webSPDZ: Versatile MPC on the Web

Thomas Buchsteiner[1], Karl W. Koch[1,2], Dragos Rotaru[3], Christian Rechberger[1]

[1] Graz University of Technology, Austria
christian.rechberger@tugraz.at
thomas.buchsteiner@gmail.com
karl.koch@tugraz.at
[2] Secure Information Technology Center Austria (A-SIT), Graz, Austria
[3] Gateway, New York, USA
dragos@mygateway.xyz

Abstract. Multi-party computation (MPC) has become increasingly practical in the last two decades, solving privacy and security issues in various domains, such as healthcare, finance, and machine learning. One big caveat is that MPC sometimes lacks usability since the knowledge barrier for *regular* users can be high. Users have to deal with, e.g., various CLI tools, private networks, and sometimes even must install many dependencies, which are often hardware-dependent.

A solution to improve the usability of MPC is to build browser-based MPC engines where each party runs within a browser window. Two examples of such an MPC web engine are JIFF and the web variant of MPyC. Both support an honest majority with passive corruptions.

`webSPDZ`: Our work brings one of the most performant and versatile general-purpose MPC engines, MP-SPDZ, to the web. MP-SPDZ supports ≥40 MPC protocols with different security models, enabling many security models on the web. To port MP-SPDZ to the web, we use Emscripten to compile MP-SPDZ's C++ BackEnd to WebAssembly and upgrade the party communication for the browser (WebRTC or WebSockets). We call the new MPC web engine webSPDZ. As with the native versions of the mentioned MPC web engines, MPyC-Web and JIFF, webSPDZ outperforms them in our end-to-end experiments.

We believe that webSPDZ brings forth many interesting and practically relevant use cases. Thus, webSPDZ pushes the boundaries of practical MPC: making MPC more usable and enabling it for a broader community.

Keywords: Privacy-Enhancing Technology· Private Computations· Multi-Party Computation· MP-SPDZ· MPC Web Engine· webSPDZ.

1 Introduction

With the ever-increasing gathering and availability of data, also the amount of use cases to learn from data increased virtually in all areas of our lifes. E.g., in the areas of (i) health care, (ii) smart driving, or (iii) smart energy management.

To realize such computations in a privacy-preserving way, multi-party computation (MPC) has proven to be an *increasingly* practical cryptographic building block. E.g., MPC has been used to protect privacy in (i) detection of early brain tumors through inference of MRI images [32], (ii) reducing fuel consumption through speed advisory systems for vehicles [30], or (iii) saving energy through smart-meter systems in buildings [44]. Already in 2008, the first-known large-scale real-life use case of MPC took place: the popular Danish sugar-beet trading [13]. Moreover, in 2024 MPC has been used to perform *real-life* cross-hospital analytics [7].

One primary aspect to practically enable MPC are ready-to-use MPC engines. Especially general-purpose engines are of interest as they *virtually* allow arbitrary computations with a dynamic range of participating parties.

Motivation. Imagine a meeting where everyone should participate in a spontaneous privacy-preserving survey. To actively join an MPC computation, one needs to setup the respective MPC engine. E.g., to install all requirements and configure the engine for each party. Although many MPC engines provide Docker containers with tutorial programs, it is still an initial hurdle to get everything running; especially if one is not a *technical expert*.

Bogdanov et al. [11] conducted an end-user survey regarding the potential use of MPC in various areas and use cases. Their survey results underline the relevance of MPC for *technical non-experts*. E.g., in the area of biomedical data Wirth et al. [48] explained the relevance of learning from shared data while ensuring the privacy of thereof. As such, Wirth et al. chose MPC as their privacy-preserving computational building block; although they also noted its practical limitations:

> "SMPC is still rarely used in real-world biomedical data sharing activities due to several barriers, including its technical complexity and lack of usability." [48]

Barak et al. [8] reported the same relevance of *commoditizing MPC* in 2018, and explained the importance and advantage of the web browser for end users:

> "A far more compelling way of carrying out the above study would be to have each company run its own copy of the MPC. However, ..., none of these companies would install software and would only use a browser interface." [8]
> ...
> "As the browser is becoming the new OS for end users, additional features and capabilities are constantly added ... open the possibility of achieving high performance MPC directly in the browser." [8]

Ballhausen et al. [7] reported the use of a native general-purpose MPC engine to analyze real-world cancer-related data across hospitals. Though, they also mentioned practical hurdles of using MPC in this project: "*...included a complex tech stack, demanding implementation, complicated technology management...*". Hence, it is a relevant issue to make MPC more accessible and usable for a broader audience.

One of the most promising cross-platform solutions is to bring MPC engines to the web browser. Users *just* need a compatible browser. Moreover, the portable nature of web-based solutions enables the use of MPC for many end-user devices; e.g., PCs, smartphones, tablets, or even "wearables" like smartwatches in the near future [42]. The smooth setup of web-based solutions with the cross-platform aspect, enables the advantages of MPC for nearly everyone.

State of the Art. Already in 2018, Barak et al. [8] have performed MPC in the web browser and presented benchmarks of several experiments. They built upon `Emscripten`, but reported issues of not having "Single Instruction, Multiple Data" (SIMD) [47] in the browser. However, while the web-app's source code is available on GitHub [16], we did not find the source code of their MPC engine for the browser.

The "JavaScript library for building web apps that employ MPC" (`JIFF`) [22] and the "Web Variant of Multiparty Computation in Python" (`MPyC-Web`) [35] are two *recent popular* general-purpose examples for performing MPC on the web. Though, both engines only support one security model (passive security using Shamir's secret sharing with honest majority) and `JIFF` needs a central server for party communication. Although messages sent between parties are end-to-end encrypted, the central server presents a single point of failure, could be a communication bottleneck, and could infer certain computation-related metadata, like the amount of communication rounds.

Hence, while running MPC on the web is possible, one cannot choose from various security models for general-purpose computations with peer-to-peer (P2P) party communication and a dynamic amount of participating parties.

Our Contributions. With this work, we enable for the first time MPC computations on the web supporting different security models and virtually > 40 MPC protocols with active and passive corruptions as well as honest or dishonest majority.

We achieve this variety by bringing the native general-purpose MPC engine "Multi-Protocol SPDZ - A Versatile Framework for MPC" (`MP-SPDZ`) [27] to the web browser, dubbed `webSPDZ`. We built upon `MP-SPDZ` due its variety of protocols, active development such as frequent bug fixes and updates, and regular stable releases every few months as well as "popularity" as general-purpose MPC engine which has about 900 stars on GitHub and more than 250 forks.

Inspired by the pioneering work of Barak et al. [8], we use `Emscripten` to compile `MP-SPDZ`'s C++ backend to WebAssembly (Wasm), which is *usually* faster than JavaScript in the web browser [38,45]. For P2P-based party communication in the browser, we build upon "Web Real-Time Communication" (`WebRTC`).

Moreover, `webSPDZ` achieves the fastest MPC runtimes in the web browser in our end-to-end benchmarks. For benchmarking, we choose a simple MPC program to benchmark, yet one that is often used as a building block, which is the dot-product between two secret vectors. As MPC protocols, we use the secret-sharing schemes of Shamir (to practically compare `webSPDZ` with existing MPC web engines) and Replicated. To showcase a real-life case, we answer "who pays for dinner?" via MPC in the browser on three smartphones in $\sim$0.13s.

Outline. Section 3 describes the developed MPC web engine, `webSPDZ`. Section 4 evaluates and discusses the runtime performance of `webSPDZ`, `MPyC-Web`, and `JIFF`, as well as `webSPDZ`'s native variant (`MP-SPDZ`, which is dubbed `natSPDZ` in this section). Section 5 describes the related work in more detail. Additionally, Section 2 presents supporting preliminaries and Section 6 concludes this work.

