SecFloat: Accurate Floating-Point meets Secure 2-Party Computation

Background

Notation: 记安全参数为 λ, 秘密共享的整数长度为 l-bit.

MPC方面，假设我们拥有以下基础知识：

- 可以使用基于公钥密码等方式实现 1-out-of-k OT_l，单次通信量 $2\lambda + kl$。
- 基于 2-party 的加法 secret sharing 的加法与标量乘几乎免费（无通信开销）。
- 可以使用 beaver triple 实现 2-party 的安全乘法操作。一次安全乘法，总通信量为 $2l$。

在浮点数储存方面，我们使用 IEEE754 Float32（单精度标准）：

- $(-1)^S \times (1.M) \times 2^{E-bias}$
- 符号位 S 1位、指数为 E 7 位、尾数 M 为 23 位、$bias = 127$
- 四舍六入五成双（round-to-nearest-ties-to-even）

IEEE 754 加法

不失一般性，设 $x \geq y$，为了计算 $z = x + y$，其中：

$$x = (-1)^{S_x} \cdot (1.M_x) \cdot 2^{E_x}, \quad y = (-1)^{S_y} \cdot (1.M_y) \cdot 2^{E_y}$$

则步骤如下：

1. 指数对齐。若 $E_x > E_y$，将 y 的尾数右移： $M_y \leftarrow \frac{1.M_y}{2^{E_x - E_y}}, \quad E \leftarrow E_x, \quad M_x \leftarrow (1.M_x).$
2. 尾数相加。考虑S_x, S_y的符号情况。若同号，则计算 $M = M_x + M_y$。若异号，则用大者减小者得到差的绝对值$M = |M_x - M_y|$，并附上正确的符号S。
3. 规范化。显然 $\max\{M_x + M_y, |M_x - M_y|\} < 4$，故$M$右移最多只需一次（对应指数$E$加1即可）。若 $M = 0$，则直接跳到第四步。其余情况反复左移并自减E，直到$M \in [1, 2)$ 时停止。
4. 写入结果。最后将 M 的首位 1 剔除，将最后的 S, E, M 重新作为新的 32 位浮点结果。
5. 在 $M_y \leftarrow \frac{1.M_y}{2^{E_x - E_y}}$、$M = M_x + M_y$、$M = |M_x - M_y|$以及对$M$右移一位这四个步骤中，会涉及到舍入问题。

IEEE 754 乘法

乘法由于不涉及数的大小比较，因此步骤相对比较简单。

1. 符号位异或。$S = S_x \oplus S_y$.
2. 指数相加。$E = E_x + E_y - bias$.
3. 尾数相乘。$M = (1.M_x) \cdot (1.M_y)$，结果在 $[1, 4)$ 范围内。
4. 规范化。如果结果 ≥ 2，将 E 自增并将 M 右移一位，最后剔除 M 最高位的 1.
5. 在 $M = (1.M_x) \cdot (1.M_y)$ 与 M 右移一位这两步涉及到舍入操作。

舍入方法

我们令 G, R, S 分别为 M 最低位之后的后三位。则“四舍六入五成双”的规则可以形式化表述为如下伪代码：

```
if (GRS > 0b100):
    M = M + 1
elif (GRS == 0b100 and M[:-1] == 1):
    M = M + 1
```

注意舍入之后有可能需要再次规范化。

Building Blocks

下面用自底向上的方式，讲清楚整个协议是如何构建的。

$\mathcal{F}_{MUX}$

我们需要实现的理想功能是，`MUX(b,x)=(b==1?x:0)`。即对于布尔 secret share $[c] = (c_0, c_1), c_i \in \{0, 1\}$ 和算术 secret share $[a] = (a_0, a_1), a \in Z_n$，输出 $[a \cdot c]$。步骤如下：

