The Cost of IEEE Arithmetic in Secure Computation

David W. Archer[1], Shahla Atapoor[2], and Nigel P. Smart[2,3]

[1] Galois Inc., Portland, USA.
[2] imec-COSIC, KU Leuven, Leuven, Belgium.
[3] University of Bristol, Bristol, UK.
`dwa@galois.com`,
`shahla.atapoor@kuleuven.be`,
`nigel.smart@kuleuven.be`

Abstract. Programmers are used to the rounding and error properties of IEEE double precision arithmetic, however in secure computing paradigms, such as provided by Multi-Party Computation (MPC), usually a different form of approximation is provided for real number arithmetic. We compare the two standard variants using for LSSS-based MPC, with an implementation of IEEE compliant double precision using binary circuit-based MPC. We compare the relative performance, and conclude that the addition cost of IEEE compliance maybe too great for some applications. Thus in the secure domain standards bodies may wish to examine a different form of real number approximations.

1 Introduction

Multi-Party Computation (MPC) is a technique which enables a set of parties to compute a function on their own joint private input, whilst at the same time revealing nothing about their private inputs to each other; other than what can be deduced from the output of the function. Most MPC protocols fall into one of two broad categories: garbled circuits or linear-secret-sharing-scheme-based (LSSS-based) MPC. The garbled-circuit approach began with the work of Yao [Yao86], who gave a two party protocol in which one party 'garbles' (or encrypts) a binary circuit (along with their input) and then another party 'evaluates' the garbled circuit using their own input. By contrast, the LSSS-based approach [BGW88, CCD88] involves the parties dividing each secret value into several *shares*, over a finite field $\mathbb{F}_p$, and perform computations on the shares, and then reconstruct the secret at the end by combining the shares to determine the output.

Since their invention in the mid 1980s such technologies have come a long way. The garbled circuit (GC) based approach of Yao was generalized to the honest-majority multi-party setting by Beaver et al. [BMR90]. Where now the collection of parties "garble" the binary circuit, and then later the same collection of parties jointly evaluating the garbled circuit. Recent work [HSS17, WRK17] (which we denote by HSS and WRK respectively) have given very efficient multi-party protocols for binary circuits using this methodology for the dishonest majority setting.

The other line of LSSS based work has also had considerable recent practical advances, e.g. [BDOZ11, DPSZ12], for the case of large prime p and full threshold access structures, or [CGH+18] for the case of honest majority style access structures for a large prime. For 'small prime' LSSS based MPC Araki et al. [ABF+17] propose a method for the threshold case of $(n, t) = (3, 1)$ using replicated sharing. This method is relatively easy to generalize to any Q2 access structure represented by replicated sharing.

In all settings, the state-of-the-art is now protocols which provide so-called active (a.k.a. malicious security), namely they allow honest parties to detect when adversarial parties arbitrarily deviate from the protocol. In addition, recent work has also focused on combining the two approaches so as to get the advantages of working with function representations given by binary circuits, as well as function representations tailored to the LSSS-based approaches (working with large basic data types, i.e. integers modulo p). The work [AOR$^+$19], implemented in the Scale-Mamba system [ACC$^+$21], combines the HSS protocol [HSS17] in the GC world with either the SPDZ protocol [DPSZ12] (for full threshold adversaries) or the Smart-Wood protocol [SW19] (for non-full threshold adversaries) in the LSSS world. This conversation is itself based on so-called daBits, introduced in [RW19].

In real applications one does not want to work either directly with binary circuit based representations of functions, nor does one want to work with integers modulo p. After all many practical real world applications involve processing real numbers. But real numbers are not native datatypes in MPC systems, just as they are not native in normal computing, thus approximations have to be made. In standard (in-the-clear) computing we approximate real numbers by floating point operations; with the IEEE-754 representation being the standard.

For efficiency reasons much prior work on real number arithmetic in secure computation has mainly focused on fixed point operations. This follows from the work of Catrina and Saxena [CS10], who showed how to perform efficient fixed point operations within an LSSS-based MPC system. Over the years various authors have examined efficient LSSS-based MPC arithmetic on such fixed point numbers, e.g. [AS19, KLR16, Lie12]. However, as is well known in standard in-the-clear computation, fixed point arithmetic is not particularly suitable for many real world applications.

Using a floating point representation is possible with an MPC system. By utilizing binary circuits one can implement the IEEE-754 representation as a circuit and then execute them, so as to produce true IEEE-754 compliant secure floating point operations. However, unlike normal circuit design, the binary circuits for GC-operations are composed purely of AND, XOR and NOT gates; with the major cost associated to AND gates. Thus we need to produce circuits which are optimized for this cost metric and not the usual electronic circuit style cost metrics. We present a solution to this circuit creation problem in this work, and present gate counts and run-times of our IEEE-754 compliant circuits.

Utilizing an LSSS-based MPC system one can often obtain a more efficient form of floating point arithmetic, using the methodology of [ABZS13]. This tries to emulate the real number arithmetic, not in a computer whose native operations are modulo a power-of-two (as in IEEE-754 format), but in a computing machine whose native arithmetic is modulo a large prime number p. The methodology of [ABZS13] gives a number of tunable values, such as the usual mantissa and exponent sizes, but also a 'statistical security' parameter which controls how close 'leaked' values are from uniformly random values[4]. The paper [KW15] also implements floating point arithmetic but in a passively secure honest-majority three party setting, utilizing secret sharing modulo 2^v. The work *approximates* IEEE arithmetic using a technique similar to [ABZS13], but tuned for the case of sharing modulo 2^v. This work was extended and improved in [KLR16].

In this work, we compare the binary circuit based approach for IEEE-754 arithmetic (which follows the IEEE-754 standard with respect to round errors), with the approach of [ABZS13] which follows a different approach to rounding errors. As remarked above a modern trend is to utilize the two types of MPC systems together. Thus we also present methodologies to convert from the IEEE-754 representation to the representation used in [ABZS13]. Our work is most closely related to that of [PS15], which combines, as we do, a binary circuit approach for IEEE 754 arithmetic

[4] The closer they are to uniformly random, then the less information is leaked.

with a linear secret sharing approach for other MPC operations. They also use the CBMC-GC approach to obtain the binary circuits. However, their MPC implementation is restricted to passive security and the honest-majority three party setting (as opposed to our actively secure approach for general access structures). The evaluation of the circuits in [PS15] is performed by classical garbling, as opposed to our approach (in the honest majority setting) of utilizing an LSSS-based protocol. Thus in the three party case, we achieve roughly a five fold performance improvement for basic operations such as addition and multiplication, whilst at the same time we achieve active security.

We aim to answer the question; what is the performance penalty one incurs from requiring adherence to the IEEE-754 standard? We present run-times for all the above variants, and compare these against run-times for the fixed point representation mentioned above. We only examine the basic floating point operations of addition, multiplication and division. For higher level functions (such as sin, cos, sqrt, log etc) there are often specific protocols in the secure computation domain for these functions which differ from the 'standard' methodologies, see [AS19, ABZS13]. We conclude that the performance penalty for utilizing IEEE-754 compliant arithmetic as opposed to a tailored form of arithmetic for MPC computations may be too great.

2 Preliminaries

In this section, we introduce three forms of approximating real numbers, as well as our MPC-black box. The key set in all our methodologies is $\mathbb{Z}_{\langle k \rangle}$, which we define as the set of integers $\{x \in \mathbb{Z} : -2^{k-1} \leq x \leq 2^{k-1} - 1\}$, which we embed into $\mathbb{F}_p$ via the map $x \mapsto x \pmod{p}$, assuming $k < \log_2 p$. We let $[n]$ denote the set $\{1, \ldots, n\}$. We assume a statistical security parameter sec, which one can think of as being equal to 40. This measures the statistical distance between various distributions in the underlying protocols.

2.1 MPC-Black Box

Here we describe our two MPC systems (one GC-based and one LSSS-based) and how they fit together using the 'Zaphod' methodology given in [AOR+19]. The systems are in the active security with abort paradigm, i.e. if a malicious party deviates from the protocol then the protocol will abort with overwhelming probability.

LSSS-Based MPC: We let $[\![\cdot]\!]_p$ denote an (authenticated) linear secret sharing scheme (LSSS) over the finite field $\mathbb{F}_p$ (for a large prime p) which realizes either a full threshold access structure or a Q2-access structure (a Q2-access structure [Mau06], for readers not familiar with this terminology, can be thought of as a generalization of a threshold system in which the threshold t satisfies $t < n/2$). There are various MPC protocols that enable actively secure MPC with abort to be carried out using such an LSSS, e.g. [DPSZ12, SW19, CGH+18].

The simplest LSSS is the one which supports full threshold access structures. In this situation, a secret $x \in \mathbb{F}_p$ is held secure by n parties, by each party holding a value $x_i \in \mathbb{F}_p$ so that $x = \sum x_i \pmod{p}$. This produces an unauthenticated sharing which we denote by $\langle x \rangle_p$. To ensure correctness in the presence of active adversaries such a secret sharing needs to be authenticated with a distributed MAC value, i.e. each party also holds $\gamma_i \in \mathbb{F}_p$ such that $\alpha \cdot x = \sum \gamma_i$ for some fixed global secret MAC key α; see [DPSZ12, DKL+13] for details and how one can define MPC in this context in the pre-processing model. Such authentication is called a SPDZ-style MAC, and the combined sharing of x we denote by $[\![x]\!]_p$.

