pinocchio computations) and link the kth component of this commitment to the jth variable of the block using a modified $\langle z_j\rangle_1 = \langle\beta_i\cdot(x^k+v_j+w_j+y_j)\rangle_1$. We use one block per external commitment that the proof refers to. The witness (which is not

Extractable Trapdoor Commitment Scheme Family $(\mathsf{G0}^1, \mathsf{Gc}^1, \mathsf{C}^1)$**:**

- $\mathsf{G0}^1$: Fix $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ and random s. Return $\mathsf{crs} = (\{\langle x^i\rangle_1, \langle x^i\rangle_2\}_{i=0,\ldots,d})$, $\mathsf{td} = s$.
- Gc^1: Pick random α. Return $\mathsf{ck} = (\langle 1\rangle_1, \langle \alpha\rangle_2, \langle x\rangle_1, \langle \alpha x\rangle_2, \ldots, \langle x^d\rangle_1, \langle \alpha x^d\rangle_2)$
- C^1: Return $(r\langle 1\rangle_1 + \mathbf{v}_1\langle x\rangle_1 + \mathbf{v}_2\langle x^2\rangle_1 + \ldots, r\langle \alpha\rangle_2 + \mathbf{v}_1\langle \alpha x\rangle_2 + \mathbf{v}_2\langle \alpha x^2\rangle_2 + \ldots)$

Key generation G^1**:** Fix a QAP of degree at most d, and let $v_j(x), w_j(x), y_j(x)$ be as in Pinocchio. Fix random, secret $\alpha_v, \alpha_w, \alpha_y, \beta, r_v, r_w$. Let $r_y = r_v r_w$. Let $z_j(x) = x^j + r_v v_j + r_w w_j + r_y y_j$ if $j \le W$ and $z_j(x) = r_v v_j + r_w w_j + r_y y_j$ otherwise. Evaluation key ($i = 1, \ldots, n$, $j = 1, \ldots, d$):

$$\langle x^j\rangle_1, \langle r_v v_j\rangle_1, \langle r_v t\rangle_1, \langle \alpha_v r_v v_j\rangle_2, \langle \alpha_v r_v t\rangle_2 \langle r_w w_j\rangle_1, \langle r_w t\rangle_1, \langle \alpha_w r_w w_j\rangle_1, \langle \alpha_w r_w t\rangle_1 \langle r_y y_j\rangle_1,$$
$$\langle r_y t\rangle_1, \langle \alpha_y r_y y_j\rangle_2, \langle \alpha_y r_y t\rangle_2 \langle \beta_i z_{(i-1)d+j}\rangle_1, \langle \beta_i z_{nd+j}\rangle_1, \langle \beta_i\rangle_1, \langle \beta_i r_v t\rangle_1, \langle \beta_i r_w t\rangle_1, \langle \beta_i r_y t\rangle_1$$

Verification key ($i = 1, \ldots, n$): $(\langle \alpha_v\rangle_2, \langle \alpha_w\rangle_1, \langle \alpha_y\rangle_2, \langle \beta_i\rangle_2, \langle \beta_i\rangle_1, \langle r_y t\rangle_2)$.

Proof generation P^1**:** Let $\mathbf{u}_i = \mathsf{C}^1_{\mathsf{ck}_i}(\mathbf{v_i}; r_i)$, and let $\mathbf{w}$ be the witness such that $(\mathbf{v_1}, \ldots, \mathbf{v_n}; \mathbf{w})$ is a solution to the QAP. Generate random $\delta_{v,i}, \delta_{w,i}, \delta_{y,i}$. Compute $\mathbf{h}$ as the coefficients of polynomial $((\sum_j \mathbf{x}_j \cdot v_j(x) + \delta_v \cdot t(x)) \cdot (\sum_j \mathbf{x}_j \cdot w_j(x) + \delta_w \cdot t(x)) - (\sum_j \mathbf{x}_j \cdot y_j(x) + \delta_y \cdot t(x)))/t(x)$. Return ($i = 1, \ldots, n$; $[\cdot]$ means only if $i = 1$):

$$\langle V_i\rangle_1 = \textstyle\sum_{j=1}^d \mathbf{v}_{\mathbf{i},j} \langle r_v v_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \mathbf{w}_j \langle r_v v_{nd+j}\rangle_1\right] + \delta_{v,i}\langle r_v t\rangle_1, \langle \alpha_v V_i\rangle_2 = \ldots$$
$$\langle W_i\rangle_1 = \textstyle\sum_{j=1}^d \mathbf{v}_{\mathbf{i},j} \langle r_w w_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \mathbf{w}_j \langle r_w w_{nd+j}\rangle_1\right] + \delta_{w,i}\langle r_w t\rangle_1, \langle \alpha_w W_i\rangle_1 = .$$
$$\langle Y_i\rangle_1 = \textstyle\sum_{j=1}^d \mathbf{v}_{\mathbf{i},j} \langle r_y y_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \mathbf{w}_j \langle r_y y_{nd+j}\rangle_1\right] + \delta_{y,i}\langle r_y t\rangle_1, \langle \alpha_y Y_i\rangle_2 = \ldots$$
$$\langle Z_i\rangle_1 = \textstyle\sum_{j=1}^d \mathbf{v}_{\mathbf{i},j} \langle \beta_i z_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \mathbf{w}_j \langle \beta_i z_{nd+j}\rangle_1\right] + r_i\langle \beta_i\rangle_1 + \delta_{v,i}\langle \beta_i r_v t\rangle_1$$
$$\langle H\rangle_1 = \textstyle\sum_j \mathbf{h}_j \langle x^j\rangle_1. \qquad + \delta_{w,i}\langle \beta_i r_w t\rangle_1 + \delta_{y,i}\langle \beta_i r_y t\rangle_1$$

Proof verification V^1**:** Letting $\mathsf{ck}_i = (\ldots, \langle \alpha_i\rangle_2)$, $\mathbf{u}_i = (\langle C_i\rangle_1, \langle \alpha_i C_i\rangle_2)$, check that:

$$e(\langle C_i\rangle_1, \langle \alpha_i\rangle_2) = e(\langle 1\rangle_1, \langle \alpha_i C_i\rangle_2); \;\; e(\langle V_i\rangle_1, \langle \alpha_v\rangle_2) = e(\langle \alpha_v V_i\rangle_1, \langle 1\rangle_2); \qquad \textbf{(C,V)}$$
$$e(\langle \alpha_w\rangle_1, \langle W_i\rangle_2) = e(\langle 1\rangle_1, \langle \alpha_w W_i\rangle_2); \;\; e(\langle Y_i\rangle_1, \langle \alpha_y\rangle_2) = e(\langle 1\rangle_1, \langle \alpha_y Y_i\rangle_2); \qquad \textbf{(W,Y)}$$
$$e(\langle V_i\rangle_1 + \langle Y_i\rangle_1 + \langle C_i\rangle_1, \langle \beta_i\rangle_2) \cdot e(\langle \beta_i\rangle_1, \langle W_i\rangle_2) = e(\langle Z_i\rangle_1, \langle 1\rangle_2); \qquad \textbf{(Z)}$$
$$e(\langle V\rangle_1, \langle W\rangle_2) \cdot e(\langle Y\rangle_1, \langle 1\rangle_2)^{-1} = e(\langle H\rangle_1, \langle r_y t\rangle_2). \qquad \textbf{(H)}$$

(where $\langle V\rangle_1 = \langle V_1\rangle_1 + \ldots + \langle V_n\rangle_1$, $\langle W\rangle_2 = \langle W_1\rangle_2 + \ldots + \langle W_n\rangle_2$, $\langle Y\rangle_1 = \langle Y_1\rangle_1 + \ldots$)

Figure 4: Pinocchio-Based Adaptive zk-SNARK $(\mathsf{G}^1, \mathsf{P}^1, \mathsf{V}^1)$

committed to externally) is included in the first block, with the normal Pinocchio element $\langle z_j\rangle_1 = \langle \beta_1 \cdot (v_j + w_j + y_j)\rangle_1$ just checking internal consistency as usual. The verification procedure changes slightly: $\langle V\rangle_1$ is no longer extended to $\langle V^+\rangle_1$ to include public I/O (which we do not have); instead, the $(\mathbf{Z})$ check ensures consistency with the corresponding commitment, for which there is a new correctness check $(\mathbf{C})$.

The precise construction is shown in Figure 4 with full background and details provided in [117]. This construction contains details on how to add randomness to make the proof zero-knowledge; and it

Construction	Comm. Size	Proof size	Prover computation		Verif comp
			non-crypt. op.	crypt. op.	
Geppetto	3 gr. el.	8 gr. el.	$\Theta(D\log D)$	$\Theta(D)$	$4n+12$ pair.
Hash First+Pinocchio	2 gr. el.	9n+1 gr. el.	$\Theta(d\log d)$	$\Theta(d)$	$13n+3$ pair.
Hash First+Pinocchio*	2 gr. el.	5n+5 gr. el.	$\Theta(d'\log d')$	$\Theta(d')$	$8n+8$ pair.
Our zk-SNARK I	2 gr. el.	7n+1 gr. el.	$\Theta(d\log d)$	$\Theta(d)$	$11n+3$ pair.
Our zk-SNARK I*	2 gr. el.	3n+5 gr. el.	$\Theta(d'\log d')$	$\Theta(d')$	$7n+7$ pair.
Our zk-SNARK II	2 gr. el.	3n+8 gr. el.	$\Theta(d\log d)$	$\Theta(d)$	$6n+12$ pair.

Table 6: Comparison between Pinocchio-based SNARKs (n: number of commitments; d is QAP degree; $d' \leq d$ is QAP degree with optimization; $D \geq d$ is fixed QAP degree)

shows how additional $\langle \alpha \cdot \rangle_1$ elements are added to obtain an extractable trapdoor commitment family $(\mathsf{G0}^1, \mathsf{Gc}^1, \mathsf{C}^1)$.

8.2 Smaller Proofs and Comparison to Literature

We now compare the concrete efficiency of our proposal, including two (mutually incompatible) optimizations that decrease the size of the above zk-SNARK, to two related proposals from the literature.

In Table 6, we provide a detailed comparison of our zk-SNARKs with two similar constructions: the Geppetto protocol due to [31] (which is also zero-knowledge but not adaptive); and the "hash first" approach applied to Pinocchio [52] (which is adaptive but not zero-knowledge). Geppetto is Protocol 2 from [31].

Geppetto is the most efficient construction, but apart from not being adaptive, it also requires all computations to be fixed and of the same size, making it inefficient for small computations when they are combined with large ones. Our construction outperforms Hash First+Pinocchio, essentially adding zero knowledge for free. Which optimization is best depends on n and $d' - d$.

8.3 Secure/Correct Adaptive Function Evaluation

In this section, we sketch how our zk-SNARK can be used to perform "adaptive function evaluation": privacy-preserving verifiable computation on committed data. We consider a setting in which multiple mutually distrusting *data owners* want to allow privacy-preserving outsourced computations on their joint data. A *client* asks a computation to be performed on this data by a set of *workers*. The input data is sensitive, so the workers should not learn what data they are computing on (assuming up to a maximum number of workers are passively corrupted). On the other hand, the client wants to be guaranteed the computation result is correct, for instance, with respect to a commitment to the data published by the data owner (making no assumption on which data owners and/or workers are actively corrupted). The difference in assumptions for the privacy and correctness guarantees is motivated by settings where data owners together choose the computation infrastructure (so they feel active corruption is unlikely) but need to convince an external client (e.g. a medical reviewer) of correctness. We work in the CRS model, where a trusted party (who is otherwise not involved in the system) performs one-time key generation.

