New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning\textsuperscript{a}

Presented at S&P 2019

Ivan Damgård\textsuperscript{1}  Daniel Escudero\textsuperscript{1}  Tore Frederiksen\textsuperscript{2}  Marcel Keller\textsuperscript{3}  Peter Scholl\textsuperscript{1}  Nikolaj Volgushev\textsuperscript{2}

\textsuperscript{1}Aarhus University, Denmark
\textsuperscript{2}Alexandra Institute, Denmark
\textsuperscript{3}Data61, CSIRO, Australia

\textsuperscript{a}Supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreements No 669255 (MPCPRO), No 731583 (SODA) and the Danish Independent Research Council under Grant-ID DFF6108-00169 (FoCC).
Introduction
MPC

Alice → $x_1$ → Trusted Party → $x_3$ → Charlie

Bob → $x_2$ → Trusted Party → $x_4$ → Dave
Many different approaches to MPC

Circuits over $\mathbb{F}_2$
- Garbled Circuits
- BMR
- GMW
- ...

Circuits over $\mathbb{F}_p$
- BGW
- BeDOZa
- SPDZ
- MASCOT
- ...

2
Many different approaches to MPC

- Sharemind, Bogdanov et al. ESORICS 2008
- Replicated SS with Active Security, Furukawa et al. EUROCRYPT 2017
- Passive-To-Active Compiler, Damgård et al. CRYPTO 2018
- SPD$_{Z_{2k}}$, Cramer et al. CRYPTO 2018.
Our Contribution

New sub-protocols for $\text{SPD}Z_{2^k}$

We expand $\text{SPD}Z_{2^k}$ with a series of sub-protocols to enhance the potential range of applications.

- Arithmetic-Binary share conversions
- Random-bit generation
- Bit-decomposition
- Secure truncation, comparison and equality check.
We implement the $\text{SPDZ}_{2^k}$ protocol in Java, as part of the \textit{FRamework for Efficient Secure COmputation} (FRESCO).

- Our implementation contains several optimizations that can be of independent interest.
- In the microbenchmarks we observe several improvements with respect to other protocols over fields.
Applications to Secure Machine Learning

We illustrate the benefits of our techniques by performing certain ML tasks in SPD\(\mathbb{Z}_{2^k}\) and observe several improvements with respect to other protocols over fields. We consider:

- Secure evaluation of Decision Trees
- Secure evaluation of Support Vector Machines
$\text{SPD}_{\mathbb{Z}_{2^k}}$
SPD$\mathbb{Z}_{2^k}$ in a nutshell

Additive Authenticated Secret-Sharing over $\mathbb{Z}_{2^k}$

$x \in \mathbb{Z}_{2^k}$ is shared, denoted by $[x]_{2^k}$, if

- Each $P_i$ has $x^i, \alpha^i, m^i \in \mathbb{Z}_{2^{k+s}}$
- $\sum x^i \equiv_{k+s} x'$ with $x' \equiv_k x$
- $\sum \alpha^i \equiv_{k+s} \alpha$, where $\alpha \in \mathbb{Z}_{2^s}$ is a random global key
- $\sum m^i \equiv_{k+s} \alpha \cdot x'$.

$x \equiv y \mod 2^\ell$ is abbreviated by $x \equiv_\ell y$
\[\sum_{i=1}^{n} x_i \equiv k + s\]
\begin{equation}
\equiv_{k+s}
\end{equation}

\begin{align*}
\frac{k+s}{x^1} + \frac{k+s}{x^2} + \cdots + \frac{k+s}{x^n} \equiv_{k+s} \frac{k}{x'}
\end{align*}
Secure computation with preprocessing

Input phase

\[ [x_i]_{2^k} = (x_i - r_i) + [r_i]_{2^k} \]

where \( x_i \) are the inputs and \((r_i, [r_i]_{2^k})\) is preprocessed.