For threshold Q2-access structures, which can tolerate up to t corrupt parties out of n, where $t < n/2$, one can define the secret sharing scheme using Shamir's secret sharing [Sha79]. In this method, a secret $x \in \mathbb{F}_p$ is shared as the zero'th coefficient of a polynomial $f(X)$ of degree t with player i being given the share $f(i)$. Share reconstruction can be performed using Lagrange interpolation. Active security of the underlying MPC protocol is achieved using the error detection properties of the Reed-Solomon code associated to the Shamir sharing, see [SW19]. Thus we automatically obtain a sharing $[\![x]\!]_p$ which authenticates itself to be correct.

We can also consider LSSS-based MPC over a small prime p, for example $p = 2$. Utilizing Shamir sharing is impossible here, without using relatively expensive field extensions, thus it is common in this situation to represent the Q2-access structure via replicated sharing. This is only possible when the number of maximally unqualified sets[5] (i.e. $n!/(t! \cdot (n-t)!)$) is 'small'. For such Q2-access structures over $\mathbb{F}_2$ we use a generalization of the method of [ABF$^+$17]. This protocol uses Maurer's passively secure multiplication protocol [Mau06] to generate passively secure multiplication triples over $\mathbb{F}_2$. These are then turned into actively secure triples using the method of [ABF$^+$17][Protocol 3.1]. Finally, the triples are consumed in a standard secret sharing based online phase, using the methodology of [SW19] to ensure active security, whilst using the techniques of [KRSW18] to reduce the total amount of communication performed. Again, the use of Q2 access structures ensures that the sharing authenticates itself. We let $[\![x]\!]_2$ denote such an authenticated sharing of a bit $x \in \mathbb{F}_2$ using this replicated sharing.

GC-Based n-party MPC: For full threshold access structures we cannot use the trick of utilizing replicated sharing, thus to execute binary circuits in the full threshold case we turn to general n-party Garbled Circuit based MPC; i.e. constant round protocols based on the HSS protocol [HSS17]. Being an n-party GC protocol the data is still secret shared between the parties, but with a different sharing to that used above. We first pick a large finite field of characteristic two, in our case we select $\mathcal{K} = \mathbb{F}_{2^{128}}$. The size is determined so that an event with probability $1/|\mathcal{K}|$ can be considered negligible. For each element $x \in \mathbb{F}_2$ (resp. $\mathcal{K}$), we denote $\langle x \rangle_2$ (resp. $\langle x \rangle_{\mathcal{K}}$) the *unauthenticated* additive sharing of x over $\mathbb{F}_2$ (resp. $\mathcal{K}$), where $x = \sum_{i \in [n]} x_i$, with party P_i holding $x_i \in \mathbb{F}_2$ (resp. $\mathcal{K}$).

To obtain active security with abort we need to authenticate these sharings. However, for technical reasons, this is done using a BDOZ-style MAC introduced by Bendlin et al. [BDOZ11], as opposed to a SPDZ-style MAC, introduced in [DPSZ12]. In particular every party P_i authenticates their share b_i towards party P_j, for each $j \neq i$, by holding a MAC $M_i^j \in \mathcal{K}$, such that $M_i^j = K_j^i + b_i \cdot \Delta_j \in \mathcal{K}$, where P_j holds the local key $K_j^i \in \mathcal{K}$ and the fixed global MAC key $\Delta_j \in \mathcal{K}$. This defines an n-party authenticated representation of a bit, denoted by $[\![b]\!]_2$, where $b = \sum_{i=1}^{n} b_i$ and each P_i holds the bit-share b_i, $n-1$ MACs M_i^j, $n-1$ local keys K_i^j and Δ_i, i.e. $[\![b]\!]_2 = \{b_i, \Delta_i, \{M_i^j, K_i^j\}_{j \neq i}\}_{i \in [n]}$.

We denote by $[\![b]\!]_2$ a bit that is linearly secret shared according to $\langle \cdot \rangle_2$ and *authenticated* according to the pairwise MACs. The extension to vectors of shared bits is immediate. We let $[\![\mathbf{b}]\!]_2$ denote a secret sharing of a vector $\mathbf{b}$, the sharing of the i-bit will be denoted by $[\![\mathbf{b}^{(i)}]\!]_2$, whilst the i-bit of the clear vector $\mathbf{b}$ will be denoted by $\mathbf{b}^{(i)}$. Using such sharings one can create an n-party MPC protocol to evaluate binary circuits, see [HSS17] for details.

[5] A maximally unqualified set is a set of parties which cannot recover the secret, but for which adding an arbitrary additional player will make the set qualified.

Combining the Binary and Large Prime Variants: To combine the binary and large prime worlds we use the Zaphod methodology [AOR+19], which we briefly recap on. Our interest is in the function dependent online phase, so we focus on the online functionalities only. When using the mapping from binary to arithmetic circuits the data being transferred has bit length bounded by 64. In other words every data item x is an element of $\mathbb{Z}_{\langle 2^{64}\rangle}$. We will be interested in the 'large prime' regime of Zaphod, so we will require that $64 + \mathsf{sec} < \log_2 p$, for a statistical security parameter sec. This guarantees that selecting $\log_2 p$ bits b_i at random and then forming, for $x \in \mathbb{Z}_{\langle 2^{64}\rangle}$,

$$x + \sum_{i=0}^{\lceil \log_2 p \rceil} b_i \cdot 2^i \pmod{p}$$

will statistically hide the value of x.

Functionality $\mathcal{F}_{\mathsf{MPC}}$ - Zaphod Evaluation

The functionality runs with parties $P_1, \ldots, P_n$ and an ideal adversary Adv. Let A be the set of corrupt parties. Given a set I of valid identifiers, all values are stored in the form $(\mathit{varid}, \mathit{domain}, x)$, where $\mathit{varid} \in I$, $\mathit{domain} \in \{\mathbb{F}_2^{64}, \mathbb{F}_p\}$ and $x \in \mathit{domain}$. We assume p is restricted as in the main text.

Initialize: On input (Init) from all parties, the functionality activates.
If (Init) was received before, do nothing.

Input: On input $(\mathit{Input}, P_i, \mathit{varid}, \mathit{domain}, x)$ from P_i and $(\mathit{input}, P_i, \mathit{varid}, \mathit{domain})$ from all other parties, with varid a fresh identifier, store $(\mathit{varid}, \mathit{domain}, x)$.

Evaluate: Upon receiving $(\{\mathit{varid}_j\}_{j\in m}, \mathit{varid}, \mathit{domain}, C_{\bar{f}})$, from all parties, where $\bar{f} : \{\mathit{domain}\}^m \to \mathit{domain}$ and varid is a fresh identifier, if $\{\mathit{varid}_j\}_{j\in[m]}$ were previously stored, proceed as follows:
1. Retrieve $(\mathit{varid}_j, \mathit{domain}, x_j)$, for each $j \in [m]$
2. Store $(\mathit{varid}, \mathit{domain}, x_{m+1} \leftarrow \bar{f}(x_1, \ldots, x_m))$

Output: On input $(\mathit{Output}, \mathit{varid}, \mathit{domain}, \mathit{type})$, from all parties with $\mathit{type} \in \{0, \ldots, n\}$ (if varid is present in memory):
1. If $\mathit{type} = 0$ (**Public Output**): Retrieve (varid, y) and send it to Adv. If the adversary sends $\mathsf{Deliver}$, send y to all parties.
2. Otherwise (**Private Output**): Send (varid) to Adv. Upon receiving $\mathsf{Deliver}$ from Adv, send y to P_i.

Abort: The adversary can at any time send abort, upon which send abort to all honest parties and halt.

Figure 1. Functionality $\mathcal{F}_{\mathsf{MPC}}$ - Zaphod Evaluation

In Figure 1 and Figure 2 we provide the functionalities for our MPC black box. Each value in $\mathcal{F}_{\mathsf{MPC}}$ is uniquely identified by an identifier $\mathit{varid} \in I$, where I is a set of valid identifiers, and a domain set $\mathit{domain} \in \{\mathbb{F}_p, \mathbb{F}_2^{64}\}$. One can see $\mathcal{F}_{\mathsf{MPC}}$-*Zaphod Evaluation* as a combination of two MPC black boxes, specified by the set assigned to *domain*, along with two conversion routines, namely $\mathsf{ConvertToField}$ and $\mathsf{ConvertToBinary}$, given in $\mathcal{F}_{\mathsf{MPC}}$- *Zaphod Conversion.* If $\mathit{domain} = \mathbb{F}_p$, the MPC black box provides arithmetic operations over the finite field $\mathbb{F}_p$, whereas if $\mathit{domain} = \mathbb{F}_2^{64}$, it enables one to execute arbitrary binary circuits over binary vectors of length 64, i.e. function with arguments and results in the set $\mathbb{F}_2^{64}$.