In the Appendix of [117] we provide a precise security model that captures the above security guarantees by ideal functionalities.

8.3.1 Our Construction

We now present our general construction based on multi-party computation and *any* adaptive zk-SNARK. At a high level, to achieve secure adaptive function evaluation, the workers compute the function using multi-party computation (MPC), guaranteeing privacy and correctness under certain conditions. However, when these conditions are not met, we still want to achieve correct adaptive function evaluation, i.e., we still want to ensure a correct computation result. To achieve this, the workers also produce, using MPC, a zk-SNARK proof of correctness of the result.

We require a MPC protocol in the outsourcing setting, i.e., with separate inputters (in our case, the data owners and the client), recipients (the client) and workers. The protocol needs to be reactive, so that the data owners can provide their input before knowing the function to be computed and secure even if any number of data owners the client are actively corrupted. Security of the MPC protocol will generally depend on how many workers are corrupted; our construction will realise secure adaptive function evaluation (as opposed to just correct adaptive function evaluation) exactly when the underlying MPC protocol is secure.

Our protocol is shown in Figure 5. It uses an MPC protocol with the above properties, a trapdoor commitment family, and an adaptive zk-SNARK, instantiated for the function to be computed. The protocol relies on a trusted party that generates the key material of the zk-SNARK, but is otherwise not involved in the computation. Each data owner has an input $\mathbf{a_i} \in \mathbb{F}^d$ and the client has an input $\mathbf{a_c} \in \mathbb{F}^{d'}$ and a function $f : (\mathbb{F}^d)^n \times \mathbb{F}^{d'} \to \mathbb{F}^{d-d'}$ that it wants to compute on the combined data. Internal variables of the MPC protocol are denoted $[\![\cdot]\!]$.

8.4 Prototype and Distributed Medical Research Case

In this section, we present a proof-of-concept implementation of our second zk-SNARK construction and our Adaptive Trinocchio protocol. Computations can be specified at a high level using a Python frontend; executed either locally or in a privacy-preserving way using multi-party computation; and then automatically proven and verified to be correct by a C++ backend. We show how computations can be performed on the same committed data coming from multiple data owners (with key material independent from input length, and optionally in a privacy-preserving way). Below we discuss this for aggregate survival statistics on two patient populations. Another that we prototyped is the "log-rank test": a common statistical test whether there is a statistically significant difference survival rate between the populations.

8.4.1 Application to Medical Survival Analysis

We have applied our prototype to (adaptively) perform computations on survival data about two patient populations. In medical research, survival data about a population is a set of tuples (n_j, d_j), where n_j is the number of patients still in the study just before time j and d_j is the number of deaths at time j. We assume both populations are distributed among multiple hospitals, that each commit to their contributions $(d_{j,1}, n_{j,1}, d_{j,2}, n_{j,2})$ to the two populations at each time.

Aggregate Survival Data Our first computation is to compute an aggregate version of the survival data, where each block $\{d_{j,1}, n_{j,1}, d_{j,2}, n_{j,2}\}_{j=1}^{25}$ of 25 time points is summarised as $(\sum_j d_{j,1}, n_{1,1}, \sum_j d_{j,2}, n_{1,2})$. The function SUMM computing this summary is shown in Algorithm 1. Function SUMM translates into a QAP on 26 commitments: as input, for each time point j, a commitment $\sum_i c_{i,j}$ to the combined survival data $([\![\mathbf{d_{i,1}}]\!], [\![\mathbf{n_{i,1}}]\!], [\![\mathbf{d_{i,2}}]\!], [\![\mathbf{n_{i,2}}]\!])$ from the different hospitals i at that time (using the fact that commitments are homomorphic); as output, a commitment to $([\![d'_1]\!], [\![n'_1]\!], [\![d'_2]\!], [\![n'_2]\!])$.

Protocol Adaptive Trinocchio

(Data provider has $\mathbf{a_i} \in \mathbb{F}^d$; client has $\mathbf{a_c} \in \mathbb{F}^{d'}$, function $f : (\mathbb{F}^d)^n \times \mathbb{F}^{d'} \to \mathbb{F}^{d-d'}$.)

1. The trusted party generates a system-wide CRS crs of the trapdoor commitment family, and commitment keys $\mathsf{ck}_1, \ldots, \mathsf{ck}_n, \mathsf{ck}_c$ for the data owner and client. This material is distributed to all parties.
2. Each data owner computes commitment $c_i = \mathsf{C}_{\mathsf{ck}_i}(\mathbf{a_i}, r_i)$ to its input $\mathbf{a_i} \in \mathbb{F}^d$ using randomness r_i, and publishes it on a bulletin board.
3. The data owners, workers, and client use the MPC protocol to do the following:
   - Each data owner provides input $\mathbf{a_i}$ and randomness r_i
   - For each i, compute $c_i' = \mathsf{C}_{\mathsf{ck}_i}([\![\mathbf{a_i}]\!], [\![r_i]\!])$; if $c_i \neq c_i'$ then abort
4. The client provides function f to the trusted party. The trusted party determines a QAP Q computing f and a function f' solving Q, and performs key generation of the adaptive zk-SNARK (where one commitment combines the client's input and output). The client gets verification key crsv; the workers get Q, f', and the corresponding evaluation key crsp.
5. The data owners, workers, and client continue with the MPC from step 3:
   - Client: provide input $\mathbf{a_c}$
   - Compute $([\![\mathbf{b}]\!]; [\![\mathbf{w}]\!]) \leftarrow f'([\![\mathbf{a_1}]\!]; \ldots; [\![\mathbf{a_n}]\!]; [\![\mathbf{a_c}]\!])$
   - Compute $[\![c_c]\!] \leftarrow \mathsf{C}_{\mathsf{ck}_c}([\![\mathbf{a_c}]\!], [\![\mathbf{b}]\!]; [\![r_c]\!])$ for random r_c
   - Compute $[\![\pi]\!] \leftarrow \mathsf{P}(\mathsf{crs}, \{\mathsf{ck}_i\}, \ldots, \mathsf{ck}_n, \mathsf{ck}_c, \mathsf{crsp}; c_1, \ldots, c_n, [\![c_c]\!]; [\![\mathbf{a_1}]\!]; \ldots; [\![\mathbf{a_n}]\!]; [\![\mathbf{a_c}]\!], [\![\mathbf{b}]\!]; [\![r_1]\!], \ldots, [\![r_n]\!], [\![r_c]\!]; [\![\mathbf{w}]\!])$
   - Open outputs $[\![\mathbf{b}]\!], [\![r_c]\!], [\![c_c]\!], [\![\pi]\!]$ to the client
6. The client checks whether $\mathsf{V}(\mathsf{crs}, \mathsf{ck}_1, \ldots, \mathsf{ck}_n, \mathsf{ck}_c, \mathsf{crsv}; c_1, \ldots, c_n, c_c; \pi) = 1$ and $c_c = \mathsf{C}_{\mathsf{ck}_c}(\mathbf{a_c}, \mathbf{b}; r_c)$ and if so, returns computation result $\mathbf{b}$.

Figure 5: The Adaptive Trinocchio protocol

Performance Table 7 shows the performance of our proof-of-concept implementation for computing aggregate survival data (on a modern laptop). As input, we used the "btrial" data set included in R's "kmsurv" package (on which we indeed reproduce R's `survdiff` result) of 175 data points. Apart from having one data point per commitment, we also experiment with having a "block size" of 25 or 175 data points. For aggregation, we use one QAP per 25 data points or per commitment, whichever is more.

We time the performance of running the computation, producing the proof, and verifying it, with or without MPC. As expected, MPC induces a large overhead for the computation. This holds especially for the logrank test (due to the many fixed-point computations), which we cover in the full paper. MPC also incurs an overhead for proving: this is because of the many exponentiations with $|\mathbb{F}|$-sized secret shares rather than small witnesses. Note that proving is faster than computing with MPC: the underlying operations are slower [109], but the QAP proof is in effect on a verification circuit that is smaller than the circuit of the computation itself [46].

Algorithm 1 Anonymized survival data computation

Require: $[\![\mathbf{d_1}]\!], [\![\mathbf{n_1}]\!], [\![\mathbf{d_2}]\!], [\![\mathbf{n_2}]\!]$: block of survival data points for two populations
Ensure: $([\![d'_1]\!], [\![n'_1]\!], [\![d'_2]\!], [\![n'_2]\!])$ aggregated survival data for the block
1: **function** SUMM($[\![\mathbf{d}_{i,1}]\!], [\![\mathbf{d}_{i,2}]\!], [\![\mathbf{n}_{i,1}]\!], [\![\mathbf{n}_{i,2}]\!]$)
2: **return** $(\sum_i [\![\mathbf{d_1}_{,i}]\!], [\![\mathbf{n_1}_{,1}]\!], \sum_i [\![\mathbf{d_2}_{,i}]\!], [\![\mathbf{n_2}_{,1}]\!])$

	Computation (function): 0.0s (w/o MPC) / 0.1s (w/ MPC) Computation (function+witness): 0.0s (w/o MPC) / 0.1s (w/ MPC)		
	BS=1	**BS=25**	**BS=175**
Aggregate	QAP degree: 3 Prover: 0.3s/0.4s Verifier: 1.2s/1.5s	QAP degree: 3 Prover: 0.1s/0.1s Verifier: 0.2s/0.2s	QAP degree: 57 Prover: 0.0s/0.0s Verifier: 0.0s/0.0s

Table 7: Performance: computation/proving/verification; with/without MPC

9 Oblivious Verification of Inexact String Matches

We present a technique to verify that a search string occurs at a certain position in a reference string with a certain maximum edit distance, where the search string, position, and reference string can all stay hidden with respect to the party or parties performing the verification. Inexact searching for substrings is a well-known general problem in computer science, whose applications include DNA matching[7], which was what primarily motivated this work. SODA deliverable D2.2 presents our algorithm for privacy preserving DNA sequence alignment.

9.1 Background

It is a well-known problem to search for a substring in a string, while allowing for a certain number of mismatches. One important application is in DNA sequence alignment, where short samples of DNA are matched with a reference genome to find their location (mismatches are needed both because of read errors and because DNA slightly differs from person to person). The number of mismatches is usually formalized by means of the *edit* or *Levenshtein distance*, which is defined as the total number of single-character insertions, deletions, or replacements needed to transform one string into the other. The actual list of changes needed to transform one string into the other is known as the *edit script*.

Finding a substring with a maximal edit distance is a computationally expensive task, for which many precise and heuristic algorithms are known. Here we will use the technique from [89], which is an exact algorithm proposed in the context of DNA sequence alignment. Techniques are also known to solve this problem using multi-party computation [116], e.g., in the case where a patient does not want to share his DNA with a healthcare provider, while the healthcare provide does not want to share the search query.