Addition gates

\[ [x + y]_{2^k} = [x]_{2^k} + [y]_{2^k} \]

Multiplication gates

\[ [x \cdot y]_{2^k} = [c]_{2^k} + (x - a) \cdot [b]_{2^k} + (y - b) \cdot [a]_{2^k} + (x - a)(y - b) \]

where \(([a]_{2^k}, [b]_{2^k}, [c]_{2^k})\) is preprocessed with \( c = a \cdot b \).
Primitives for MPC Modulo $2^k$
Random Bit

$\mathbb{Z}_2 \rightarrow \mathbb{Z}_{2^k}$

$\mathbb{Z}_{2^k} \rightarrow \mathbb{Z}_2$

$\langle \cdot \rangle \rightarrow [\cdot]_2$

$\mathbb{Z}_2$ Triple

BitDec

SVM

Decision Trees
Generating Random Bits \([b]_{2^k}\) (Intuition)

1. Sample \([r]_{2^k}\) at random and let \([a]_{2^k} = [r^2]_{2^k}\).
2. Open \(a\). Let \(c\) be some square root of \(a\).
3. Compute \([d]_{2^k} = c^{-1}[r]_{2^k}\).
   - Now \(d\) is a random square root of 1, so \(d \in R \{-1, +1\}\).
4. Output \([b]_{2^k}\), where \(b = (d + 1)/2\).
Generating Random Bits $[b]_{2^k}$ (Intuition)

**Ideal Protocol**

1. Sample $[r]_{2^k}$ at random and let $[a]_{2^k} = [r^2]_{2^k}$.
2. Open $a$. Let $c$ be some square root of $a$.
3. Compute $[d]_{2^k} = c^{-1}[r]_{2^k}$.
   - Now $d$ is a random square root of 1, so $d \in \mathbb{R}\{-1, +1\}$.
4. Output $[b]_{2^k}$, where $b = (d + 1)/2$. 
Generating Random Bits $[b]_{2^k}$ (Intuition)

**Actual Protocol**

1. Sample $[r]_{2^{k+2}}$ at random, where $r$ is odd, and let $[a]_{2^{k+2}} = [r^2]_{2^{k+2}}$.
2. Open $a$. Let $c$ be some square root of $a$.
3. Compute $[d]_{2^{k+2}} = c^{-1}[r]_{2^{k+2}}$
   - Now $d$ is a random square root of $1 \mod 2^{k+2}$, so $d \in_R \{-1, +1, -1 + 2^{k+1}, +1 + 2^{k+1}\}$.
4. Output $[b]_{2^k}$, where $b \equiv_k (d + 1)/2$. 


Local reduction modulo 2.\textsuperscript{a}

\textsuperscript{a}In fact, it is reduction modulo 2^{s+1} for the extra \( s \) “MAC” bits.

1. Sample a random bit \([r]_{2^k} (r \in \mathbb{Z}_2)\)
2. Convert \([r]_{2^k}\) to \([r]_2\).
3. Open \([c] = [b]_2 \oplus [r]_2\)
4. Output \([b]_{2^k} = [r]_{2^k} + [c]_{2^k} - 2[r]_{2^k}[c]_{2^k}\)
Bit Decomposition: $[x]_{2^k} \rightarrow ([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k})$

1. Sample random bits $[r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k}$ and let $[r]_{2^k} = \sum_{i=0}^{k-1} 2^i [r_i]_{2^k}$.
2. Compute $[a]_{2^k} = [x]_{2^k} - [r]_{2^k}$ and open $a$.

4. Compute the binary circuit

$([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k}) = \text{ADD} ((a_0, \ldots, a_{k-1}), ([r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k})).$
Bit Decomposition: \([x]_{2^k} \rightarrow ([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k})\)

1. Sample random bits \([r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k}\) and let \([r]_{2^k} = \sum_{i=0}^{k-1} 2^i[r_i]_{2^k}\).
2. Compute \([a]_{2^k} = [x]_{2^k} - [r]_{2^k}\) and open \(a\).
3. Convert \(([r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k})\) to \(([r_0]_2, \ldots, [r_{k-1}]_2)\).
4. Compute the binary circuit

\[
([x_0]_2, \ldots, [x_{k-1}]_2) = \text{ADD} ((a_0, \ldots, a_{k-1}), ([r_0]_2, \ldots, [r_{k-1}]_2)).
\]
5. Convert the result \(([x_0]_2, \ldots, [x_{k-1}]_2)\) to \(([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k})\).
Implementation and Benchmarks
Online Phase - Micro Operations

Throughput in elements per second for the online phase of micro operations over 1 Gbps network. The factor columns express the runtime improvement factor of $\text{SPDZ}_{2k}$ over SPDZ in FRESCO.