$\mathcal{F}_{\mathsf{MPC}}$- *Zaphod Conversion* permits parties to switch between the two MPC black boxes. Note that we have defined $\mathsf{ConvertToBinary}$ and $\mathsf{ConvertToField}$ to ensure that they are mutual inverses of each other (if the $\mathbb{F}_p$-input element is fewer than 64 bits in length when in the centered interval $(-p/2, \ldots, p/2])$. The algorithms to implement the conversion functions given below are given in [AOR+19]. To ease notation we will write these two operations, for $x \in \mathbb{Z}_{\langle 2^{64} \rangle}$, as $[\![x]\!]_p \leftarrow \mathsf{convert}([\![\mathbf{x}]\!]_2)$ and $[\![\mathbf{x}]\!]_2 \leftarrow \mathsf{convert}([\![x]\!]_p)$. A trivial modification of the protocol 'Convert To Field' allows us to perform an unsigned conversion (i.e. when we think of the bits $[\![\mathbf{x}]\!]_2$ representing an integer $x \in [0, \ldots, 2^{64})$, which we will denote by $[\![x]\!]_p \leftarrow \mathsf{convert}^u([\![\mathbf{x}]\!]_2)$.

Functionality $\mathcal{F}_{\mathsf{MPC}}$ - Zaphod Conversion

Convert To Field: On input $(Convert, varid_1, \mathbb{F}_2^{64}, varid_2, \mathbb{F}_p)$:
1. Retrieve $(varid_1, \mathbb{F}_2^{64}, \mathbf{x})$ and convert $\mathbf{x}$ to an element $y \in \mathbb{F}_p$ by setting $y \leftarrow -x_{63} \cdot 2^{63} + \sum_{i=0}^{62} x_i \cdot 2^i$.
2. Store $(varid_2, \mathbb{F}_p, y)$.

Convert To Binary: On input $(Convert, varid_1, \mathbb{F}_p, varid_2, \mathbb{F}_2^{64})$:
1. Retrieve $(varid_1, \mathbb{F}_p, x)$ as an integer in the range $(-p/2, \ldots, p/2)$.
2. Express $y = x \pmod{2^{64}}$ as $y = \sum_{i=0}^{64} y_i \cdot 2^i$ for $y_i \in \{0, 1\}$
3. Consider the values y_i as elements in $\mathbb{F}_2$ and pack them into a vector $\mathbf{y} \in \mathbb{F}_2^{64}$.

Figure 2. Functionality $\mathcal{F}_{\mathsf{MPC}}$ - Zaphod Conversion

To describe our protocols we will let $[\![\mathbf{x}]\!]_2$, for a boldface $\mathbf{x}$, denote a vector of 64 secret shared bits in the binary arithmetic based side of the computation. We let $[\![x]\!]_2$ for a non-boldface value x denote a single shared bit $x \in \{0, 1\}$ in the binary side of the computation. An execution of **Output**, for *type* $= 0$, we will denote by $x \leftarrow \mathsf{Open}([\![x]\!]_p)$ (resp. $\mathbf{x} \leftarrow \mathsf{Open}([\![x]\!]_2)$.

We let $[\![\mathbf{x}]\!]_2 + [\![\mathbf{y}]\!]_2$ (resp. $[\![x]\!]_2 \cdot [\![y]\!]_2$) denote the execution of the Garbled Circuit to evaluate the addition (resp. multiplication) of two 64-bit integers. Likewise, we let $[\![\mathbf{x}]\!]_2 \oplus [\![\mathbf{y}]\!]_2$ denote the bit-wise XOR of the two secret 64-bit bitstrings $\mathbf{x}$ and $\mathbf{y}$, and $[\![x]\!]_2 \cdot [\![\mathbf{y}]\!]_2$ denote the secure bitwise AND of the single shared bit x with the secret shared bits of the vector $\mathbf{y}$. Clearly operations such as left/right-shift (i.e. $[\![x]\!]_2 \ll 3$) can be done for free, as they are just moving data around in memory. We let $[\![\mathbf{x}]\!]_2^{(i)}$ denote the i-th bit of $\mathbf{x}$, with bit zero representing the least significant bit.

A value $[\![x]\!]_p$ will always represent a shared value $x \in \mathbb{F}_p$. We let $[\![x]\!]_p + [\![y]\!]_p$ (resp. $[\![x]\!]_p \cdot [\![y]\!]_p$) denote addition (resp. multiplication) modulo $\mathbb{F}_p$. If we know $x \in \mathbb{Z}_{\langle 2^{64} \rangle}$ then we can also perform shift operations such as $[\![x]\!]_p \ll 3$, however these are more expensive and require specific protocols which are outlined in [Cd10] (amongst other works).

2.2 IEEE-754

The accepted standard for floating-point arithmetic operations is the IEEE-754 standard, first published in 1985. We use version IEEE-754-2008, published in August 2008. We note that the relevant international standard, ISO/IEC/IEEE 60559:2011 is identical to IEEE-754-2008. IEEE-754 defines representations of finite numbers, infinities, and non-numeric values (so-called "NaNs"); a set of defined arithmetic operations on representable numbers; rounding modes to be used in generating correct outputs of those operations; and a variety of error conditions that may arise during those operations. In this paper we concentrate on the double precision binary format of the IEEE-754 standard.

The IEEE representation has 64 bits in total with the least significant bit (lsb) b_0 (which we assume placed to the right) and the most significant bit (msb) b_{63} (which we assume placed to the left). The three parts of the representation are the sign bit, in position b_{63}, the exponent, in positions $b_{62} \cdots b_{52}$, and the manitissa in positions $b_{51} \cdots b_0$.

If the sign bit is equal to one, then the number is negative. The eleven bits of exponent represent a number $e \in [0, \ldots, 2047]$, which becomes the 'real' exponent after subtracting the number 1023. The mantissa gives an integer $m \in [0, \ldots, 2^{52} - 1]$, although this is not the 'mathematically correct' mantissa as the initial trailing one bit in the msb position has already been deleted. There are two special cases for e, corresponding to the cases of the eleven bits being all zeros or all ones. If $e = 0$ and $m = 0$ then this represents the value zero (which can be positive or negative depending on b_{63}). If $e = 2047$ and $m = 0$ then we have either $+\infty$ or $-\infty$ depending on the sign bit b_{63}. When $e = 2047$ and $m \neq 0$ then we have the special number `NaN`, meaning 'Not a Number'. Thus assuming $e \neq 2047$ the real number represented by this representation is given by

$$(-1)^{b_{63}} \cdot \Big(1 + \sum_{i=1}^{52} (b_{52-i} \cdot 2^{-i})\Big) \cdot 2^{e-1023}.$$

2.3 Secure-Floats via $\mathbb{F}_p$-Arithmetic

The standard floating point representation in LSSS-based MPC over a finite field $\mathbb{F}_p$ of odd characteristic is due to [ABZS13], which itself builds on the work of [Cd10]. Unlike [ABZS13] we always carry around a value err, which the reader should think of as a value which corresponds to the `NaN` in IEEE-754 arithmetic.

Floating point numbers are defined by two global, public integer parameters (ℓ, k) which define the size of the mantissa and the exponent respectively. Each floating point number is represented as a five-tuple $(v, p, z, s, \mathsf{err})$, where

- $v \in [2^{\ell-1}, 2^{\ell})$ is an $\ell + 1$-bit significant with its most significant bit always set to one (note here the msb is not dropped as in the IEEE format).
- $p \in \mathbb{Z}_{\langle k \rangle}$ is the signed exponent.
- z is a bit to define whether the number is zero or not.
- s is a sign bit (equal to zero if non-negative).
- err is the error flag (equal to zero if no rounding or arithmetic error has occurred, it holds a non-zero value otherwise).

Thus assuming $\mathsf{err} = 0$ this tuple represents the value $u = (1 - 2 \cdot s) \cdot (1 - z) \cdot v \cdot 2^p$. We adopt the conventions that when $u = 0$ we also have $z = 1, v = 0$ and $p = 0$, and when $\mathsf{err} \neq 0$ then the values of v, p, z and s are meaningless.

Such errors can be triggered by 'higher level' functions such as trying to compute $\sqrt{-1}$ or $\log(-1)$, they can be triggered by division by zero operations or an underflow/overflow operation. An underflow or overflow occurs when the value p from a computation falls out of the range $\mathbb{Z}_{\langle k \rangle}$. When an error occurs in an operation the err flag is incremented by one. Thus the operation $z \leftarrow x/0 + y$ results in $z.\mathsf{err} = 1$, whereas $z \leftarrow x/0 + y/0$ results in $z.\mathsf{err} = 2$ (assuming in both cases $x.\mathsf{err} = y.\mathsf{err} = 0$.