For the related problem of computing the edit distance between two strings (i.e., matching the whole two strings instead of finding one string in the other), algorithms are known that compute the edit distance and the corresponding edit script at the same time [18, 83]. Given the edit script, it is easy to design an algorithm to verify that two strings have edit distance at most the number of errors in the edit script. We did not find in the literature an algorithm that performs inexact search and returns the edit script, or an algorithm that verifies that an edit script is correct. However, both would seem likely to exist.

9.2 Privacy-preserving edit string verification

As discussed above, it is easy to check correctness of an an edit script in order to check that a search string occurs in a reference string with a maximal edit distance. However, to perform this a check, access to the search string and reference string is needed. In settings where these strings are confidential, this is not desirable. We present a data-oblivious algorithm for edit string verification that can be used to perform this check in a privacy-preserving way, for example in the following setttings:

- A client lets a search be performed by third party, and wishes to obtain a guarantee that the search result is correct, without having access to the reference string. This can be done by running the data-oblivious algorithm as a verifiable computation according to section 8 [117].
- Same as in the first point, but where the search is performed in a privacy-preserving way using multi-party computation [117].

[7] we like to acknowledge Fatih Turkmen for the initial suggestion to apply verifiable computation to DNA search results

- Same as in the first point, but where instead of by a non-interactive verifiable computation, the verification is performed by an interactive multi-party computation protocol between the client and the third party.
- A search is performed in a privacy-preserving way between several parties (e.g., one holding the reference string and another holding the search string). For efficiency reasons, the search is performed using a protocol for multi-party computation that is passively secure but actively private, i.e., a protocol that prevents leakage of information to malicious participants but does not guarantee correctness. To obtain active security, i.e., a guarantee of correctness of the result, the parties perform the edit string verification algorithm using an actively secure multi-party computation.

For edit string verification, obviously the edit string needs to be computed in the first place. As discussed above, algorithms are known that compute edit strings at the same time as computing the edit distance. In the privacy-preserving setting, no such algorithm is known. Hence, we developed a privacy-preserving version of the inexact search algorithm of [89] that computes the edit string.

9.3 Approach

Our first idea is a novel representation of the edit script that is suitable for oblivious computation and verification. Traditionally, an edit script is represented as a list of changes, e.g., "replace an A by a C at location 2". However, such a representation is not amenable to easy verification, especially in the data-oblivious setting. Instead, we propose to represent the edit script as a sequence of operations $\in \{\text{match}, \text{mismatch}, \text{insertion}, \text{deletion}\}$ describing in sequence how the two strings can be matched against each other. For instance, the sequence $(\text{match}, \text{mismatch}, \text{match}, \text{match}, \text{deletion})$ describes how string "ATTCC" can be transformed into string "AGTC". Since the sequence contains two non-matches, the edit distance between the two strings is at most two.

Our second idea is to allow a "dummy" operation in the above representation of edit scripts. Note that the length of an edit script can reveal information about how a search string was matched in a certain reference string. For instance, consider the search string "AAGC". In reference string "AATC", this search string can be matched with edit distance at most one with edit script $(\text{match}, \text{match}, \text{mismatch}, \text{match})$. In reference string "AATGC", the edit script with edit distance at most one has length five: (match, match, insertion, mismatch, match). Hence the length of the edit script leaks information about the underlying reference string. However, the length of the edit script is always at most the length of the search string plus the number of mismatches. We propose to use dummy operations to pad the edit script so that it always attains this maximum length.

Our third idea is to compute the edit script (including dummy operations) while performing inexact search in a data-oblivious way. We do this by following the recursive approach of [89] and making sure that each recursive call made along the way adds enough dummy operations to make the edit script attain its maximal length.

Our fourth idea is to perform edit script verification by sequentially going through the reference string, search string, and edit script at the same time. We keep secret position counters in the reference and search strings and, for each operation in the edit script, we verify that the characters at these secret positions match the operation given in the edit script. If this succeeds, the maximum number of errors is not exceeded, and the end of the search string is reached, the verification has succeeded.

9.4 Construction

9.4.1 Inexact Search with Edit Script Computation

Our data-oblivious algorithm to perform inexact search while computing the edit script given in Algorithm 2. This algorithm can be implemented in a privacy-preserving way using any framework for multiparty computation such as VIFF. Alternatively, the algorithm can be implemented in a non-privacy-preserving way to get an algorithm for inexact search that returns the edit script (including dummy operations) as well as returning the search result. (In this case, the conditional combine in line 22 would be replaced by calling InexRecur either for the match or the mismatch case, as appropriate.)

As discussed, this algorithm is an adaptation of the inexact search algorthm from [89]. In our description, we have abstracted away from how the actual searching in the reference string happens (lines 11 and 3); in practice, this is done with an auxiliary data structure called the Burrows-Wheeler matrix. Note that [89] is specific to DNA (or, more generally, strings from a small alphabet); however, the principe of computing the edit script along with the search can be generalized to other search algorithms. Focussing on how the edit script is computed, we ensure that the edit script has length $\mathsf{len}([W])+z$ by its initialization (lines 4 and 6) and via the recursive calls for insertion (line 9), deletion (line 14), match (line 16), and mismatch (line 19). Note that ensuring this invariant requires us to add a dummy item for insertion and match, but not for deletion and mismatch. Results from multiple recursive calls are combined in a data-oblivious way using the Combine function, that, given two return values, simply takes the first one if it contains a match and otherwise the second one. For the case of match and mismatch, we make the algorithm data-oblivious by evaluating both the match and the mismatch branch and obliviously combining them with CCombine, which selects the match or mismatch result depending on the oblivious test $[w] = b$.

9.4.2 Edit script verification

Our data-oblivious edit script verification algorithm is shown in Algorithm 3. This algorithm can be implemented in a privacy-preserving way using any framework for multiparty computation, e.g. VIFF. In this case, **assert***s* are performed by public zero tests, as available in VIFF, and secret indexing $[c] \leftarrow [A]_{[b]}$ is done as described in, e.g., [45]. Alternatively, it can be implemented for verifiable computation, for instance using the VIFF-based framework from [117]. In this case, the production of the verifiable comptation proof can either be performed in a privacy-preserving way or not. Note that [117] details oblivious construction for arithmetic computations and the operator $[b] \leftarrow [a] \neq 0$. Based on this, the operations used in Algorithm 3 can be implemented as follows:

- $[a] \leftarrow ([b] = c)$ is computed as $[a] \leftarrow 1 - ([b-c] \neq 0)$
- **assert**$([a] \neq b)$ is done by computing $[x] \leftarrow [(a-b) \neq 0]$, opening x and checking it is zero
- **assert**$([a] = 0)$ is done by opening a and checking it is zero
- Computing $[c] \leftarrow [A]_{[b]}$ is done by converting $[b]$ into unit vector $([b_1], \ldots, [b_{\mathsf{len}(|A|)}])$ with 1 at position b, proving its correctness by checking that $[b_i]$ equals $([b] = i)$ for all i, and computing $[c] \leftarrow [b_1] \cdot [A_1] + [b_2] \cdot [A_2] + \ldots$

The implementation is a reasonably straightforward implementation of the idea presented above. As discussed, we go through the elememts $[S_i]$ of the edit script, keeping track of the current location $[xpos]$ in the reference string and $[wpos]$ in the search string. We need to check that the number

Algorithm 2 Inexact search

Require: X: reference string, $[W]$: unmatched part of search string, R: matched part of search string, z: number of allowed mismatches

Ensure: Returns $([f],[p],[S])$ where, if there is no match then $f=0$; otherwise, $f=1$, p is the position of a match, and S the corresponding edit script

1: **function** InexRecur($X,[W],R,z$)
2: **if** len($[W]$) $=0$ **then** ▷ empty search string: we have found a match
3: $p \leftarrow$ location of any occurence of R in X
4: **return** $([1],[p],([\text{dummy},\ldots,\text{dummy}]))$ ▷ z dummies
5: $([W'],[w]) \leftarrow [W]$ ▷ split into last character and remainder
6: $([f],[p],[S]) \leftarrow ([0],[0],([\text{dummy},\ldots,\text{dummy}]))$ ▷ $z+\text{len}([W])$ dummies
7: **if** $z>0$ **then**
8: $([f'],[p'],[S']) \leftarrow \text{InexRecur}(X,[W'],R,z-1)$ ▷ insertion
9: $([f],[p],[S]) \leftarrow \text{Combine}(([f],[p],[S]),([f'],[p'],[\text{insertion}]|||[\text{dummy}]|||[S']))$
10: **for** $c \in \{A,C,G,T\}$ **do**
11: **if** $b||R$ occurs in X **then**
12: **if** $z>0$ **then** ▷ deletion
13: $([f'],[p'],[S']) \leftarrow \text{InexRecur}(X,[W],b||R,z-1)$
14: $([f],[p],[S]) \leftarrow \text{Combine}(([f],[p],[S]),([f'],[p'],[\text{deletion}]|||[S']))$
15: $([f_1],[p_1],[S_1]) \leftarrow \text{InexRecur}(X,[W'],b||R,z)$ ▷ match
16: $[S_1] \leftarrow [\text{match}]|||[S_1]$
17: **if** $z>0$ **then** ▷ mismatch
18: $([f_2],[p_2],[S_2]) \leftarrow \text{InexRecur}(X,[W'],b||R,z-1)$
19: $[S_1] \leftarrow [\text{mismatch}]|||[\text{dummy}]|||[S_1]$
20: **else**
21: $([f_2],[p_2],[S_2]) \leftarrow ([0],[0],([\text{dummy},\ldots,\text{dummy}]))$ ▷ $z+\text{len}([W])\times$
22: $([f'],[p'],[S']) \leftarrow \text{CCombine}([w]=b,([f_1],[p_1],[S_1]),([f_2],[p_2],[S_2]))$
23: $([f],[p],[S]) \leftarrow \text{Combine}(([f],[p],[S]),([f'],[p'],[S']))$
24: **return** $([f],[p],[S])$

Require: $([f],[p],[S]),([f'],[p'],[S'])$: two InexRecur return values
Ensure: Returns $([f],[p],[S])$ if $f=1$, $([f'],[p'],[S'])$ otherwise
25: **function** Combine($([f],[p],[S]),([f'],[p'],[S'])$)
26: **return** $([f]+[f']-[f]\cdot[f'],[f]\cdot[p]+(1-[f])\cdot[p],([f]\cdot[S_1]+(1-[f]\cdot[S_1'],\ldots))$

Require: $[b]$: choice bit, $([f],[p],[S]),([f'],[p'],[S'])$: two InexRecur return values
Ensure: Returns $([f],[p],[S])$ if $b=1$, $([f'],[p'],[S'])$ otherwise
27: **function** CCombine($[b],([f],[p],[S]),([f'],[p'],[S'])$)
28: **return** $([b]\cdot[f]+(1-[b])\cdot[f'],[b]\cdot[p]+(1-[b])\cdot[p],([b]\cdot[S_1]+(1-[b]\cdot[S_1'],\ldots))$

of errors does not exceed the bound z; we do this by checking inside the loop that the number of errors $[nerrors]$ encountered so far does not equal $z+1$ (line 10). (Alternatively, one can verify that $[nerrors \leq z]$ at the end.) We also need to check that the current positions in X and W do not exceed their bounds. However, note that the current position is allowed to be the length plus one, in case no