SETUP: P_0 持有 a_0, c_0，P_1 持有 a_1, c_1。

- P_0 与 P_1 各自生成随机数 $r_0, r_1 \in Z_n$。
- P_0 根据 c_0 的值设置 (s_0, s_1)。若 $c_0 = 0$，令 $(s_0, s_1) = (-r_0, -r_0 + a_0)$，否则为 $(-r_0 + a_0, -r_0)$。
- P_0 作为发送方与 P_1 做一轮 1-out-of-2 OT，P_0 提供两条消息 (s_0, s_1)，P_1 提供 c_1 并最终获得 $x_1 = s_{c_1}$。
- P_1 根据 c_1 的值设置 (t_0, t_1)。若 $c_1 = 0$，令 $(t_0, t_1) = (-r_1, -r_1 + a_1)$，否则为 $(-r_1 + a_1, -r_1)$。
- 再做一轮 OT，P_0 获得 $x_0 = t_{c_0}$。
- P_0 贡献 $r_0 + x_0$，P_1 贡献 $r_1 + x_1$。加起来是 $(r_0 + r_1) + (x_0 + x_1)$。

我们列举出四种可能的 (c_0, c_1) 情况：

> $(s_0, s_1) = (-r_0 + c_0 a_0, -r_0 + a_0(1 - c_0))$, $(t_0, t_1) = (-r_1 + c_1 a_1, -r_1 + a_1(1 - c_1))$
> $x_0 = t_{c_0}, x_1 = s_{c_1}$

c_0	c_1	x_0	x_1	$(r_0 + r_1) + (x_0 + x_1)$
0	0	$-r_1 + c_1 a_1$	$-r_0 + c_0 a_0$	0
0	1	$-r_1 + c_1 a_1$	$-r_0 + a_0(1 - c_0)$	a
1	0	$-r_1 + a_1(1 - c_1)$	$-r_0 + c_0 a_0$	a
1	1	$-r_1 + a_1(1 - c_1)$	$-r_0 + a_0(1 - c_0)$	0

结果刚好等于 $[a \cdot c]$，因此协议正确。

通信量为两轮 OT 的开销，也就是 $2(\lambda + 2l) = 2\lambda + 4l$。但 CryptoFlow2 的 3.1.1 节可以将通信开销优化到 $2\lambda + 2l$。

$\mathcal{F}_{AND}$

这是显然的，直接使用 beaver triple 实现，通信量为 $(\lambda+16)+4$。（见 CryptoFlow2 的附录 A1 节）

$\mathcal{F}_{OR}$

我们有 $[x \vee y] = [x \oplus y] \oplus [x \wedge y]$。因此令 $[x \wedge y] = (z_0, z_1)$，因此每一方直接计算 $x_i \oplus y_i \oplus z_i$ 即可。通信量与 $\mathcal{F}_{AND}$ 相同，也为 $\lambda + 20$。

$\mathcal{F}_{EQ}$

我们将 x, y 按照 m-bit 进行分块。考虑对某个块 x_j, y_j 进行比较。我们使用 1-out-of-2^m OT：

- P_0 随机选取 $(eq_{0,j})_0$，并对 $k \in [0, 2^m-1]$ 准备消息 $t_{j,k} = (eq_{0,j})_0 \oplus (x_j == k)$.
- P_0 将 2^m 个消息作为 OT 的输入，P_1 输入 y_j，并获得 $(eq_{0,j})_1$.
- 当且仅当 $x_j = y_j$ 时，我们有 $(eq_{0,j})_0 \oplus (eq_{0,j})_1 = 1$，这就完成了 x_j, y_j 的秘密EQ判断。

我们可以通过树状结构与 $\mathcal{F}_{AND}$，计算 $(eq_{1,j})_i = (eq_{0,j})_i \wedge (eq_{0,j+m})_i$，将 m-bit 比较拓展到 $2m$-bit，最后完成整个长度的比较。通信量为 $\lceil \frac{l}{m} \rceil (2\lambda + 2^m) + \lceil \frac{l}{m} \rceil (\lambda + 20)$，轮数为 $\log l$.

$\mathcal{F}_{GT/LT}$

仍然是分块的思路，首先计算块长度为 m 的结果 $1\{x_j < y_j\}$，还是可以使用 1-out-of-2^m OT 完成。P_0 只需随机生成 $(lt_{0,j})_0$, 并准备 2^m 个消息 $t_{j,k} = (lt_{0,j})_0 \oplus 1\{x_j < k\}$.