Following the documentation of Scale-Mamba [ACC+21] we refer to such a floating point value as an sfloat (secure-float) when the values $(v, p, z, s, \mathsf{err})$ are all secret shared. In which case, we write $([\![v]\!]_p, [\![p]\!]_p, [\![z]\!]_p, [\![s]\!]_p, [\![\mathsf{err}]\!]_p)$. Unlike in [ABZS13] the value err is kept permanently masked. When an sfloat value $([\![v]\!]_p, [\![p]\!]_p, [\![z]\!]_p, [\![s]\!]_p, [\![\mathsf{err}]\!]_p)$ is unmarked, first the value $[\![b]\!]_p \leftarrow ([\![\mathsf{err}]\!]_p = 0)$ is computed. Then the five values $([\![b]\!]_p \cdot [\![v]\!]_p, [\![b]\!]_p \cdot [\![p]\!]_p, [\![b]\!]_p \cdot [\![z]\!]_p, [\![b]\!]_p \cdot [\![s]\!]_p, 1 - [\![b]\!]_p)$ are opened.

In this way no information leaks, including how many errors were accumulated, when a value with $\mathsf{err} \neq 0$ is transferred from the secure to the insecure domain.

Arithmetic is implemented using the algorithms in [ABZS13], with correctness and security maintained as long as $2 \cdot \ell + \mathsf{sec} < \log_2 p$, for the statistical security parameter sec. In particular, $2^{-\mathsf{sec}}$ represents the statistical distance between values leaked by the algorithms in $\mathbb{F}_p$, and uniformly random values chosen from $\mathbb{F}_p$.

2.4 Fixed Point Representation

For comparison, we also compare how the above two methods compare against the standard LSSS-based approach to approximating floating point values; namely a fixed point representation first given in [CS10], which also utilizes the work in [Cd10]. We define $\mathbb{Q}_{\langle k,f \rangle}$ as the set of rational numbers $\{x \in \mathbb{Q} : x = \overline{x} \cdot 2^{-f}, \overline{x} \in \mathbb{Z}_{\langle k \rangle}\}$. We represent $x \in \mathbb{Q}$ as the integer $x \cdot 2^f = \overline{x} \in \mathbb{Z}_{\langle k \rangle}$, which is then represented in $\mathbb{F}_p$ via the mapping used above. Thus $x \in \mathbb{Q}$ is in the range $[-2^e, 2^e - 2^{-f}]$ where $e = k - f$. As we are working with fixed point numbers we assume that the parameters f and k are public. For the algorithms to work (in particular fixed point multiplication and division) we require that $f < k$ and $2 \cdot k + \mathsf{sec} < \log_2 p$, again for the statistical security parameter sec. Again the documentation of Scale-Mamba [ACC+21] we refer to such a fixed point value as an sfix when the value $\overline{x}$ is secret shared as $[\![\overline{x}]\!]_p$.

3 Generating Circuits for IEEE Arithmetic

In this section, we implement circuits for floating-point arithmetic operations which are suitable for MPC based on binary circuit computation. While some of this machinery (with respect to NaN computations etc) may at first glance appear superfluous, leading to more complex circuits than necessary; however, floating point implementations that lack this machinery may fail to meet the expectations of real-world users. Thus for our work to be as relevant to real-world settings as possible, we choose to implement fully IEEE-754-compliant circuit designs.

Unfortunately, because IEEE-754 is the recognized standard, and because floating-point performance is a key competitive metric for hardware implementations such as those found in modern microprocessors, IEEE-754 compatible arithmetic circuit designs are generally proprietary and optimized for the hardware. In our application we require combinatorial circuits which are optimized for low AND-depth. To achieve our goal of IEEE-754 compliance without readily available compliant circuit designs, we choose to generate our own standard-compliant circuit designs. To do so, we choose one of the few thoroughly tested open-source IEEE-754 compliant software libraries, the Berkeley SoftFloat library, release 2c [Hau18], from which we automatically derive circuit designs using a C-to-circuit compiler designed for MPC applications (CBMC-GC) [BHWK16].

For each circuit of interest in this work, we started with the relevant main function in the SoftFloat library, and modified the circuit to make it acceptable as input to the CBMC-GC compiler. The necessary edits in each case were minor: changing the name of the relevant function to `mpc_main` so that CBMC-GC would compile that function; renaming the input and output arguments to match the expectations of the compiler and modifying the code to contain only a single output assignment statement and return point, with a specific output naming convention, at the end of the function. Figure 3 shows as an example the original SoftFloat code for IEEE-754 compliant 64-bit floating-point addition, whilst Figure 4 shows the same function after modification for compilation by CBMC-GC.

```
float64 float64_add(float64 a,float64 b)
{ flag aSign, bSign;
  aSign = extractFloat64Sign(a);
  bSign = extractFloat64Sign(b);
  if ( aSign == bSign )
  { return addFloat64Sigs(a,b,aSign); }
  else
  {  return subFloat64Sigs(a,b,aSign); } }
```

Fig. 3. The original SoftFloat C-code for IEEE-754 double addition

```
void mpc_main(float64 INPUT_X_a,
              float64 INPUT_Y_b)
{ flag aSign, bSign;
  float64 temp_output, OUTPUT_A_x;
  aSign = extractFloat64Sign(INPUT_X_a);
  bSign = extractFloat64Sign(INPUT_Y_b);
  if ( aSign == bSign )
   { temp = addFloat64Sigs(INPUT_X_a,
                    INPUT_Y_b,aSign);
     goto done;  }
  else { temp =
     subFloat64Sigs(INPUT_X_a,
            INPUT_Y_b,aSign);
    goto done; }
  done:
    OUTPUT_A_X = temp; }
```

Fig. 4. SoftFloat C-code after modification for CBMC-GC

CBMC-GC is a compilation pipeline designed to convert software functions written in the C programming language into circuit specifications suitable for use in secure multi-party computation – most particularly, garbled circuit computation. The compiler CBMC proceeds by first unrolling loops in the source algorithm. Next, the compiler converts the algorithm into static single-assignment form. Finally, the compiler uses multiple passes of optimization techniques with the aim of reducing gate count in the resulting circuit or reducing depth of that circuit. Optimization begins with conversion of the circuit into and-inverter graph (AIG) form – Boolean networks comprised of 2-input AND gates and inverter gates. During AIG construction, CBMC-GC employs structural hashing [MCJB05] to prevent addition of redundant AND gates, resulting in an AIG with *partial canonicity*. The compiler uses constant propagation techniques to further reduce unnecessary gates (those that result in constant outputs due to one or more inputs being constant). The compiler also applies heuristic re-write rules to reduce subcircuit complexity, and employs SAT sweeping [ZKKS06] to identify additional circuit nodes that realize equivalent logical functions, and then remove such redundancies.

When optimizing for minimum depth, CBMC-GC precedes these optimization passes with a step of "aggregating" gates – parallelizing otherwise sequential structures of gates in order to achieve lower circuit depth. For the addition circuit, we specify no compiler flags. For the multiplication circuit, we specify the "–low-depth" flag, so that the compiler optimizes for the lowest possible gate depth. For the division circuit, which is programmed using an iterative approximation loop, we specify compiler flags that limit the loop unrolling to a factor of 24. We choose this unrolling depth to closely match the number of stages typically used in modern microprocessor floating-point division pipelines, thus the output is (experimentally) indistinguishable from the output of the circuits used in a modern microprocessor. Finally, we use the circuit-utils tool in the CBMC-GC tool suite to convert the optimized circuit into the "Bristol Fashion" used by the Scale-Mamba suite. We have the Bristol format converter remove OR gates as well, resulting in circuits that contain only AND, XOR, and INV gates.

Eventually, we obtain the following circuit sizes for our three basic operations of IEEE-754 compliant floating-point addition, multiplication, and division:

	No. ANDs	No. XORs	No. INVs	AND Depth
add	5385	8190	2062	235
mul	19626	21947	3326	129
div	82269	84151	17587	3619

In the above table, we also present the AND-depth of the circuit; since when operating in the Q2-domain using an LSSS-based MPC enginer modulo 2 the dominant cost is not the number of AND gates but the depth of the AND gates in the circuit.

4 Converting Between Representations

Converting between the two representations is mathematically trivial from a functional perspective. However, our conversion needs to be executed in the secure domain, and thus we need to ensure that no sensitive data leaks during the conversion and in addition, we must use constructions which can be executed reasonably efficient in the secure domain.

We first present two trivial extensions to the conversion algorithms in [AOR+19] which were given earlier. The conversion algorithms work by utilizing a correlated randomness source, called daBits. A call $(\llbracket b \rrbracket_2, \llbracket b \rrbracket_p) \leftarrow \mathsf{daBits}$ produces a doubly-shared bit $b \in \{0,1\}$, which is shared with respect to the two different methodologies. Having such a correlated randomness source allows us to perform conversions. See [AOR+19, RW19] for how such correlated randomness is produced.

The first conversion algorithm, which we denote by $\llbracket x \rrbracket_p \leftarrow \mathsf{convert}(\llbracket x \rrbracket_2)$, converts a single bit x from the binary-world to the LSSS-world. This is executed as follows:

1. $(\llbracket r \rrbracket_2, \llbracket r \rrbracket_p) \leftarrow \mathsf{daBits}$.
2. $\llbracket v \rrbracket_2 \leftarrow \llbracket x \rrbracket_2 \oplus \llbracket r \rrbracket_2$.
3. $v \leftarrow \mathsf{Open}(\llbracket v \rrbracket_2)$.
4. $\llbracket x \rrbracket_p \leftarrow v + \llbracket r \rrbracket_p - 2 \cdot v \cdot \llbracket r \rrbracket_p$.

