AGENDA

• Introducing the JavaScript/WebAssembly port
• Browser example (simple integers)
• Performance Review
• Javascript (Node.js)/WebAssembly examples
  • Threshold FHE
  • Proxy Re-Encryption
  • Buffer-based serialization
• Considerations and Future Work
JavaScript/WebAssembly Port

Performing homomorphic operations in Browser or NodeJS
Introducing the Javascript / WebAssembly port

- [https://gitlab.com/palisade/palisade-wasm](https://gitlab.com/palisade/palisade-wasm)

- We are developing an npm package that provides access to the Palisade homomorphic encryption library to JavaScript / Typescript users
  - Currently in alpha, API is currently subject to change

- Currently optimized for the following schemes
  - BGVrns
  - CKKS
  - FHEW

- WebAssembly environment is typically limited to 4GB RAM
Using the JavaScript/WebAssembly Port
Supported Examples

- **BINFHE**
  - Boolean circuits
  - Boolean circuits with buffer-based serialization

- **PKE**
  - PRE buffer
  - Simple Integer
  - Simple Integer with BGVns scheme
  - Simple Integer with buffer-based serialization
  - Simple Real Numbers
  - Threshold FHE

All examples demonstrated in previous Palisade webinars are supported by the WebAssembly Port
While the JS API docs are still a work in progress, the names in the JS port closely follow the underlying C++ functions (with some slight alterations)

<table>
<thead>
<tr>
<th>C++</th>
<th>JS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CryptoContext&lt;DCRTPoly&gt;</td>
<td>CryptoContextDCRTPoly</td>
</tr>
<tr>
<td>EvalMult(Ciphertext,Ciphertext)</td>
<td>EvalMultCipherCipher</td>
</tr>
<tr>
<td>AND</td>
<td>BINGATE.AND</td>
</tr>
</tbody>
</table>
Capabilities

- All examples described in previous Palisade webinars can be re-created in JavaScript by using the WebAssembly port.

- WebAssembly users can perform homomorphic encryptions over:
  - Boolean circuits
  - Integers
  - Real Numbers

- Encryption keys can be serialized and deserialized to communicate over the network with peers:
  - Since file access from WebAssembly is somewhat complex, only buffer serialization is currently supported.

```cpp
// Serialize cryptocontext in C++
Serial::SerializeToFile(
    DATA_FOLDER + "/cryptocontext.txt",
    cryptoContext,SerType::BINARY);

// Serialize cryptocontext in JavaScript
const cryptoContextBuffer = 
    module.SerializeCryptoContextToBuffer(
        cryptoContext,module.SerType.BINARY);
```
Examples: Web Demo

Simple Integers
Typescript bindings

- The JavaScript port will also feature typescript bindings, which can catch errors at compile time rather than runtime
  - Also provides accurate autocomplete!

- This provides many of the structural guarantees of C++ while maintaining the flexibility of JavaScript’s OO and functional patterns
Documentation