2 Preliminaries for `webSPDZ`

In this section, we first briefly describe MPC with the related security models and protocols, and define the term "general-purpose MPC (web) engine". For further information, the interested reader can check, e.g., Smart [41] and Lindell[28]. Then, we show the main tools that bring `MP-SPDZ` to the web browser: Wasm, `Emscripten`, and `WebRTC`.

2.1 MPC-related Building Blocks

MPC enables various parties to jointly compute a function over secret inputs. While no party learns the input of other parties or intermediate results, all or a subset of dedicated parties receive the function outputs. MPC computations use a dedicated MPC protocol under a specific security model. To practically enable computations using generic MPC protocols/programs, parties use so-called "MPC engines".

MPC Security Models. One usually decides first how many parties can an adversary corrupt and the type of corruption. Depending on the amount of parties that can be (statically) corrupt, protocols largely classify into two categories:

1. Honest majority where at most half of the parties are corrupt.
2. Dishonest majority which allows up to all but one corrupted parties.

For types of corruptions, the MPC literature states primarily two:

1. Passive or semi-honest security where corrupted parties try to learn as much as possible from the protocol transcript.
2. Active or malicious security where corrupted parties can arbitrarily deviate from the protocol by sending malformed data.

MPC Protocols. To perform MPC computations, various protocols provide different levels of security. In this paper, we focus on protocols that work with arithmetic circuits. We built upon the secret-sharing protocols Shamir [15,39,2] and Replicated [6] for 3 parties. These protocols provide passive security in an honest-majority setting. Moreover, we use the replicated protocol for 4 parties by Dalskov et al. [18], which provides active security.

Since the most difficult task in MPC is to multiply two secrets, we use protocols that build upon "Beaver triples [9]". These Beaver triples are generated in an (i) input-independent preprocessing phase, and consumed during the (ii) actual (online) computation phase where parties get to use their secret inputs [29].

General-Purpose MPC Web Engines. With MPC engines, we denote a framework which enables parties to *actively participate* in MPC computations. If an MPC engine can be run in a web browser, we denote it as MPC web engine. With general-purpose, we denote an MPC engine which *virtually* supports arbitrary computations and a dynamic number of participating parties.

2.2 Web-Browser-related Building Blocks

To bring `MP-SPDZ` to the web browser, we primarily built upon three pillars: (1) Wasm to run code in the browser, (2) `Emscripten` to compile `MP-SPDZ`'s C++ BackEnd to Wasm, and (3) `WebRTC` for P2P communication in the browser.

Wasm [21]. WebAssembly (Wasm) has been developed to run code in the web browser safe, fast, portable, and compact [38]. As such, Wasm is a low-level binary instruction format, which uses only integers and floats, and is supported by most popular browsers [46]. An essential browser feature for our work is native 64-bit memory, which, e.g., Chrome and Firefox recently enabled by default. Since Wasm's pure runtime is close to native, it usually runs alongside JavaScript to speed up certain code parts.

Usually, Wasm is a compilation target and not written by hand. Thus, developers write native code and then compile it to Wasm (e.g., via `Emscripten`).

Emscripten [19]. Is the most-popular (open-source) tool that compiles native C/C++ code to Wasm. `Emscripten` builds upon an LLVM frontend and uses a backend that generates JavaScript code [50]. The toolchain is well-documented and provides many features. Some important features for our work are a set of already ported libraries to Wasm and many APIs that allow easy integration of, e.g., file systems, Wasm workers (threads), or C++ support for JavaScript objects.

The `Emscripten` compiler `emcc` replaces the native compiler and provides a variety of flags to, e.g., optimize performance and code size. `emcc` outputs a Wasm and JavaScript file. The JavaScript file loads the Wasm file and provides an API to interact with the compiled Wasm modules. For testing purposes, `emcc` also auto-generates an HTML file.

WebRTC [21]. Google released the "Web Real-Time Communication" (`WebRTC`) project as an open-source framework in 2011. Since then, `WebRTC` has been further developed and standardized by the World Wide Web Consortium (W3C). By now, most popular browsers support the technology.

`WebRTC` is a collection of APIs that enables encrypted peer-to-peer (P2P) communication between web browsers without additional plugins. Every `WebRTC` connection is encrypted using the protocols Datagram Transport Layer Security (DTLS) for key exchange and Secure Real Time (SRT) for encrypted data transmission. For further details on `WebRTC`'s security architecture, the interested reader can check, e.g., the Internet Engineering Task Force (IETF) [25].

The connection-building process is called Interactive Connectivity Establishment (ICE). First, each peer uses a STUN (Session Traversal Utilities for NAT) server to dissolve the public IP address and create an ICE candidate; while a TURN (Traversal Using Relays around NAT) server acts as a fallback. Then, these ICE candidates are exchanged between any two peers via a signaling server. When parties establish a `WebRTC` connection, the STUN/TURN/signaling servers are not needed anymore and the whole communication is P2P. Please note that the signaling process is not part of the `WebRTC` standard as it relies on other protocols like SIP (Session Initiation Protocol) over WebSockets. E.g., Blum et al. [10] and Sredojev et al. [43] give further details on `WebRTC`.

3 webSPDZ

This section describes the "Web Variant of Multi-Protocol-SPDZ" (`webSPDZ`). Initially, we describe `webSPDZ`'s source-code approach. Then, we describe how to transform `MP-SPDZ` into an MPC web engine.

Section 3.1 describes how to cross-compile `MP-SPDZ`'s C++ backend to Wasm using `Emscripten`. Section 3.2 describes the changes to enable P2P communication in the web browser using `WebRTC`. Besides `WebRTC`, Appendix B describes an alternative communication approach using `WebSockets`. Then, Section 3.3 gives an example of performing MPC in the web browser using `webSPDZ`. Finally, Section 3.4 describes security and privacy considerations for practical instantiations of `webSPDZ`.

Source Code. We open-sourced `webSPDZ` on GitHub[4]. Since we base `webSPDZ` on `MP-SPDZ`, most of the source code is identical to `MP-SPDZ`'s C++ backend. For further information on practical use of `webSPDZ`, please refer to the repository. For instance, we show `webSPDZ`'s tech stack, provide ready-to-use test files, and highlight some additional information on transforming `MP-SPDZ` to `webSPDZ`.

[4] github.com/tbuchs/webSPDZ

3.1 Compiling MP-SPDZ's C++ backend to WebAssembly

We show how to compile `MP-SPDZ`'s C++ backend to Wasm by first compiling the library dependencies. Then we describe the relevant code changes in `MP-SPDZ` to make it compatible with wasm. Finally we show some features we had to bring in, such as compiler settings, thread management, and various APIs. For more details on compiling C++ to Wasm using `Emscripten`, please refer to, e.g., `Emscripten`'s documentation.

Porting Library Dependencies. To support the > 40 protocols, `MP-SPDZ` mainly relies on four (open-source) libraries: Boost, Libsodium, OpenSSL, and GMP. We have added pre-compiled ".a archives" of the libraries so we can link them to the Wasm version.
Boost. Includes a set of libraries that `MP-SPDZ` uses, e.g., for network communication and multithreading. We replace the original library by linking a ported version of Boost from the open-source project Ports, using the flag `-sUSE_BOOST_HEADERS=1` .
Libsodium. `Emscripten` provides the command `emconfigure` , which can replace all default environment variables with `Emscripten`-specific ones. Libsodium provides a build script for `Emscripten`, which provides the necessary commands to build it from source.
OpenSSL. We must build OpenSSL manually using several flags.Then, we must adapt the compiler path in the resulting Makefile. The resulting ".a archive" is then linked in the framework.
GMP. `MP-SPDZ` uses GMP for arithmetic operations. We can use the original GMP library without any patches since the datatype `long` has the correct size of 8 bytes in 64-bit Wasm. We focus on 64-bit Wasm since most web browsers, such as Firefox and Chrome, support it now. For our experiments with 32-bit Wasm, please refer to Appendix A *(GMP Library for 32-bit webSPDZ)*.