然后合并的时候高位优先，$1\{x < y\} = 1\{x_H < y_H\} \oplus (1\{x_H = y_H\} \wedge 1\{x_L < y_L\})$.

通信成本小于 $\lambda(4q) + 2^m(2q) + 22q$，取 $m = 4, q = l/4$ 时为 $\lambda l + 13.5l$ ，轮数 $\log l$.

$\mathcal{F}_{LUT}$

假设 LUT 有 2^m 项，每一项有 n-bit.

SETUP: P_0 随机取索引 $r \in \{0,1\}^m$，和 LUT L 混淆后的 share $T^0[i] \in Z_{2^n}, \quad \forall i \in \{0,1\}^m$.

- P_0 对每个 $s \in \{0,1\}^m$，构造 $M_s[i] = L[i \oplus r \oplus s] \oplus T^0[i], \quad \forall i \in \{0,1\}^m$.
- P_0 将这 2^m 条长度为 n-bit 的消息与 P_1（选取 s）作 1-out-of-2^m OT_n，成本为 $2\lambda + 2^m n$.
- P_1 令 $T^1 \leftarrow M_s$。现在 P_0 持有 (T^0, r)，P_1 持有 (T^1, s).
- 在线阶段，P_0 发送 $u = x_0 \oplus r$, P_1 发送 $v = x_1 \oplus s$，双方同时计算 $i^* = u \oplus v = x \oplus r \oplus s$. 最后 P_0, P_1 分别保存 $T^0[i^*], T^1[i^*]$，显然合起来是 $L[x]$。通信成本 $2m$ 可忽略。

$\mathcal{F}_{Wrap}$

如果我们要计算 $1\{a+b>2^n-1\}$，这等同于计算 $1\{2^n-1-a<b\}$。因此直接使用 $\mathcal{F}_{GT/LT}$ 即可。

$\mathcal{F}_{B2A}$

P_0, P_1 分别持有布尔共享的一位 $c=c_0\oplus c_1, c\in\{0,1\}$，并最后得到 $d=d_0+d_1 \pmod{2^n}$ 且 $d=c$。

- P_0 随机选取 $x\in Z_{2^n}$，生成二元组 (x, c_0+x)， 并与 P_1 （持有输入 c_1）执行 1-out-of-2 COT_n，P_1 拿到结果 y_1。P_0 设置 $y_0=2^n-x$.
- 双方本地线性修正，P_0 计算 $d_0=c_0-2y_0$，P_1 计算 $d_1=c_1-2y_1$.

验证一下结果，$d_0+d_1=c_0+c_1-2(y_0+y_1)$。

- 当 $c_1=0$ 时，$y_0+y_1=(2^n-x)+x=2^n$，故 $d_0+d_1=c_0+c_1 \pmod{2^n}$。
- 当 $c_1=1$ 时，$y_0+y_1=(2^n-x)+(c_0+x)=2^n+c_0$，故 $d_0+d_1=c_0+c_1-2c_0=1-c_0$. 但此时 $c=c_0\oplus 1=1-c_0$，因此 $d=c$ 仍然成立。

通信开销为一次 1-out-of-2 COT_n，成本为 $\lambda+n$.

$\mathcal{F}_{ZExt}$

我们有了前面的 $\mathcal{F}_{Wrap}$ 和 $\mathcal{F}_{B2A}$，构造 $\mathcal{F}_{ZExt}$ 便是自然的事情。对于 m-bit 的加法共享，我们尝试将其零扩展到 n-bit $(n > m)$。首先，我们要 check 这两个 share 是否有进位（使用 $\mathcal{F}_{Wrap}$），然后将得到的布尔进位 w 使用 $\mathcal{F}_{B2A}$ 转为 $Z_{2^{n-m}}$ 的算术值。

但我们只是做零扩展操作，两个 share 相加，不能在第 m 位产生进位，因此双方要在 Z_{2^n} 减掉一个 2^m 的 share，使用 $2^m *_n w$ 即可。($*_n$ 运算也可以使用 $\mathcal{F}_{MUX}$ 替代。)