The converse algorithm takes a value $\llbracket x \rrbracket_p$ in the LSSS-world, which we know to represent a value $x \in \{0,1\} \subset \mathbb{F}_p$ and converts it to a shared bit in the binary-world; an operation which we denote by $\llbracket x \rrbracket_2 \leftarrow \mathsf{convert}(\llbracket x \rrbracket_p)$. The procedure for this is a little more complex, and requires that $\mathsf{sec}+1 \leq \min\{64, \log_2 p\}$, which will hold in practice in any case. This operation proceeds as follows:

1. For $i = 0, \ldots, \mathsf{sec}$ execute $(\llbracket r_i \rrbracket_2, \llbracket r_i \rrbracket_p) \leftarrow \mathsf{daBits}$.
2. $\llbracket r \rrbracket_p \leftarrow \sum_{i=0}^{\mathsf{sec}} \llbracket r_i \rrbracket_p \cdot 2^i$.
3. $\llbracket v \rrbracket_p \leftarrow \llbracket x \rrbracket_p + \llbracket r \rrbracket_p$.
4. $v \leftarrow \mathsf{Open}(\llbracket v \rrbracket_p)$.
5. $\llbracket \mathbf{x} \rrbracket_2 \leftarrow v - \llbracket \mathbf{r} \rrbracket_2$.

This works as the value $v = x + r$ is guaranteed to hold a 64-bit unsigned integer, and it is also statistically hiding of the single bit x. The final subtraction is performed using a binary circuit for 64-bit subtraction.

We can now present our two algorithms for conversion between a secure IEEE floating point double, held as a bit vector $[\![\mathbf{x}]\!]_2$ in the binary circuit world, and an sfloat value ($[\![v]\!]_p$, $[\![p]\!]_p$, $[\![z]\!]_p$, $[\![s]\!]_p$, $[\![\mathsf{err}]\!]_p$) in the LSSS world; and vice-versa. These algorithms are given in Figure 5 and Figure 6, with the main complexity coming from needing to deal with the variable values ℓ and k representing the sizes of the mantissa and exponent in the sfloat datatype. We mark the lines which cost nothing in the secure domain with a comment of "free".

```
IEEE to sfloat

Input: $[\![x]\!]_2$.
Output: ($[\![\mathbf{v}]\!]_p, [\![\mathbf{p}]\!]_p, [\![\mathbf{z}]\!]_p, [\![\mathbf{s}]\!]_p, [\![\mathsf{err}]\!]_p$), (with parameters $\ell$ and $k$).

 1. $[\![\mathbf{s}]\!]_p \leftarrow \mathsf{convert}([\![x^{(63)}]\!]_2)$.
 2. $[\![z]\!]_2 \leftarrow ([\![x]\!]_2 = 0)$
 3. $[\![v]\!]_2 \leftarrow [\![x]\!]_2 \ll 11$.                 //free
 4. $[\![v^{(63)}]\!]_2 \leftarrow 1 - [\![z]\!]_2$.              //free
 5. If $\ell \geq 64$ then
    (a) $[\![\mathbf{v}]\!]_p \leftarrow \mathsf{convert}^u([\![v]\!]_2)$.
    (b) $[\![\mathbf{v}]\!]_p \leftarrow [\![v]\!]_p \ll (\ell - 64)$.
 6. If $\ell < 63$ then
    (a) $[\![v]\!]_2 \leftarrow [\![v]\!]_2 \gg (64 - \ell)$.        //free
    (b) $[\![\mathbf{v}]\!]_p \leftarrow \mathsf{convert}([\![v]\!]_2)$.
 7. $[\![x']\!]_2 \leftarrow [\![x]\!]_2$.                       //free
 8. $[\![x'^{(63)}]\!]_2 \leftarrow 0$.                          //free
 9. $[\![p]\!]_2 \leftarrow [\![x']\!]_2 \gg 52$.                 //free
10. $[\![p']\!]_2 \leftarrow (-(\ell - 1) - 1023) + [\![p]\!]_2$
11. $[\![\mathbf{p}]\!]_p \leftarrow \mathsf{convert}([\![p']\!]_2)$.
12. $[\![\mathbf{z}]\!]_p \leftarrow \mathsf{convert}([\![z]\!]_2)$.
13. $[\![e]\!]_2 \leftarrow ([\![p]\!]_2 = 2047)$.
14. If $k < 11$ then
    (a) $\mathsf{mask} \leftarrow 2^{64} - 1 - (2^k - 1)$.      //free
    (b) $[\![t]\!]_2 \leftarrow [\![p']\!]_2 \ \& \ \mathsf{mask}$.
    (c) $[\![e']\!]_2 \leftarrow [\![t^{(63)}]\!]_2$.                //free
    (d) $[\![e'']\!]_2 \leftarrow ([\![t]\!]_2 = 0)$.
    (e) $[\![e]\!]_2 \leftarrow [\![e]\!]_2 \oplus (1 - [\![e']\!]_2) \cdot (1 - [\![e'']\!]_2)$.
15. $[\![\mathsf{err}]\!]_p \leftarrow \mathsf{convert}([\![e]\!]_2)$.
16. Return ($[\![\mathbf{v}]\!]_p, [\![\mathbf{p}]\!]_p, [\![\mathbf{z}]\!]_p, [\![\mathbf{s}]\!]_p, [\![\mathsf{err}]\!]_p$).
```

Figure 5. IEEE to sfloat

IEEE to sfloat*:* Algorithm Figure 5, shows how we can securely convert a number from IEEE in the binary-world to sfloat in the LSSS-world. Step 1 extracts the single bit corresponding to the sign bit. This uses our optimized methodology for doing such conversions given earlier. In step 2 we test whether the shared vector $\mathbf{x}$ is equal to zero. This is done by forming $[\![z]\!]_2 \leftarrow \prod_{i=0}^{63}(1 - [\![\mathbf{x}^{(i)}]\!]_2)$.

In steps 3–6b we are extracting the mantissa v. A naive way to do this would be to take 64 bits of the IEEE representation, shift by 12 to the left and then shift back by 12 to right. Then, to obtain the 52 bit mantissa, we dd one bit in the most significant bit position which is bit position 52 and finally convert the value to the $[\![v]\!]_p$ used in sfloat, at which point we could adjust this to cope with the required value of ℓ. However, this would be very inefficient. We try to maintain as much of the bit-shifting operations in the $[\![\cdot]\!]_2$ domain, as their bit-shifts come for free. Thus we first shift left by eleven places and then add the one bit into the top position (which only needs to happen if the value $\mathbf{x}$ is non-zero, thus we use $[\![z]\!]_2$ for this). We then have two cases to consider, if $\ell \geq 64$ then we actually need to shift the value up even more, but we are already using 64-bits. Thus we convert, using the unsigned conversion routine (Step 5a) to a modulo p value, and then perform the shift up in this domain (which is relatively expensive but unavoidable given the basic instructions available to us). When $\ell < 63$ we perform the shift down in the binary domain (Step 6a) and then do the conversion (using an unsigned conversion as we know the top bit is zero).

Steps 7–9 extract the bits of the IEEE exponent p, whereas step 10 converts it to the correct exponent for the sfloat representation. Now obtaining $[\![z]\!]_p$ and $[\![p]\!]_p$ is relatively straight forward. Note, to obtain $[\![z]\!]_p$ we use the optimized bit conversion protocol given earlier in this section.

Steps 13 onwards deal with computing the error flag $[\![\mathsf{err}]\!]_p$. This is first computed using bit operations in the modulo two domain, and then converted to $[\![\mathsf{err}]\!]_p$ using the above optimized conversion (in the penultimate step). We need to set the flag err if either $[\![\mathbf{p}]\!]_2$ represents the value 2047, or there is an error introduced due to a low value of k in the sfloat representation. The first test, in step 13, is accomplished by setting $[\![e]\!]_2 \leftarrow \prod_{i=0}^{11} [\![\mathbf{p}^{(i)}]\!]_2$. The second test, in steps 14a–14e, test whether the value of $[\![\mathbf{p}']\!]_2$ lies outside the range $[-2^{k-1}, \ldots, 2^{k-1} - 1]$ or not.