Modifying MP-SPDZ implementation. Since some commands and functions are unavailable in a Wasm-based environment, we must adapt the respective code parts in `MP-SPDZ`'s C++ backend. For instance, some optimization approaches or network parts have to be revised. The following paragraphs show the most relevant code modifications to enable `webSPDZ`.
Architecture-specific inline assembly. `MP-SPDZ` optimizes runtime in many different ways. One is using inline assembly to enable high-speed operations via the CPU. However, since Wasm does not support inline assembly, we need to replace it with equivalent C++ code. Fortunately, `MP-SPDZ` provides architecture-specific implementations in many cases.
SIMD instructions. "Single Instruction, Multiple Data" (SIMD) instructions are another optimization technique in `MP-SPDZ`. For instance, MP-SPDZ uses AVX 256-bit instructions and the AES-NI and PCLMUL instruction sets to speed up AES and certain multiplications. However, e.g., these AES-related instructions are not available. Moreover, SIMD instructions in Wasm are usually

less performant than the original instruction sets. As such, we emulate many SIMD via software-specific code.
Networking. `MP-SPDZ` bases its networking on C++ (POSIX) sockets, either SSL sockets of the Boost library or default BSD sockets. Since web browsers do not support POSIX socket functions, such as `listen()` or `accept()`, `Emscripten` bridges WebSockets and POSIX sockets via proxy servers. Further, we also want P2P communication for the MPC computations on the web to enhance privacy. Therefore, we introduce a new networking layer for P2P party communication in a web browser environment. Section 3.2 *(Peer-to-Peer Party Communication in the Web Browser)* describes this new networking layer.
Makefile. In `MP-SPDZ`'s Makefile, we must change the compiler and linker flags for the used `Emscripten` APIs and the unsupported flags in a web browser environment. The remaining MP-SPDZ's Makefile is mostly unchanged. Finally, the `Emscripten` compiler `emcc` replaces the original `gcc` compiler.

Compiler Settings, Threads, and APIs. The large variety of implemented features in `Emscripten` simplifies the cross-compilation process. We use several APIs from `Emscripten` and various compiler settings. For instance, we use the compiler flags `-sMEMORY64=1`, which generates 64-bit Wasm code and `-sASYNCIFY`, which enables synchronous C++ code and asynchronous JavaScript code. We require this interplay of C++ and JavaScript code when waiting for a network response, such as input from other parties. However, such `Emscripten` compiler flags *can* lead to decreased performance and a larger code size.

Dealing with the UI Thread. The web browser uses the main thread to execute code and perform tasks in the UI. When we have a synchronous operation, such as waiting for another party's input, the UI will freeze until the operation has finished. We have implemented a solution that avoids this UI freezing by carefully implementing a thread-management solution. `webSPDZ`'s web application runs in a separate thread, using the `-sPROXY_TO_PTHREAD` compiler flag. The UI thread spawns a new thread within the application and only becomes active if needed. Within the application thread, multithreading is still possible.

However, in some cases, the application thread needs the support of the UI thread. For instance, the the `WebRTC` API must access the web browser's `Window component`, which the UI thread handles. Therefore, some networking parts can only run in the UI thread. `Emscripten` provides the `proxying.h` API to proxy between the threads. However, this proxying *can* lead to a performance decrease.

File system in the web browser. `Emscripten`'s file system API provides a POSIX-like file system in a web browser environment. We use `WasmFS`, a Wasm-based file system. `WasmFS` is faster than, e.g., the JavaScript-based filesystem.

3.2 Peer-to-Peer Party Communication in the Web Browser

Due to security and privacy considerations, We do not want to rely on an external (proxy) server for party communication within MPC computations. Thus, as a second preparation step, we enable P2P communication between MPC parties in the web browser.

To enable P2P communication, we adapt `MP-SPDZ`'s networking layer. We built upon `WebRTC` to be portable across various browsers while retaining security & privacy. To support `WebRTC` in `MP-SPDZ`'s C++ backend, we use the C++ library `libdatachannel`. The library's developer provides a ported Wasm version with a limited feature set called `libdatachannel-wasm14`. Since the library does not yet support Wasm64, we had to change the library's code for `webSPDZ` in some places. We benefited from `MP-SPDZ`'s modular C++ design when integrating the new networking layer. `MP-SPDZ` implements most communication in the `Player` class. We create a subclass called `WebPlayer` for the new `WebRTC` communication.

Further, we implement `WebRTC`'s signaling process for connection establishment. To establish a connection to a signaling web-socket server, which exchanges necessary *meta* connection information between parties, we use `Emscripten`'s WebSocket API. We implemented a JavaScript-based signaling server that accepts incoming web-socket connections from parties and forwards sent data to the other parties. When the signaling process finishes, the web-socket connection is closed and parties *purely* communicate via P2P for, e.g., MPC computations. Figure 1 shows `webSPDZ`'s networking communication with three parties. Please refer to, e.g., Section 2.2 for details on `WebRTC`'s connection establishment.

Moreover, we can introduce another networking approach for other cases with different networking requirements. As for the `WebPlayer`'s subclass, we can add another `Player`'s subclass with *relatively few* lines of code.

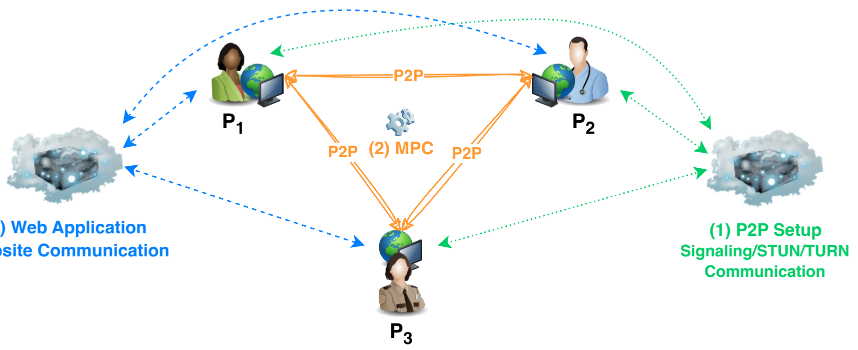


Fig. 1: `webSPDZ`'s networking communication with three parties. First (0) each party communicates with a web-app server that provides the MPC program. Next (1), each party sets up P2P channels using `WebRTC`'s signaling server. Finally (2), parties communicate via P2P for the actual MPC computation.

3.3 Running MP-SPDZ in the Web Browser using webSPDZ

This section describes the final assembly of transforming `MP-SPDZ` into `webSPDZ`. First, we describe a complete flow of performing an MPC computation in the web browser. While the execution steps are similar to the native variant of `MP-SPDZ`, we highlight the differences. The complete flow is the following:

1. **Compile webSPDZ.** After compilation, `Emscripten` outputs a `.wasm`, `.js`, and test `.html` file. We use the auto-generated `.html` file. The `.js` file serves as a bridge between the web browser's JavaScript engine and Wasm. Further, we can call the C++-backend's functions via the `.js` file.
2. **Write and compile the MPC program.** As for native `MP-SPDZ`, we write an MPC program in `MP-SPDZ`'s Python-based front-end language and compile it. Then, `MP-SPDZ`'s compiler outputs a schedule file containing the computation's meta information and a bytecode file containing low-level instructions for the C++ virtual machine.
3. **Provide webSPDZ's web application.** Now, we can host `webSPDZ` using the web-application server, which provides the `.wasm`, `.js`, and `.html` files.
4. **Parties join the MPC computation.** To start the MPC computation, each party opens `webSPDZ`'s web application via a web browser tab. Parties provide `webSPDZ`'s execution parameters as query parameters via the browser's address bar.
5. **Parties set up P2P communication.** When all required parties have joined, each party performs the P2P setup via `WebRTC`. When each party has finished the P2P setup, parties can communicate in a P2P way for the MPC computation.
6. **Parties perform the MPC computation in the web browser.** As for native `MP-SPDZ`, parties might provide input, depending on the respective MPC program, and jointly perform the MPC computation. After a successful MPC computation, pre-defined parties can read the computation results in their browser interface.