成本为 $\mathrm{Comm}(\mathcal{F}_{Wrap} + \mathcal{F}_{ZExt}) = \lambda m + 14m + \lambda + (n-m) = \lambda(m+1) + 13m + n.$

$\mathcal{F}_{TR}$

既然我们有了从小到大的 $\mathcal{F}_{ZExt}$，那自然也有反过来的 $\mathcal{F}_{TR}$。我们假设从 l-bit 截断低位的 s-bit，并输出最终的高位 $l-s$-bit：

SETUP: P_b 将原来的 share x_b 拆成 $u_b || v_b$，前者为 $l-s$ 位，后者为 s 位，可以证明：

$$TR(x, s) = u_0 + u_1 + Wrap(v_0, v_1, s)$$

开销为 $\mathrm{Comm}(\mathcal{F}_{Wrap} + \mathcal{F}_{B2A}) = (\lambda s + 14s) + (\lambda + (l-s)) = \lambda(s+1) + 13s + l.$

$\mathcal{F}_{CrossTerm}$

$\mathcal{F}_{CrossTerm}$ 与用 beaver triple 的安全乘法比较相近，但区别是后者 P_0 和 P_1 都知道 x 和 y 的一部分 share。而 $\mathcal{F}_{CrossTerm}$ 的适用条件是 P_0 独占 x，P_1 独占 y，最后各自获得长度 $l = m + n$ 的 $x * y$ 的 share.

- P_0 将自己的 x 写成二进制 $x = \sum_{i=0}^{m-1} x_i 2^i, \quad x_i \in \{0, 1\}$.
- 对 $i \in [0, m-1]$, 调用 1-out-of-2 COT_{l-i}：P_0 持有 x_i，P_1 持有 y，生成 $\langle t_i \rangle^{l-i}$ 满足 $t_i = x_i \cdot y$.
- 双方本地计算 $\langle z \rangle^l = \sum_{i=0}^{m-1} \langle t_i \rangle^{l-i}$，显然两个share加起来就是乘积$x \cdot y$.

总通信成本为 $\sum_{i=0}^{m-1} (\lambda + (l-i)) = m\lambda + ml - \frac{m(m-1)}{2} = O(m\lambda + mn)$.

$\mathcal{F}_{UMult}$

语义是双方持有 $x = x_0 + x_1 \pmod{2^m}, \quad y = y_0 + y_1 \pmod{2^n}$，目标是计算 $z = x \cdot y \in Z_{2^{m+n}}$.

这里还是不能直接做 beaver triple，因为它是 ring agnostic 的，可能会带来未知的 wrap 问题。当然一个办法是将 x, y 都扩展到足够大的环，做安全乘法后又截断回去。当然这里有两个问题，一是 $\mathcal{F}_{TR}$ 是高位截断而不是低位截断，协议需要稍微改一改；二是这种办法运算成本较高，不如接下来介绍的专用 $\mathcal{F}_{UMult}$ 协议。

我们现在要计算$(x_0+x_1)(y_0+y_1)=x_0y_0+x_1y_1+x_0y_1+x_1y_0$，前两者可以分别由 P_0,P_1 离线完成。而后两者就涉及到刚才的 $\mathcal{F}_{CrossTerm}$ 了。总之，我们有了在 P_0 存储的完整 x_0y_0，P_1 存储的 x_1y_1，以及双方都拥有的 $\langle x_0y_1\rangle,\langle x_1y_0\rangle$ 的一部分。$\mathcal{F}_{UMult}$ 的作用就是把这 4 项安全地拼起来，并且处理 wrap 的问题。

- 首先，我们要考虑 x_0+x_1,y_0+y_1 是否溢出的问题，因此需要 $\mathcal{F}_{Wrap}$ 进行检测得到两个溢出位 w_x,w_y。
- 然后我们可以利用 $\mathcal{F}_{MUX}$ 计算 `g=w_y?x:0` 与 `h=w_x:y:0` 的 share $\langle g\rangle,\langle h\rangle$，处理可能的溢出差值。
- 最终 P_b 输出 $x_by_b+\langle x_0y_1\rangle_b+\langle x_1y_0\rangle_b-2^n\langle g\rangle_b-2^m\langle h\rangle_b$.