- Doxygen for C/C++ to WASM:

```
PALISADE Lattice Crypto Library 0.0.1

A lattice crypto library for software engineers by software engineers.

CryptoContext em.csp File Reference

Typedefs
using Cmd = CryptoContext<NCryptoPoly>;```

- TypeDoc/TSDoc for WASM => Typescript (Coming soon):
Performance Review
Performance

- **GCC**
  
  bin/benchmark/lib-benchmark

  Run on (8 X 4700 MHz CPU s)

  CPU Caches:
  - L1 Data: 32 KiB (x8)
  - L1 Instruction: 32 KiB (x8)
  - L2 Unified: 256 KiB (x8)
  - L3 Unified: 12288 KiB (x1)

  Load Average: 0.09, 0.39, 0.96

- **NodeJS WASM**
  
  bin/lib-benchmark.js

- **Clang**
Performance

WebAssembly Performance Comparison per Scheme

"KeyGen" Operation

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>1108</td>
</tr>
<tr>
<td></td>
<td>1103</td>
</tr>
<tr>
<td></td>
<td>919</td>
</tr>
<tr>
<td>WASM</td>
<td>5301</td>
</tr>
<tr>
<td></td>
<td>5397</td>
</tr>
<tr>
<td></td>
<td>5013</td>
</tr>
</tbody>
</table>

- BFVrns
- CKKS
- BGVrns

PALISADE
Performance

WebAssembly Performance Comparison per Scheme

"Encryption" Operation

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Native Time (μs)</th>
<th>WASM Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFVrns</td>
<td>1058</td>
<td>5759</td>
</tr>
<tr>
<td>CKKS</td>
<td>960</td>
<td>5473</td>
</tr>
<tr>
<td>BGVrns</td>
<td>1072</td>
<td>5544</td>
</tr>
</tbody>
</table>

Legend:
- Native
- WASM
Performance

WebAssembly Performance Comparison per Scheme

"Decryption" Operation

Time (μs)

<table>
<thead>
<tr>
<th></th>
<th>Native</th>
<th>WASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFVnrs</td>
<td>207</td>
<td>1970</td>
</tr>
<tr>
<td>CKKS</td>
<td>430</td>
<td>2173</td>
</tr>
<tr>
<td>BGVnrs</td>
<td>107</td>
<td>532</td>
</tr>
</tbody>
</table>
Performance

WebAssembly Performance Comparison per Scheme

"Add" Operation

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Native Time (μs)</th>
<th>WASM Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFVrns</td>
<td>16.5</td>
<td>68.9</td>
</tr>
<tr>
<td>CKKS</td>
<td>16.3</td>
<td>69.5</td>
</tr>
<tr>
<td>BGVrns</td>
<td>21.3</td>
<td>79</td>
</tr>
</tbody>
</table>
Performance

WebAssembly Performance Comparison per Scheme

"MultReLin" Operation

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Time (μs)</th>
<th>Native</th>
<th>WASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFVrns</td>
<td>75842</td>
<td>4740</td>
<td>104</td>
</tr>
<tr>
<td>CKKS</td>
<td>696</td>
<td>104</td>
<td>696</td>
</tr>
<tr>
<td>BGVrns</td>
<td>524</td>
<td>102</td>
<td>524</td>
</tr>
</tbody>
</table>
Performance

- For integer calculations, **BGVrns** is the preferred scheme.
- For real number calculations, use **CKKS**.
- **BGVrns** and **CKKS** are optimized for WebAssembly.
- Multiplying/rotation of ciphertexts suffers approximately a **5X** slowdown with **BGVrns**, but receives a **20X** slowdown with **BFVrns**.

![WebAssembly Performance Comparison per Scheme](chart.png)
NodeJS Examples
Examples: Lives Demos

- Threshold FHE
- Proxy Re-Encryption
- Buffer-based serialization
Considerations and Future Work
Considerations of JavaScript/WebAssembly Port

- **Benefits**
  - Develop web-based homomorphic encryption applications with comparable performance when compared to native execution.

- **Limitations**
  - Currently generation of crypto context supports a limited number of parameters. This is work in progress.
  - Whereas C++ detects variables going out of scope, JavaScript users must explicitly call `.delete()` on every C++ handle they receive i.e. generated crypto context.
  - OpenMP is not available for WebAssembly yet.
  - Very limited SIMD support mostly through emulation. Only the 128-bit wide instructions from AVX instruction set are available. 256-bit wide AVX instructions are not provided.

- **Alternatives**
  - Other binding options are available where users can write their own WASM bindings.
  - C++ addons can also be used.
Future Work

● Optimize other schemes for WebAssembly

● Potential support for multi-threading by WebAssembly

● Examples to highlight potential use cases of the palisade-wasm
  ○ Develop web-based homomorphic solution
  ○ Networked Threshold-FHE, Proxy-ReEncryption

● We’d love more contributors. Please try Palisade-wasm and help us improve it.

● Please Help us spread the word!
THANK YOU!

https://palisade-crypto.org