3.4 Security & Privacy Considerations of webSPDZ

This section discusses security- and privacy-related considerations of `webSPDZ`. First, we discuss (networking) communication-related aspects. Then, we discuss (MPC) computation-related aspects.

Communication-related security & privacy. For the communication during MPC computations, parties communicate in a P2P way via `WebRTC`. `WebRTC` encrypts these communications using the Secure Real-time Transport Protocol (SRTP). We only need the signaling server for the communication setup. However, the default communication setup does not authenticate parties' messages.

We use the signaling server due to usability. With the signaling server, parties can establish a P2P communication without *manually* exchanging connection information (the ICE candidates). In sensitive cases, we could use other channels to exchange connection information. For instance, one could use a *trusted* group channel in the Signal messaging app.

Regarding external entities, we can host both the signaling and `webSPDZ`'s web application, which provides the MPC program and engine, on the same server. Since both applications run on the same server, they fall into the same trust domain. This unification of trust increases the overall trust in the system, as one already needs to trust `webSPDZ`'s web application, which is the same as trusting the content of other web applications, such as google.com or eprint.iacr.org.

We use a Google STUN server (stun.1.google.com) to translate a party's IP address and create the ICE candidates. While the respective server learns when an IP address starts communicating, we must also trust the STUN server's answers. Hence, we balance a trade-off between privacy, usability, and trust. One can change the STUN server according to their needs. For instance, many publicly available STUN servers exist. Ideally, each application/organization self-hosts its STUN/TURN servers in sensitive cases.

Computation-related security & privacy. In general, `webSPDZ` provides the same computational security & privacy as the native variant of `MP-SPDZ`. Virtually, `webSPDZ` can support the same protocols. Hence, we can select the same protocols based on the desired level of security. For instance, we can equally select the corruption mode (passive vs. active security) and trust in the parties' majority (honest vs. dishonest majority). `MP-SPDZ` gives a detailed overview of the various security models and corresponding protocols, e.g., in its GitHub repository [1].

4 Performance Evaluation of webSPDZ

In this section, we practically evaluate the performance of `webSPDZ`. Initially, we describe the MPC program for our lab-environment benchmarks, the dot·product, and our benchmarking approach. Then, we show our results for the lab-environment benchmarks and the exemplary real-life case on three Android smartphones, "Dining MPC Phones", which answers who pays for dinner.

For the lab-environment benchmarks, we used the MPC protocols Shamir (3 parties) and Replicated (3 and 4 parties). Shamir and Replicated with 3 parties provide passive security (corruption) for an honest majority. Replicated with 4 parties provides active security. For the real-life case, we used Replicated with 3 parties.

Since both the native and web variants of `MP-SPDZ` are built upon the same BackEnd and use the same FrontEnd, in this section, we dub the native variant of `MP-SPDZ` "`natSPDZ`".

4.1 Evaluation Strategy

Programs & Protocols. For the lab-environment benchmarks, we build upon the strategy from Keller [27] and compute the **dot·product** of two vectors (multiplication is one of the most expensive operations in MPC). Figure 2 shows the source code for `webSPDZ` and `natSPDZ`, respectively (both use the same FrontEnd code). We select the most comparable MPC web engines and run them for two protocols.

As MPC protocols, we first use Shamir, as all engines support it, with 3 parties for 100,000 vector elements. Further, we use Replicated with 3 and 4 parties, respectively, using `webSPDZ` and `MP-SPDZ` for 100 up to 1,000,000 vector elements. Shamir and Replicated for 3 parties provide passive security for an honest majority (threshold $t = \left\lceil \frac{\#parties}{2} \right\rceil - 1$; e.g., $t = 1$ for 3 parties). Replicated with 4 parties provides active security. We set a field size of 128 bits. Except for `JIFF`, which we set to 48 bits as it could only handle up to 53 bits in our experiments, by explicitly setting the prime modulus.

As an exemplary real-life case, the "Dining MPC Phones", we answer: "Who pays for dinner?". The program *only* reveals the party with the highest salary while keeping the parties' salaries private. Three Android smartphones (= 3 parties) use `webSPDZ` via the Google Chrome browser, using the MPC protocol Replicated.

```
start_timer(1) # Full Program
n = int(program.args[1])
start_timer(2) # Input Phase
a = sint.Array(n)
b = sint.Array(n)
a.input_from(0) # Party 0
b.input_from(1) # Party 1
stop_timer(2)
start_timer(3) # Pure Dot·Product Computation
res = sint.dot_product(a,b)
stop_timer(3)
stop_timer(1)
```

Fig. 2: Source code of the **dot·product** for the two MPC engines `webSPDZ` and `natSPDZ`. Inspired by Keller's benchmarking code [26]. # denotes a comment.

Input & Measurements. In our benchmarks, Party 0 and 1 each input a secret vector and share it among all parties. We measure the runtime for (1) the whole program (Full), from startup until the final result, (2) the input phase (creating or reading in the vectors and sharing them among all parties), and (3) the pure dot·product computation. We do not measure compile times since we can compile an MPC program in advance. Moreover, we can reuse such compilations given the same parameters. We measured each setting three times and took the average.

Selected MPC Engines & Run Environment. We benchmarked the following MPC web engines:

- "JavaScript library for building web apps that employ MPC" (`JIFF`) [22]
  - *Version:* from June 6th 2024 [23] (Commit ID: *f5a22d8*). Dot-product product implementation based on Hastings et al.'s example code [33]. To improve performance, we reduced the server's log output.
  - *Browser:* we use `node.js` to run `JIFF`. Since both `node.js` and the web browser Google Chrome run on the V8 engine, using `node.js` *should* lead to comparable or even better performance than a pure browser environment.
- "Web Variant of Multiparty Computation in Python" (`MPyC-Web`) [35]
  - *Version:* from February 16th 2024 (Commit ID: *5b72f19*). Dot-product implementation based on Keller's benchmarking code [26] *(mpyc)*.
  - *Browser:* Firefox 132.0.1. Same as `webSPDZ`, `MPyC-Web` compiles the (Python) BackEnd source to Wasm.
- "Web Variant of Multi-Protocol-SPDZ" (`webSPDZ`) (this work)
  - *Version:* based on `MP-SPDZ`'s source code from June 20th 2024 (Commit ID in `MP-SPDZ`'s repository: *18e934f*; Commit ID in `webSPDZ`'s repository: *bfffc9d*). Dot-product implementation based on Keller's benchmarking code [26] *(mp-spdz)*.
  - *Browser:* Firefox Nightly 134.0a1. `webSPDZ` compiles the (C++) BackEnd source to Wasm.

Additionally, we compare `webSPDZ` with the native variant of `MP-SPDZ`, `natSPDZ`.

Section 5 *(Related Work)* gives further details on `JIFF` and `MPyC-Web`. For an as-fair-as-possible comparison, `natSPDZ` uses the same version of `MP-SPDZ` as `webSPDZ`.

We crafted Docker containers for each engine. For `webSPDZ`, `MP-SPDZ`, and `JIFF`, each party ran in its own Docker container with a limit of 4GB RAM and 4 CPUs of an *x86_64 AMD EPYC 7502 32-Core Processor* using Linux. `webSPDZ`'s Docker containers simulated the web browser. For `MPyC-Web`, we connected the parties via browser tabs from the local machine (outside the Docker container).

Reproducibility. We have open-sourced `webSPDZ` and our benchmarking data on GitHub[5]. For instance, we provide the source code and Docker files for all benchmarked MPC engines. Based on the benchmarking machine, the results *may* vary. However, the relative runtimes of the various engines *should be* similar.