尝试把两个 share 加起来看看：$\sum_b(x_by_b+\langle x_0y_1\rangle_b+\langle x_1y_0\rangle)=(x_0+x_1)(y_0+y_1)$

而 $\sum_b(-2^n\langle g\rangle_b-2^m\langle h\rangle_b)=-2^ng-2^mh=-2^n(w_y\cdot x)-2^m(w_x\cdot y)$

合在一起，$z=(x_0+x_1)(y_0+y_1)-w_y\cdot(x\cdot 2^n)-w_x\cdot(y\cdot 2^m)$

$=(x+w_x2^m)(y+w_y2^n)-w_y\cdot(x\cdot 2^n)-w_x\cdot(y\cdot 2^m)=xy$，刚好抵消。

设$\nu=\max(m,n),l=m+n$，论文给出了具体的开销，数量级为 $O(\lambda\nu+\nu^2)$。相比于 beaver triple 做法（含生成乘法三元组）的 $O(\lambda l+l^2)$ 数量级相同，但论文声称 $\mathcal{F}_{UMult}$ 通信少 $1.5\times$.

$\mathcal{F}_{SMult}$ 与 $\mathcal{F}_{UMult}$ 原理类似，通信量完全相同，这里就略过了。

$\mathcal{F}_{DigDec}$

作用是将一个 l-bit 的 share $\langle x \rangle$，按 d-bit 分块，拆分成 $c = \lceil \frac{l}{d} \rceil$ 个 $\{\langle z_i \rangle\}_{i=0}^{c-1}$，其中 $x = z_{c-1}||z_{c-2}||\cdots||z_0$. 显然，本地进行这个操作会导致 wrap 问题。

当然解决这个问题的思路也容易想到，首先使用 $\mathcal{F}_{wrap}$ 判断 $\langle x_{[0,d-1]} \rangle$ 的两个 share 是否溢出，并得到溢出结果的 share $\langle c_0 \rangle$，令 $\langle z_0 \rangle = \langle x_{[0,d-1]} \rangle - 2^d \langle c_0 \rangle$，并将目前双方的高位 $\langle x_{[d,l-1]} \rangle$ 加上 $\langle c_0 \rangle$，最后递归执行这一过程即可。

复杂度相当于 $(c-1)$ 次 $\mathcal{F}_{wrap}$ 的成本，也就是 $(c-1)(\lambda d + 14d)$.

$\mathcal{F}_{MSNZB-P}$

检测第 i 个 d-bit 分块中最高非零位对应的全局 index，也可以理解成 $\lfloor \log_2(z_i) \rfloor + i \cdot d$，这里的具体实现方法是直接 $\mathcal{F}_{LUT}$ 解决。但有一个问题是当 $z_i = 0$ 时右边这个式子与左边实现的语义并不相同，作者这里选择让 $z_i = 0$ 的结果未定义，交给上层协议来解决。

> 这一点我不是很理解，$z_i = 0$ 的情况也可以在 $\mathcal{F}_{LUT}$ 的情况直接解决啊，又不会增加通信成本。

总之，通信量等价于一个 1-out-of-2^d $\mathrm{OT}_{\lceil \log_2 l \rceil}$ 实现，因此通信量为 $2\lambda + 2^d \lceil \log_2 l \rceil$.

$\mathcal{F}_{Zeros}, \mathcal{F}_{OneHot}$

前者的意思是判断一个长度为 d 的向量（或者 d-bit 数）是否为零。而后者是将一个 $\langle k \rangle \in [0, l-1]$ 的 share，转化为一个长度为 l 的向量 share，满足和为 $(0, 0, \cdots, 1, \cdots, 0)$，其中仅有第 k 个向量为 1.