Algorithm 3 Edit script verification

Require: $[X]$: reference string, $[W]$: search string, $[xpos]$: position in X, z: maximum number of mismatches, $[S]$: edit script of of length $\mathsf{len}([W])+z$

Ensure: **assert**s an error if the edit script is wrong, otherwise returns

1: **function** VerifyEditScript($[X],[W],[xpos],z,[S]$)
2: $[wpos] \leftarrow 1$
3: $[nerrors] \leftarrow 0$
4: **for** $i = 1,\ldots,\mathsf{len}([S])$ **do**
5: $[ismatch] \leftarrow ([S_i] = \text{match})$
6: $[ismismatch] \leftarrow ([S_i] = \text{mismatch})$
7: $[isinsertion] \leftarrow ([S_i] = \text{insertion})$
8: $[isdeletion] \leftarrow ([S_i] = \text{deletion})$
9: $[nerrors] \leftarrow [nerrors] + [ismismatch] + [isinsertion] + [isdeletion]$
10: **assert**$([nerrors] \neq z+1)$
11: **assert**$([xpos] \neq \mathsf{len}([X])+2)$
12: **assert**$([wpos] \neq \mathsf{len}([W])+2)$
13: $[x] \leftarrow ([X]||[0])_{[xpos]}$
14: $[w] \leftarrow ([W]||[0])_{[wpos]}$
15: $[err] \leftarrow [match] \cdot ([x{=}0] + [w{=}0] + [x{-}w{\neq}0]) + [mismatch] \cdot ([x{=}0] + [w{=}0])$
16: **assert**$([err] = 0)$
17: $[xpos] \leftarrow [xpos] + [ismatch] + [ismismatch] + [isdeletion]$
18: $[wpos] \leftarrow [wpos] + [ismatch] + [ismismatch] + [isinsertion]$
19: **assert**$([wpos] = \mathsf{len}([W])+1)$

match or mismatch occurs in the edit script that requires us to read at this location. Hence, we check that the current position never reaches the length plus two (lines 11 and 12). We can then compute the items $[x]$ and $[w]$ at the given locations, where we add a zero to the reference and search strings, to make sure that the read is always defined. We then check if there is an error in the edit script, which is the case if there is supposed to be a match and $[x]$ and $[w]$ do not match or are outside of the strings; or if there is supposed to be a mismatch and $[x]$ or $[w]$ is outside of the string (line 16). Finally, we ensure that the edit script has reached the end of the search string (line 19). These checks combined imply validity of the edit script, i.e., the search string did occur at the given location in the reference string with an edit distance of at most z.

9.5 Possible extensions

- The current edit script computation and verification use Levenshtein distance, i.e., mismatches, insertions, and deletions are penalized with a score of 1. Both algorithms are easily generalized to other scoring functions.
- The presented search algorithm assumes that the reference string X is public; it is also possible to devise an alternative that keeps X private. Basically, this means that lines 11 and 3 need to be evaluated in a privacy-preserving way. In particular, we need to always evaluate the "if" branch in line 12 and then ignore its result depending on the oblivious condition that $b||R$ occurs in $[X]$. Since this modification does not influence our central contribution to obliviously compute the edit script, we do not discuss it further.

- To speed up secret indexing in W (line 14 of Algorithm 3), note that the position in W is always contained in the interval $[i-z,i]$. Hence, secret indexing only needs to be done in an array of length $z+1$.
- In the verifiable computation context where proofs are produced in the clear, VerifyEditScript is inefficient on large reference strings because of the use of secret indexing. To remedy this, one can use an approach based on Merkle trees. Here, the reference string is stored in a Merkle tree, whose root is an input for the verifier. Now, the relevant portion of the reference string is obliviously authenticated with respect to this root inside the verifiable computation, after which VerifyEditScript only needs to be called with respect to this relevant portion.

10 Data-Oblivious Array Slicing with Sliding Windows

We present a data-oblivious technique for array slicing. Array slicing aims to retrieve a fixed number of elements from an array starting at a variable location. This can be done with subsequent data-oblivious single-element accesses using existing techniques, but we provide a technique based on sliding windows that is more efficient.

One particular motivation for this comes from the application of proving, in a verifiable computation, that a DNA string contains a particular substring as presented in the previous section. One of the steps in this computation is to retrieve a substring from the DNA string (on order to prove that it matches the requested substring, e.g., with some spelling mistakes. Here, the DNA string is typically very long, so being able to do this with just one linear scan through the DNA string is highly desirable.

10.1 Background

Data-oblivious algorithms are algorithms whose program flow, including memory accesses, does not depend on its inputs. Data-oblivious algorithms are useful to protect against side-channel attacks (as witnessed by the recent Spectre and Meltdown attacks on Intel processors); and for programming privacy-preserving of verifiable computations (that need to be data-oblivious by nature).

One common operation for which data-oblivious techniques are known, is indexing an array at a variable location. For instance, in the context of multi-party computation, this problem is known as "secret indexing" and [45] provides a possible solution. In this solution, the index is first converted to a bitstring $i_1, \ldots, i_N$ with a one exactly at the location corresponding to the index, and the element of array $[a_1, \ldots, a_N]$ at the index is computed as $i_1 \cdot a_1 + \ldots + i_N \cdot a_N$. Note that this requires $\mathcal{O}(N)$ operations, i.e., this approach for indexing at a variable location is linear in the size of the array. Other approaches such as Path ORAM [113] provide asymptotically better performance, but this improvement only kicks in for large arrays.

A related problem is array slicing. In array slicing, the objective is to retrieve not one element, but a (constant) number of subsequent elements. (This can be generalised to a variable number of elements with a certain interval between them, but we do not consider this generalisation here.) As far as we know, the problem of data-oblivious array slicing has not been considered in the literature.

A straightforward way to implement data-oblivious array slicing is possible based on existing techniques for data-oblivious array indexing. Namely, to retrieve n subsequent elements starting at location i, one can simply use array indexing to subsequently accesss elements $i, i+1, \ldots, i+n-1$.

Unfortunately, this naive way to implement array slicing based on array indexing is inefficient. In particular, if array indexing takes $\mathcal{O}(N)$ time, where N is the length of the array, then taking a slice of length n takes $\mathcal{O}(N \cdot n)$ time. Instead, we provide a solution that takes $\mathcal{O}(N)$ time overall.

10.2 Approach

The core idea of this is to use a "sliding window"-based approach to perform array slicing. We walk linearly through the original, source, array, and at the same time, we walk linearly through a new, target, array of the size of the resulting slice. If we are at the end of the target array, we go back to the start, and if we did not reach the end of the slice, we copy the element from the current position in the original array to the current position in the target array. This is a linear scan, so it takes $\mathcal{O}(N)$ time.

Eventually, the target array contains the intended array slice, possibly shifted by a certain amount. We can then either use this target array directly, making sure to compensate for the shift when accessing its elements; or we shift the elements from the target array so that it starts with the correct element.

Algorithm 4 Oblivious slicing

Require: $\mathbf{A}$: source array; $\mathbf{i}$: index of slice; l: length of slice

Ensure: Returns $(\mathbf{B},\mathbf{s})$ where target array B is the slice of array A at index i with length l, shifted cyclically to the right by s positions

1: **function** ArraySlice($\mathbf{A},\mathbf{i},l$)
2: $\quad\mathbf{B} \leftarrow (0,\ldots,0)$ ▷ array of length l
3: $\quad\mathbf{w} \leftarrow 1$
4: $\quad\mathbf{s} \leftarrow 0$
5: $\quad$**for** $j = 0,\ldots,|A|-1$ **do**
6: $\quad\quad\mathbf{B}[j \mod l] \leftarrow \mathbf{B}[j \mod l] + \mathbf{w}\cdot(\mathbf{A}[j] - \mathbf{B}[j \mod l])$
7: $\quad\quad\mathbf{w} \leftarrow \mathbf{w} - [\mathbf{i} = j+1-l]$
8: $\quad\quad\mathbf{s} \leftarrow \mathbf{s} + [\mathbf{i} = j]\cdot((j \mod l) - \mathbf{s})$
9: $\quad$**return** $(\mathbf{B},\mathbf{s})$

The latter can be performed by doing n "oblivious left shifts" (i.e., shifting all elements left one place if some secret condition is satisfied) at a cost of $\mathcal{O}(n^2)$ time.

10.3 Setting

The principle described here applies to any setting where data-obliviousness is needed (e.g., hardware with side channels or multi-party computation), but for a concrete embodiment one may consider verifiable computation, and in particular, a system such as Pinocchio [102] or PySNARK (https://github.com/Charterhouse/pysnark) in which the algorithms we specify below may be easily implemented as verifiable computations.

When specifying our algorithms, we use boldfaced to denote "sensitive" variables whose value should not influence the control flow. So for instance, in Algorithm 4 below, the control flow should not depend on the index $\mathbf{i}$, which may be sensitive, but it may depend on the length l, which is public. Similarly, the loop variable j is public so the control flow may depend on it (but, in turn, its value may not depend on any boldfaced variable). In verifiable computation, boldfaced variables correspond to I/O or witnesses of the proof and non-boldfaced variables correspond to constants; in multi-party computation, boldfaced variables correspond to secret-shared or encrypted data and non-boldfaced variables correspond to plain (i.e. not encrypted or secret-shared) data.

In particular, we assume that an operation $[\mathbf{a} = b]$ is available that is equal to 1 if a sensitive value $\mathbf{a}$ is equal to a non-sensitive value b, and 0 otherwise. In verifiable computation, this operation is performed in a data-oblivious way using the zero-equality gate from [102] (noting that `(a=b)?1:0` is equivalent to `1-(a-b≠0?1:0)`). In multi-party computation, this operation can be implemented with the EQZ protocol from [45].

10.4 Oblivious Slicing

Our oblivious array slicing algorithm is shown in Algorithm 4. As can be seen, the algorithm is basically a linear scan through the source array (lines 5–8). Throughout the scan, $\mathbf{w}$ represents whether we are still writing in the target, i.e., whether we are before the end of the array slice. If this is the case, we update target array $\mathbf{B}$ at location $(j \mod l)$ with the current element of the source array; otherwise, we do not change it (line 6). $\mathbf{w}$ is initially set to one, but it is set to zero as soon as the end of the slice has been reached, i.e., if j equals $\mathbf{i}+l-1$ (line 7). $\mathbf{s}$ is used to compute the shift: if we are

Algorithm 5 Oblivious shifting

Require: **B**: array, **s**: number of positions
Ensure: **C**: array B left-shifted by s positions

1: **function** ArrayShift(**B**, **s**)
2: $\quad$ $\mathbf{w} \leftarrow 1$
3: $\quad$ $\mathbf{C} \leftarrow \mathbf{B}$
4: $\quad$ **for** $i = 0, \ldots, |\mathbf{B}| - 1$ **do**
5: $\quad\quad$ $\mathbf{w} \leftarrow \mathbf{w} - [\mathbf{s} = i]$
6: $\quad\quad$ **for** $j = 0, \ldots, |\mathbf{B}| - 1$ **do** $\mathbf{C}[j] \leftarrow \mathbf{C}[j] + \mathbf{w} \cdot (\mathbf{C}[(j+1) \mod |\mathbf{B}|] - \mathbf{C}[j])$
7: $\quad$ **return C**

at the start of the slice it is set to the shift corresponding to the current index i, otherwise it remains unchanged (line 8). Finally, the computed **B** and **s** are returned (line 9).