[5] github.com/tbuchs/webSPDZ

4.2 Evaluation Results

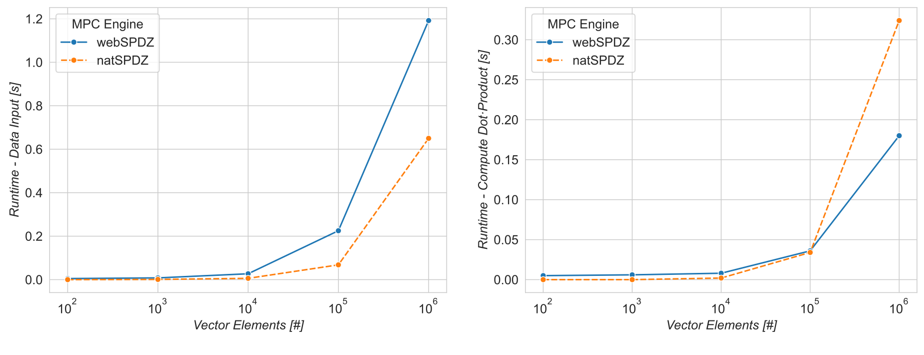
Table 1 shows the results of our lab-environment benchmarks. Additionally, Figure 3 shows Replicated runtimes with 3 parties for 100 up to 1,000,000 (1 million) vector elements for the program phases Input and Pure Computation. Figure 4 shows the “Dining MPC Phones” case evaluation on three smartphones. webSPDZ nominated the party that pays for dinner (highest salary) in ∼0.13s .

Shamir Protocol for 3 Parties. To compute the dot·product for 100,000 vector elements, `JIFF` takes overall ∼4.5min. `MPyC-Web` takes overall ∼6.6s (1.17s Input; 0.001s Pure Computation). Thus, `MPyC-Web` seems to have a heavier initialization phase. `webSPDZ` and `MP-SPDZ` take overall ∼1.2s and ∼0.12s respectively. `MP-SPDZ`’s runtime aligns with Keller’s results [27] (0.08s using 64 bits).

Replicated Protocol for 3 and 4 Parties. To compute the dot·product for 100,000 vector elements using the Replicated protocol, `webSPDZ` takes overall ∼0.26s (0.23s Input; 0.03s Pure Computation) and ∼1.83s for 3 and 4 parties, respectively. `MP-SPDZ` takes overall ∼0.072s (68ms Input; 4ms Pure Computation) and ∼0.12s for 3 and 4 parties, respectively. As for the Shamir protocol, `MP-SPDZ`’s runtime aligns with Keller’s results [27] (0.03s using 64 bits for the 3-party variant).

For the extended evaluation of Replicated with 3 parties using 100 up to 1,000,000 (1 million) vector elements, the **overall runtime** of `webSPDZ` is first similar to `MP-SPDZ` and then increases faster when inputting more data.

The **Pure Computation runtime** of `webSPDZ` is similar to `MP-SPDZ` up to 100,000 vector elements. However, interestingly, for 1,000,000 vector elements, `webSPDZ` was even *slightly* faster than `MP-SPDZ`.

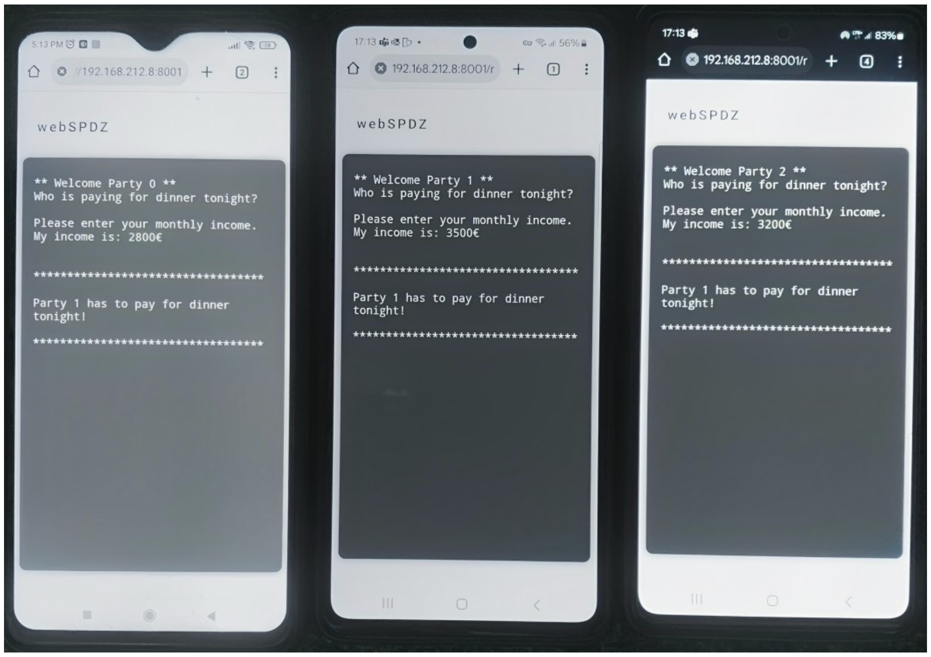


(a) Input Phase (b) Pure Computation Phase

Fig. 3: Runtime in seconds of the dot·product for various vector elements in an unrestricted (LAN-like) network. Using the MPC protocol Replicated for 3 parties with the MPC engines `webSPDZ` and `MP-SPDZ` (dotted line).

Table 1: Time in seconds to compute the dot·product for 100,000 vector elements, using the MPC protocols Shamir and Replicated with 3 parties in an unrestricted (LAN-like) environment. In an honest-majority setting with passive security, where 2 parties provide input. Averaged over 3 runs and party with max. Full runtime shown. Since only `webSPDZ` supports protocols beyond Shamir, we do not show Replicated timings for the other MPC web engines.

MPC Engines	**Shamir (3P)**			**Rep. Ring (3P)**		
	Full	Input	Pure Comp.	Full	Input	Pure Comp.
Web						
JIFF (48-bit)	265.11	49.76	215.34	⊥	⊥	⊥
MPyCweb (128-bit)	6.58	1.17	**0.001**	⊥	⊥	⊥
webSPDZ (128-bit)	**1.16**	**1.04**	0.11	0.26	0.23	0.03
Native						
natSPDZ (128-bit)	0.12	0.11	0.007	0.072	0.068	0.004



```
** Welcome Party 0 **
Who is paying for dinner tonight?

Please enter your monthly income.
My income is: 2800€

**********************************

Party 1 has to pay for dinner
tonight!

**********************************
```

```
** Welcome Party 1 **
Who is paying for dinner tonight?

Please enter your monthly income.
My income is: 3500€

**********************************

Party 1 has to pay for dinner
tonight!

**********************************
```

```
** Welcome Party 2 **
Who is paying for dinner tonight?

Please enter your monthly income.
My income is: 3200€

**********************************

Party 1 has to pay for dinner
tonight!

**********************************
```

Fig. 4: "Dining MPC phones". Three Android smartphones evaluated who pays for dinner in ~0.13s, using `webSPDZ` (Replicated) via the Google Chrome browser.

Σ **Summary.** One of the main runtime bottlenecks is the input phase. In the web browser, inputting data seems costlier than in a native environment, probably because of higher networking costs.

Further, for `JIFF`, the Pure Computation runtime, and `MPyC-Web`, the initialization phase, took relatively long too. Although `JIFF` uses only a field size of 48 bits, it is the slowest engine in our experiments. We assume that JIFF's slow runtime originates from (i) JavaScript and (ii) routing each party message via a coordination server. Further, `MPyC-Web` has the fastest Pure Computation runtime in our benchmarks. The fast runtime *might* originate from running all parties and the setup server in the same Docker container. In this case, we assume that `MPyC-Web`'s runtime is equal to or faster than running each party in an individual Docker container.