这两个分别可以用 1-out-of-2^d OT 和 1-out-of-l OT_l 实现，成本分别为 $2\lambda + 2^d$ 和 $2\lambda + l^2$。虽然后者成本较高，但只在父协议 $\mathcal{F}_{MSNZB}$ 的最后调用一次，因此总成本还能接受。

$\mathcal{F}_{MSNZB}$

含义是给定一个输入 $\langle x \rangle^l$，计算出 l 的最高零位的值 k，输出向量 share 满足和为 $(0, 0, \cdots, 1, \cdots, 0)$，其中仅有第 k 个向量为 1。 为了简单起见，这里我假设 $\mathcal{F}_{MSNZB-P}$ 是定义良好的（也就是处理了 $z_i = 0$ 的情况）。令 $\iota = \lceil \log_2 l \rceil$:

- 首先计算输入 $\langle x \rangle^l$ 做 $\mathcal{F}_{DigDec}$ 得到 $\langle y_i \rangle^d$，然后对每个 i 调用 $\mathcal{F}_{MSNZB-P}$ 得到 $\langle u_i \rangle^\iota$.
- 调用 $\mathcal{F}_{Zeros}$ 得到布尔分享 $\langle v_i \rangle$，用 $\mathcal{F}_{AND}$（也就是 beaver triple）对于任意 $i = c-2, \cdots, 0$ 做 $w_i = w_{i+1} \wedge v_{i+1}$. 这样 $w_i = \prod_{j>i} v_j$. 这样最高非零位的值便是满足 $w_i = 1$ 的从大到小的最后一个。
- 对最高位 digit，令 $\langle z'_{c-1} \rangle^\iota = \langle u_{c-1} \rangle^\iota$，对于剩下 $i = c-2, \cdots, 0$，$z'_i = \mathrm{MUX}(w_i, u_i)$.
- 最后本地算出 $\tilde{z} = \sum_{i=0}^{c-1} z'_i$，并调用 $\mathcal{F}_{OneHot}$ 得到最终的布尔向量 $\{\langle z_k \rangle\}_{k \in [0, l-1]}$。

总结

原语	依赖的原语	功能	通信开销
$\mathcal{F}_{MUX}$	$\binom{2}{1}-\mathrm{OT}_l$	长度l位的三目运算符	$2\lambda+2l$
$\mathcal{F}_{OR}$	$\mathcal{F}_{AND}$, i.e. beaver triple	逻辑或	$\lambda+20$
$\mathcal{F}_{EQ}$	$\binom{2^m}{1}-\mathrm{OT}, \mathcal{F}_{AND}$	长度l位的算术相等	$<\frac{3}{4}\lambda l+9l$
$\mathcal{F}_{LT/GT}$	$\binom{2^m}{1}-\mathrm{OT}, \mathcal{F}_{AND}$	长度l位的算术比较	$<\lambda l+14l$
$\mathcal{F}_{LUT}$	$\binom{2^m}{1}-\mathrm{OT}_n$	长度m位，结果 n 位的查找表	$2\lambda+2^m n$
$\mathcal{F}_{ZExt}$	$\mathcal{F}_{Wrap}, \mathcal{F}_{B2A}$	m 位零扩展到 n 位	$\lambda(m+1)+13m+n$
$\mathcal{F}_{TR}$	$\mathcal{F}_{Wrap}, \mathcal{F}_{B2A}$	l 位截取高 $l-s$ 位	$\lambda(s+1)+l+13s$
$\mathcal{F}_{UMult}, \mathcal{F}_{SMult}$	$\mathcal{F}_{CrossTerm}, \mathcal{F}_{Wrap}, \mathcal{F}_{MUX}$	长度m,n位的无/有符号乘	$O(\lambda l+l^2)$
$\mathcal{F}_{MSNZB}$	$\mathcal{F}_{DigDec}, \mathcal{F}_{MSNZB-P}, \mathcal{F}_{OneHot}$等	长度l位的最高非零位index	$\leq\lambda(5l-4)+l^2$

Primitives

我们终于构建了 secfloat 论文对应的所有基础协议，现在我们转入本篇论文构造的重要原语。

$\mathcal{F}_{FPcheck}$