As discussed, ArraySlice returns a shifted target array **B** plus a shift **s**. The element at the **i**th position in this array can be retrieved in a data-oblivious way by computing **j**, where $j = (s+i) \mod |\mathbf{B}|$, in a data-oblivious way, and then accessing **B** at location **j** in a data-oblivious way, as discussed above. Alternatively, we can shift the whole array **B** by **s** positions in a data-oblivious way to obtain non-shifted array **C**. For instance, this can be done as shown in Algorithm 5 by repeatedly left-shifting by one position at most $|\mathbf{B}|$ times. Specifically, in this algorithm, **w** determines whether we still need to apply more left-shifts; it is set to zero if **s** left-shifts have taken place (line 5) and if it is non-zero, the array is left-shifted by one position (line 6).

11 Privacy-Preserving Verifiable Fixed-Point Division

We present a procedure that allows provers to cryptographically prove that a value contained in a commitment or encryption is a fixed-point multiplication or fixed-point division of two other committed or encrypted numbers. The proof can be constructed by a single prover, or by multiple provers who do not individually know the sensitive values they are computing on.

In various cryptographic systems, there is a need to prove that a computation was correctly performed:

- When outsourcing a computation, systems for verifiable computation such as Pinocchio [102] allow a client to efficiently verify that the output was correct, possibly more efficiently than performing the computation itself;
- The above setting can also be instantiated in a privacy-friendly way, without the workers knowing what sensitive values they are computing on, using systems like Trinocchio [109];
- When parties input sensitive data to a privacy-preserving computation (e.g., e-voting), they may need to prove that this input data is in the correct input domain for the computation, e.g., that numbers are within a certain range or textual input contains only valid characters;

In general, these and other applications require cryptographic tools to prove that a set of input values hidden in some representation (an encryption or a commitment) satisfies a certain property.

The fundamentals of proving that committed or encrypted values satisfy a certain property are well known. Such a property is typically modelled by the fact that a certain predicate on the values should satisfy true. The function to compute this predicate is usually generically modelled as a circuit. One type of circuit that is commonly used are binary circuits, e.g., [76]. Another type is arithmetic circuits, i.e., circuits consisting of additions and multiplications of natural numbers, possibly modulo a prime. We focus here on the arithmetic circuit representation.

In theory any property can be modelled as a circuit, but in practice this leads to very large circuits so tailored solutions are needed for particular properties. For instance, a large body of work has researched how to efficiently prove that a value lies within a certain range, e.g., [41], and [102] shows efficient proofs of correctness of bit decompositions and of correct application of the delta-function.

However, practical computations often take place on non-integers, for instance fixed-point numbers. That is, we take a stored integer c to represent value $\tilde{c} = c \cdot 2^{-k}$ for some fixed k. Addition and subtraction of fixed-point numbers coincide with addition and subtraction of the underlying stored integers. However, for multiplication and division this is not the case. E.g., multiplying $\tilde{c} = c \cdot 2^{-k}$ and $\tilde{d} = d \cdot 2^{-k}$ involves first multiplying $\tilde{c}$ and $\tilde{d}$ but then also truncating the last k bits. It is not clear how this truncation operation can be efficiently represented as an arithmetic circuit. The literature does not seem to provide any solutions to this problem.

Here we provide tailored solutions to efficiently prove correctness of fixed-point multiplication and division. These solutions apply both in the case where the prover is a single party that knows which values it is building proofs on, but also in the distributed setting (e.g., [107, 108]) where work is distributed between multiple provers in a privacy-preserving way.

Fixed-point multiplications and divisions occur in many natural applications. In particular, in the privacy-preserving setting, applications include linear programming [24] for multiplication and the logrank test for Kaplan-Meier survival analysis (in this project) for division. The logrank test for testing difference between two Kaplan-Meier survival curves (or more generally, any χ^2 test) requires proving correctness of fixed-point divisions and multiplications. In a proof-of-concept implementation,

we have demonstrated that the techniques can be used to provide cryptographic proof of correctness of this logrank test.

11.1 Approach

11.1.1 Multiplication

Suppose natural numbers c and d represent fixed-point values $\tilde{c} = c \cdot 2^{-k}$ and $\tilde{d} = d \cdot 2^{-k}$, and we have computed e such that $\tilde{e} = e \cdot 2^{-k} \approx \tilde{c} \cdot \tilde{d}$. For e to be correct, we need to prove that e was provided by a correct truncation. The crucial observation underlying this intervention is the fact that correctness is captured by the property that $e - c \cdot d \in [-2^k, 2^k]$, or equivalently, that $e - c \cdot d + 2^k \geq 0$ and $2^k - (e - c \cdot d) \geq 0$; and that this property can be verified efficiently.

This suggests the following procedure. We prove that the two $(\leq k+1)$-bit numbers $e - c \cdot d + 2^k$ and $2^k - (e - c \cdot d)$ are positive. We do this (in the arithmetic circuit setting) by making $(k+1)$-bit bit decompositions of the two numbers and proving that they are correct. Namely, we determine the bits $a_0, \ldots, a_k$ of, e.g., $e - c \cdot d + 2^k$ and prove that i) all a_i are bits, by proving that $a_i \cdot (1 - a_i) = 0$; ii) $a_0, \ldots, a_k$ is indeed a bit decomposition of $e - c \cdot d + 2^k$, by proving that $e - c \cdot d + 2^k = a_0 + a_1 \cdot 2 + \ldots + a_k 2^k$.

Working this out in a straightforward way, we get the following procedure:

1. **Given**: c, d, e, where e is the fixed-point multiplication of c, d
2. Determine $e - c \cdot d + 2^k$, $2^k - (e - c \cdot d)$.
3. Compute bit decompositions $a_0, \ldots, a_k$, $b_0, \ldots, b_k$ of those numbers. (By correctness of e, these numbers are at most $k+1$ bits long.)
4. Prove that all a_i, b_i are 0 by proving that $a_i \cdot (1 - a_i)$, $b_i \cdot (1 - b_i)$ are zero
5. Prove that all the bit decompositions are correct by proving that $e - c \cdot d + 2^k = a_0 + a_1 \cdot 2 + \ldots + a_k 2^k$, $2^k - (e - c \cdot d) = b_0 + b_1 \cdot 2 + \ldots + b_k 2^k$.

11.1.2 Division

Say values stored natural numbers c and d represent fixed-point values $\tilde{c} = c \cdot 2^{-k}$ and $\tilde{d} = d \cdot 2^{-k}$, and we have computed e such that $\tilde{e} = e \cdot 2^{-k} \approx \tilde{c}/\tilde{d}$.

For e to be correct, we need that $\tilde{c} \approx \tilde{d}\tilde{e}$, so $c2^{-k} \approx d2^{-k} \cdot e2^{-k}$, so $c \cdot 2^k \approx d \cdot e$. The crucial observation in this case is that a one-off error in e increases the value of $c \cdot 2^k - d \cdot e$ by d, so for e to be correct we need $c \cdot 2^k - d \cdot e \in [-d, d]$; and this can be efficiently verified if an upper bound on the bitlength of d is known. Namely, we prove this by proving that $d + c \cdot 2^k - d \cdot e \geq 0$ and $d + d \cdot e - c \cdot 2^k \geq 0$. If d is at most K bits (i.e., $|\tilde{d}| \leq 2^{K-k}$)), then both numbers are at most $K+1$ bits so we need a $K+1$-bit bit decomposition. Note that we require an upper bound on the size of d — but in practical applications such an upper bound is often available.

Hence similarly to above we obtain the following procedure:

1. **Given**: c, d, e, where e is the fixed-point division of c by d, and an upper bound K on the bitlength of d
2. Determine $d + c \cdot 2^k - d \cdot e$ and $d + d \cdot e - c \cdot 2^k$

3. Compute bit decompositions $a_0, \ldots, a_K$, $b_0, \ldots, b_K$ of those numbers. (By correctness of e these numbers are at most $K+1$ bits long.)

4. Prove that all a_i, b_i are 0 by proving that $a_i \cdot (1-a_i)$, $b_i \cdot (1-b_i)$ are zero

5. Prove that all the bit decompositions are correct by proving that $d + c \cdot 2^k - d \cdot e = a_0 + a_1 \cdot 2 + \ldots + a_K 2^K$, $d + d \cdot e - c \cdot 2^k = b_0 + b_1 \cdot 2 + \ldots + b_K 2^K$.

11.2 Verifiable Fixed-Point Division

Realizations can be classified among two axes: first, what kind of proof system is used, and second, whether or not the sensitive data is hidden from the worker.

Concerning the kind of proof system, we need a commitment (or encryption) scheme and zero-knowledge proofs (or arguments) that commitments satisfy polynomial relations. Notable possible proof systems are:

- For computations on the integers (not modulo): use the Fujisaki-Okomoto commitment scheme, homomorphic addition, and the multiplication proofs from [56]
- Use an additively homomorphic commitment scheme modulo a prime, such as Pedersen commitments or ElGamal encryption [106], and proofs of correct multiplication based on Σ-protocols
- Use the Pinocchio verifiable computation scheme and model the proofs as a "quadratic arithmetic program" [102]

We now present two cases based on whether or not sensitive data is hidden from the worker.

One case does not hide sensitive data from the worker. For instance, consider a scenario where a medical researcher wants to prove correctness of a statistical test on a dataset, where the researcher has access to the dataset but does not want to disclose it to a reviewer. In this scenario, the researcher is the "prover" and the reviewer is the "verifier". To prove correctness of the statistical test, different computations need to be verified, e.g., additions and subtractions, but also fixed-point multiplications. At this point, the prover has already proven correct the computation of the committed values to be multiplied or divided; as a next step, the correctness of this multiplication or division needs to be proven.

This case can be based on all three proof systems mentioned above. The procedure for multiplication/division is implemented as follows:

1. **Given**: prover knows c, d, e; prover and verifier know respective commitments C, D, E

2. The prover provides a commitment to $c \cdot d$ (for multiplication) or $d \cdot e$ (for division) to the verifier

3. The prover provides commitments to the bits of the bit decompositions to the verifier

4. The prover proves correctness of the bit decomposition commitments using the zero-knowledge proofs (or arguments), which the verifier verifies

5. The prover proves correspondence of the bit decomposition commitments using the zero-knowledge proofs (or arguments), which the verifier verifies

Another case does hide the sensitive data. In the medical research scenario, also the researcher is not allowed to see the dataset. Hence, the computation is done on behalf of the researcher by multiple cloud computation parties (Amazon, Google, Microsoft). These cloud computation parties act as "provers" where the researcher and reviewers can both check the results from the cloud computation parties hence act as "verifiers".