Hence, `webSPDZ` is the **fastest and most versatile MPC web engine** (in our end-to-end benchmarks). Virtually, `webSPDZ` can support each `MP-SPDZ`'s MPC protocols (>40).

5 Related Work

In this section, we compare recent related work regarding MPC engines which either run in the web browser (showing additional details for `JIFF` and `MPyC-Web`), or are specifically designed to enhance usability. For each engine, we also show the year of appearance until the last active commit if the developers have open-sourced it. Finally, we sum up gathered lessons learnt from the related work.

In the last decade, several MPC engines have been developed. The majority of these engines aims to be as fast as possible. On this quest, these engines have been built upon different programming languages. For instance:

- **C++**: `MP-SPDZ` [27], `MOTION` [14], `ABY3` [34], or `Sequre` [40].
- **Rust**: `swanky` [20].
- **Python**: `MPyC` [3].
- **Java:** `FRESCO` **[5]**.
- **JavaScript:** `JIFF` **[22]**.
- **"SecreC 2" [37]**: Which has been specifically designed for MPC and is used by, e.g., `Sharemind` [12].

For further MPC engines, please refer to, e.g., Rotaru's "Awesome MPC" list [4], Hastings et al.'s SoK paper [24], or Keller's MPC-engine overview in the `MP-SPDZ` paper [27].

While these MPC engines provide relatively fast computations and/or a variety of MPC protocols with different security models, most require tech-related knowledge to set everything up. Further, for each engine setup, specific MPC knowledge would be beneficial, e.g., which protocol and security model to use. These tech-related and MPC knowledge prerequisites are inherent entry barriers for most end users.

Thus, some MPC engines address usability, too, such as Sharemind [12] or `EasySMPC` [48]. Further, two engines provide a way to perform MPC in a web browser: JIFF and MPyC-Web. These web-compliant engines inherently enhance usability for end users as they *just* need to open a compatible browser tab.

5.1 Other MPC Web Engines

As briefly mentioned in Section 1 *(Introduction)*, already in 2018, Barak et al. [8] have performed MPC in the web browser.

Barak et al. developed a large-scale application which consisted of an administrative web app, as well as a design and implementation of a dedicated MPC protocol for low-bandwidth participants, dubbed HyperMPC. As such, they even benchmarked a scenario with *500 participating parties*. The web app provided a user-friendly way to provide input and join the computation. The computation could be joined, e.g., via the web browser, a mobile device, a cloud service, or an IoT device.

In the appendix, Barak et al. briefly describe their way of performing MPC in the web browser. They built upon, e.g., Emscripten but reported issues of not having SIMD in the web browser. Though, while the web-app's source code is available on GitHub [16], we neither found the source code of HyperMPC, nor of their approach to compile the engine for a web browser environment.

The "JavaScript library for building web apps that employ MPC" (JIFF) and the "Web Variant of Multiparty Computation in Python" (MPyC-Web) are two *recent popular* examples for performing MPC in the web browser.

JIFF (2017–2024–. . .) [22]. Is a general-purpose MPC web engine based on JavaScript, created by the "Boston multiparty group". Its source code is publicly available on GitHub [23]. MPC computations use Shamir's secret sharing for passive security in a pre-processing and online phase. The pre-processing phase operates in an honest-majority setting. The online phase operates in a dishonest-majority setting. Thus, overall JIFF operates in an honest-majority setting.

For party communication, JIFF uses a central coordination server. Besides the usual trust assumptions on honest parties in MPC, this central server presents one of the main privacy concerns and communication bottlenecks. Moreover, the "Boston multiparty group" also provides JIGG, which operates on garbled circuits.

MPyC-Web (2023–2024–. . .) [35]. Is a general-purpose MPC web engine based on Python and Wasm, created by Schoenmakers, Nikolov, and others. MPyC-Web builds upon the "Native Variant of Multiparty Computation in Python" (MPyC-Nat)[3] and uses PyScript to compile the Python source code to Wasm and the JavaScript library PeerJS for peer-to-peer communication via WebRTC. Its source code is publicly available on GitHub [35]. MPC computations use Shamir's secret sharing for passive security in an honest-majority [31].

5.2 Usability-focused MPC Engines

Besides MPC web engines which inherently provide enhanced usability, other MPC engines specifically address usability. E.g., `EasySMPC`, which has been created due to a lack of usability for the target audience in the biomedical sector. Or `Sharemind`, which aims to provide an easy-to-use statistics tool similar to the plaintext computation tool “R”.

EasySMPC (2020–2023) [48]. Is an MPC engine based on Java, created by Wirth, Kussel, Müller, Hamacher, and Prasser. `EasySMPC`’s source code is publicly available on GitHub [49]. MPC computations use arithmetic secret sharing, with a dynamic number of participating parties. Though, `EasySMPC` only supports additions and subtractions. As `JIFF`, the engine needs a *kind of central server* as their party communication uses emails. As communication alternative, it supports a dedicated (micro) server.

`EasySMPC` focuses on usability in the biomedical sector, offering a dedicated (native) Java-based desktop app. One of their main usability advantages is a graphical user interface, where users can enter input as in a web form. While the Java desktop app worked for their specific use case, such a special-purpose native approach limits the scope of portability.

Sharemind (2008–2024–. . .) [12]. Is an MPC engine based on “SecreC 2”, created by Bogdanov, Laur, and Willemson. `Sharemind`’s source code regarding the “SecreC 2” standard library or SDK is publicly available on GitHub [17]. MPC computations use additive secret sharing, fixed to 3 parties, supporting arbitrary computations.

One focus of Sharemind is easy-to-use statistics. As such, the engine has been used by a *recent* cross-hospital data analytics project [7]. The project operated on *real-world* cancer-related patient data.

5.3 Σ All in All

While MPC engines exist that address usability either specifically or inherently (web-based), the main limitation of the related work is the versatility of MPC protocols and security models. Each investigated engine supports one protocol in a passive honest-majority setting. `webSPDZ` *currently* supports five protocols and virtually as many as `MP-SPDZ` (>40).

Besides, some engines use a central server for communication, such as `JIFF` or `EasySMPC`, which adds privacy concerns. `webSPDZ` needs a server for the setup phase, e.g., to share the MPC’s Wasm code and arrange connection information, but the parties use a P2P connection for the actual MPC computations.

One significant usability-enhancing aspect of some engines is the graphical interface. `EasySMPC` even allows input to be provided purely via their graphical interface. While `webSPDZ` provides easy-to-access computations via a web browser tab, parties usually must provide an input file.

6 Future Work

In this section, we outline potentially interesting future work. First, upgrading webSPDZ with a focus on MPC protocols, performance, and usability. Then, to further evaluate the performance of webSPDZ itself and in comparison with other MPC (web) engines, by performing further benchmarks for a variety of MPC settings. Moreover, we show miscellaneous potentials, such as use cases, run environments, and verifiability of computations.

Upgrading webSPDZ. webSPDZ currently supports five MPC protocols: passive and active Shamir, Replicated Ring for 3 and 4 parties, and semi2k (2 parties). Porting protocols from MP-SPDZ to webSPDZ takes some manual effort. By smoothening or even automating the porting process, we can enable all (>40) MP-SPDZ-supported protocols faster.

webSPDZ's browser-related tools, Wasm, Emscripten, and WebRTC, will *likely* be further developed. Thus, over time, Wasm-related operations in the browser will *likely* get faster. As soon as more browser features are supported and integrated in webSPDZ, e.g., more/full support for SIMD or improvements in the filesystem API, we expect a performance increase for webSPDZ.

Parties currently provide program parameters as query parameters in the web browser's address bar. Especially for non-tech users, providing a graphical way to configure relevant MPC settings, such as the number of parties or used protocol, and providing input would increase usability.