sfloat to IEEE

```
Input: $([\![v]\!]_p, [\![p]\!]_p, [\![z]\!]_p, [\![s]\!]_p, [\![\mathsf{err}]\!]_p)$, (with parameters $\ell$ and $k$).
Output: $[\![\mathbf{x}]\!]_2$.

 1. $[\![v']\!]_p \leftarrow [\![v]\!]_p - 2^{\ell-1} \cdot (1 - [\![z]\!]_p)$.
 2. $\ell' \leftarrow \ell$.
 3. If $\ell > 64$ then
    (a) $\ell' \leftarrow 53$
    (b) $[\![v']\!]_p \leftarrow [\![v']\!]_p \gg (\ell - 53)$.
 4. $[\![\mathbf{v}]\!]_2 \leftarrow \mathsf{convert}([\![v']\!]_p)$.
 5. If $\ell' < 53$ then $[\![\mathbf{v}]\!]_2 \leftarrow [\![\mathbf{v}]\!]_2 \ll (53 - \ell')$.          //free
 6. Else $[\![\mathbf{v}]\!]_2 \leftarrow [\![\mathbf{v}]\!]_2 \gg (\ell' - 53)$.                    //free
 7. $[\![\mathsf{ok}]\!]_p \leftarrow ([\![\mathsf{err}]\!]_p == 0)$.
 8. $[\![\mathsf{ok}]\!]_2 \leftarrow \mathsf{convert}([\![\mathsf{ok}]\!]_p)$.
 9. $[\![\mathbf{p}]\!]_2 \leftarrow \mathsf{convert}((1 - [\![z]\!]_p) \cdot ([\![p]\!]_p + \ell + 1023 - 1))$.
10. If $k > 11$ then
    (a) $[\![\mathbf{t}]\!]_2 \leftarrow [\![\mathbf{p}]\!]_2$ & 0xFFFFFFFFFFFFF800
    (b) $[\![\mathsf{ok}]\!]_2 \leftarrow [\![\mathsf{ok}]\!]_2$ & $([\![\mathbf{t}]\!]_2 == 0)$.
11. $[\![\mathbf{s}]\!]_2 \leftarrow \mathsf{convert}([\![s]\!]_p)$.
12. $[\![\mathbf{x}]\!]_2 \leftarrow ([\![\mathbf{p}]\!]_2 \ll 52) \oplus [\![\mathbf{v}]\!]_2$.                   //free
13. $[\![\mathbf{x}^{(63)}]\!]_2 \leftarrow [\![\mathbf{s}]\!]_2$.
14. $[\![\mathbf{z}]\!]_2 \leftarrow \mathsf{convert}([\![z]\!]_p)$.
15. NaN $\leftarrow$ 0x7FF0000000000001
16. $[\![\mathbf{x}]\!]_2 \leftarrow (1 - [\![\mathbf{z}]\!]_2)$ & $[\![\mathbf{x}]\!]_2$.
17. $[\![\mathbf{x}]\!]_2 \leftarrow ([\![\mathsf{ok}]\!]_2$ & $[\![\mathbf{x}]\!]_2) \oplus ((1 - [\![\mathsf{ok}]\!]_2)$ & NaN$)$.
18. Return $[\![\mathbf{x}]\!]_2$.
```

Figure 6. sfloat to IEEE

sfloat *to IEEE.* Algorithm Figure 6, shows how we can convert a number from sfloat to IEEE. In steps 1–6, we are computing the mantissa of IEEE from $[\![v]\!]_p$. To do so, first we need to get rid of the most significant bit of $[\![v]\!]_p$ (step 1) as the sfloat representation stores the msb, whereas IEEE does not. Since bit-shifting in the $[\![\cdot]\!]_p$ domain is expensive, whereas bit-shifting in the $[\![\cdot]\!]_2$ domain costs nothing we do a complete shift (step 3) when $\ell > 64$ down to 53-bits, but when $\ell < 64$ we delay the shifting until we have converted $[\![v]\!]_p$ to $[\![\mathbf{v}]\!]_2$ (step 5).

In step 7–8, we are dealing with the error flag which says if we should end up with NaN or not. To make it easier to follow we define ok flag which will be one when the error flag is zero. In step 9, we are converting the exponent of the sfloat into the IEEE exponent. When $k > 11$ this value could overflow the allowed IEEE representation, so this is detected in step 10 and the ok flag is updated accordingly.

The sign bit is dealt with in step 11 using the optimized bit conversion protocol which was explained earlier. In steps 12–13, we need to pack all the preceding values together to obtain the IEEE representation. This leaves us with dealing with the two special values of zero and NaN. These are dealt with in steps 14-17, where we use a fixed NaN value of 0x7FF0000000000001.

5 Experimental Results

In this section, we give results on our implementation of different operations namely, addition, multiplication, division, as well as our conversion algorithms of IEEE to sfloat and sfloat to IEEE which are presented in Section 4. These are given in the Full Threshold case for $n = 2, \ldots, 8$ parties and in the threshold case for (n, t) values with $t < n/2$ and $n = 3, \ldots, 8$. In the latter case, we do not present run times for $(n, t) = (8, 3)$ as in this situation the number of maximally unqualified sets starts to become too big for our replicated secret sharing based technique for evaluating binary circuits. For the large prime sharing in the threshold case we utilize Shamir sharing.

The experiments were done in Scale-Mamba version 1.11 [ACC+21]. The sfloat data type was instantiated using mantissas with bit length $l = 53$ and (signed) exponents with bit length $k = 11$. To satisfy the required equation $2 \cdot \ell + \kappa < \log_2 p$, for sfloat, we used a prime with 148 bit length and statistical security parameter $\kappa = 40$. In all experiments we measured the online run time averaged over 500 runs.

For the basic operations on IEEE values using binary circuit based MPC (Figure 7) the runtime of the Full-Threshold variants grows with the number of parties. The average online time per operation in this case ranges for addition from $0.025s$ (for $n = 2$) to $0.216s$ (for $n = 8$), for multiplication from $0.092s$ (for $n = 2$) to $0.672s$ (for $n = 8$), for division from $0.634s$ (for $n = 2$) to $3.27s$ (for $n = 8$). For the threshold variants we see a range of run-times depending on the precise values of (n, t), growing as functions of n and t. The range of values is in the threshold case for addition from $0.013s$ to $0.035s$, for multiplication from $0.022s$ to $0.041s$, for division from $0.191s$ to $0.643s$. So depending on the number of parties and the precise operation the performance of the threshold variants are between 2 and 6 times faster than their full threshold counterparts. This behavior is to be expected as the IEEE operations are performed via binary circuits. In the full threshold case these are done via the HSS protocol, which is low round complexity but requires a lot of data to be sent and a lot of computation to be performed. When in the threshold case this is performed using a replicated LSSS, which requires higher round complexity but the total amount of data sent and computation performed is less.

We next turn to the sfix operations. Recall sfix addition is a local operation, and involves no communication thus it is very fast in all situations irrespective of the underlying access structure or the number of parties, taking roughly $0.2\mu s$ on average per addition. For multiplication, the run times scales with the number of parties, and the threshold, ranging from 0.001s to 0.0004s for $n = 8$ in the Full Threshold case. For division, again, the runtime scales with the number of parties; ranging from 0.0026s for $n = 2$ to 0.0082 for $n = 8$ in the Full Threshold case. For the case of threshold Shamir sharing the times are roughly twice as fast as in the Full Threshold case. See Figure 8 for details. Thus using sfix instead of IEEE arithmetic equates to savings (in execution times) of orders of magnitude.

The main issue with sfix operations is the inherent precision loss in fixed-point operations, thus we next turn to examine the performance (in Figure 9) of the sfloat operations. Recall these are approximations to real numbers much like standard IEEE arithmetic, but not exactly the same, as they are more flexible and tunable to the MPC environment. Much like the sfix operations, the sfloat operations do not require the execution of binary circuits. The run time of the Shamir based threshold access structures is about half that of the Full Threshold access structures we tested. In the Full Threshold case, we obtain execution times ranging from 0.005s-0.018s for addition, 0.002s-0.008s for multiplication and 0.004s-0.010s for division.

In summary for the basic operations we have a trade off between accuracy sfix-sfloat-IEEE and speed IEEE-sfloat-sfix. For multiplication the performance improvement between sfloat and sfix is a factor of two, whereas the performance improvement between sfloat and IEEE multiplication is a factor of around one hundred.

Finally, we turn to the conversion routines between sfloat and IEEE representations given in Section 4. We see that since these operations require a combination of LSSS-based operations over the large field, as well as binary circuit based operations over $\mathbb{F}_2$, there is less of a pronounced difference between the run times for Full Threshold and those for thresholds with $t < n/2$. The conversion from IEEE to sfloat is roughly 2-4 times faster than the conversion from sfloat to IEEE. The timings are given in Figure 10.