This case is similar to the previous one, except that instead of a single prover holding the data, the data is now secret-shared between multiple provers and proofs are computed in a distributed way. For instance, using Shamir secret sharing between three provers. This case can be based on the second and third proof systems above. The steps for multiplication/division are performed as follows:

1. **Given**: c, d, e Shamir secret-shared between multiple provers; provers and verifier know respective commitments C, D, E. (Note: e is computed by MPC fixed-point multiplication/division protocols [45])
2. The provers run an MPC protocol to obtain shars of $c \cdot d$ (for multiplication) or $d \cdot e$ (for division); provide commitment shares to the verifier who reconstructs the commitment
3. The provers run an MPC protocol (cf. [45]) to obtain secret-shared bit decompositions of the appropriate values; provide commitment shares to the verifier who reconstructs the commitment
4. The provers prove correctness of the bit decomposition commitments using the zero-knowledge proofs (or arguments) in a distributed way (ElGamal: [46]; Pinocchio: [108]; Pedersen: natural extension to [46]), which the verifier verifies
5. The prover proves correspondence of the bit decomposition commitments using the zero-knowledge proofs (or arguments), which the verifier verifies

Note: to use the first (Fujisaki-Okomoto) commitment scheme while hiding sensitive inputs from provers, MPC over the integers [115] can be used. It is not clear if the bit decomposition protocols from [45] translate to this setting — if they do, then a privacy-preserving variant based on the first commitment scheme is also possible.

12 Revisiting the Implementation of Finite-Field Arithmetic Modulo p

Several software libraries for multiprecision arithmetic exist. One of the most well-known is the *GNU Multi-Precision library*, or GMP [66]. Higher-level libraries for multiprecision arithmetic and number theory, like Victor Shoup's *Number Theory Library* (NTL) [112] rely on GMP for high-performance basic multiprecision arithmetic.

Many modern cryptographic applications, like elliptic-curve-based schemes and multiparty computation frameworks, require multiprecision arithmetic with numbers whose bit-lengths are in the order of 100–500 bits. For this bit-range, the use of multiprecision-arithmetical functions designed for arbitrary bit-lengths (meaning that storage must be allocated dynamically) is typically overkill and could incur a significant runtime overhead; fixed-size arrays seem the better choice. For fixed-bitsize multiprecision numbers, GMP offers low-level functions whose names are prefixed with "`mpn_`". Yet other multiprecision-arithmetic libraries wrap those low-level GMP functions into a more convenient interface and augment them with additional functionality. An example is the LIBFF finite-field library from Virza et al. [103], which is used in the Zcash implementation of the Zerocash protocol.

The code base of GMP consists of a reference implementation in C, combined with many parts written in hand-optimized assembly for specific CPU targets. Also portions of LIBFF are written in hand-optimized assembly code. Motivated by the ongoing development of high-level systems programming languages (e.g., Modern C++, D, Rust) as well as optimizing compilers (e.g., LLVM [88]), we investigate how close we can get to the run-time performance of existing hand-optimized assembly implementations by means of implementing a library of basic routines for multiprecision arithmetic [19] in Standard C++17, at the time of writing the most recent C++ language dialect. In some sense, our implementations might be viewed as an attempt to write "optimizing-compiler-friendly" C++ code.

In this work, we do not address the question of whether our code is constant-time, and leave this as further work. Some multiprecision-arithmetic libraries targeted at cryptography applications use assembly language to have more control over constant-time behavior. On the other hand, there is some fairly recent work on formal verification of constant-time C code, e.g. [1]. Unfortunately, the tool presented in [1], `ct-verif`, does not (yet) seem to be compatible with C++.

Note that in this work we do not focus on achieving speedups by attempting to improve the algebraic complexity of multiprecision arithmetic; such efforts can be viewed as being "orthogonal" to (and would still benefit from) the techniques presented here.

An important aspect of our design is the frequent use of `std::array`, the fixed-size array type (introduced in C++11) from the Standard Library. We use `std::array` to pass multiprecision integers into functions, which gives the compiler full knowledge about the length of the multiprecision integer, hence enabling more opportunities for optimization.

Our library is header-only, mostly because of our frequent use of C++ templates. An advantage of the header-only property is that the compiler can inline function bodies of our library into client code (which can result in significant performance gains), whereas for separately compiled libraries (such as NTL) inlining cannot occur without enabling *link-time optimization* (LTO). Note that one drawback of using a library with lots of templated code is an increase in compilation times.

We highlight several aspects where our library achieves an improvement in terms of runtime performance or usability over state-of-the-art libraries for multiprecision and finite-ring/finite-field arithmetic. We want to emphasize that our library should not be viewed as a full replacement for, say, GMP, NTL or LIBFF, because our library lacks many of the functionalities that such (more mature) libraries provide.

Compile-Time Computations. *Constant expressions* are a notable feature of Modern C++ (introduced in C++11, but extended in more recent versions of the language). Constant expressions, marked by the `constexpr` keyword, are expressions that can be evaluated at run time as well as at compile time. We have paid special attention to ensuring that all of our implementations of the basic algorithms for multiprecision arithmetic are valid `constexpr` functions. By doing so (and with the help of some Template Metaprogramming [118]), we have created the first library for *compile-time multiprecision arithmetic*. Such compile-time computations can be convenient for specifying pre-computations along with the rest of the code, without paying any runtime overhead for these pre-computations. For example, suppose that in some function we want to hard-code a multiprecision integer, represented as a human-readable (base-10) digit string. NTL, for example, will parse such a string into a multiprecision integer at runtime, and this happens every time the function is executed, unless the programmer takes explicit care of caching the parsed number. With our library, the string representation of the big integer is parsed at compile time, hence incurs zero runtime overhead. Another example is arithmetic modulo some integer q, where the modulus q is constant throughout the program. The reduction modulo q is typically a computationally expensive operation, for which special methods have been devised, like Barrett reduction or the use of Montgomery representation [96]. Our library directly supports to "hard-code" this modulus into the Barrett reduction or Montgomery reduction procedures, and to perform the required modulus-dependent pre-computation at compile-time.

12.1 "Compiler-Friendly" Function-Parameter Passing

One aspect of our work is the design of the function prototypes, in other words, the way in which the arguments are passed into, and out of, a function.

Let us first illustrate how a state-of-the-art library for multiprecision arithmetic, GMP, passes multiprecision integers into functions. In GMP, the prototype of the `mpn_add_n` function, for adding equi-sized multiprecision integers looks as follows:

```
mp_limb_t mpn_add_n (mp_ptr rp, mp_srcptr up, mp_srcptr vp, mp_size_t n);
```

The names `mp_ptr` and `mp_srcptr` represent pointer types, hence, the function takes three pointers, pointing to the beginning of the memory of the first and second operand and of the result, and it takes the length `n`. Note that `mp_limb_t` represents the data type of the limbs.

In our library, the prototype of our function for adding equi-sized multiprecision integers looks as follows.

```
template <typename T, size_t N>
constexpr auto add(big_int<N, T> a, big_int<N, T> b);
```

The `big_int` type is a thin wrapper around `std::array`; a `big_int<N, T>` type may safely be read as a `std::array<T, N>` type. Note that by using this function prototype, the compiler knows exactly the sizes of the inputs. This allows for more optimization opportunities, which sometimes leads to faster code, as indicated by the benchmarks in Section 12.4.1. The arguments are passed by value, and the return type, `big_int<N + 1, T>`, is returned by value as well. Although the use of pass-by-value might give the impression that many unnecessary copies will be made, from experiments we have observed that a modern optimizing compiler removes/elides unnecessary copies (and in some cases, C++17 actually guarantees copy elision).

12.2 Compile-Time "Tricks"

12.2.1 Representing Compile-Time Arguments

To represent multiprecision integers that are guaranteed to be known at compile time, we use *parameter packs* (a C++11 feature) which give rise to *variadic templates* (a template class or template function with a variable number of template parameters). To conveniently pass those parameter packs as "ordinary" function arguments, we leverage the `std::integer_sequence` type, which was introduced in C++14.

An example of the use of `std::integer_sequence` is in our function that implements Barrett reduction:

```
template <typename T, size_t N, T... Modulus>
constexpr auto
barrett_reduction(big_int<N, T> x, std::integer_sequence<T, Modulus...>);
```

Here, note that `Modulus` is a variadic template, as indicated by the ellipsis, but can be passed as the second function argument.

12.2.2 Enforcing Compile-Time Execution of `constexpr` Functions

We also use the `std::integer_sequence` type for another, yet related, purpose. Given any `constexpr` function, and constant-expression arguments to that function, the compiler *may* decide to execute that function at compile time, but is not forced to do so. Sometimes, we want to enforce the compiler to perform some computation at compile time. One "trick" to achieve this is to let that `constexpr`-function produce a result of type `std::integer_sequence`, which is then used at another place as input argument.

12.2.3 Easy Initialization from a Literal

With the C++11 feature *User-Defined Literals*, we can define a custom suffix (we have chosen the suffix `_Z`) for defining multiprecision-integer literals in base-10 notation, which gets parsed at compile-time.

Example.

```
auto modulus = 144740111546645463731260070504932198000989141205031_Z;
```

The `_Z` user-defined literal returns an `std::integer_sequence`, which means that this return value is guaranteed to be a constant expression. If desired, the object can then be converted into a `big_int` type.[8]

12.2.4 Fast Modular Reductions with Automatic Compile-Time Precomputations

In Section 12.2.1 we have already shown the function prototype of our `barrett_reduction` function. Let us now show how to invoke it, with a modulus that is known at compile time:

[8] Whether a `big_int` type qualifies as a constant expression depends on the context where it appears in the code. For example, if a `big_int` is created in the body of a `constexpr` function, then it qualifies as such; if it is passed into a `constexpr` function as a function argument, then it does not qualify as a constant expression inside the `constexpr` function.

```
auto num = to_bigint(54766245287875756986436_Z);
auto reduced = barrett_reduction(number, 118059162071741130344 9_Z);
assert(reduced == to_bigint(459030734874837027782_Z));
```

Because `barrett_reduction` is a `constexpr` function, the following code is also valid:

```
constexpr auto num = to_bigint(54766245287875756986436_Z);
constexpr auto reduced = barrett_reduction(number, 118059162071741130344 9_Z);
static_assert(reduced == to_bigint(459030734874837027782_Z));
```

Performing computations in the finite ring $\mathbb{Z}/q\mathbb{Z}$. When working in a finite ring or finite field with a fixed modulus, the latter should be defined once and for all. For this scenario, we have created the `ZqElement` class, which supports operations in the ring $\mathbb{Z}/q\mathbb{Z}$.

First, we declare the type of the ring by calling the `Zq` template function, which creates a dummy instance of the `ZqElement` class, of which we take the type using C++'s `decltype` keyword:

```
using GF101 = decltype(Zq(1267650600228229401496703205653_Z));
// define the type of a 101-bit prime field
```

Now, we can create instances of our newly created type, and perform arithmetic using the overloaded operators in the ring:

```
GF101 x(873219109665139280029863897 6_Z);
GF101 y(27349736_Z);

auto sum = x + y;
auto prod = x * y;
```

12.3 Composition with Higher-Level Libraries

Because our `big_int` type is essentially an array, it is easy and safe (i.e., no risk of memory leaks) to use the type inside some other type, like a standard library container, such as `std::vector`.