Another aspect is the loading of MPC programs. Parties access MPC programs via a URL from webSPDZ's app server. If parties compile an MPC program directly in the browser, trust in the overall system can be increased.

Further Benchmarks & MPC Engines. On the one hand, one could evaluate the performance more comprehensively by testing various MPC settings, such as network delays, number of parties, or MPC protocols. For instance, Lorünser & Wohner [31] benchmarked the native variants of MP-SPDZ and MPyC and observed a relative runtime change between the two engines for different network delays. It would be interesting to see if the same holds for the engines' web variant. Moreover, measuring webSPDZ more fine-grained could lead to valuable insights into potential runtime bottlenecks. Next to runtime performance, communication (number of protocol rounds and network data) could also be measured.

On the other hand, one could evaluate further MPC web engines and their native parts. The leading question hereby could be, how far are we from closing the gap between MPC native and MPC web engines? Especially with continuous feature development in the web browser in mind, this might lead to surprising discoveries. Due to our positive experience with the compilation tool Emscripten, one could experiment with porting further C++-based MPC native engines to the web browser. For instance, MOTION [14] or Sequre [40], or the Java-based engine FRESCO [5] using tools like wasmer-java or TeaVM. Moreover, since Sequre was faster than native MP-SPDZ for some of their benchmark settings [40], one could investigate if the same holds for the web.

Miscellaneous. Since we compiled `MP-SPDZ`'s C++ BackEnd to Wasm, one could investigate further run targets. For instance, IoT devices that support Wasm in a smart-home environment. Such run targets even further broaden the applicability of MPC.

Moreover, many approaches exist to verify, e.g., the correctness of a program run or create proofs for secret-shared data. For instance, Ozdemir and Boneh [36] use zero-knowledge proofs for distributed secrets. One could investigate if "Verifiable MPC" is equivalent on the web?

7 Conclusions

Our vision is to enable MPC for everyone by making it simple to use the technology. One promising solution is to use web-based MPC engines, which enable a party to actively join an MPC computation by *simply* opening a web browser tab.

While some MPC engines address end-user usability specifically or inherently (web-based), each investigated engine supports one protocol in a passive, honest-majority setting. Thus, the main limitation of the state-of-the-art is the versatility of MPC protocols and security models. Besides, some engines use a central server for communication (e.g., `JIFF` or `EasySMPC`), which adds privacy concerns and potential bottlenecks.

That is why we created `webSPDZ`, the web variant of `MP-SPDZ`, one of the most performant and flexible general-purpose MPC engines, which supports > 40 protocols with different security models. Hence, `webSPDZ` enables many security models on the web. To transform `MP-SPDZ` into an MPC web engine, we use (i) `Emscripten` to compile `MP-SPDZ`'s C++ BackEnd to WebAssembly and (ii) `WebRTC` to enable peer-to-peer (P2P) party communication in the browser.

As with the native variants of the two recent MPC web engines, `MPyC-Web` and `JIFF`, `webSPDZ` outperforms them in our end-to-end experiments. Compared to native engines, one of the main bottlenecks in the browser seems to be the phase of inputting data. For pure computation, `webSPDZ` performed almost identically to its native variant and even outperformed it for one setting. The reason for the slower input phase could originate from, e.g., a slower file system or less efficient P2P party communication in the browser. Closing the gap between MPC native and web engines would be one of the interesting future works. As one usability-enhancing aspect, `webSPDZ` could add a GUI for, e.g., inputting data.

We believe that `webSPDZ` brings forth many interesting and practically relevant use cases. Thus, `webSPDZ` pushes the boundaries of practical MPC: to make MPC more usable and enabling it for a broader community.

Acknowledgments. This work received funding from (i) **PREPARED**, a project funded by the Austrian security research programme KIRAS of the Federal Ministry of Finance (BMF) and (ii) **LICORICE**, a European Union's Horizon Europe project (no. 101168311).

References

1. data61 / Marcel Keller: MP-SPDZ: A versatile framework for multi-party computation (2024), retrieved November 13th from github.com/data61/MP-SPDZ
2. Abspoel, M., Dalskov, A.P.K., Escudero, D., Nof, A.: An efficient passive-to-active compiler for honest-majority MPC over rings. In: Sako, K., Tippenhauer, N.O. (eds.) Applied Cryptography and Network Security - 19th International Conference, ACNS 2021, Kamakura, Japan, June 21-24, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12727, pp. 122–152. Springer (2021), doi.org/10.1007/978-3-030-78375-4_6
3. et al., B.S.: Mpyc: Multiparty computation in python (landing page) (2024), retrieved November 13th from www.win.tue.nl/ berry/mpyc
4. et al., D.R.: awesome-mpc: A curated list of multi party computation resources and links (2024), retrieved November 25th 2024 from github.com/rdragos/awesome-mpc
5. Alexandra Institute: FRESCO - a FRamework for Efficient Secure COmputation (2015), retrieved November 25th 2024 from github.com/aicis/fresco
6. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 805–817. ACM (2016), doi.org/10.1145/2976749.2978331
7. Ballhausen, H., Corradini, S., Belka, C., Bogdanov, D., Boldrini, L., Bono, F., Goelz, C., Landry, G., Panza, G., Parodi, K., Talviste, R., Tran, H.E., Gambacorta, M.A., Marschner, S.: Privacy-friendly evaluation of patient data with secure multiparty computation in a european pilot study. npj Digit. Medicine **7**(1) (2024), doi.org/10.1038/s41746-024-01293-4
8. Barak, A., Hirt, M., Koskas, L., Lindell, Y.: An end-to-end system for large scale P2P mpc-as-a-service and low-bandwidth MPC for weak participants. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 695–712. ACM (2018), doi.org/10.1145/3243734.3243801
9. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings. Lecture Notes in Computer Science, vol. 576, pp. 420–432. Springer (1991), doi.org/10.1007/3-540-46766-1_34
10. Blum, N., Lachapelle, S., Alvestrand, H.: Webrtc: real-time communication for the open web platform. Commun. ACM **64**(8), 50–54 (2021), doi.org/10.1145/3453182
11. Bogdanov, D., Kamm, L., Laur, S., Pruulmann-Vengerfeldt, P.: Secure multi-party data analysis: end user validation and practical experiments. IACR Cryptol. ePrint Arch. p. 826 (2013)
12. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.) Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008), doi.org/10.1007/978-3-540-88313-5_13

13. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Financial Cryptography. Lecture Notes in Computer Science, vol. 5628, pp. 325–343. Springer (2009)
14. Braun, L., Demmler, D., Schneider, T., Tkachenko, O.: MOTION - A framework for mixed-protocol multi-party computation. ACM Trans. Priv. Secur. **25**(2), 8:1–8:35 (2022), doi.org/10.1145/3490390
15. Cramer, R., Damgård, I., Maurer, U.M.: General secure multi-party computation from any linear secret-sharing scheme. In: Preneel, B. (ed.) Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding. Lecture Notes in Computer Science, vol. 1807, pp. 316–334. Springer (2000), doi.org/10.1007/3-540-45539-6_22
16. cryptobiu: Mpc simulation framework (github) (2018), retrieved November 13th 2024 from github.com/cryptobiu/MATRIX
17. Cybernetica: Sharemind sdk - development tools for apps on sharemind (2015), retrieved November 26th from github.com/sharemind-sdk
18. Dalskov, A.P.K., Escudero, D., Keller, M.: Fantastic four: Honest-majority four-party secure computation with malicious security. In: Bailey, M.D., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 2183–2200. USENIX Association (2021), usenix.org/conference/usenixsecurity21/presentation/dalskov
19. Emscripten: Emscripten is a complete compiler toolchain to webassembly, using llvm, with a special focus on speed, size, and the web platform. (2024), retrieved November 28th 2024 from emscripten.org
20. Galois, Inc.: swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky (2019), retrieved November 25th 2024 from github.com/aicis/fresco
21. Google: Real-time communication for the web (2024), retrieved November 28th 2024 from webrtc.org
22. multiparty group, B.U.: Jiff: Javascript library for building applications that rely on secure multi-party computation (landing page) (2024), retrieved November 13th from multiparty.org/jiff
23. multiparty group, B.U.: Jiff: Javascript library for building web-based applications that employ mpc (github) (2024), retrieved November 13th from github.com/multiparty/jiff
24. Hastings, M., Hemenway, B., Noble, D., Zdancewic, S.: Sok: General purpose compilers for secure multi-party computation. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 1220–1237. IEEE (2019), doi.org/10.1109/SP.2019.00028
25. (IETF), I.E.T.F.: Webrtc security architecture (2021), retrieved November 28th 2024 from datatracker.ietf.org/doc/html/rfc8827
26. Keller, M.: Benchmarks for various multi-party computation frameworks (2020), retrieved November 13th from github.com/mkskeller/mpc-benchmarks
27. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (2020), doi.org/10.1145/3372297.3417872
28. Lindell, Y.: Secure multiparty computation. Commun. ACM **64**(1), 86–96 (2021), doi.org/10.1145/3387108

29. Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 259–276. ACM (2017), doi.org/10.1145/3133956.3133999
30. Liu, M., Cheng, L., Gu, Y., Wang, Y., Liu, Q., O'Connor, N.E.: MPC-CSAS: multi-party computation for real-time privacy-preserving speed advisory systems. IEEE Trans. Intell. Transp. Syst. **23**(6), 5887–5893 (2022)
31. Lorünser, T., Wohner, F.: Performance comparison of two generic mpc-frameworks with symmetric ciphers. In: ICETE (2). pp. 587–594. ScitePress (2020)
32. Lytvyn, O., Nguyen, G.T.: Secure multi-party computation for magnetic resonance imaging classification. In: ANT/EDI40. Procedia Computer Science, vol. 220, pp. 24–31. Elsevier (2023)
33. Marcella Hastings et al.: Sample code and build environments for mpc frameworks (2024), retrieved November 13^{th} 2024 from github.com/MPC-SoK/frameworks
34. Mohassel, P., Rindal, P.: Aby^3: A mixed protocol framework for machine learning. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 35–52. ACM (2018), doi.org/10.1145/3243734.3243760
35. Nikolov, E., Schoenmakers, B.: Mpyc web (github) (2024), retrieved November 13^{th} 2024 from github.com/e-nikolov/mpyc-web
36. Ozdemir, A., Boneh, D.: Experimenting with collaborative zk-snarks: Zero-knowledge proofs for distributed secrets. In: Butler, K.R.B., Thomas, K. (eds.) 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022. pp. 4291–4308. USENIX Association (2022), usenix.org/conference/usenixsecurity22/presentation/ozdemir
37. Randmets, J., et al.: Programming languages for secure multi-party computation application development. Ph.D. thesis, University of Tartu Tartu, Estonia (2017)
38. Rossberg, A., Titzer, B.L., Haas, A., Schuff, D.L., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F., Holman, M.: Bringing the web up to speed with webassembly. Commun. ACM **61**(12), 107–115 (2018)
39. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979). https://doi.org/10.1145/359168.359176, https://doi.org/10.1145/359168.359176
40. Smajlovic, H., Shajii, A., Berger, B., Cho, H., Numanagic, I.: Sequre: a high-performance framework for rapid development of secure bioinformatics pipelines. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS Workshops 2022, Lyon, France, May 30 - June 3, 2022. pp. 164–165. IEEE (2022). https://doi.org/10.1109/IPDPSW55747.2022.00040, https://doi.org/10.1109/IPDPSW55747.2022.00040
41. Smart, N.P.: Computing on encrypted data. IEEE Secur. Priv. **21**(4), 94–98 (2023), doi.org/10.1109/MSEC.2023.3279517
42. Song, J., Jin, M., Woo, H.: A fast browsing model for web contents on wearable devices. In: ICCE. pp. 376–377. IEEE (2017)
43. Sredojev, B., Samardzija, D., Posarac, D.: Webrtc technology overview and signaling solution design and implementation. In: Biljanovic, P., Butkovic, Z., Skala, K., Mikac, B., Cicin-Sain, M., Sruk, V., Ribaric, S., Gros, S., Vrdoljak, B., Mauher, M., Sokolic, A. (eds.) 38th International Convention on Information and Communication Technology, Electronics and Microelectronics,

MIPRO 2015, Opatija, Croatia, May 25-29, 2015. pp. 1006–1009. IEEE (2015), doi.org/10.1109/MIPRO.2015.7160422

44. Thoma, C., Cui, T., Franchetti, F.: Secure multiparty computation based privacy preserving smart metering system. In: 2012 North American Power Symposium (NAPS). pp. 1–6 (2012). https://doi.org/10.1109/NAPS.2012.6336415
45. Wang, W.: Empowering web applications with webassembly: Are we there yet? In: ASE. pp. 1301–1305. IEEE (2021)
46. WebAssembly: Webassembly: Feature extensions (2024), retrieved November 28th 2024 from webassembly.org/features
47. Wikipedia: Single instruction, multiple data (simd) (2024), retrieved April 29th 2024 from en.wikipedia.org/wiki/Single_instruction,_multiple_data
48. Wirth, F.N., Kussel, T., Müller, A., Hamacher, K., Prasser, F.: EasySMPC: a simple but powerful no-code tool for practical secure multiparty computation. BMC Bioinform. **23**(1), 531 (2022)
49. Wirth, F.N., Kussel, T., Müller, A., Prasser, F.: Easysmpc - no-code secure multi-party computation (2020), retrieved November 26th from github.com/easy-smpc/easy-smpc
50. Zakai, A.: Emscripten: an llvm-to-javascript compiler. In: Lopes, C.V., Fisher, K. (eds.) Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011. pp. 301–312. ACM (2011), doi.org/10.1145/2048147.2048224

A GMP Library for 32-bit webSPDZ

In earlier versions of `webSPDZ`, we experimented with 32-bit Wasm. However, since, e.g., the web browsers Firefox and Chrome now support 64-bit Wasm, we focus on 64-bit Wasm in this paper. For the interested reader, in this section, we describe one of our main challenges of 32-bit `webSPDZ`, porting the C++ library GMP to 32-bit Wasm.

As GMP relies on C89, where the datatype `long long` is unavailable, porting GMP to Wasm requires manual effort. Since the datatypes `long` and `size_t` have only 4 bytes in 32-bit architectures like Wasm32, we need the datatype `long long` as it has 8 bytes in any architecture.

Luckily, one of the developers of GMP has been very helpful and provided a patched version of GMP. After adding the patch to the library, we can define the GMP datatype `mp_limb_t` as 8 bytes, equal to a `long long`. Then, we can compile GMP to Wasm using `Emscripten`. We configure cross-compilation to Wasm using the following command: `emconfigure ./ configure -disable-assembly -host none -enable-cxx ABI=longlong CFLAGS=-fPIC CXXFLAGS=-fPIC`. The patch helped a lot in porting the library, though it was not officially available in the GMP repository when we experimented with 32-bit `webSPDZ`.

B Alternative Party Communication using WebSockets

Besides the described peer-to-peer (P2P) party communication using `WebRTC`, we experimented with `WebSockets` to increase performance. Each (communication) thread can open a `WebSocket` connection. Thus, `WebSockets` enable multithreading for sending and receiving data, while `WebRTC` can only send and receive data via the browser's main thread. However, we did not see a significant performance increase in our experiments. Moreover, `WebSocket` communication needs a server, while `WebRTC` communication is P2P (for the MPC computation).

For further details, e.g., Emscripten's documentation shows various ways to use `WebSockets` with `Emscripten`.