<table>
<thead>
<tr>
<th></th>
<th>$k = 32$</th>
<th>$k = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPDZ$_{2k}$ ($\sigma = 26$)</td>
<td>SPDZ ($\sigma = 26$)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>687041</td>
<td>141346</td>
</tr>
<tr>
<td>Equality</td>
<td>15334</td>
<td>3213</td>
</tr>
<tr>
<td>Comparison</td>
<td>9153</td>
<td>1769</td>
</tr>
</tbody>
</table>
Online Phase for SVMs Evaluation

Online phase benchmarking of SVM evaluation over 1 Gbps network. The factor columns express the runtime improvement factor of SPDZ\(_{2k}\) over SPDZ in FRESCO. Times are in milliseconds per sample.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Num. Classes, Features</th>
<th>Batch Size</th>
<th>[k = 32, \sigma = 26]</th>
<th>[k = 64, \sigma = 57]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>SPDZ(_{2k})</td>
<td>SPDZ</td>
</tr>
<tr>
<td>CIFAR</td>
<td>10, 2048</td>
<td>1</td>
<td>82 ms</td>
<td>214 ms</td>
</tr>
<tr>
<td>MIT</td>
<td>67, 2048</td>
<td>1</td>
<td>379 ms</td>
<td>1318 ms</td>
</tr>
<tr>
<td>ALOI</td>
<td>463, 128</td>
<td>1</td>
<td>242 ms</td>
<td>857 ms</td>
</tr>
<tr>
<td>CIFAR</td>
<td>10, 2048</td>
<td>5</td>
<td>39 ms</td>
<td>168 ms</td>
</tr>
<tr>
<td>MIT</td>
<td>67, 2048</td>
<td>5</td>
<td>225 ms</td>
<td>1101 ms</td>
</tr>
<tr>
<td>ALOI</td>
<td>463, 128</td>
<td>5</td>
<td>162 ms</td>
<td>741 ms</td>
</tr>
</tbody>
</table>
Online phase benchmarking of evaluation of decision trees over 1 Gbps network. The factor columns express the runtime improvement factor of SPDZ\(_{2k}\) over SPDZ in FRESCO. Times are in milliseconds per sample.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Depth, Num. Features</th>
<th>Batch Size</th>
<th>(k = 32, \sigma = 26)</th>
<th>(k = 64, \sigma = 57)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>SPDZ(_{2k})</td>
<td>SPDZ</td>
</tr>
<tr>
<td>Hill Valley</td>
<td>3, 100</td>
<td>1</td>
<td>21 ms</td>
<td>24 ms</td>
</tr>
<tr>
<td>Spambase</td>
<td>6, 57</td>
<td>1</td>
<td>48 ms</td>
<td>104 ms</td>
</tr>
<tr>
<td>Diabetes</td>
<td>9, 8</td>
<td>1</td>
<td>80 ms</td>
<td>215 ms</td>
</tr>
<tr>
<td>Hill Valley</td>
<td>3, 100</td>
<td>5</td>
<td>6 ms</td>
<td>10 ms</td>
</tr>
<tr>
<td>Spambase</td>
<td>6, 57</td>
<td>5</td>
<td>14 ms</td>
<td>40 ms</td>
</tr>
<tr>
<td>Diabetes</td>
<td>9, 8</td>
<td>5</td>
<td>41 ms</td>
<td>185 ms</td>
</tr>
</tbody>
</table>
Triple Generation Throughput

- SPDZ$_{2^k}$ ($k = 32, \sigma = 26$)
- SPDZ$_{2^k}$ ($k = 64, \sigma = 57$)
- Mascot (128 bit field)
- Overdrive ($k = 64$ (128 bit field), $\sigma = 57$)
- Overdrive ($k = 32$ (64 bit field), $\sigma = 40$)

(a) WAN (50 Mbps, 100 ms latency)
(b) LAN (1 Gbps, 0.1 ms latency)
(c) LAN (10 Gbps, 0.1 ms latency)
Total theoretical communication complexity counted in (kilo-, mega-, giga-) bytes for the two-party case. Values for SPDZ are based on Overdrive in Low Gear. For bit triples we use the optimized TinyOT WRK protocol.