Example: Matrix Multiplication via the Eigen library. Suppose that we need to use matrices, and the associated operations, like matrix multiplication. NTL, for example, defines its own dedicated vector and matrix class. As an alternative, we would like to compose our `big_int` or `Zq_Element` type with an existing matrix class. As a concrete working example, let us consider *Eigen* [67], a C++ library for linear algebra that offers a templated Matrix class whose element type can be specified as a template parameter. Suppose that we would like to define matrix multiplication in the field modulo the 101-bit prime $p = 1267650600228229401496703205653$.

The first step is to provide Eigen with information about our type by means of a traits class. We can do this by means of some "boilerplate" code, which we will not show here. Then, we can define the matrix type (in this example, a fixed-size 3-by-3 matrix) over our field and perform an actual computation:

```
using Mx33 = Eigen::Matrix<GF101, 3, 3>;
Mx33 A, B, C;
```

```
A <<  2_Z,   4_Z,  6_Z,
     10_Z,  11_Z, 12_Z,
      1_Z, 100_Z, 30_Z;

B << 5_Z,  3_Z,  9_Z,
     8_Z,  6_Z, 55_Z,
     3_Z, 17_Z,  2_Z;

C = A * B;
```

12.4 Running-Time Benchmarks

We have run the benchmarks described below on a MacBook Pro (2017) quad-core 2300 MHz Intel Core i5 7th generation "Kaby Lake" processor with 16 GB memory. The "Turbo Boost" feature (which is a dynamic overclocking feature on some Intel CPUs) has been explicitly disabled using a small program called "Turbo Boost Switcher". We have used Apple's LLVM compiler Version 9.0.0, with `-O3` optimizations; we did not enable LTO. For obtaining the actual timings, we have used Google's Benchmark library [51]. The timings that we show are the user-CPU timings (as opposed to the "wall-clock timings"), represented as *mean*±*std.dev.*, obtained using the `-benchmark_repetitions=5` option. The minimum value in a table row is shown in boldface.

Note that we apply the functions that we benchmark to random inputs *generated at runtime*, to rule out the possibility that the compiler can already compute the result at compile time (in that case, the code that we want to benchmark would disappear from the benchmark's hot loop).

12.4.1 Multiplication

Table 12.4.1 shows benchmarks for multiplication. We compare between GMP, NTL and our code. Note that the multiplication function in GMP (and NTL, which uses GMP as a back-end) perform some case distinctions at runtime (to decide which algorithm to use). Hence, we also compare to the internal function `__gmpn_mul_basecase` (which is not part of GMP's public API), which is a multiplication function tailored to small operands.[9] The benchmarks indicate that our implementation slightly beats GMP's `__gmpn_mul_basecase` for small n (the number of limbs). Beyond 4 limbs our implementation starts losing from GMP in terms of speed. By compiling with `-march=native` and by enforcing function-inlining, we can somewhat improve the performance of our implementation, especially for 7 and 8 limbs. Note that such inlining can be enforced using compiler attributes: `[[gnu::always_inline]]`. Results obtained by using different combinations of the two compilation options are shown in Table 9, 10 and 11.

12.4.2 Montgomery Multiplication

With respect to Montgomery multiplication, we compare our library against LIBFF. Table 12 shows a benchmark for Montgomery multiplication of 4-limb operands. The 4-limb specialization of the `mul_reduce` function of LIBFF is written by hand in assembly, and implements the CIOS method (Coarsely Integrated Operand Scanning) [84]. From inspecting the compiler-generate assembly code of the benchmark program, we have observed that LIBFF's Montgomery multiplication code gets inlined. We beat the performance of LIBFF by forcing the compiler to also inline our `montgomery_mul`

[9] We wish to thank Fredrik Johansson for pointing this out.

Table 8: Time (CPU time±stddev) spent to multiply two n-limb operands (limb size: 64-bit). Comparison between: (a) our implementation of $O(n^2)$-time schoolbook multiplication, (b) the same algorithm but then defined inside a function with a "pointer-to-array + array-size"-argument-passing-style function prototype (as opposed to using `std::array`), (c) GMP's `mpn_mul` function, (d) GMP's internal `__gmpn_mul_basecase` function (not part of GMP's public API) and (e) NTL's `mul` function.

n	Our library	pointer-style f.p.	GMP	GMP (internal)	NTL
2	**5** ± 0 ns	**5** ± 0 ns	13 ± 0 ns	**5** ± 0 ns	23 ± 0 ns
3	**11** ± 0 ns	12 ± 0 ns	22 ± 0 ns	14 ± 0 ns	34 ± 0 ns
4	**18** ± 0 ns	20 ± 0 ns	29 ± 0 ns	20 ± 0 ns	39 ± 3 ns
5	38 ± 0 ns	40 ± 0 ns	36 ± 0 ns	**28** ± 0 ns	48 ± 1 ns
6	55 ± 0 ns	57 ± 0 ns	45 ± 0 ns	**37** ± 0 ns	54 ± 0 ns
7	73 ± 1 ns	76 ± 0 ns	56 ± 1 ns	**50** ± 1 ns	65 ± 0 ns
8	128 ± 1 ns	101 ± 2 ns	69 ± 0 ns	**61** ± 1 ns	76 ± 1 ns

Table 9: Same experiment as Table 12.4.1, but compiled with forced inlining of our multiplication functions.

n	Our library	pointer-style f.p.	GMP	GMP (internal)	NTL
2	**5** ± 0 ns	**5** ± 0 ns	13 ± 0 ns	**5** ± 0 ns	22 ± 0 ns
3	**11** ± 0 ns	**11** ± 0 ns	23 ± 1 ns	14 ± 0 ns	33 ± 2 ns
4	**18** ± 0 ns	20 ± 0 ns	29 ± 0 ns	20 ± 0 ns	38 ± 0 ns
5	38 ± 0 ns	40 ± 0 ns	36 ± 0 ns	**28** ± 0 ns	45 ± 0 ns
6	55 ± 0 ns	57 ± 0 ns	45 ± 0 ns	**37** ± 0 ns	54 ± 0 ns
7	73 ± 1 ns	76 ± 0 ns	56 ± 0 ns	**48** ± 0 ns	65 ± 1 ns
8	93 ± 1 ns	99 ± 0 ns	68 ± 1 ns	**60** ± 0 ns	76 ± 0 ns

Table 10: Same experiment as Table 12.4.1, but compiled with `-march=native` flag.

n	Our library	pointer-style f.p.	GMP	GMP (internal)	NTL
2	**4** ± 0 ns	11 ± 0 ns	13 ± 0 ns	5 ± 0 ns	22 ± 0 ns
3	**9** ± 0 ns	15 ± 0 ns	23 ± 1 ns	14 ± 0 ns	32 ± 0 ns
4	**18** ± 0 ns	23 ± 0 ns	29 ± 0 ns	20 ± 0 ns	39 ± 1 ns
5	34 ± 0 ns	40 ± 0 ns	36 ± 0 ns	**28** ± 0 ns	45 ± 1 ns
6	50 ± 0 ns	57 ± 0 ns	46 ± 1 ns	**38** ± 0 ns	54 ± 1 ns
7	69 ± 1 ns	76 ± 1 ns	56 ± 1 ns	**48** ± 0 ns	66 ± 2 ns
8	102 ± 0 ns	99 ± 0 ns	68 ± 0 ns	**60** ± 0 ns	76 ± 0 ns

Table 11: Same experiment as Table 12.4.1, but compiled with `-march=native` flag and forced inlining of our multiplication functions.

n	Our library	pointer-style f.p.	GMP	GMP (internal)	NTL
2	**4** ± 0 ns	11 ± 0 ns	13 ± 0 ns	5 ± 0 ns	22 ± 0 ns
3	**9** ± 0 ns	15 ± 0 ns	22 ± 0 ns	14 ± 0 ns	31 ± 0 ns
4	**18** ± 0 ns	23 ± 0 ns	29 ± 0 ns	20 ± 0 ns	38 ± 0 ns
5	34 ± 0 ns	41 ± 0 ns	36 ± 0 ns	**28** ± 0 ns	46 ± 0 ns
6	50 ± 0 ns	57 ± 1 ns	45 ± 0 ns	**37** ± 0 ns	54 ± 0 ns
7	66 ± 0 ns	77 ± 1 ns	56 ± 0 ns	**48** ± 1 ns	65 ± 1 ns
8	85 ± 0 ns	99 ± 0 ns	67 ± 0 ns	**60** ± 0 ns	76 ± 0 ns

Table 12: Time (CPU time) spent to perform Montgomery multiplication two 4-limb operands (limb size: 64-bit). Comparison between our "textbook implementation" of Montgomery multiplication (following [96, Chap. 14]), and LIBFF's `mul_reduce` function.

n	Our library		LIBFF `mul_reduce`
	(no inlining)	(enforced inlining)	
4	87 ns	39 ns	66 ns

code. If we do not enforce inlining, then LIBFF's (inlined) assembly code is faster, which seems to indicate that in this case the gap in running time is merely due to function-call overhead.

In other words, it seems that the use of hand-written inline assembly in LIBFF triggered the compiler to inline the multiplication function in our benchmark program, and that this inlining operation achieves a performance lead over NTL; LIBFF's hand-written assembly language for the 4-limb case in the multiplication function (that is the one that we benchmarked) actually seems to be slower than what the compiler generates out of our C++ implementation.

12.4.3 Modular Exponentiation

Table 12.4.3 shows benchmarking results for modular exponentiation. Given a fixed 200-bit modulus, we raise a 195 bit operand to an exponent of 122 bits. Again, we observed that applying forced inlining of `montgomery_mul` (a subroutine of our modular-exponentiation function `mod_exp`) has a big impact on the performance.

Table 13: Comparison of modular exponentiation: given a modulus of 200 bits, we raise a 195 bit operand to an exponent of 122 bits. We compare our "textbook implementation" of Montgomery-multiplication-based modular exponentiation (following [96, Chap. 14]), to NTL's `power` function and LIBFF's power operator.

Our library		LIBFF	NTL
(no inlining)	(enforced inlining)		
15474 ns	6594 ns	10530 ns	16862 ns

12.5 Conclusion

The running-time benchmarks indicate that, with respect to multiprecision integer arithmetic, the combination of high-level code and modern compilers has become a serious alternative for hand-optimized assembly code. Furthermore, with modern high-level systems languages, e.g. Modern C++, we can create easy-to-use APIs without paying in terms of runtime overhead.

The functions in our library enable multi-precision and modular arithmetic at *compile-time*, which is useful when all inputs to some computation are known at compile-time (and in this case those computations will have zero cost at runtime). For our targeted small-number-of-limbs scenario, the *run-time* performance of the *same* functions is competitive with existing state-of-the-art libraries for multiprecision arithmetic. The code base of our library is pure C++ and much more concise than, say, the parts of GMP that implement the same functionality.

We have released our work as an open-source library [19] under a permissive license (Apache 2) and we invite others to contribute to its development. At the same time, we hope that some of the techniques presented in this work can contribute to or influence the development of the existing software libraries for cryptography and number theory.

References

[1] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, 2016. USENIX Association.

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 805–817. ACM Press, October 2016.