Acknowledgments

This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the FWO under an Odysseus project GOH9718N, by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. FA8750-19-C-0502, and by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA) via Contract No. 2019-1902070006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either express or implied, of the ERC, DARPA, the FWO, ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

ABF$^+$17. Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

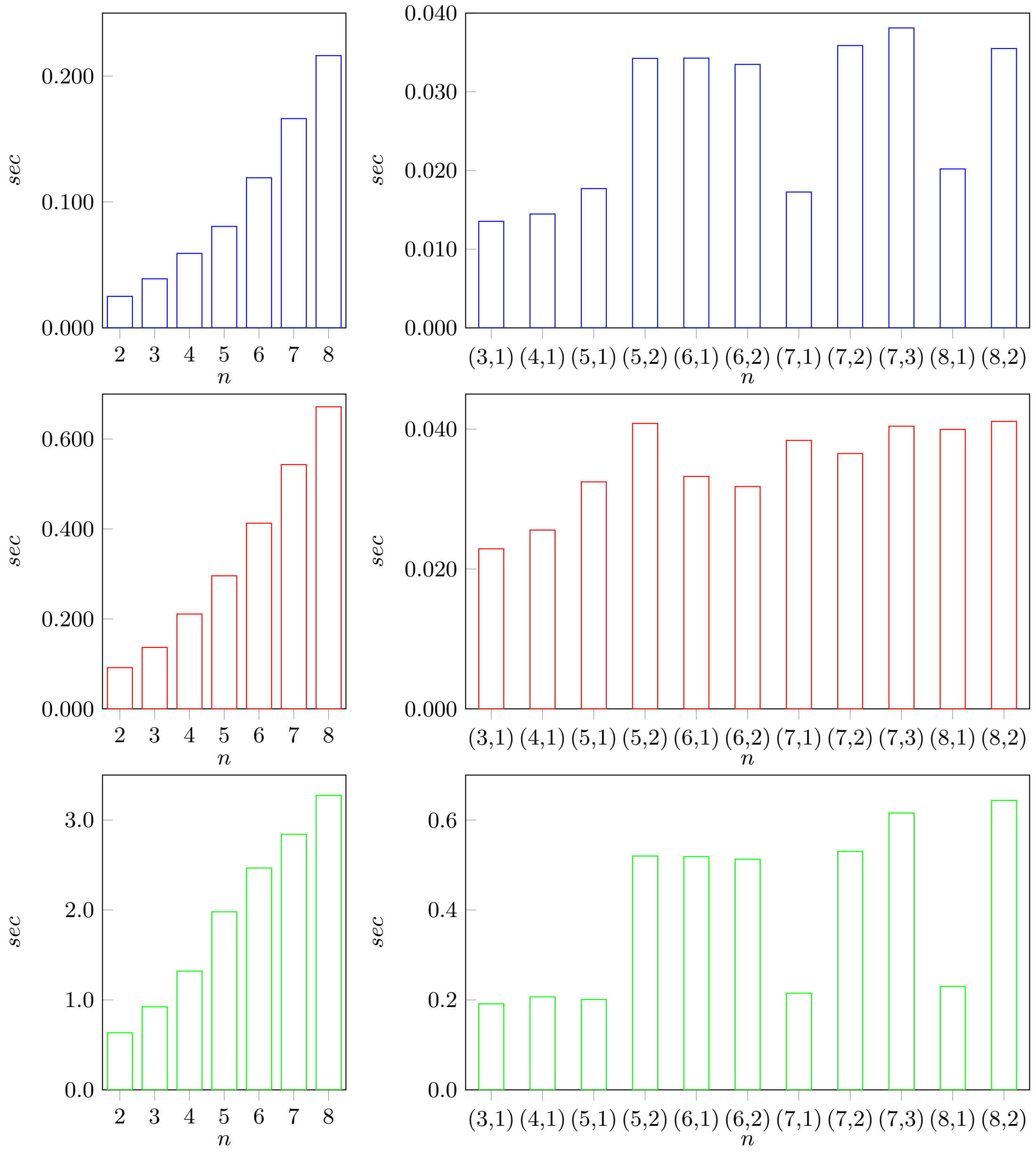

Fig. 7. Execution time in sec to execute the binary circuit based IEEE-754 operations addition (blue), multiplication (red), division (green) for Full Threshold access structure with n players (left column) and Shamir access structures (n, t) (right column)

ABZS13. Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *ISOC Network and Distributed System Security Symposium – NDSS 2013*, San Diego, CA, USA, February 24–27, 2013. The Internet Society.

ACC$^+$21. Abdelrahaman Aly, Kelong Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE and MAMBA v1.11: Documentation, 2021.

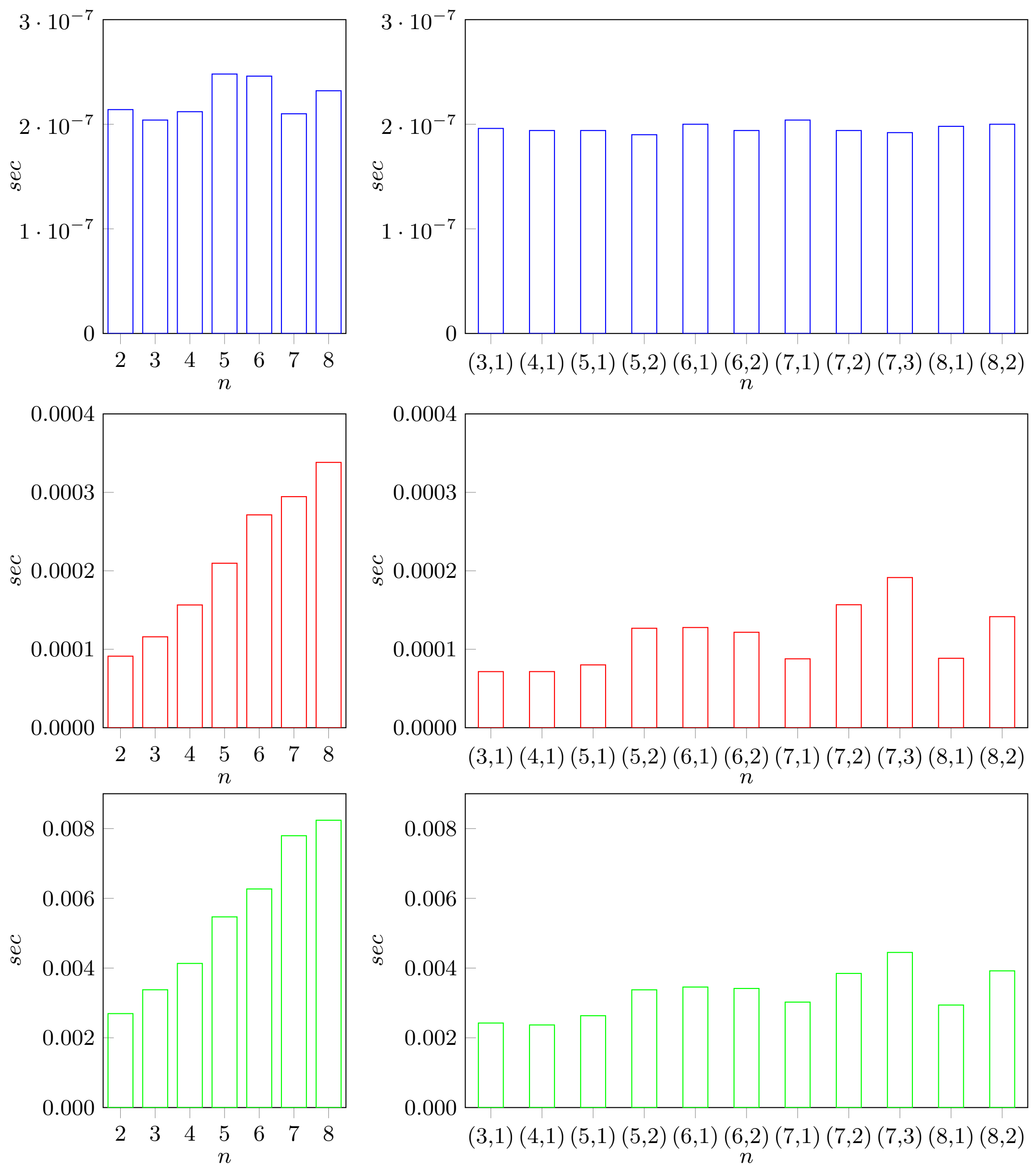

Fig. 8. Execution time in sec to execute the LSSS-based sfix operations addition (blue), multiplication (red), division (green) for Full Threshold access structure with n players (left column) and Shamir access structures (n, t) (right column)

AOR+19. Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In Michael Brenner, Tancrède Lepoint, and Kurt Rohloff, editors, *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 33–44. ACM, 2019.