<table>
<thead>
<tr>
<th></th>
<th>$k = 32$, $\sigma = 26$</th>
<th>$k = 64$, $\sigma = 57$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPD$\mathbb{Z}_{2^k}$</td>
<td>SPDZ (64 bit field)</td>
</tr>
<tr>
<td></td>
<td>Preprocessing</td>
<td>Online</td>
</tr>
<tr>
<td>Comparison</td>
<td>627 KB</td>
<td>46 B</td>
</tr>
<tr>
<td>Equality</td>
<td>486 KB</td>
<td>24 B</td>
</tr>
<tr>
<td>DTREE (Diabetes)</td>
<td>209 MB</td>
<td>131 KB</td>
</tr>
<tr>
<td>SVM (ALOI)</td>
<td>908 MB</td>
<td>1.44 KB</td>
</tr>
</tbody>
</table>
Conclusions

- We implemented the SPD$\mathbb{Z}_{2^k}$ protocol along with practical primitives for MPC mod $2^k$.
- We saw up to a 5-fold improvement in computation for various tasks, and up to a 85-fold reduction in online communication costs for secure comparison, as compared to the field setting.

Future Work

- Close the gap for the preprocessing.
- Expand the range of applications for computation modulo $2^k$. 
Thank you!
How to Compute the Binary Circuit?

SPD$\mathbb{Z}_{2^k}$ does not support triple generation for $k = 1$.

**Share Conversion** $\langle x \rangle \rightarrow [x]_2$

Let $\langle \cdot \rangle$ denote any authenticated sharing mechanism over $\mathbb{Z}_2$ (e.g. TinyOT shares).

- **INPUT:** Sharings $\langle x_1 \rangle, \ldots, \langle x_m \rangle$.
- **OUTPUT:** Binary sharings $[x_1]_2, \ldots, [x_m]_2$. 

1. Sample $s$ additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$. 
1. Sample $s$ additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.

2. Each party $P_i$ inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$.
1. Sample $s$ additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.

2. Each party $P_i$ inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$.

3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1]^2, \ldots, [x_m]^2$ and $[r_1]^2, \ldots, [r_s]^2$. 
1. Sample $s$ additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.

2. Each party $P_i$ inputs the shares $x^i_1, \ldots, x^i_m, r^i_1, \ldots, r^i_s$ to obtain $[x^i_1], \ldots, [x^i_m], [r^i_1], \ldots, [r^i_s]$.

3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1]_2, \ldots, [x_m]_2$ and $[r_1]_2, \ldots, [r_s]_2$.

4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow R \{0, 1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.
1. Sample $s$ additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.

2. Each party $P_i$ inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$.

3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1^2], \ldots, [x_m^2]$ and $[r_1^2], \ldots, [r_s^2]$.

4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0, 1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.

5. For each $j$, open $[r_j^2] \oplus \sum_{i=1}^{m} \chi_{i,j} [x_i^2] \text{ as } y_j$ and $\langle r_j^2 \rangle \oplus \sum_{i=1}^{m} \chi_{i,j} \langle x_i \rangle \text{ as } y_j'$. 
1. Sample $s$ additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.

2. Each party $P_i$ inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$.

3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1], \ldots, [x_m], [r_1], \ldots, [r_s]$.

4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0, 1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.

5. For each $j$, open $[r_j] \oplus \sum_{i=1}^m \chi_{i,j} \cdot [x_i]$ as $y_j$ and $\langle r_j \rangle \oplus \sum_{i=1}^m \chi_{i,j} \cdot \langle x_i \rangle$ as $y'_j$.

6. Check that $y_j = y'_j$ for all $j$. If not, abort.
1. Sample $s$ additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.

2. Each party $P_i$ inputs the shares $x^i_1, \ldots, x^i_m, r^i_1, \ldots, r^i_s$ to obtain $[x^i_1], \ldots, [x^i_m], [r^i_1], \ldots, [r^i_s]$.

3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1], \ldots, [x_m]$ and $[r_1], \ldots, [r_s]$.

4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0,1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.

5. For each $j$, open $[r_j] \oplus \sum_{i=1}^{m} \chi_{i,j} \cdot [x_i]$ as $y_j$ and $\langle r_j \rangle \oplus \sum_{i=1}^{m} \chi_{i,j} \cdot \langle x_i \rangle$ as $y'_j$.

6. Check that $y_j = y'_j$ for all $j$. If not, abort.

7. Output the sharings $[x_1], \ldots, [x_m]$. 