[3] Gilad Asharov, Daniel Demmler, Michael Schapira, Thomas Schneider, Gil Segev, Scott Shenker, and Michael Zohner. Privacy-preserving interdomain routing at internet scale. *PoPETs*, 2017(3):147, 2017.

[4] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Heidelberg, April 2012.

[5] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In *EUROCRYPT*, pages 673–701, 2015.

[6] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A fast provably secure cryptographic hash function. *IACR Cryptology ePrint Archive*, 2003:230, 2003.

[7] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *2015 IEEE Symposium on Security and Privacy*, pages 271–286. IEEE Computer Society Press, May 2015.

[8] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.

[9] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[10] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 578–590. ACM Press, October 2016.

[11] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[12] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[13] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 663–680. Springer, Heidelberg, August 2012.

[14] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[15] Daniel Bleichenbacher and Phong Q. Nguyen. Noisy polynomial interpolation and noisy Chinese remaindering. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 53–69. Springer, Heidelberg, May 2000.

[16] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Heidelberg, October 2008.

[17] Dan Boneh. Finding smooth integers in short intervals using CRT decoding. In *32nd ACM STOC*, pages 265–272. ACM Press, May 2000.

[18] BOOST documentation. Sequence algorithms, 2012. http://erikerlandson.github.io/algorithm/libs/algorithm/doc/html/algorithm/Sequence.html.

[19] Niek J. Bouman et al. CTBignum: C++ library for compile-time and run-time big-integer and modular arithmetic, 2018. https://github.com/niekbouman/ctbignum.

[20] Gabriel Bracha. An $O(\lg n)$ expected rounds randomized byzantine generals protocol. In *17th ACM STOC*, pages 316–326. ACM Press, May 1985.

[21] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. *IACR Cryptology ePrint Archive*, 2015:472, 2015.

[22] Ignacio Cascudo, Ivan Damgård, Bernardo David, Nico Döttling, and Jesper Buus Nielsen. Rate-1, linear time and additively homomorphic UC commitments. In *CRYPTO*, pages 179–207, 2016.

[23] Ignacio Cascudo, Ivan Damgård, Bernardo Machado David, Irene Giacomelli, Jesper Buus Nielsen, and Roberto Trifiletti. Additively homomorphic UC commitments with optimal amortized overhead. In *PKC*, pages 495–515, 2015.

[24] Octavian Catrina and Sebastiaan de Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS 2010*, volume 6345 of *LNCS*, pages 134–150. Springer, Heidelberg, September 2010.

[25] Yan-Cheng Chang and Chi-Jen Lu. Oblivious polynomial evaluation and oblivious neural learning. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 369–384. Springer, Heidelberg, December 2001.

[26] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988.

[27] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *CRYPTO 2018*, 2018. https://eprint.iacr.org/2018/570.

[28] Gil Cohen, Ivan Bjerre Damgård, Yuval Ishai, Jonas Kölker, Peter Bro Miltersen, Ran Raz, and Ron Rothblum. Efficient multiparty protocols via log-depth threshold formulae. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:107, 2013.

[29] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 466–485. Springer, Heidelberg, December 2014.

[30] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of NSDI*, 2017.

[31] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE Computer Society Press, May 2015.

[32] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO 2018*, 2018.

[33] Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 596–613. Springer, Heidelberg, May 2003.

[34] Ivan Damgård, Bernardo Machado David, Irene Giacomelli, and Jesper Buus Nielsen. Compact VSS and efficient homomorphic UC commitments. In *ASIACRYPT*, pages 213–232, 2014.

[35] 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 Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[36] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the minimac protocol for secure computation. In *SCN*, pages 398–415, 2014.

[37] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[38] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In *CRYPTO*, pages 558–576, 2010.

[39] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO 2018*, 2018.

[40] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[41] Ivan Damgård and Rune Thorbek. Non-interactive proofs for integer multiplication. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 412–429. Springer, Heidelberg, May 2007.

[42] Ivan Damgård and Rasmus Winther Zakarias. Fast oblivious AES A dedicated application of the minimac protocol. In *AFRICACRYPT*, pages 245–264, 2016.

[43] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.

[44] Bernardo Machado David, Ryo Nishimaki, Samuel Ranellucci, and Alain Tapp. Generalizing efficient multiparty computation. In Anja Lehmann and Stefan Wolf, editors, *ICITS 15*, volume 9063 of *LNCS*, pages 15–32. Springer, Heidelberg, May 2015.

[45] Sebastiaan de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming*. PhD thesis, Eindhoven University of Technology, 2012.

[46] Sebastiaan de Hoogh, Berry Schoenmakers, and Meilof Veeningen. Certificate validation in secure computation and its use in verifiable linear programming. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 265–284. Springer, Heidelberg, April 2016.

[47] Yvo Desmedt and Kaoru Kurosawa. How to break a practical MIX and design a new one. In *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 557–572. Springer, 2000.

[48] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *NDSS*, 2017.

[49] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 2263–2276. ACM Press, October / November 2017.

[50] Nico Döttling, Daniel Kraschewski, and Jörn Müller-Quade. David & Goliath oblivious affine function evaluation - asymptotically optimal building blocks for universally composable two-party computation from a single untrusted stateful tamper-proof hardware token. Cryptology ePrint Archive, Report 2012/135, 2012. http://eprint.iacr.org/2012/135.

[51] Dominic Hamon Eric Fiselier et al. Google Benchmark - a library to support the benchmarking of functions, 2014. https://github.com/google/benchmark.

[52] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 1304–1316. ACM Press, October 2016.

[53] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710. ACM Press, May 1992.

[54] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic UC commitments. In *TCC*, pages 542–565, 2016.

[55] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *ASIACRYPT*, pages 711–735, 2015.

[56] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 16–30. Springer, Heidelberg, August 1997.

[57] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 225–255, 2017.

[58] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, May 2017.

[59] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[60] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *17th ACM PODC*, pages 101–111. ACM, June / July 1998.

[61] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[62] Satrajit Ghosh, Jesper Buus Nielsen, and Tobias Nilges. Maliciously secure oblivious linear function evaluation with constant overhead. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 629–659. Springer, Heidelberg, December 2017.

[63] Niv Gilboa. Two party RSA key generation. In *CRYPTO*, pages 116–129, 1999.

[64] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[65] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[66] Torbjörn Granlund et al. GNU MP 6.1.2 multiple precision arithmetic library, 2016. https://gmplib.org/.

[67] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[68] Carmit Hazay. Oblivious polynomial evaluation and secure set-intersection from algebraic PRFs. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 90–120. Springer, Heidelberg, March 2015.

[69] Carmit Hazay and Yehuda Lindell. Efficient oblivious polynomial evaluation with simulation-based security. Cryptology ePrint Archive, Report 2009/459, 2009. http://eprint.iacr.org/2009/459.

[70] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. TinyKeys: A new approach to efficient multi-party computation. In *CRYPTO 2018*, 2018.

[71] 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, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[72] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[73] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 600–620. Springer, Heidelberg, March 2013.

[74] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, Heidelberg, August 2008.

[75] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, Heidelberg, March 2009.

[76] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 955–966. ACM Press, November 2013.

[77] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 115–128. Springer, Heidelberg, May 2007.

[78] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[79] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 830–842. ACM Press, October 2016.

[80] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. LNCS, pages 158–189. Springer, Heidelberg, 2018.

[81] Aggelos Kiayias and Moti Yung. Cryptographic hardness based on the decoding of reed-solomon codes. *IEEE Trans. Information Theory*, 54(6):2752–2769, 2008.

[82] Joe Kilian. Founding cryptography on oblivious transfer. In *20th ACM STOC*, pages 20–31. ACM Press, May 1988.

[83] P. Klier. Method for computing the minimum edit distance with fine granularity suitably quickly, December 25 2007. US Patent 7,313,555.

[84] Ç. Kaya Koç, Tolga Acar, and Burton S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[85] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 54–70. Springer, Heidelberg, August 2013.

[86] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, Heidelberg, August 2014.

[87] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In *CRYPTO*, pages 495–512, 2014.

[88] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[89] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[90] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 259–276. ACM Press, October / November 2017.

[91] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 259–276. Springer, Heidelberg, August 2011.

[92] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, Heidelberg, August 2015.

[93] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from BMR and SHE. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 554–581. Springer, Heidelberg, October / November 2016.

[94] Helger Lipmaa. Prover-efficient commit-and-prove zero-knowledge SNARKs. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 185–206. Springer, Heidelberg, April 2016.

[95] Ueli M. Maurer. Secure multi-party computation made simple (invited talk). In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 14–28. Springer, Heidelberg, September 2003.

[96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[97] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 591–602, 2015.

[98] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM STOC*, pages 245–254. ACM Press, May 1999.

[99] Moni Naor and Benny Pinkas. Oblivious polynomial evaluation. *SIAM J. Comput.*, 35(5):1254–1281, 2006.

[100] 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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[101] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *NDSS*, 2017.

[102] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.

[103] Aleksejs Popovs, Eran Tromer, Madars Virza, Howard Wu, et al. C++ library for finite fields and elliptic curves. https://github.com/scipr-lab/libff.

[104] Michael O Rabin. How to exchange secrets with oblivious transfer. 1981.

[105] Peter Rindal and Roberto Trifiletti. Splitcommit: Implementing and analyzing homomorphic UC commitments. *IACR Cryptology ePrint Archive*, 2017:407, 2017.

[106] Berry Schoenmakers. Cryptography 2 (2WC13) / Cryptographic Protocols (2WC17) Lecture Notes, 2014. Version 1.0.

[107] Berry Schoenmakers and Meilof Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. Cryptology ePrint Archive, Report 2015/058, 2015. http://eprint.iacr.org/2015/058.

[108] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-friendly outsourcing by distributed verifiable computation. Cryptology ePrint Archive, Report 2015/480, 2015. http://eprint.iacr.org/2015/480.

[109] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 346–366. Springer, Heidelberg, June 2016.

[110] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[111] Bhavani Shankar, Kannan Srinathan, and C Pandu Rangan. Alternative protocols for generalized oblivious transfer. In *International Conference on Distributed Computing and Networking*, pages 304–309. Springer, 2008.

[112] Victor Shoup. A library for doing number theory. http://www.shoup.net/ntl.

[113] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. Cryptology ePrint Archive, Report 2013/280, 2013. http://eprint.iacr.org/2013/280.

[114] Stephen R Tate and Ke Xu. On garbled circuits and constant round secure function evaluation. *CoPS Lab, University of North Texas, Tech. Rep*, 2:2003, 2003.

[115] Rune Thorbek. *Linear Integer Secret Sharing*. PhD thesis, University of Aarhus, 2009.

[116] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient dna searching through oblivious automata. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 07*, pages 519–528. ACM Press, October 2007.

[117] Meilof Veeningen. Pinocchio-based adaptive zk-SNARKs and secure/correct adaptive function evaluation. In Marc Joye and Abderrahmane Nitaj, editors, *AFRICACRYPT 17*, volume 10239 of *LNCS*, pages 21–39. Springer, Heidelberg, May 2017.

[118] Todd Veldhuizen. Using C++ template metaprograms. In *C++ gems*, pages 459–473. SIGS Publications, Inc., 1996.

[119] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[120] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of EUROSAM '79*, 1979.