AS19. Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th Inter-*

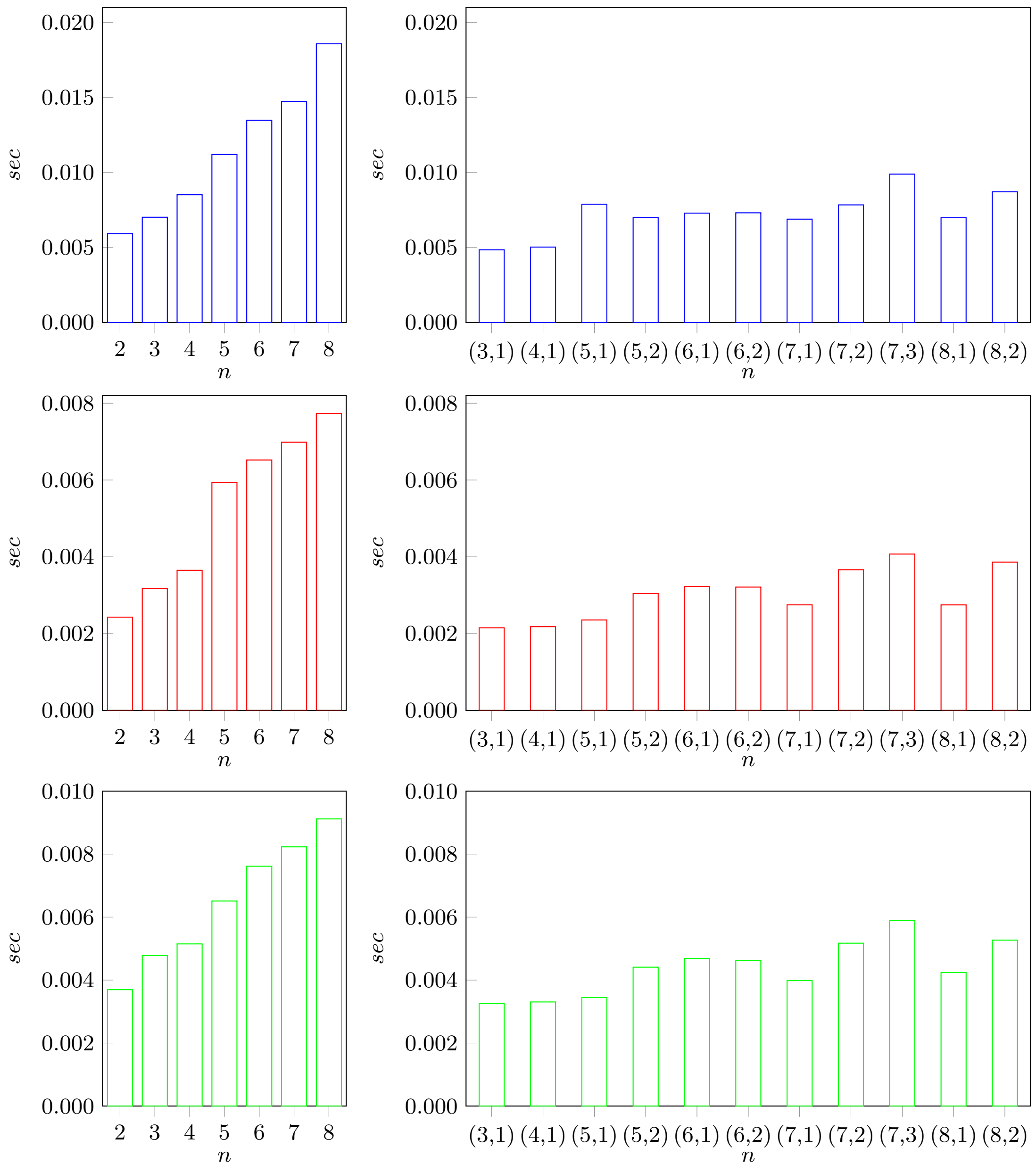

Fig. 9. Execution time in sec to execute the LSSS-based sfloat operations addition (blue), multiplication (red), division (green) for Full Threshold access structure with n players (left column) and Shamir access structures (n, t) (right column)

national Conference on Applied Cryptography and Network Security, volume 11464 of *Lecture Notes in Computer Science*, pages 509–529, Bogota, Colombia, June 5–7, 2019. Springer, Heidelberg, Germany.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

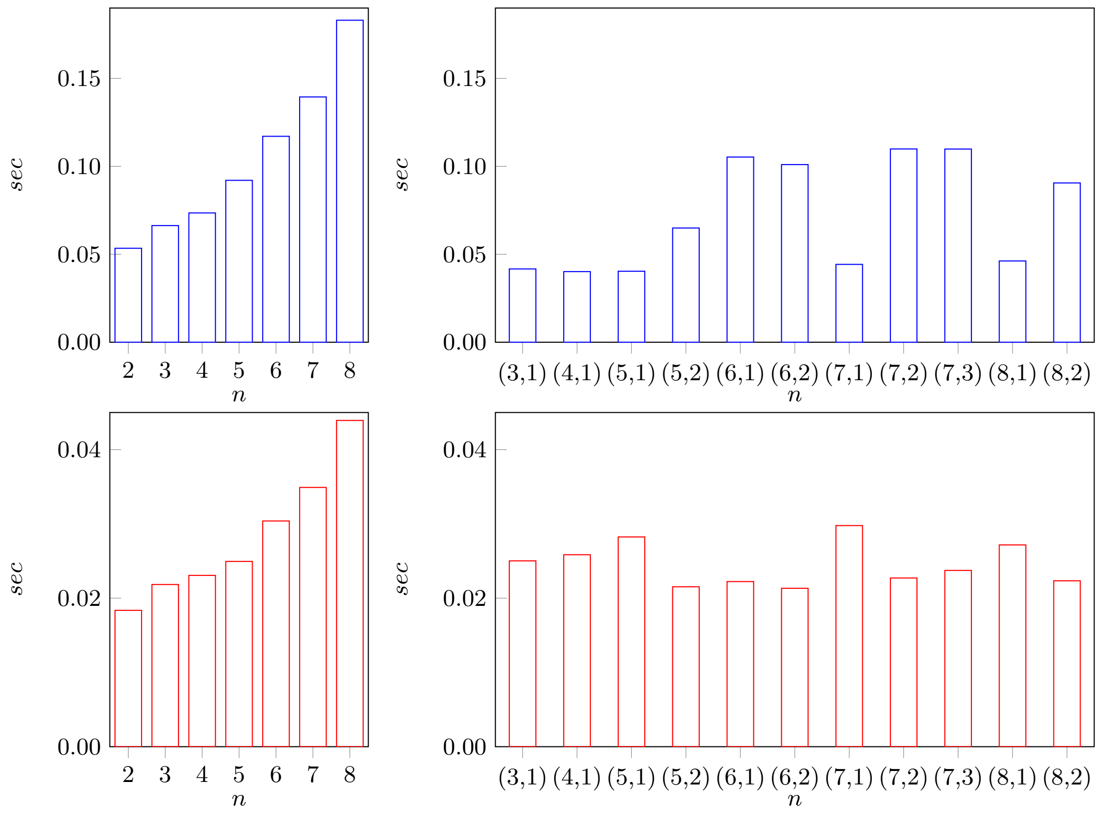

Fig. 10. Execution time in sec to execute the conversion operations sfloat to IEEE-764 (blue) and IEEE-764 to sfloat (red) for Full Threshold access structure with n players (left column) and Shamir access structures (n, t) (right column)

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

BHWK16. Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. Compiling low depth circuits for practical secure computation. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016: 21st European Symposium on Research in Computer Security, Part II*, volume 9879 of *Lecture Notes in Computer Science*, pages 80–98, Heraklion, Greece, September 26–30, 2016. Springer, Heidelberg, Germany.

BMR90. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press.

Cd10. Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10: 7th International Conference on Security in Communication Networks*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199, Amalfi, Italy, September 13–15, 2010. Springer, Heidelberg, Germany.

CGH$^+$18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. Cryptology ePrint Archive, Report 2018/570, 2018. `https://eprint.iacr.org/2018/570`.

CS10. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010: 14th International Conference on Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50, Tenerife, Canary Islands, Spain, January 25–28, 2010. Springer, Heidelberg, Germany.

DKL$^+$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

Hau18. John Hauser. Berkeley SoftFloat, 2018.

HSS17. Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 598–628, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

KLR16. Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 271–287, Christ Church, Barbados, February 26, 2016. Springer, Heidelberg, Germany.

KRSW18. Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Reducing communication channels in MPC. In Dario Catalano and Roberto De Prisco, editors, *SCN 18: 11th International Conference on Security in Communication Networks*, volume 11035 of *Lecture Notes in Computer Science*, pages 181–199, Amalfi, Italy, September 5–7, 2018. Springer, Heidelberg, Germany.

KW15. Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *Int. J. Inf. Sec.*, 14(6):531–548, 2015.

Lie12. Manuel Liedel. Secure distributed computation of the square root and applications. In Mark Dermot Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience - 8th International Conference, ISPEC 2012, Hangzhou, China, April 9-12, 2012. Proceedings*, volume 7232 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 2012.

Mau06. Ueli M. Maurer. Secure multi-party computation made simple. *Discret. Appl. Math.*, 154(2):370–381, 2006.

MCJB05. Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert Brayton. Fraigs: A unifying representation for logic synthesis and verification, 2005.

PS15. Pille Pullonen and Sander Siim. Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *Lecture Notes in Computer Science*, pages 172–183, San Juan, Puerto Rico, January 30, 2015. Springer, Heidelberg, Germany.

RW19. Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India*, volume 11898 of *Lecture Notes in Computer Science*, pages 227–249, Hyderabad, India, December 15–18, 2019. Springer, Heidelberg, Germany.

Sha79. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

SW19. Nigel P. Smart and Tim Wood. Error detection in monotone span programs with application to communication-efficient multi-party computation. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 210–229, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

WRK17. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 39–56, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

ZKKS06. Qi Zhu, Nathan Kitchen, Andreas Kuehlmann, and Alberto L. Sangiovanni-Vincentelli. SAT sweeping with local observability don't-cares. In Ellen Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 229–234. ACM, 2006.